A Formal Approach to Architectural Design Patterns

P.S.C. Alencar[1], D.D. Cowan[1], C.J.P. Lucena[2]

[1] Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada
[2] Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brazil

Abstract. In this paper we introduce a formal approach to architectural design patterns based on an object-oriented model integrated with a process-oriented method for describing the patterns. The object-oriented model is based on the Abstract Data View (ADV) concept, which is a formal model for subjectivity in that it explicitly distinguishes between two kinds of objects, namely application objects and object views. The formalism allows the definition and application of design patterns by considering both the process program for the pattern tasks and the interconnected objects and views resulting from a particular pattern instantiation. The approach can be used to describe design patterns at many different architectural levels, and this is illustrated by presenting patterns for the master-slave, pipes-and-filters, layered systems, adapter, observer, and composite.

1 Introduction

Design patterns can be viewed as a means to achieve large-scale reuse by capturing successful software development design practice within a particular context [12, 7, 6]. Patterns should not be limited in what they can describe and can be used to encapsulate good design practices at both the specification and implementation levels. Thus, design patterns can be applied at many different levels of abstraction in the software development life-cycle, and can focus on reuse within architectural design as well as detailed design and implementation. In fact, a system of patterns for software development should include patterns covering various ranges of scale, beginning with patterns for defining the basic architectural structure of an application and ending with patterns describing how to implement a particular design mechanism in a concrete programming language.

Most published research [12, 6] in design patterns has been described in a structured but informal notation, and has focused on implementation-oriented patterns rather than architectural ones. For example, one publication [12] contains descriptions of patterns using text and diagrams, and has grouped patterns into three major categories. These descriptions can be viewed as an informal recipe or process for producing instantiations of specific patterns in languages such as Smalltalk or C++. Even when architectural issues are considered [7], the software architectural design is expressed only through informal patterns.

An architectural pattern is based on selected types of components and connectors, together with a control structure that governs their execution.

In this paper we describe a formal approach to design patterns which encompasses patterns at different levels of granularity ranging from architectural to implementation descriptions. There are two aspects to design patterns that are considered in this presentation: the process of producing specific instantiations of a design pattern, and the use of formally defined components or objects to substitute in these instantiations.

If the process is defined through a process language with formal syntax and semantics, then any ambiguities in the process of design pattern instantiation should be eliminated. Reducing or even eliminating ambiguity should make it easier to derive code consistently and perhaps even lead to some automation of the code production for the particular instantiation of a design pattern [1]. Substituting formally defined components into an instantiation could permit a formal reasoning process about the resulting system. We currently have established two different frameworks for reasoning about designs[3, 5] of this type.

Recent investigations[5] have shown how both a formal model and a prototype can be derived from a single component-based specification, thus providing a strong link between formalism and implementation.

The formally defined components are based on the Abstract Data View (ADV) approach[8, 9, 10] which uses a formal model [3, 4] to achieve separation by dividing designs into two types of components: objects and object views, and by strictly following a set of design rules. Specific instantiations of views as represented by Abstract Data Views (ADVs) and objects called Abstract Data Objects (ADOs) are substituted into the design pattern realization while maintaining a clear separation between view and object. Currently the ADV and ADO components are specified using temporal logic and the interconnection between components is described in terms of category theory.

Each design pattern has an associated process program that describes how to substitute these components to create a specific instantiation. In fact, this framework can be seen as a formal approach for a system of design patterns.

2 Abstract Data Views: the Model and Its Schema

In this section we describe the Abstract Data View (ADV) model and associated schema. This model allows us to create design patterns at various levels of granularity ranging from program design to software architectures, while maintaining a clear separation of concerns among the components.

A model of the ADV/ADO concept showing how these two types of objects interact is presented in Figure 1. An ADO is an object in the object-oriented sense, but has no direct contact with the "outside" world. As an object, an ADO has state and a public interface that can be used to query or change this state. An ADO is abstract since we are only interested in the public interface. An ADV is an ADO augmented to support the development of general "views" of ADOs, where a view could include a user or network interface or an adaptation of the

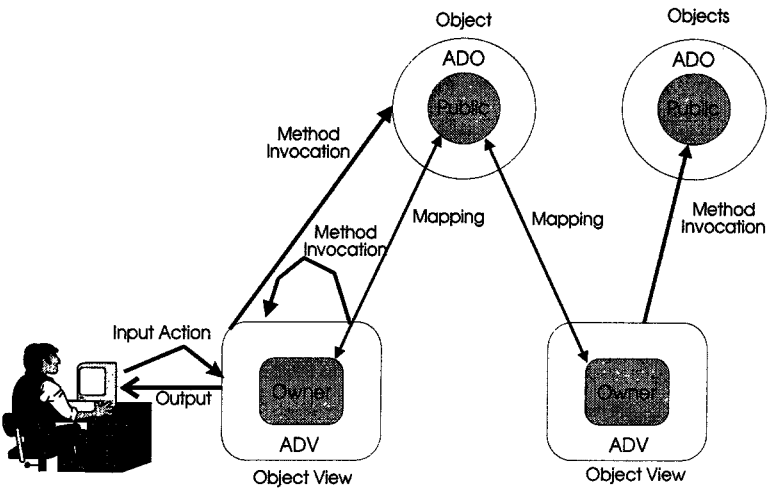

Fig. 1. An ADV/ADO interaction model

public interface of an ADO to change the way the ADO is "viewed" by other ADOs. A view may change the state of an associated ADO either through an input action (event) as found in a user interface, or through the action of another ADO.

Since an ADV is conceived to be separate from an ADO and yet specify a view of an ADO, the ADV should incorporate a formal association with its corresponding ADO. The formal association consists of: a naming convention, a method of ensuring that the ADV view and the ADO state are consistent, and a method of changing the ADO state from its associated ADV. Because an ADV is an object with properties which assist the designer in maintaining a clear separation of concerns we have chosen to give this "special" object a unique identity.

In order to maintain a separation of concerns, an ADV knows the name of any ADO to which it is connected, but an ADO does not know the name of its attached ADVs. The name of the ADO connected to an ADV is represented in the ADV by a placeholder variable called "owner" which is shown in Figure 1.

If the state of an ADO is changed then any part of the state that may be viewed by a connected ADV through the ADO's public interface must be consistent with that change. A morphism or mapping is defined between the ADV and ADO that expresses this invariant, and of course, uses the naming convention previously described. In addition, an ADV may query or change the state of a connected ADO through its normal public interface.

Figure 1 illustrates many of these concepts. The user depicted in the Figure causes an input action that is received by the ADV acting as a user interface. This action can cause a method invocation in which the ADV changes or queries its own state or the state of its associated ADO. If the state of the ADO changes through some other action, then the mapping ensures that the ADO and the user

interface ADV are consistent. The other ADV in Figure 1 acts as an interface between two ADOs, and changes in the ADO are similarly reflected in this ADV through the mapping. Thus, the mapping can force a method invocation to occur which changes another ADO.

ADV ADV_Name [*For* | *On*] ADO_Name
- *Declarations*
  - *Data Signatures* - sorts and functions
  - *Attributes* - observable properties of objects
  - *Causal Actions* - list of possible input actions
  - *Effectual Actions* - list of possible effectual actions
  - *Nested ADVs* - allows composition, inheritance, sets, ...
- *Static Properties*
  - *Constraints* - constraints in the attributes values
  - *Derived Attributes* - non-primitive attribute descriptions
- *Dynamic Properties*
  - *Interconnection* - description of the communication process among objects
  - *Valuation* - the effect of events on attributes
  - *Behavior* - behavioral properties of the ADV

End ADV_Name

Fig. 2. A descriptive schema for an ADV.

In summary we observe that there are two types of ADVs: an ADV which acts as an interface between two different media, and an ADV which acts as an interface between two ADOs operating in the same medium. Although there are two types of ADVs, they are natural extensions of each other.

The ADV/ADO model was originally conceived to address the same concerns as the MVC [19] paradigm. However, the ADV/ADO model is more general in its approach in that it explicitly models both interfaces to the external world and interfaces between objects. Emphasis on the interface as a special type of object encourages the designer to address separation of concerns explicitly in a design. In addition, the ADV/ADO model is formally defined and can support both reasoning and implementation with the same model.

The separation between views and objects makes it possible to use several ADVs to create different views for a single collection of ADOs. In this case, both ADOs and their associated views must be consistent. For example, a clock ADO could have a digital view, an analog view, or both. We call consistency among the different ADVs *horizontal consistency*, while consistency between the visual object (ADV) and its ADO is called *vertical consistency*. These consistency properties must be guaranteed by the specification of ADVs, ADOs, and their environment.

ADVs and ADOs have distinct roles in a software system. As a consequence,

they are described by different schemas. These schemas are not the actual objects of the system, but rather provide descriptions of their static and dynamic properties and declarations of entities that are used within the scope of the object. Such schemas are presented in detail in [4].

The specification syntax of the whole schema which is based on ones described in [13] is presented essentially through a temporal logic formalism [4, 22]. Every ADV or ADO structure is subdivided into three sections. A declaration part contains a description of all of the elements that compose the object including sorts, functions, attributes, and actions. The static properties part defines all the properties which do not affect the state of the object. Dynamic properties establish how the states and attribute values of an object are modified during its lifetime.

Figure 2 shows the structure of the schemas to be used in the specification of ADVs. Causal actions correspond to input events while effectual actions correspond to method invocations. The header of the schema has the name of the ADV and the name of its associated ADO. ADO schemas, which are not illustrated here, have a similar structure to ADVs, except that ADOs do not support causal actions, and they do not contain references to any ADVs.

These formal ADV/ADO schemas are based on temporal logic and some tools from category theory (institutions). This approach is strongly based on Maibaum and Fiadeiro's combination of temporal logic and category theory [11] that was initially developed for the purpose of formalizing modularization techniques for reactive systems. We capture the ADV semantics in logic by using temporal logic to describe the ADV and ADO components and their properties, morphisms (or mappings) to describe the relationship between these components (through a concept related to interpretation between theories), and tools from category theory (institutions) to specify systems of interconnected ADVs and ADOs (the structuring mechanisms). For more details on the issues treated in this section see [3].

The formal category theory tool used here, called the theory of institutions (and its associated tools), was introduced by Goguen and Burstall [15] and allows the theories of a logic to be shown to constitute a category whose morphisms correspond to property preserving translations between their languages. These translations or mappings are also known as interpretations between theories, and have been used to model relationships between abstract and concrete specifications [25], or to model mappings between different notions of software architecture [23].

We adopt a temporal logic with a (global) discrete linear time structure similar to those used in [21], since this allows easier assessment of the support for modular specification that is described. We also use the fact that temporal logics may be defined that satisfy, to some extent, institution [14], and hence, that temporal theories may be used as modularization units for concurrent system specification.

The formal specifications of ADVs and ADOs are provided as theory presentations [3] and a categorical account of the ADV/ADO specifications is provided

through a category of temporal theories. A morphism or mapping of theory presentations is a signature morphism that defines a theorem preserving translation between the two theory presentations and a locality property. Morphisms capture the relationship that exists between two ADV/ADO theory presentations. Thus, morphisms can be used to express a system as a diagram showing an interconnection of its parts. Formally, this diagram is a directed multigraph in which the nodes are labeled by ADV/ADO specifications, and the edges by the specification morphisms.

3 A Formal Description of Design Patterns

The ADV model supports reuse since it divides an application into a set of specialized objects (separation of concerns) each of which may be used in other designs. However, we would like to "glue" these objects into reusable systems, that is, systems which are easily maintained over time. Design patterns as proposed in [12] support this form of reuse. Each design pattern is a meta-description of a solution for a problem that occurs frequently in software design. The application of the meta-description results in several objects connected together to form a specific instantiation of such a design solution.

Operator **Pattern Name**
  Objective - description of the intent of the pattern
  Parameters - external elements used in the pattern definition
  Subtasks - description of pattern in primitive constructors
  Consequences - how the pattern supports its objective
  Product Text - language-dependent specification of pattern
End Operator

Fig. 3. Development constructor structure for a design pattern.

The acceptance of reusable descriptions, such as design patterns, is highly dependent on easily comprehensible definitions and unambiguous specifications. We address both issues in a single formalism for design pattern application.

In order to formalize the application of design patterns we introduce development constructors which are based on schemas that indicate how to apply a pattern. We define design pattern constructors to consist of a language-independent part and a product text specification, where a specific language is adopted; this approach is similar to that described in [20].

The language-independent part of the structure should clearly define the characteristics of a design pattern. According to [12], a pattern is composed of four essential elements: *pattern name*, *problem statement*, *solution*, and *consequences*.

Appropriate pattern names are usually important factors to assist developers in the specification of a system. In the case of reusable modules, the vocabulary of patterns could be one way of guiding the user to choosing suitable modules for the solution of particular problems.

A problem statement is a description of the circumstances in which to apply a design pattern, and clarifies the pattern objectives. In the development constructor structure shown in Figure 3, such a statement is described by an *Objective* section.

Applying a pattern in the context of a specific problem requires a process description, and so we specify this process in terms of primitive development constructors and parameters. The primitive constructors applied to pattern construction are organized in a section of the schema called *Subtasks*, while input parameters used in this process are declared in the *Parameters* section.

The consequences of an application of a pattern provide a description of the results of using such structure in a software system. The roles of the components within the pattern objectives are also illustrated. This section may be helpful in evaluating the suitability of a pattern in a specific context. These ramifications are specified in the *Consequences* section of a pattern schema.

The language-dependent part of the pattern constructors describes the result of the application of a pattern as a specific formal representation. Since design patterns are solution abstractions, a template of the pattern should be a helpful instrument in guiding the user to a particular specification. Such templates are illustrated in the pattern development constructors using the formalism of ADV/ADO schematic representations described in Section 2.

4 Formal Design Patterns

In this section we provide a formal description of some design patterns at both the program design and architectural levels. We have chosen the adapter, observer, and composite design patterns to illustrate formal descriptions of patterns used to support program designs and the master-slave, pipes and filters, and layers as typical patterns that are used to describe system architectures. In this way we illustrate how the same basic formalism may be used to specify software at two different levels of abstraction. We present the implementation patterns first, since they have appeared elsewhere [12], and the interested reader may wish to compare the two different styles of presentation.

4.1 Program Design Patterns

We have chosen the adapter, observer and composite patterns from [12], to illustrate how the pattern constructor described in Figure 3 is applied. There is a substantial difference between the pattern specifications presented in [12], and the specifications introduced in this paper. The patterns in [12] are based on OMT diagrams, informal descriptions in English and C++ templates, and are much closer to the implementation level than the version of the pattern

Operator Design Pattern Adapter
Objective Modify the interface of an object
Parameters Objects: ADAPTEE, TARGET;
Subtasks 1 - Specify Adaptation Object: ADAPTEE, TARGET → ADAPTER
1.1 - Create Object: → ADAPTER
1.2 - Compose Objects: ADAPTEE, ADAPTER → ADAPTER
1.3 - Inherit Objects: TARGET, ADAPTER → ADAPTER
1.4 - Specify Links: ADAPTEE, ADAPTER → ADAPTER
Consequences ADAPTER will contain most of ADAPTEE functionality available through the TARGET object interface
Product Text *ADV/ADO* ADAPTER
Declarations
...
Nested ADVs/ADOs
Compose ADAPTEE;
Inherit TARGET;
...
Dynamic Properties
...
Interconnection
With ADV/ADO ADAPTEE
TargetActions ⟼ *AdapteeActions*;
End ADAPTER
End Operator

Fig. 4. Specification of adapter pattern constructor.

descriptions we present in this paper. Our version of the patterns is based on the pattern constructors and the specification formalism associated with the ADV design approach. Although we only present three patterns from [12] in this paper, our notation can handle a much broader range of patterns at the program design level as illustrated in [1].

In the following design pattern examples, the pattern specifications provide explanations and directions to instantiate a design through the use of well-defined development operators and incomplete object schemas. The development operators are sequentially numbered in a section called *Subtasks*, while the object schemas are defined in the language-dependent section called *Product Text*, which provides reusable patterns for the program design. Other sections complete the design pattern specifications by providing additional information.

The first pattern we describe is the *adapter*, and its corresponding specification schema is shown in Figure 4. The objective of the *adapter* is to modify

Operator Design Pattern Observer
Objective Define dependency between objects
Parameters Objects: SUBJECT;
Subtasks 1 - Specify Observers to Objects: SUBJECT → OBSERVERs
1.1 - Create Object: → OBSERVERs
1.2 - Compose Objects: SUBJECT, OBSERVERs → OBSERVERs
1.3 - Specify Links: OBSERVERs → OBSERVERs
Consequences OBSERVERs monitor changes in the state of SUBJECT to update its own state
Product Text *ADV* OBSERVERs for ADO SUBJECT
Declarations
...
Attributes
ObserverAttributes;
Actions
ObserverQueryActions;
ObserverChangeActions;
...
Dynamic Properties
Interconnection
With ADO SUBJECT
SubjectAttributes ⟼ *ObserversAttributes*;
SubjectQueryActions ⟼ *ObserversQueryActions*;
SubjectChangeActions ⟼ *ObserversChangeActions*;
...
End OBSERVERs
End Operator

Fig. 5. Specification of observer pattern constructor.

the interface of a given object to conform to the needs of a client object. It is generally used to produce compatibility between two objects. The adapter might be seen as an object which is a wrapper for another object, and it can be used for adaptations of interface objects (ADVs) as well as application objects (ADOs). An adapter object could also be regarded as a view (ADV) for an application object (ADO).

The subtasks which specify how to instantiate a pattern are given in a task or function notation of the form: "$f : x \rightarrow y$" where f is a function, x is a list of parameters for the function f, and y is the result of applying the function f. In the *adapter* pattern the function "Create Object" returns a copy of the ADAPTER Product Text while the function "Inherit Objects" takes the two arguments TARGET and ADAPTER and returns the modified Product Text for ADAPTER. In the context of C++ the ADAPTER text would have the

Operator Design Pattern Composite

Objective Compose objects into tree structures to represent part-whole hierarchies

Parameters Objects: COMPONENT;

Subtasks

1 - Create a Tree Structure.
- 1.1 - Instantiate Concrete Object: COMPONENT → COMPOSITE
- 1.2 - Instantiate Concrete Object: COMPONENT → LEAFs
- 1.3 - Compose Objects: LEAFs, COMPOSITE → COMPOSITE
- 1.4 - If Subtree is needed:
  - 1.4.1 - Recursively Create SubTrees (Step 1)
  - 1.4.2 - Compose Objects: SubCOMPOSITE, COMPOSITE → COMPOSITE

Consequences A tree structure composed of LEAF objects and COMPOSITE objects is created, where the last ones represent the internal nodes of the tree

Product Text

```
ADV/ADO COMPOSITE
    Declarations
        ...
        Attributes
            ComponentType: ADO COMPONENT;
        Nested ADVs/ADOs
            Set CompSet of ComponentType;
            Inherit Component;
    ...
End COMPOSITE

ADV/ADO LEAF
    Declarations
        ...
        Nested ADVs/ADOs
            Inherit Component;
```

Fig. 6. Specification of composite pattern constructor.

words "inherit from TARGET" inserted in the appropriate location.

The *observer* design pattern, illustrated in Figure 5, is an example of a connecting pattern. The main objective of this pattern is to define a one-to-many dependency between objects, so that the dependent objects can monitor changes in one object. In the specification introduced here, we assume that *observer* objects are views (ADVs) of application objects, which we call *subject* in the pattern constructor.

The current specification approach differs from the design proposed in [12], in that the link between a view and its application object is represented by the ADV *mapping* design mechanism, which was explained in Section 2. This approach does not describe the implementation of the link, but indicates a mapping or morphism between elements of the objects involved. In contrast, the design described in [12] proposes a design technique that is closer to the implementation than the proposed *mapping.*

Figure 6 describes the specification of the *composite* design pattern. This pattern defines a hierarchical structure of objects sharing part-whole relationships. In such a relationship between objects, a composite object performs the "whole" role, while *leaf* and other *composite* objects represent the "parts."

The elements composing the resulting tree structure have uniform interfaces, since all of them inherit the tree interface from the abstract class called *component.* Additionally, these elements might be defined by ADVs or ADOs, since the *composite* design pattern might be used to structure both user interface objects and application objects.

The product text in Figure 6 has an attribute that contains the name of an ADO. Thus, it is possible to specify dynamic object structures where the names change over time.

4.2 Architectural Design Patterns

There are situations in a software system in which it is necessary to replicate a particular service in order to achieve fault tolerance, safety, and correctness. The redundant services are needed because in these situations it is not desirable to have a single supplier for a critical service. In these cases a design pattern called Master-Slave can be used. This pattern consists of a master component and a set of at least two slave components. The slaves are independent components that each provide the same service, but may use different solution strategies for providing that service. The master is the only component to which the clients of the service communicate, and the service is only accessible through the master's public interface. The master is responsible for invoking the slaves and for producing the final result which is computed from the results returned by the slaves. Thus, the master does not provide the service directly, but delegates the same task (the particular service) to several independent suppliers, and then returns the selected result to its clients.

From this discussion we can see that the Master-Slave design pattern has three kinds of participants: the client, the master and the slaves. The client requires a certain service in order to complete its own task. The master organizes the invocation of replicated services and decides which of the results returned by its slaves is to be passed to its clients, while the slaves are responsible for implementing the critical service. The Master-Slave Design Pattern is specified in Figure 7 with the *Product Text* for the Client and Master shown in Figure 8 and the *Product Text* for the Slave shown in Figure 9.

Pipes-and-filters is another important architectural pattern which is often used when software systems are composed of several independent complete sub-

Operator Design Pattern Master-Slave

Objective	Handle the computation of replicated services
Parameters	Objects: CLIENT;
Subtasks	1 - Specify Master Object: CLIENT $\rightarrow$ MASTER 1.1 - Create Object: $\rightarrow$ MASTER 1.2 - Compose Objects: CLIENT, MASTER $\rightarrow$ MASTER 1.3 - Specify Links: MASTER $\rightarrow$ MASTER 2 - Specify Slave Objects: MASTER $\rightarrow$ SLAVE$_i$ $i = 1, \ldots, N$ 2.1 - Create Objects: $\rightarrow$ SLAVE$_i$ $i = 1, \ldots, N$ 2.2 - Compose Objects: MASTER, SLAVE$_i$ $\rightarrow$ MASTER $i = 1, \ldots, N$ 2.3 - Specify Links: MASTER $\rightarrow$ MASTER
Consequences	The replicated service provided by the SLAVE$_i$ (the slaves) is offered to CLIENT through the MASTER after selection
Product Text	*ADV/ADO* CLIENT ... *End* CLIENT *ADV/ADO* MASTER ... *End* MASTER *ADV/ADO* SLAVE$_i$ $i = 1, \ldots, N$ ... *End* SLAVE$_i$

Fig. 7. Specification of the Master-Slave pattern constructor.

tasks. The subtasks are performed in a sequential or parallel order, and communicate with each other only by exchanging streams of data. The objective is to obtain highly reusable, interchangeable, and maintainable applications. This pattern consists of filter and pipe components. The filter components transform data and are responsible for a particular, independent, complete subtask of an application. The filter's interface consists of an input and output stream only. A filter reads streams of data from its inputs, processes this data, and produces streams of data as its outputs. The filter preserves any ordering relationship between the input and output data. The transformation of input is usually done locally and incrementally, and thus the output may begin before the input is completely read. A filter should not know the identity of the filters preceding or following it in the computation sequence, and so filters may not share state in order to preserve this independence. The pipe components (the pipes) are the connections between the filters that transmit data. There are normally two different ways to realize pipes: they may be links between filters (such as message calls) or they may be separate components (such as repositories or sensors). The responsibility of a pipe is to transmit data between filters, possibly converting the data format between the inout and output filter.

Product Text ADV/ADO CLIENT
Declarations
...
Nested ADVs/ADOs
...
Attributes
ClientAttributes;
Actions
ComputeTask;
...
End CLIENT
ADV/ADO MASTER
Declarations
...
Nested ADVs/ADOs
...
Attributes
MasterAttributes;
Actions
OutCom; *Service*; $\overline{Service}$;
SelectResult($Res_1, \ldots, Res_N$);
...
Dynamic Properties
...
Interconnection
With ADV/ADO CLIENT
$Service \longmapsto ComputeTask$;
With ADV/ADO SLAVE$_i$ $i = 1, \ldots, N$
$\overline{Service} \longmapsto ServiceSlave_1$;
...
$\overline{Service} \longmapsto ServiceSlave_N$;
Behavior
$s_1 \wedge Service \rightarrow s_2$
$s_2 \wedge \overline{Service} \rightarrow s_3$
...
End MASTER

Fig. 8. The Master and Client product text for the Master-Slave pattern constructor.

The Pipes-and-Filters Architectural Pattern is specified in Figure 10 where the product text is shown in Figure 11. The Pipe-and-Filter architectural pattern can be viewed as an arrangement of producer-consumer structures. Each filter can be seen as a consumer of input, either from a user or from the output of some other filter, and as a producer of output, either for the user or as input to some other filter.

Another important architectural design pattern is the layered architecture.

```
Product Text ...
        ADV/ADO SLAVE_i i = 1,...,N
            Declarations
                ...
                Attributes
                    Slave_iAttributes;
                Actions
                    ServiceSlave_i;
                ...
                Dynamic Properties
                    ...
        End SLAVE_i
End Operator
```

Fig. 9. The Slave product text for the Master-Slave pattern constructor.

```
Operator Design Pattern Pipes-and-Filters
    Objective       Structure applications that can be divided into several
                    completely independent subtasks performed in a strongly
                    determined sequential or parallel order
    Subtasks        1 - Specify Filter Objects: → FILTER_i i = 1,2
                        1.1 - Create Objects: → FILTER_i i = 1,2
                    2 - Specify Pipe (Object): → PIPE
                        2.1 - Create Object: → PIPE
                        2.2 - Compose Objects: FILTER_1,FILTER_2,PIPE → PIPE
                        2.4 - Specify Links: PIPE → PIPE
    Consequences    The subtasks are connected through the PIPEs
    Product Text    ADV/ADO FILTER_1
                        ...
                    End FILTER_1
                    ADV/ADO FILTER_2
                        ...
                    End FILTER_2
                    ADV/ADO PIPE
                        ...
                    End PIPE
```

Fig. 10. Specification of the Pipes-and-Filters pattern constructor.

This pattern can be used to structure applications that can be organized hierarchically, and its use implies that each layer provides services to the layer above and serves as a client to the layer below. The specification in Figure 12 considers the particular case in which we have one layer object built on the top of the object $LAYER_i$. The top layer object, referred to as $LAYER_{i+1}$, is connected to $LAYER_i$ by the object view $CONNECTOR_i$.

Product Text ADV/ADO FILTER$_1$
Declarations
Attributes
$FILTER_1$ *Attributes*;
Actions
$SubTask_1$; $ActCom_{out}$; $ActCom_{in}$; $\overline{ActCom_{in}}$;
...
Dynamic Properties
...
Interconnection
With ADV/ADO OUTCOMPONENT
$\overline{ActCom_{in}} \longmapsto OutCompAction$;
Behavior
$s_1 \wedge \overline{ActCom_{in}} \rightarrow s_2$
$s_2 \wedge ActCom_{in} \rightarrow s_3$
...
End FILTER$_1$
ADV/ADO FILTER$_2$
Declarations
...
Nested ADVs/ADOs
...
Attributes
$FILTER_2$ *Attributes*;
Actions
$SubTask_2$;
...
End FILTER$_2$
ADV/ADO PIPE
Declarations
...
Nested ADVs/ADOs
...
Attributes
PIPEAttributes;
Actions
AddResult; *RetrieveResult*;
...
End PIPE

Fig. 11. The Filter and Pipe product text for the Pipes-and-Filters pattern constructor.

Operator Design Pattern Layered-System

Objective Used to structure applications organized hierarchically

Subtasks
1 - Specify First Layer Object: $\rightarrow$ LAYER$_i$
1.1 - Create Object: $\rightarrow$ LAYER$_i$
2 - Specify Second Layer Object: $\rightarrow$ LAYER$_{i+1}$
2.1 - Create Object: $\rightarrow$ LAYER$_{i+1}$
2.2 - Create Connector Object: $\rightarrow$ CONNECTOR$_i$
2.3 - Specify Links: LAYER$_i$,LAYER$_{i+1}$,CONNECTOR$_i$ $\rightarrow$ CONNECTOR$_i$

Consequences Each layer provides services to the layer above and serves as a client to the layer below

Product Text

ADV/ADO LAYER$_i$
Declarations
Attributes
LAYER$_i$Attributes
Actions
LAYER$_i$Action
...
End LAYER$_i$
ADV/ADO LAYER$_{i+1}$
Declarations
Attributes
LAYER$_{i+1}$Attributes
Actions
LAYER$_{i+1}$Action
...
End LAYER$_{i+1}$
ADV/ADV CONNECTOR$_i$
Declarations
Attributes
CONNECTOR$_i$Attributes
Actions
CONNECTOR$_i$Action
Dynamic Properties
...
Interconnection
With ADO LAYER$_i$
CONNECTOR$_i$*Action* $\longmapsto$ LAYER$_i$Action
With ADO LAYER$_{i+1}$
CONNECTOR$_i$*Action* $\longmapsto$ LAYER$_{i+1}$Action
End CONNECTOR$_i$

Fig. 12. Specification of the Layered System pattern constructor.

This pattern indicates how the ADV model can be viewed as a formalism [3] supporting the integration of subject-oriented programming [16, 17] and design patterns [12, 7, 6], establishing the relationship of object views and objects with

subjects and subject activations. The object LAYER_1 can be, for example, an ADO House and the object LAYER_2 and its associated CONNECTOR_1 can be thought of as the subject activation related to a buyer view of the House. In this case, an extrinsic value of the House (an attribute of the ADO Buyer) can be computed by an action or method defined in the CONNECTOR_1 using the intrinsic attributes of the ADO House such as the size and number of rooms. Besides allowing the specification of layered systems, we can also formalize in principle other applications of the subject model such as, for example, the specification of multiple views of data, code debugging from a stable or a temporary perspective, and the management of the versions of a system by introducing subjects as a mechanism to associate different state with a single object.

5 Conclusions

In this paper we describe a formal approach for capturing software designs expressed by design patterns, and have demonstrated our techniques by describing design patterns at both the architectural and program design levels of software description. Our process programming approach to design patterns allows us to define primitive design pattern tasks or constructors that can then be used to produce specific instantiations of a design. We believe that this approach clarifies both the application and structure of the design patterns. In addition, the patterns have been specified so as to incorporate the concept of objects (ADOs) and object views (ADVs).

Using this formal approach including objects and object views directs us toward several important results. By using formally defined components we are able to reason about the design and prove properties as shown in [3, 5]. Of course systems often do not yield to formal approaches because of their size and complexity. However, the formal approach could still produce useful results in that the models generated could be used to aid in the testing process [5] by serving as a basis for test case generation [18], or by providing a means for measuring test coverage.

We are also investigating code generation from design patterns incorporating ADO and ADV schemas. Experiments based on the process program description have shown that design patterns can yield corresponding C++ schemas which can be completed by the designer through an interactive dialogue. In fact, we are currently experimenting with C++ code generation by constructing an interactive tool to generate and complete the schemas [2]. Thus, we should be able to produce a single design representation from which we can both reason about formally specified properties, and generate most of the code.

In addition to providing a basis for code generation and reasoning, this formal approach can be used to address several other important issues. This approach can indicate possible steps towards the definition of a unique process language vocabulary that can describe the interconnection mechanisms at the object, module, and architectural description levels. It can also provide a foundation for the definition of an integrated formal approach to software system

specification and design that considers various levels of abstraction [23, 24]. The formal components can be specified at different levels of detail and thus various program/system/architecture features such as data, control, functionality, behavior, communication, concurrency/distribution/timing concerns can be included in the description as it is developed. Such an approach using ADVs has been briefly described in [10]. Using a uniform formal approach similar to the one we describe in this paper can enhance the quality of a design as the architects and designers work from a single system description.

6 Note to the Reader

The technical reports mentioned in this paper are available through our World Wide Web site at http://csg.uwaterloo.ca/.

References

1. P.S.C. Alencar, D.D. Cowan, D.M. German, K.J. Lichtner, C.J.P. Lucena, and L.C.M. Nova. A Formal Approach to Design Pattern Definition & Application. Technical Report CS-95-34, University of Waterloo, Waterloo, Ontario, Canada, August 1995.
2. P.S.C. Alencar, D.D. Cowan, K.J. Lichtner, C.J.P. Lucena, and L.C.M. Nova. Tool Support for Formal Design Patterns. Technical Report CS-95-36, University of Waterloo, Waterloo, Ontario, Canada, August 1995.
3. P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. A Logical Theory of Interfaces and Objects. Technical Report CS-95-15, University of Waterloo, Waterloo, Ontario, Canada, 1995.
4. P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and L.C.M. Nova. Formal specification of reusable interface objects. In *Proceedings of the Symposium on Software Reusability (SSR'95)*, pages 88–96. ACM Press, 1995.
5. P. Bumbulis, P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. Combining Formal Techniques and Prototyping in User Interface Construction and Verification. In *2nd Eurographics Workshop on Design, Specification, Verification of Interactive Systems (DSV-IS'95)*. Springer-Verlag Lecture Notes in Computer Science, 1995. to appear.
6. P. Coad. *Object Models: Strategies, Patterns & Applications.* Yourdon Press, 1995.
7. J.O. Coplien and D.C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, 1995.
8. D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, January 1993.
9. D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Application Integration: Constructing Composite Applications from Interactive Components. *Software Practice and Experience*, 23(3):255–276, March 1993.
10. D.D. Cowan and C.J.P. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design. *IEEE Transactions on Software Engineering*, 21(3):229–243, March 1995.
11. J. Fiadeiro and T. Maibaum. Temporal Theories as Modularization Units for Concurrent System Specification. *Formal Aspects of Computing*, 4(4), 1992.

12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
13. Martin Gogolla, Stefan Conrad, and Rudolf Herzig. Sketching Concepts and Computational Model of TROLL Light. In *Proceedings of Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, Berlin, Germany, March 1993. Springer.
14. J. Goguen and R. Burstall. *Introducing Institutions*, volume 164 of *Lecture Notes in Computer Science.* Springer-Verlag, 1984.
15. J. Goguen and R. Burstall. Institutions: Abstract Model Systems Theory. *Journal of the ACM*, 39(1):95–146, 1992.
16. William Harrison and Harold Ossher. Subject-Oriented programming (A Critique of Pure Objects). In *OOPLSA'93.* ACM, 1993.
17. William Harrison, Harold Ossher, Randal B. Smith, and Ungar David. Subjectivity in Object-Oriented Systems, Workshop Summary. In *OOPLSA'94.* ACM, 1994.
18. Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.
19. Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *JOOP*, pages 26–49, August September 1988.
20. N. Levy and G. Smith. A Language Independent Approach to Specification Construction. In *Proceedings of the SIGSOFT'94*, New Orleans, LA, USA, December 1994.
21. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.
22. Zohar Manna and Amir Pnueli. *The temporal logic of reactive systems: Specification.* Springer-Verlag, 1992.
23. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.
24. Y. V. Srinivas and R. Jullig. Specware: Formal Support for Composing Software. In *Proceedings of the Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, 1995.
25. W. M. Turski and T. S. E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.