Language Support for Design Patterns

Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: Jan.Bosch@ide.hk-r.se
www: http://www.pt.hk-r.se/~bosch

Abstract

Design patterns have proven to be useful for the design of object-oriented systems. The power of a design pattern lies in its ability to provide generic solutions that can be specialised for particular situations. The implementation of design patterns has received only little attention and we have identified two relevant problems associated with the implementation. First, the traceability of a design pattern in the implementation is often insufficient; often the design pattern is 'lost'. Second, implementing design patterns may present significant implementation overhead for the software engineer. Often, a, potentially large, number of simple methods has to be implemented with trivial behaviour, e.g. forwarding a message to another object. In this paper, the layered object model (**LayOM**) is presented. **LayOM** provides language support for the explicit representation of design patterns in the programming language. **LayOM** is an extended object-oriented language in that it contains several components that are not part of the conventional object model, such as *states*, *categories* and *layers*. Layers are used to represent design patterns at the level of the programming language and example layer types for four design patterns are presented. **LayOM** is supported by a development environment that translates **LayOM** code into C++. The generated C++ code can be used as any C++ code for the development of applications. An important aspect of **LayOM** is that the language itself is extensible. This allows new design patterns to be added to the language model.

1 Introduction

Design patterns are becoming increasingly popular as a mechanism to describe solutions to general design problems that can be customised to particular applications. Several authors, e.g. [Gamma et al. 94, Pree 94, Coplien & Schmidt 95], have written about design patterns. Several categories of design patterns have been proposed for general, i.e. domain independent, patterns, but others have proposed patterns for particular domains. The number of available design patterns is growing constantly.

Most authors propose design patterns as a mechanism specifically used during design. The relation to the implementation level is often discussed in terms of example or template code that allows the software engineer to conveniently transform the design pattern. In general, a traditional object-oriented language such as C++ is used. The disadvantage of a traditional language such as C++ is that no support for the representation of design patterns is provided by the language. This leads to traceability and implementation overhead problems related to the implementation of design patterns.

In this paper, the layered object model (**LayOM**) is proposed as an alternative approach. **LayOM** is a language model that provides explicit support for modelling constructs used during object-oriented design, such as *design patterns*, but also *relations* between objects and *abstract object state*. An object in **LayOM** consists, next to instance variables and methods, of states, categories and layers. Layers encapsulate the object and intercept messages that are send to and by the objects. The layers are organised into classes and each layer class represents a concept, such as a relation with another object or a design pattern. **LayOM** is supported by a development environment that translates classes and applications defined in **LayOM** into C++ code. The generated C++ code can then be used to construct applications, either direct or integrated with existing C++ code. The advantage of using **LayOM** instead of a traditional object-oriented language is

that it does not suffer from the identified problems related to the implementation of design patterns. **LayOM** can be used without losing compatibility with legacy systems and code developed elsewhere as it is translated to C++. The resulting C++ code can be integrated in other code as any C++ program.

The remainder of this paper is organised as follows. In the next section, design patterns are introduced and the problems associated with the implementation of design patterns are discussed. In section 3, the layered object model is presented. Section 4 describes the implementation of two structural and two behavioural design patterns as layers of LayOM. Section 5 compares the presented ideas to related work and the paper is concluded in section 6.

2 Design Patterns

Patterns, as a concept, originates from the work by Christopher Alexander [Alexander *et al.* 77], an architect who developed patterns for designing e.g. a house. Each pattern describes a recurring problem and a generic solution that can be adopted for particular situations. A set of patterns can be organised in a *pattern language*, thus providing a set of composable solutions for problems in a particular domain.

The concept of patterns has been adopted in object-oriented as *design patterns*. [Gamma *et al.* 94] define design patterns as *descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context*. The authors present a catalogue of 23 design patterns, organised in three categories depending on the pattern's purpose. *Creational patterns* are concerned with object creation, whereas *structural patterns* address the composition of classes and objects. *Behavioural patterns* are concerned with the ways in which classes or objects interact and distribute responsibility.

However, the domain of patterns is not limited to the design patterns discussed by [Gamma *et al.* 94]. For example, [Pree 95] discusses, next to the design pattern catalogue by [Gamma *et al.* 94], object-oriented patterns [Coad 92], coding patterns, framework cookbooks and formal contracts [Helm *et al.* 90] as patterns for solving general design problems. Also, in [Coplien & Schmidt 95], several design pattern related papers are presented. Due to the fact that design patterns have only recently become popular in the object-oriented community, no generally accepted classification of design patterns exists.

The description of design patterns has been kept very brief for reasons of space. We refer to [Buschmann & Meunier 95, Coad 92, Coplien & Schmidt 95, Gamma *et al.* 94, Mattsson 95, Pree 95] for a more extended overview of design patterns.

2.1 Problems of Implementing Design Patterns

We focus in this paper on the implementation in an object-oriented language of the design patterns that are contained in an object-oriented design model. We have experienced problems when implementing the design patterns in a traditional object-oriented language as C++. These problems are primarily related to the traceability of design patterns in the implementation and the implementation overhead of design patterns.

- **Traceability**: The traceability of a design pattern is often lost because the programming language does not support a corresponding concept. The software engineer is thus required to implement the pattern as distributed methods and message exchanges. This problem has also been identified by [Soukup 95].
- **Implementation Overhead**: The implementation overhead problem is due to the fact that the software engineer, when implementing a design pattern, often has to implement several methods with only trivial behaviour, e.g. forwarding a message to another object or method. This leads to significant overhead for the software engineer and decreased understandability of the resulting code.

Examples of these problems can be found in section 4. To address the problems, we propose a solution within the context of the layered object model, discussed in section 3. The layered object model is an extensible object model and its objects are encapsulated by so-called *layers*. In this paper, we illustrate how a design pattern can be implemented as a layer type. The software engineer can instantiate the layer type corresponding to a design pattern and associate a specification with the layer that specialises the behaviour of the layer type for the particular context. We illustrate our approach by

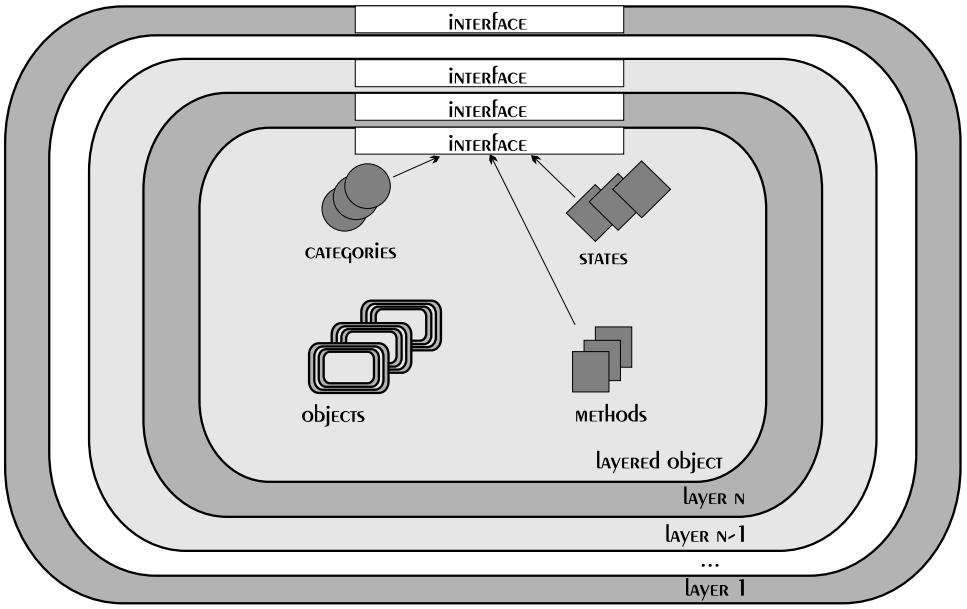

Figure 1. The layered object model

applying it to four design patterns from the design pattern catalogue by [Gamma *et al.* 94]. The reason for using this catalogue is that the semantics of these design patterns are relatively well defined and address relevant design problems.

3 Layered Object Model

The layered object model is an extended object model, i.e. it defines additional components next to the traditional object model components such as layers, states and categories. In figure 1, an example **LayOM** object is presented. The layers encapsulate the object, so that messages send to or by the object have to pass the layers. Each layer, when it intercepts a message, converts the message into a passive message object and evaluates the contents to determine the appropriate course of action. Layers can be used for various types of functionality. Layer classes have been defined for the representation of relations between objects, discussed in section 3.1 and 3.2. In this paper, we present the representation of design patterns through the use of layers.

A **LayOM** object contains, as any object model, instance variables and methods. The semantics of these components is very similar to the conventional object model. The only difference is that instance variables can have encapsulating layers adding functionality to the instance variable. In figure 2, an example **LayOM** class `TextEditWindow` is shown, containing one instance variable `loc`.

A *state* in **LayOM** is an abstraction of the internal state of the object. In **LayOM**, the internal state of an object is referred to as the *concrete state*. Based on the object's concrete state, the software engineer can define an externally visible abstraction of the concrete state, referred to as the abstract state of an object. The abstract object state is generally simpler in both the number of dimensions, as well as in the domains of the state dimensions. In figure 2, the abstract state `distFromOrigin` is shown. It abstracts the location of the mouse and the window origin into a distance measure. We refer to [Bosch 94b] for an extended description of abstract object state.

A category is an expression that defines a client category. A client category describes the discriminating characteristics of a subset of the possible clients that should be treated equally by the class. For example, the class in figure 2 defines a `Programmer` client category, restricting the use of the object to instances of class `Programmer` and its subclasses. The behavioural layer types use categories to determine whether the sender of a message is a member of a client category. If the sender is a member, the message is subject to the semantics of the specification of the behavioural layer type instance.

A layer, as mentioned, encapsulates the object and intercepts messages. It can perform all kinds of behaviour, either in response to a message or otherwise. Previously, layers have primarily been used to represent relations between objects. In **LayOM**, relations have been classified into structural relations, behavioural relations and application-domain relations.

```
class TextEditWindow
   layers
      rs : RestrictState(Programmer,accept all when distFromOrigin<100 otherwise reject);
      pin : PartialInherit(Window, *, (moveOrigin));
      po : PartOf(TextEditor);
   variables
      loc : Location;
   methods
      moveOrigin(newLoc : Location) returns Boolean
         begin
            loc := newLoc;
            self.updateWindow;
         end;
   states
      distFromOrigin returns Point
         begin return ((lox.x - self.origin.x).sqr + (lox.y - self.origin.y).sqr).sqrt; end;
   categories
      Programmer
         begin sender.subClassOf(Programmer); end;
end; // class TextEditWindow
```

Figure 2. Example LayOM class TextEditWindow

Structural relation types define the structure of a class and provide *reuse*. These relation types can be used to *extend* the functionality of a class. Inheritance and delegation are examples of structural relation types.

The second type of relations are the *behavioural relations* that are used to relate an object to its clients. The functionality of the class is used by client objects and the class can define a behavioural relation with each client (or client category). Behavioural relations *restrict* the behaviour of the class. For instance, some methods might be restricted to certain clients or in specific situations.

The third type of relations are *application domain relations*. Many domains have, next to reusable application domain classes, also application domain relation types that can be reused. For instance, the *controls* relation type is a very important type of relation in the domain of process control. In the following two sections, structural and behavioural relation layer types will be discussed. For information on application-domain relation types, we refer to [Bosch 94a, Bosch 95b].

The class in figure 2 has three layers. The `PartOf` layer defines an instance of `TextEditor` as a part of the class. The `PartialInherit` layer defines that class `TextEditWindow` inherits all methods from class `Window`, except method `moveOrigin`. The `RestrictState` layer restricts access to instances of class `Programmer` and its subclasses, but only if the distance between the mouse location and the window origin is less than 100 units.

Next to an extended object model, the layered object model is also an *extensible* object model, i.e. the object model can be extended by the software engineer with new components. **LayOM** can, for example, be extended with new layer types, but also with structural components, such as *events*. The notion of extensibility, which is a core feature of the object-oriented paradigm, has been applied to the object model itself. Object model extensibility may seem useful in theory, but in order to apply it in practice it requires extensibility of the translator or compiler associated with the language. In the case of **LayOM**, classes and applications are translated into C++. The generated classes can be combined with existing, hand-written C++ code to form an executable. The **LayOM** translator is based on *delegating compiler objects* [Bosch 95a], a concept that facilitates modularisation and reuse of compiler specifications and extensibility of the resulting compiler. The implementation of the **LayOM** translator is discussed in section 3.3.

3.1 Structural Relation Layers

Structural relation types, as described above, define the *structure* of an application. A class uses the structural relations to *extend* its behaviour and the class can be seen as the *client*, i.e. the class that obtains functionality provided by other classes. Generally, three types of structural relations are used in object-oriented systems development: *inheritance*, *delegation* and *part-of*. These types of relation all provide some form of *reuse*. The inherited, delegated or part object provides behaviour that is reused by, respectively, the inheriting, dele-

relation type	inheritance	delegation	part-of
default	Inherit	Delegate	PartOf
conditionality	ConditionalInherit	ConditionalDelegate	ConditionalPartOf
partiality	PartialInherit	PartialDelegate	PartialPartOf
conditional and partial	CondPartInherit	CondPartDelegate	CondPartPartOf

Table 1. Structural relation type identifiers

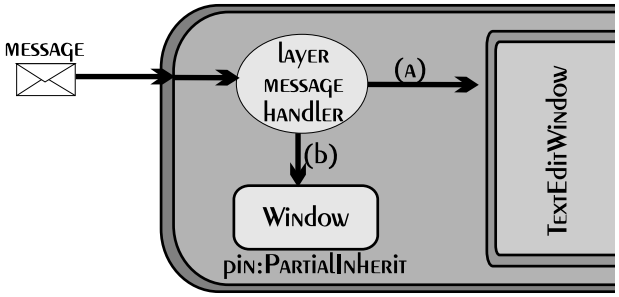

Figure 3. The `PartialInherit` layer

gating or whole object. Therefore, next to referring to these relation types as *structural*, we can also define them as *reuse* relations.

Orthogonal to the discussed relation types one can recognise two additional dimensions of describing the extended behaviour of an object, i.e. *conditionality*, and *partiality*. Conditionality indicates that the reusing object limits the reuse to only occur when it is in certain states. Partially indicates that the reusing object reuses only part of the reused object.

We consider it very important that a relation between two classes or objects, regardless of its complexity, is modelled as a single entity within the model. An alternative approach would be to define a collection of orthogonal constructs and to decompose a relation between objects into instances of each orthogonal construct, as is done in, e.g. [Bergmans 94]. This approach does, however, not represent a conceptual entity in analysis and design as an entity in the language model.

The different aspects of a structural relation, i.e. its type, partiality and conditionality, form a three-dimensional space which contains all possible combinations. In compliance with our modelling principle, we define a relation type for each combination. This results in twelve structural relation types. These structural relation types are shown in table 1.

Due to space constraints, it is not possible to describe the syntax and semantics of all structural relation types. Instead, one layer of type *Partial Inheritance* is described in detail. The semantics of the other layer types can be deduced by the reader.

In figure 2, class `TextEditWindow` contains a partial inheritance layer with the following configuration:

```
pin: PartialInherit(Window, *, (moveOrigin));
```

The semantics of the partial inheritance layer is, that a part of the interface of the inherited class is reused or excluded. The name of this layer is `pin` and its type is `PartialInherit`. The layer type accepts three arguments. The first argument is the name of the class that is inherited from; `Window` in this case. The second argument is a '*' or a list of interface elements and indicates the interface elements that are to be inherited. The '*' in this example indicates that all interface elements are inherited. The third argument can also contain a '*' or a list of interface elements. In this example, the list only consists of one element: `moveOrigin`. The semantics of layer `pin` is that class `TextEditWindow` inherits the complete interface of class `Window`, except for `moveOrigin`.

Note that, different from other object-oriented languages, the software engineer has to explicitly specify which interface elements of the superclass are to be inherited (or reused). A method is not automatically overridden by subclass method with the same name.

In figure 3, the implementation of this semantics is illustrated. The partial inheritance layer is the second layer of class `TextEditWindow`. There is the most outer layer, shown around layer `pin` and an

No factor	One factor	Two factors	Three factors
RestrictClient	RestrictState	RestrictStateAndConc	RestrictStateConcAndTime
	RestrictConc	RestrictStateAndTime	
	RestrictTime	RestrictConcAndTime	

Table 2. Behavioural relation type identifiers

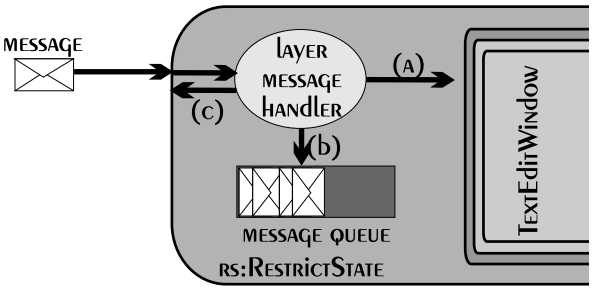

Figure 4. The `RestrictState` layer

inner layer, shown around `TextEditWindow`. All inheritance layers create an instance of the inherited superclass. In figure 3, an instance of class `Window` is shown. The layer contains a message handler, that, for each received message, determines whether the message is passed on inwards or outwards or that it is redirected to the instance of class `Window`. In the figure, an incoming message is shown. The message is reified and handed to the message handler. The message handler reads the selector field of the message and compares it with the (partially) inherited interface of class `Window`. If the selector is part of the set of interface elements, the message is redirected to the instance of class `Window` (situation (b) in figure 3). If the selector does not match with the interface of class `Window`, it is not redirected but forwarded to the next layer (situation (a) in figure 3).

For an extensive description of the semantics of the other structural relation types we refer to [Bosch 94a, Bosch 95b].

3.2 Behavioural Relation Layers

In the previous section, we discussed relations where the object containing the relation is the client, i.e. the object 'extends' itself with the structural relations. In this section we discuss relation types between the object and its clients. These relation types, generally, constrain the access of clients and the behaviour of the object in some way.

In order to keep the type and number of clients of the object open ended, we define client categories and for each message is determined whether the sender of the message is a member of a client category. The message is then subject to the behavioural relation(s) defined for that client category.

A relation between the object and a client category can be defined by a number of behavioural constraints. The types of constraints are *client-based access*, *state-based access*, *concurrency* and *real-time*. The layered object model takes a different approach to defining modelling constructs when compared to the conventional approaches. Rather than aiming at defining 'clean', orthogonal constructs, we try to define constructs that correspond to conceptual entities. Hence, we are convinced, supported by the object-oriented analysis and design methods, that the concept of a relation between objects is a conceptual entity and that the different aspects of this relation should be defined as part of this relation, rather than as several unrelated orthogonal constructs that, when combined, provide equivalent behaviour.

In table 2, the behavioural relation (or layer) types are shown. Each relation type can be viewed as a location in the space build by the four constraint dimensions defined above. However, a relation always requires a client category to be specified. This results in three dimensions which can be part of or not part of the relation. For each combination, a relation type is defined.

Again, due to space constraints, we are unable to discuss the semantics of all behavioural relation types. Instead, we will discuss the semantics of the `RestrictState` layer type in detail.

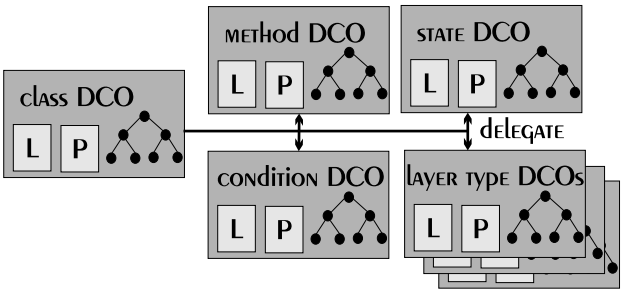

Figure 5. Overview of the LayOM compiler

The `RestrictState` layer type restricts the access for a particular client category when the object is in a certain states. Class `TextEditWindow` (see figure 2) has a `RestrictState` layer with the following configuration syntax:

```
rs : RestrictState(Programmer, accept all when
      distFromOrigin < 100 otherwise reject);
```

The semantics of this layer specification is the following. Clients that are classified as members of the `Programmer` client category can access all methods of the object provided that `distFromOrigin` is less than or equal to 100. If it is larger than 100, the message is rejected, i.e. an error message is returned to the client object that sent the message.

In figure 4, the functionality of the `RestrictState` layer type is presented graphically. The layer will intercept messages sent to the object. If the message is sent by an object that is not classified as a member of the client category that is indicated by the layer specification, the message will just be passed on to the next layer. Otherwise, the selector of the message is matched with the identifier list in the layer specification. If it matches, the state that is referred to in the specification is evaluated. Depending on the use of the keyword `unless` or `when`, the message will be passed on to the next layer (see (a) in figure 4). If the message is not passed on, the message is stored in the message queue if the `delay` keyword is used (see (b) in figure 4) or, in case of the use of keyword `reject`, the message is discarded and an error message is sent to the sender object (see (c) in figure 4).

3.3 Implementation

The implementation of the layered object model is based on the notion of *delegating compiler objects* (DCOs) [Bosch 95a]. A DCO is an object that compiles a part of the syntax. It consists of one or more lexers, one or more parsers and a parse graph. The nodes in the parse graph have the ability to generate code for itself. In case of the **LayOM** class compiler, it consists of a class DCO, method DCO, state DCO, category DCO and a DCO for each layer type. In figure 5, the structure of the **LayOM** compiler is shown.

Each DCO definition results in a class and a DCO object can instantiate another DCO and delegate control to it. The delegated DCO will perform its task and return control to the delegating DCO when it is finished. The **LayOM** compiler DCOs generate C++ output code. **LayOM** code is either a class or an application. A **LayOM** class is compiled into a C++ class and a **LayOM** application is compiled into a C++ main program. The generated C++ class can be incorporated in any C++ program.

An advantage of using the DCO approach is that it supports extensibility of the language very well. When the software engineer wants to add a new concept to the language, all that is required is a DCO defining the syntax and code generation information of that particular concept. The new DCO is added to the set of DCOs in the existing compiler and it can be used. Except for some minor modifications in the DCOs that should instantiate the new DCO, no changes are required.

The concept of delegating compiler objects is supported by a tool[1] that allows the software engineer to compose compilers by instantiating one or more DCOs. Each DCO can subsequently be assigned a lexer and a parser. Each compiler has a base DCO that is initially instantiated. The tool also provides editing facilities for specifying parsers, lexers and parse graph node classes. For a more detailed description of the delegating compiler object concept and the associated tool, we refer to [Bosch 95a].

1. The tool is currently in prototype stage and runs only on the Sun Solaris platform. Our aim is to make it available through our ftp site during 1996.

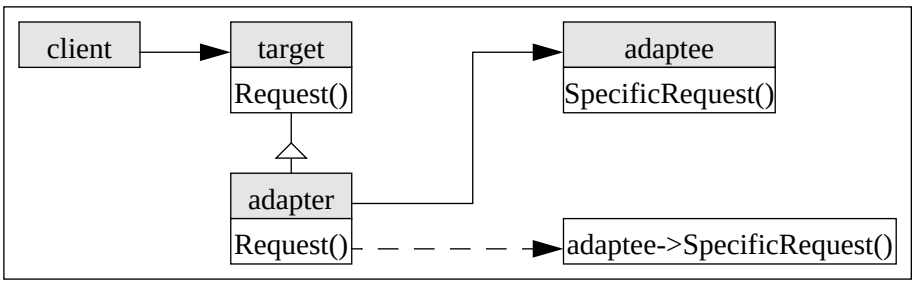

Figure 6. Structure of `Adapter` design pattern

4 Design Patterns as Language Constructs

Most design patterns have a well-defined semantics that can be used as the basis for defining language constructs that explicitly support the representation of the design pattern in the programming language. We consider design patterns as a very important technique used during design. However, we have identified problems associated with the implementation of patterns in the conventional object-oriented languages. These languages, like C++, provide insufficient support to implement design patterns in a traceable, efficient manner. Traceability is problematic because several design patterns are lost during implementation. In addition, some design patterns require the definition of a, potentially large, number of methods with very trivial behaviour, e.g. forwarding a message to a nested object. The latter is inefficient from the perspective of the software engineer.

In this section, we present a number of **LayOM** layer classes that represent design patterns. The design patterns have been selected from the collection defined by [Gamma *et al.* 94], in particular from the structural and behavioural design patterns. Since the collection contains seven structural design patterns and 11 behavioural design patterns, we are unable to describe layer types for all these design patterns for reasons of space. Instead we limit ourselves to describing two design patterns from the structural and behavioural category. The structural design patterns layer types that are discussed are `Adapter` and `Facade`. The layer types for behavioural design patterns that are discussed are `State` and `Mediator`.

4.1 Structural Design Patterns

As described in section 2, the structural design patterns are used to define parts of the structure of the system. These patterns are concerned with the composition of classes and objects. In [Gamma *et al.* 94], seven design patterns are described. However, as mentioned, traditional object-oriented languages have difficulty in implementing design patterns in a traceable and efficient manner. The solution proposed in this paper is to make use of the layered object model for the implementation of design models that make use of patterns. As **LayOM** is an extensible object model, the object model can be extended by, among others, adding new layer types. We have used this facility to define layer types that implement the functionality of design patterns. Due to reasons of space, we can only show two examples of structural design pattern layers, i.e. `Adapter` and `Facade`. These layer types are discussed in the following two sections.

4.1.1 Adapter

Intent. The `adapter` design pattern is used to convert the interface of a class into another interface that is expected by its clients. The adapter design pattern allows classes to cooperate that otherwise would be incompatible due to the differences in expected interfaces.

Problem. In a conventional object-oriented language, the adapter is implemented as an object that forwards the calls, after adaptation, to the adaptee, i.e. the adapted object. In figure 6, the structure of an adapter for object adaptation as presented in [Gamma *et al.* 94] is shown. Class adaptation is not shown in this figure.

Although the `adapter` indeed allows classes to work together that otherwise could not, there is, at least, one important disadvantage associated with the implementation of the pattern. The disadvantage is that for every element of the interface that needs to be adapted, the software engineer has to define a method that forwards the call to the actual method `SpecificRequest`. Moreover, in case of object adaptation, also those requests that otherwise would not have required adaptation have to be forwarded as well, due to the intermediate adapter object.

Solution. In the layered object model, the functionality of the Adapter design pattern does not require a separate object (or class) to be defined. Instead, a layer of type Adapter is defined that provides the functionality associated with the design pattern. The layer can be used as part of a class definition, in which case it represents class adaptation. It can also be defined for an object thus representing object adaptation. The syntax of layer type Adapter is the following:

```
<id> : Adapter(accept <mess-sel>+ as
      <new-mess-sel>, accept <mess-sel>+
      as <new-mess-sel>, ...);
```

The semantics of the layer type is that a message with a message selector <mess-sel> that is specified in the layer is passed on with a new selector <new-mess-sel>. The adapter layer type also allows more than one message selector to be translated to a new message selector. The layer will translate both messages send to the object encapsulated by the layer and messages send by the object.

The Adapter layer can be used for class adaptation by defining a new adapter class consisting only of two layers. Below, an example class adapter is shown:

```
class adapter
   layers
      adapt : Adapter(accept mess1 as
         newMessA, accept mess2, mess3
         as newMessB);
      inh : Inherit(Adaptee);
end; // class adapter
```

The example class adapter translates a mess1 message into a newMessA message and a mess2 or mess3 message into a newMessB message. The methods newMessA and newMessB are presumably implemented by class Adaptee and the Inherit layer will redirect these and other messages to the instance of class Adaptee that is contained within the layer.

Adaptation at the object level can be achieved by encapsulating the object with an additional layer upon instantiation. In this case, the adaptation will only be effective for this particular instance and not for the other instances of the same class. Below, an example of an adapted object declaration is shown.

```
...
// object declaration
adaptedAdaptee : Adaptee with layers
   adapt :  Adapter(accept mess1 as newMessA,
            accept mess2, mess3 as newMessB);
end;
...
```

The instance adaptedAdaptee will be extended with an additional layer of type Adapter that adapts its interface to match the interface expected by its clients. The Adapter layer will be the most outer layer of the object, intercepting all messages going into and out of the object.

Layer type Adapter can also be used in an inverted situation, i.e. a situation where a single client needs to access several server objects, but the client expects an interface different from the interface offered by the server objects. In this case, the client object (or its class) can be extended with an Adapter layer translating messages send by the client into messages understood by the server objects.

Evaluation. The Adapter layer type allows the software engineer to translate the Adapter design pattern directly into the implementation, without losing the pattern. There is a clear one-to-one relation between the design and the implementation. A second advantage is that the software engineer is not required to define a method for every method that needs to be adapted. The specification of the layer is all that is required. In addition, in case of object adaptation, the software engineer, in the traditional implementation approach, also needs to define a method for the methods of the adapted class that do not have to be adapted. When using the Adapter layer, this is avoided.

A disadvantage of the Adapter layer type definition presented in this paper is that the arguments of the message will be passed on as sent. In some situations, one would like to pass the arguments on in a different order or add or remove some arguments. However, this functionality will be added in the next version of the Adapter layer.

4.1.2 Facade

Intent. The Facade design pattern is used to provide a single, integrated interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that simplifies the use of the subsystem.

Problem. The structure of a subsystem incorporating the Facade design pattern often looks as in figure 7. The subsystem is defined as a class containing the classes that are part of the subsystem. The function of the subsystem class is basically twofold. The first is the coordination between the classes in the subsystem, whereas the second function is to provide an integrated interface to cli-

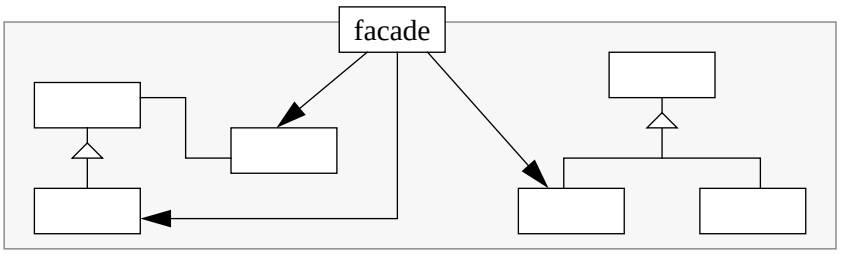

Figure 7. Structure of the Facade **design pattern**

ents of the subsystem. By defining the subsystem as a class, the first function can be dealt with in an acceptable manner. However, the second function, which often is the forwarding of messages to objects within the subsystem is problematic. The traditional approach is to define a method in the subsystem class that forwards a message sent by a client to the appropriate object inside the subsystem. The disadvantage of this approach is that for every message send by a client the subsystem class has to define a method. The number of methods might easily grow very large and defining these methods is much work for just forwarding a message. A second disadvantage is that the traceability of the design pattern in the implementation is lost.

Solution. As a solution to the identified problems associated with the traditional implementation approach one can, within the layered object model, make use of the Facade layer type. The Facade layer type provides the functionality of forwarding messages to objects that are part of the subsystem. A layer of type Facade is defined as follows:

```
<id> : Facade(forward <mess-sel>+ to <object>,
        forward <mess-sel>+ to <object>, ...);
```

The behaviour of a Facade layer is the following. It matches the selector of the message with the message selectors defined in <mess-sel>+, the message will be forwarded to the object <object>. The Facade layer can contain several forwarding elements, allowing the subsystem class to forward to several objects using a single Facade layer.

A subsystem class using a Facade layer can be defined as follows:

```
class facade
   layers
      face : Facade(forward mess1, mess2 to
             PartO1, forward mess3 to PartO2);
      PartO1 : PartOf(ClassOfO1);
      PartO2 : PartOf(ClassOfO2);
   ...
end; // class facade
```

As described in section 3, the layered object model models relations between objects as layers. These relations include the *part-of* relation, which is implemented as the PartOf layer type. The Facade layer forwards messages matching mess1 and mess2 to PartO1, which is an object contained in the subsystem. Messages matching mess3 are forwarded to object PartO2.

Evaluation. The use of the Facade layer type has two main advantages over the traditional implementation techniques. The first advantage is that the software engineer does not have to define a, possibly large, number of trivial methods that just pass on a message to one of the objects in the subsystem. The second advantage is that there is a direct correspondence between the design pattern that is used at the design level and the Facade layer defined in the implementation.

4.2 Behavioural Design Patterns

Behavioural design patterns focus on algorithms and cooperation and interaction between objects. In addition to the structure of a group of objects and classes, these design patterns are concerned with the communication between these objects and classes. In [Gamma *et al.* 94] the collection of behavioural design patterns consists of 11 patterns. For reasons of space only two of these patterns are discussed in this paper, i.e. State and Mediator.

4.2.1 State

Intent. The State pattern is used in situations where the behaviour of the object depends on the internal state of the object. Thus, when the object changes a relevant aspect of its state, it changes behaviour.

Problem. The implementation approach suggested by [Gamma *et al.* 94] is to define an abstract superclass defining the methods that change behaviour, depending on the state. This class is subclassed by as many concrete subclasses as there are different

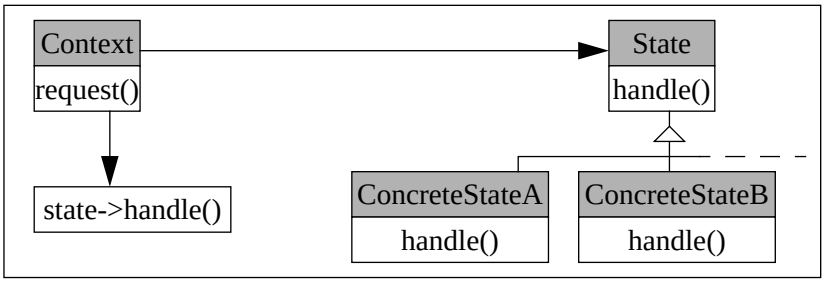

Figure 8. Structure of the `State` design pattern

relevant states and associated behaviours. These classes are used by the object that is supposed to show state-dependent behaviour, referred to as `Context`. The `Context` object always contains an instance of one of the concrete state subclasses. When the `Context` object changes state, it replaces the instance with an instance of another concrete subclass. In figure 8, the structure of the `State` design pattern is shown.

Although this implementation approach is very appropriate within the traditional object-oriented languages, the approach has, at least two problems associated with it. The first problem is that this approach assumes that there is a relatively small number of boolean states, that all require a unique implementation for each method in the set of state-dependent methods. However, in several cases the required dynamicity in the object behaviour is not as well structured as this. We refer to [Bosch 94b] for a more extended discussion of these problems. The second problem is that the `Context` object has to implement a trivial method for each state-dependent method implemented by the `State` class.

Solution. Similar to the solutions presented for the problems associated with the aforementioned design patterns, we define a layer of type `State` that allows the software engineer to specify, depending on the object state, the appropriate method or object for a message requesting state-dependent behaviour. As described in section 3, **LayOM** provides the notion of *abstract state*. Abstract state is an abstraction of the internal, concrete object state that presents the relevant state of an object at its interface. The `State` layer type makes use of the abstract object state. The syntax of the `State` layer type is defined below:

```
<id> : State(if <state-expr> forward
       <mess-sel>+ to [<mess-sel> | <object>]
       if <state-expr> forward <mess-sel>+
       to [<mess-sel> | <object>], ...);
```

The semantics of the `State` layer type is that a message received by the layer is evaluated for each element of the layer specification if the state expression `<state-expr>` evaluates to true and the selector of the message matches with one of the specified message selectors, the message is forwarded to either the method indicated by `<mess-sel>` or the object `<object>`.

The `State` layer type can be used in two scenarios. If the situation is such that a number of well-defined states exist that each have an associated behaviour, then the software engineer can define a concrete subclass for each state and declare it as a part of the `context` class. The `State` layer can then be used to direct messages to the appropriate state object. Below, an example class specification is shown. The `handle` message is state dependent. Depending on the value of `stateA` and `stateB`, the message is directed to part object `ConStA` (concrete state A) or `ConStB`.

```
class context
   layers
      st : State(if stateA forward handle
         to ConStA, if stateB forward handle
         to ConStB);
      ConStA : PartOf(ConcreteStateA);
      ConStB : PartOf(ConcreteStateB);
   ...
   states
      stateA returns Boolean
         begin ... end;
   ...
end; // class context
```

If the state dependent behaviour of the object is not as well structured as in the previous scenario, the software engineer can define different methods for a message selector and use the `State` layer to direct the message to the appropriate method. Below, an example of this approach is shown. Depending on the value of `stateA` and `stateB`, the handle message is directed to either method `handleImp1` or `handleImp2`.

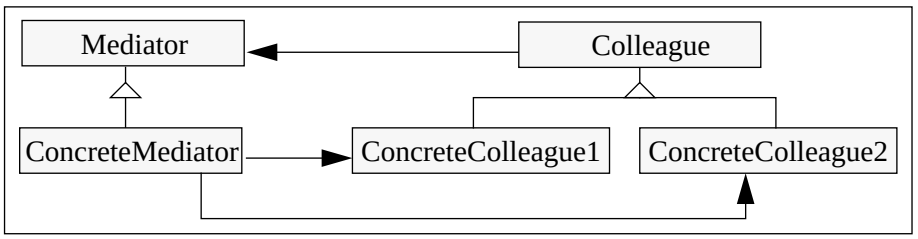

Figure 9. Structure of the `Mediator` design pattern

```
class context
   layers
      st : State(if stateA forward handle to
         handleImp1(), if stateB forward
         handle to handleImp2());
   ...
   methods
      handleImp1 returns ...
   ...
end; // class context
```

Evaluation. The `State` layer type has two advantages, when compared to traditional implementation techniques. First, the software engineer does not have to write a, possibly large, set of trivial methods forwarding the message to the appropriate method, depending on a state. Secondly, the relation between the design pattern and its implementation is kept, thereby improving traceability.

4.2.2 Mediator

Intent. The `Mediator` design pattern encapsulates the interaction of a set of objects. The `Mediator` decreases the coupling between the objects since they do not refer to each other directly. The `Mediator` can be replaced independently, allowing one to vary the interaction between the objects.

Problem. The implementation approach for the `Mediator` design pattern is to decouple the functionality of an object from its interaction with other objects. The interaction of a group of objects is encapsulated in a separate object, indicated as the Mediator. As shown in figure 9, a class `Mediator` has a reference to each object in the set that it mediates. Each object in the set (`Colleague`) has a reference to its mediator. Thus instead of sending a message to one of its colleagues directly, it sends the message to the `Mediator` that will forward the message to the appropriate object.

Although this approach separates the functionality of an object from its interaction with other objects and thus making it more reusable and flexible, one can identify some problems in the implementation when using a traditional object-oriented language. First, the `Mediator` consists of a set of methods that can be called by the colleague objects. The methods, in general, only consist of a call to the colleague object that is supposed to take care of the message. This is much work, but it also leads to the second problem: The structure of the interactions between the objects is lost in the implementation and can not be traced from design to implementation.

Solution. As a solution, the layer type `Mediator` is proposed. The `Mediator` layer is part of a mediator class and contains the specification of the interactions between the objects. The syntax of the `Mediator` layer type is shown below.

```
<id> : Mediator(forward <mess-sel>+ from
         <client> to <object>,
         forward <mess-sel>+ from <client>
         to <object>, ... );
```

The semantics of the `Mediator` layer is that a message in the `<mess-sel>+` set of message selectors sent by a client object `<client>` is forwarded to an object `<object>`. The `<client>` specification is based on the client categories of **LayOM**. The software engineer can specify a client category for each relevant subset of the interacting objects. Instead of a defined client category, the keyword `Any` can be used matching all possible clients.

Below an example class `ConcreteMediator` is shown. The `Mediator` layer forwards all messages with selector `messA` to object `ConcreteColleague1` and all messages `messB` sent by object `ConcreteColleague1` to object `ConcreteColleague2`.

```
class ConcreteMediator
   layers
      med : Mediator(forward messA from Any
               to ConcreteColleague1,
               forward messB from
               ConcreteColleague1 to
               ConcreteColleague2);
   ...
end; // class ConcreteMediator
```

Evaluation. The `Mediator` layer type solves the identified problems. First, the software engineer

does not have to write a series of methods only for forwarding purposes. The specification of a single layer is sufficient. Secondly, the layer specification shows very clearly the structure of the interaction between the objects. The traceability is thus preserved.

4.3 Composition

The four design pattern layer types discussed in section 4.1 and 4.2 and several other design pattern layer types not discussed in this paper can be composed in class descriptions. For instance, a layer of type `Adapter` and a layer of type `Mediator` can be composed in a class `ConcreteMediator`. The `Adapter` can change the selector of certain messages so that the `Mediator` can forward the message to colleague objects that otherwise could not have been in the same group due to incompatible interfaces.

This compositionality is very important in object-oriented framework development. In this domain, designers are increasingly making use of design patterns for describing the design. An important problem is that some classes in the framework might play a role in two or three design patterns. The implementation of these classes in a traditional object-oriented language is very complicated and the traceability of the individual design patterns is very poor. An implementation of these classes in **LayOM** does not suffer from these problems.

5 Related Work

[Soukup 95] is, to the best of our knowledge, the only author that specifically addresses the implementation aspects of patterns. Soukup discusses the implementation of design patterns and identified three problems. First, patterns often get lost during implementation. Second, composed patterns can lead to large clusters of mutually dependent classes. Third, although design pattern authors present the classes that make up the implementation of the design patterns, no library of concrete design pattern classes has been described. Soukup addresses this by implementing design patterns as C++ classes. When evaluating this approach, one can conclude that the traceability of design patterns in the implementation is improved. However, only a few pattern classes are presented and we are uncertain whether all patterns can be implemented as classes. We consider several patterns, e.g. the patterns addressed in this paper, unsuited for representation as classes. A second aspect is that the implementation efficiency problems are not addressed by Soukup's approach.

Most authors on design patterns generally propose an approach to implement their patterns. However, these implementations generally suffer from the identified traceability and implementation efficiency problems.

6 Conclusion

The problems associated with the implementation of design patterns in traditional object-oriented languages have been identified and discussed in this paper. These problems can be classified as the lack of traceability of design patterns in the implementation and the overhead for the software engineer when implementing design pattern.

A solution within the context of the layered object model has been introduced. The layered object model (**LayOM**) is an extended and extensible object model. It is *extended* because it contains *states*, *categories* and *layers* as additional components. **LayOM** is an *extensible* object model because it can be extended with new components and new layer types. In this paper, the identified problems were addressed by defining a layer type for a design pattern. Four design patterns, `Adapter`, `Facade`, `State` and `Mediator`, and the associated layer types have been presented as examples of the applicability of our approach. We have illustrated that these layers do not suffer from the aforementioned traceability and overhead problems.

We consider the layered object model a superior approach for practitioners, as **LayOM** is supported by an advanced implementation environment that translates **LayOM** classes and applications to C++ code. The generated C++ code can be used in combination with arbitrary C++ programs to generate applications. Thus, the software engineer using **LayOM** has the advantages of the extended expressiveness and avoids potential disadvantages as being limited to a particular environment language because the environment generates C++ code.

A second advantage of the layered object model and its supporting environment, which is not illustrated in this paper, is the extensibility of the object model. This allows the software engineer to define layer types for design patterns that were not availa-

ble. The environment is capable of extending the translator due to the fact that it is constructed using the concept of *delegating compiler objects*.

As part of future work, design patterns from different sources will be incorporated in the layered object model. Several authors have reported on application domain specific design patterns. The extensibility of the layered object model makes it very suitable to incorporate these design patterns.

Acknowledgement

Michael Mattsson and Petra Bosch provided valuable comments on earlier versions of this paper.

References

[Alexander *et al.* 77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel, 'A Pattern Language,' *Oxford University Press*, New York 1977.

[Bergmans 94] L. Bergmans, 'Composing Concurrent Objects,' *PhD Dissertation*, Department of Computer Science, University of Twente, 1994.

[Bosch 94a] J. Bosch, 'Relations as Object Model Components,' accepted for publication in *Journal of Programming Languages.*

[Bosch 94b] J. Bosch, 'Abstracting Object State,' *submitted.* Also Research Report 10/94, University of Karlskrona/Ronneby.

[Bosch 95a] J. Bosch, 'Delegating Compiler Objects - An Object-Oriented Approach to Crafting Compilers,' accepted for publication in *Proceedings Compiler Construction '96.*

[Bosch 95b] J. Bosch, 'Layered Object Model - Investigating Paradigm Extensibility,' *Ph.D. dissertation*, Department of Computer Science, Lund University, November 1995.

[Buschmann & Meunier 95] F. Buschmann, R. Meunier, 'A System of Patterns,' in *Pattern Languages of Program Design*, J.O. Coplien, D.C. Schmidt (eds.), pp. 325-343, Addison-Wesley, 1995.

[Coad 92] P. Coad, 'Object-Oriented Patterns,' *Communications of the ACM*, Vol. 35, No. 9, pp. 152-159, September 1992.

[Coplien & Schmidt 95] J.O. Coplien, D.C. Schmidt, *Patterns Languages of Program Design*, Addison-Wesley, 1995.

[Gamma *et al.* 94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Helm *et al.* 90] R. Helm, I. Holland, D. Ganghopadhyay, 'Contracts: Specifying Behavioral Compositions in Object-Oriented Systems,' *OOPSLA '90*, pp. 169-180, 1990.

[Mattsson 95] M.M. Mattsson, 'Object-Oriented Frameworks - a survey of methodological issues', *Licentiate thesis* (in preparation), Department of Computer Science, Lund University, 1995.

[Pree 95] W. Pree, *Design Patterns for Object-Oriented Software Engineering*, Addison-Wesley, 1995.

[Soukup 95] J. Soukup, 'Implementing Patterns,' in *Pattern Languages of Program Design*, J.O. Coplien, D.C. Schmidt (eds.), pp. 395-412, Addison-Wesley, 1995.