PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 27/95

# Integrating Design Patterns and Subject-Oriented Programming within the ADV Framework

P. S. C. Alencar
D. D. Cowan
C. J. P. Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 27/95
Editor: Carlos J. P. Lucena August, 1995

# Integrating Design Patterns and Subject-Oriented Programming within the ADV Framework *

P. S. C. Alencar
D. D. Cowan
C. J. P. Lucena

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

**In charge of publications:**
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br

# Integrating Design Patterns and Subject-Oriented Programming within the ADV Framework[*]

P.S.C. Alencar [†] D.D. Cowan [‡]
Computer Science Departament
Computer Systems Group, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
e-mails: alencar@csg.uwaterloo.ca, dcowan@csg.uwaterloo.ca,

C.J.P. Lucena [§]
Departamento de Informática,
Ponntifícia Universidade Católica do Rio de Janeiro -PUC/RJ
Rio de Janeiro, RJ, Brasil
e-mail: lucena@inf.puc-rio.br

PUC-RioInf.MCC

August 28, 1995

[*]The work described here has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the National Research Council of Brazil (CNPq), and WATCOM.

[†]P. S. C. Alencar is a Visiting Professor in the Computer Science Department at the University of Waterloo, Waterloo, Ontario, Canada. alencar@csg.uwaterloo.ca

[‡]D.D. Cowan is a Professor in the Computer Science Department at the University of Waterloo, Waterloo, Ontario, Canada. dcowan@csg.uwaterloo.ca

[§]C.J.P. Lucena is a Professor in the Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brazil, and an Adjunct Professor in the Computer Science Department at the University of Waterloo. lucena@csg.uwaterloo.ca

**Abstract**: Design patterns and subject-oriented programming are two object-oriented software implementation techniques directed toward reuse. Subject-oriented programming supports reuse at the component level, while design patterns attempt to reuse "good programming practices." In this paper we recast these two approaches to reuse in a formal framework based on Abstract Data Views (ADVs) and process programs, and then demonstrate through an example how they can be integrated. Basing a design approach on a formal model allows us to reason about the properties of a specific design, and also create tools to assist with the generation of code.

**Keywords**: Object-oriented Programming, Subject-oriented Programming, Design Patterns, Abstract Data Views, User Interfaces, Interfaces, Formal Methods, Development Constructors.

**Resumo**: Padrões de design e programação orientada a "subjects" são duas técnicas de implementação de software orientadas a objetos dirigidas para reuso. Programação orientada a "subjects" suporta reuso no nível dos componentes, enquanto que padrões de design promovem o reuso de "boas práticas" de programação. Neste artigo nós remodelamos estas duas técnicas para reuso em uma abordagem formal baseada em Visões Abstratas de Dados (ADVs) e programas de processo e, então, demonstramos através de um exemplo como elas podem ser integradas. O embasamento da abordagem para design em um modelo formal nos permite raciocinar sobre propriedades de um design específico e também criar ferramentas de suporte para geração de código.

**Palavras-chave**: Programação orientada a Objetos, Programação Orientada a "Subjects", Padrões de Design, Visões Abstratas de Dados, Interfaces com o Usuário, Interfaces, Métodos Formais, Construtores de Desenvolvimento.

## 1 Introduction

In this paper we examine two object-oriented software construction techniques within the context of a formal framework. These two approaches, namely subject-oriented programming [HO93] and design patterns [GHJV95] address different types of reuse within the software implementation process. Subject-oriented programming emphasizes component reuse through separation of intrinsic and extrinsic attributes, while design patterns address reuse of "good programming practices." A unified method integrating both approaches should yield highly maintainable systems composed of reusable objects. Examining these two concepts at a formal level provided several advantages: we could define design patterns formally as process programs; we could include within the pattern descriptions an explicit statement of separation of concerns as exemplified by subject-oriented programming; and we could reason about designs within existing frameworks. In addition, this formalism clarifies the application of patterns and provides a basis for creating tools to support the generation of code from designs.

The process of subject-oriented programming through the use of subjects and subject activations, separates the intrinsic and extrinsic behavior and properties of an object. This clear separation of concerns allows objects to be coupled to each other through subject activations without violating encapsulation, a necessary condition if objects are to be distributed and reused as off-the-shelf certified components. However, subject-oriented programming which supports the reuse of components, is not sufficient to ensure that a software system can be effectively maintained over its entire lifetime. In addition to reuse of components, we should use a process which produces designs general enough to avoid re-design when addressing future problems and requirements.

In order to produce such maintainable designs from reusable components we need to amalgamate a reusable design process with the concepts embodied in subject-oriented programming. With appropriate re-structuring design patterns as described in [GHJV95] can support that approach.

Our approach is based on the Abstract Data View (ADV) [CILS93a, CILS93b, CL95] formal design model which visualizes software systems as composed of two types of components: objects (Abstract Data Objects or ADOs) and object views (Abstract Data Views or ADVs). An object view can be both a viewer and modifier of an object's state. We accomplish our objective of combining these two viewpoints by first showing how the ADV model and subject-oriented programming are related, and that object views are the design analog of subjects. We then show how to specify design patterns more formally through a process programming language and descriptive schemas based on the ADV model. In this context we consider ADVs and ADOs as the basic building blocks of software design supported by various assembly techniques including primitive constructors such as composition, and acquaintance (interconnection) and complex constructs called design patterns.

## 2 The Abstract Data Views Concept

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 that it has a 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 interface or an adaptation of the 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. Figure 1 illustrates both these uses of ADVs.

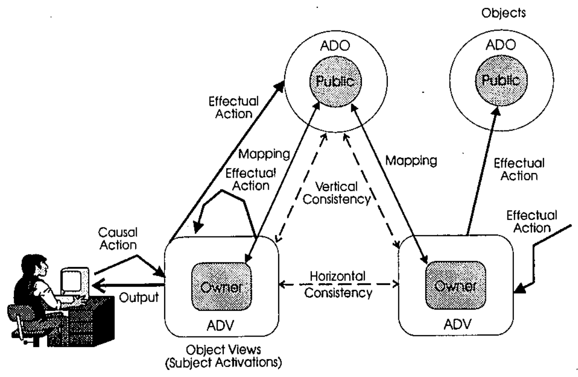


Figure 1: An ADV/ADO interaction model

Both ADVs and ADOs can be acted upon by actions to change or query their state. Actions can be divided into two categories: causal actions and effectual actions. We use the term causal actions to denote the input events acting on an ADV when it is acting as a user interface or interface to some other media. Causal actions are triggered from outside the system and internal objects are not able to generate this type of action. Effectual actions are the actions generated directly or indirectly by a causal action, and are supported by both ADVs and ADOs. The triggering of an effectual action by another action will normally be a synchronous process. An effectual action can be viewed as the activation or call of a method or procedure that is part of the public interface of an ADO or ADV.

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.

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 variables that is labeled "owner" in Figure 1.

If the state of an ADO is changed then any part of the state that is viewed by a connected ADV 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. This mapping or morphism links the ADV with the part of the state that is accessible through the ADO public interface, and so preserves encapsulation. The mapping is a design concept, and several strategies can be used for its implementation. In addition, an ADV may query or change the state of a connected ADO through its normal public interface.

Because of the separation between view and object 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 represented by ADVs must be consistent. For example, where ADVs are part of a user interface, a

```
ADO ADO_Name
    Declarations
        Data Signatures      - sorts and functions
        Attributes           - observable properties of objects
        Effectual Actions    - list of possible effectual actions
        Nested ADOs          - allows composition, inheritance, sets, ...
    Static Properties
        Constraints          - constraints in the attributes values
        Derived Attributes   - non-primitive attribute descriptions
    Dynamic Properties
        Initialization       - list of initialization actions
        Interconnection      - description of the communication process among objects
        Valuation            - the effect of events on attributes
        Behavior             - behavioral properties of the ADO
End ADO_Name
```

Figure 2: A descriptive schema for an ADO

clock ADO could have a digital view, an analog view or both, and they must show the same time. We call consistency among the different ADVs *horizontal consistency*, while consistency between the visual object (ADV) and its associated ADO is called *vertical consistency*. These consistency properties which are illustrated in Figure 1 must be guaranteed by the specification of ADVs, ADOs, and their environment.

## 3 Specification Schemas for ADVs and ADOs

In this section we describe abstract schemas for the specification of ADVs and ADOs [ACLN95]. These schemas which are based on the ones in [CGH92, GCH93, GVH$^+$94] describe how objects modify their state through associated actions. A schema is divided into three sections: declarations, static properties, and dynamic properties, and aspects of these sections use a temporal logic formalism [ACL95b, MP92].

A declaration part contains a description of all the elements that compose the object including sorts, functions, attributes, and actions; the status of these elements is public unless declared otherwise. The static properties part defines all the properties which do not affect the state of the object. Their definitions are always based on values stored by primitive attributes. Dynamic properties establish how the states and attribute values of an object are modified during its lifetime.

### 3.1 Schemas for ADOs

Figure 2 shows the structure of the schemas used in the specification of ADOs.

The *data signature* section of an object consists of a set $S$ of sort names, and a set $F$ of functions. A sort declaration represents an association of a set of values to a sort name. A function specification associates a specific operation to a function name. Among sort expressions we may have the basic abstractions such as *integer* and *string*, and the application-specific abstractions including object instances or user-defined sorts. Sort constructors, such as *set* and *union*, can also be applied to compose complex sort expressions.

*Attributes* denote the state or the set of features of an object that can change over time. *Attributes* can be used by other objects to report on the current state of an object, since *Attributes* are its observable properties. Attribute values can only be affected by actions, defined in the same schema as the attribute to be changed. We can subdivide types of actions into two subtypes: observer and change actions. The set of actions includes create and destroy which are object management procedures for objects derived from this specification.

In the *Nested ADO* section we can introduce the list of all the *component* objects of a composite ADO with the specification constructors that support nesting. Composite objects are responsible for creating the instance of the components.

The static properties of an object are represented by closed formulas. *Constraint* formulas refer to properties that must be true for all time, and *derived attribute* formulas define the derivation rules for the non-primitive attributes from the primitive (base) attributes. A derived attribute is an attribute in that it is a property of the object, but computing the derived attribute does not change the state of the object [Rum91].

The *initialization* of an object is defined by means of effectual actions that initialize attribute values of this object when the object is created. Before the creation of an object every attribute has the value *undefined.*

The *interconnection* section is described by morphisms of actions and attributes. These morphisms or mappings have several uses. They can establish how component objects relate to the composite object, or they can be an instrument to define synchrony between actions of different objects. The intended interpretation of a morphism is that the mapped attributes must always have the same values and the mapped actions must happen simultaneously.

The definitions of morphism are organized by objects, and the morphisms between the current object and other objects are specified by expressions of the form $element1 \longmapsto element2$ where *element*2 always refers to attributes or actions which are defined inside the current object. An important consideration is that an ADO schema cannot contain a reference to any ADV element, since the ADO has no knowledge about the existing ADVs.

The valuation properties of an object describe the changes occurring in attribute values of this object as an immediate consequence of a triggered action. However, the valuation rules are applied only if all the specified pre-conditions for the occurrence of the action are satisfied.

Behavior description is in general a complex task, and we have chosen temporal logic formulas to describe the object behavior. The behavior section defines the sequencing of the actions and changes in the state of the object.

## 3.2 Schemas for ADVs

Figure 3 contains a schema for an ADV as a user interface view, and as we can see by comparing Figures 2 and 3, the ADO and ADV schemas are similar, since they both incorporate object properties. However, there are some distinctions that clearly differentiate their roles.

One difference is found in the header of the schemas. Since an ADO has no knowledge about the existence of ADVs, the header of an ADO schema has only the definition of the ADO name; while, the header of the ADV schema contains both *ADV_Name* and *ADO_Name* declaring an association

```
ADV ADV_Name For 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/output actions
        Nested ADVs          - allows composition, inheritance, sets, ...
    Static Properties
        Constraints          - constraints in the attributes values
        Derived Attributes   - non-primitive attribute descriptions
    Dynamic Properties
        Initialization       - list of initialization actions
        Interconnection      - description of the communication process among objects
        Valuation            - the effect of events on attributes
        Behavior             - behavioral properties of the ADV
End ADV_Name
```

Figure 3: A descriptive schema for an ADV

of the ADO with an ADV. The description of this association is normally defined further in the *interconnection* section. An alternative structure to represent the interconnections between an ADV and an ADO could use another ADV to specify this relationship. This approach allows more flexibility in describing the interconnection. In addition, an ADO has no definition of causal actions in its structure, thus every ADO action is effectual. However, ADVs as interfaces to other media may receive causal (external) actions, which should be declared in a particular section of the schema. This distinction clarifies that the ADO has no interface properties.

ADVs can also be used to specify the way in which one ADO views another. In this context the ADV schema can be modified to specify the relationship between ADOs. For example, in Figure 4 the *client ADO* labeled $ADO_i$ views the *component ADO* labeled $ADO_j$ using $ADV_{ij}$ and thus, $ADV_{ij}$ supports the detailed definition of a view between the objects represented by $ADO_i$ and $ADO_j$.

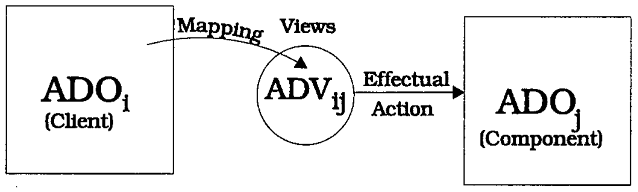


Figure 4: An ADV providing a view between two ADOs

In the ADV approach, the view (ADV) knows the identity of the two participating ADOs, but the two ADOs do not know the identity of the view (ADV). Thus, both the *client* ADO name and the *component* ADO name must be specified in the ADV. The *client* ADO name should appear in the ADV declaration, and the *component* ADO name in the statement that induces an effectual action or message.

Whenever ADO names are declared in the ADV, two placeholder variables *client_owner* and *component_owner* become available inside an ADV instance. The variables *client_owner* and *component_owner* are associated with the *client* and *component* ADO instances respectively. These variables refer to the two ADO instances and allow controlled access and naming. Notice that the concept of *component_owner* is not required in the ADV when it is used as an interface to an external medium because there is no corresponding ADO. This method of specifying the connection to the *component* ADO instance ensures that the *client* instance has no knowledge of the view of the *component* ADO instance.

The *client* ADO instance can be modified only through its operations, but read-only access to its state can be provided through the variable *client_owner*. The *client_owner* variable can then be used in the state invariant for the ADV instance and, in a restricted form, in the pre- and post-conditions, thus, allowing the "appearance" of an ADV to conform to the state of an ADO instance. This connection between the ADO instance and ADV instance ensures that the ADV instance has current knowledge of the state of the ADO instance.

The schema which supports interfaces between ADOs, contains only statements that induce effectual actions, since there is no causal action to trigger an event which would cause an input message. ADO instances do not trigger events and ADV instances as views between ADOs connect only ADO instances together. There can be multiple effectual action statements with the same *client* ADO instance, since different changes in an ADO instance can trigger different *component* ADOs. From an operational viewpoint the ADV can be viewed as a process which responds to changes in the *client* ADO state and responds to these changes by activating the interface of the *component* ADO.

The view specified in Figures 4 supports only uni-directional communication and does not allow the *component* ADO instance to communicate with the *client*. In order to allow bi-directional communication we must use a pair of ADVs. This concept and the schemas for ADVs as views between objects are described fully in [ACL95a].

In summary we observe that there are two types of ADVs which are natural extensions of each other: an ADV which acts as an interface between two different media, and can accept causal actions and instigate effectual actions; and an ADV which induces only effectual actions and acts as an interface between two ADOs operating in the same medium.

## 4 The ADV Model and Subject-Oriented Programming

In this section we illustrate that the ADV model is a formal design model for subject-oriented programming by using a simple example and a tabular comparison based on the tuples in [HO93]. The ADV model does not necessarily address all the implementation-oriented issues mentioned in [HO93, HOSD94]. Nevertheless, the main features of the subject-oriented approach are captured within our ADV formalism.

### 4.1 An Informal Example

The ADV/ADO model supports the concept of intrinsic and extrinsic properties without violating encapsulation, and allows the composition of an application from objects or other applications. Figure 5 contains a simple example that illustrates these concepts. The house in Figure 5 represented

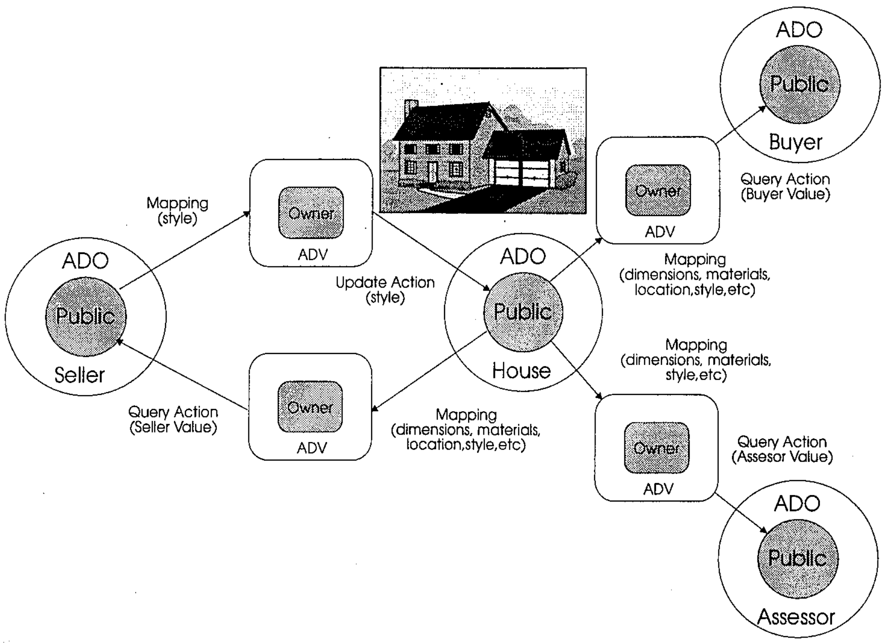


Figure 5: Several views of a house

by an ADO has several intrinsic properties including: materials, dimensions, style and location. These properties are accessed by a number of ADVs (subject activations) through a mapping, and then a different extrinsic value is computed by each ADV. Each of these values is made available to other objects (ADOs) by an effectual action or method. The mapping ensures that the house ADO does not have to know the identity of the view or the ADOs making the query about the value.

The seller ADO in the Figure decides to modify the style of the house through some construction. Hence, we require another ADV connecting the seller to the house to allow the style to be modified. This ADV shows a mapping from style controlled by the seller to the ADV which then causes an update to the style attribute of the house. As soon as the style changes for the house object the mapping connected to the other ADVs ensures that they automatically update their values. The objects in this example do not have to know the identity of other objects, since they are connected through an ADV or subject activation.

## 4.2 A Conceptual Comparison

The subject model is based on the notions of subjects and subject activations. According to Harrison and Ossher [HO93] "A subject is a collection of state and behavior specifications reflecting a particular gestalt, a perception of the world at large such as is seen by a particular application or tool." The essential characteristic of subject-oriented programming is that different subject activations can separately query and change the state of an object without the object being aware of any of the subject activations, and any of the subject activations being aware of each other. Thus, a subject activation can specify extrinsic behavior by utilizing the intrinsic behavior of the object. Although the ADV model uses schemas and temporal logic, the model can support concepts analogous to subjects and subject activations, and in fact was originally conceived to deal with similar issues at a different level of design. There are a number composition rules for ADVs including nesting and loose interconnection.

An informal comparison of the models based on the concepts in [HO93] is summarized in the table in Figure 6. In the table we consider how intrinsic and extrinsic entities are distinguished by both models and we compare how the extrinsic entities are described within the two models. We also compare the way that the extrinsic entities are composed, their extrinsic entity-oriented universes, their extrinsic entity universe activations, and their extrinsic entity activations. This informal comparison indicates that the subject-oriented and the ADV models are conceptually equivalent although addressing the same problem at different levels of abstraction. A more detailed formal comparison and extensive example based on the tuples in [HO93], is presented in [ACL95a].

| Concepts and Models | Subject Model | ADV Model |
|---|---|---|
| Intrinsic Entity | Usual Class | Intrinsic ADO Schema |
| Extrinsic Entity | Subject | ADV Schema |
| Extrinsic Entity Description | Class Names | Extrinsic ADO Schemas |
| | Interfaces | Extrinsic ADO Interfaces |
| | Class Description (Functions) | Extrinsic ADO Descriptions |
| | Superclass Function | ADO Structure Info |
| Extrinsic Entity Composition | Composition Rules: Nesting, Various Other Forms | Interconnection Modes: Nesting; Design Patterns |
| | Sequence of Components (Subjects or Compositions) | Sets or Sequences of ADVs (or Interconnected ADVs) |
| Extrinsic Entity-Oriented Universe | Composition of Subjects | Interconnected ADVs |
| | Subject Universe Activation | ADV Universe Activation |
| Extrinsic Entity Universe Activation | Set of Object Identifiers | Set of Intrinsic ADO Instances |
| | Subject Activation | ADV Activation |
| Extrinsic Entity | A Subject | An ADV Schema |
| | State Function | Extrinsic ADO Instance State Info |
| Activation | Instance-of Function | Extrinsic ADO Schema State and Behavior Info |

Figure 6: A comparison between the subject-oriented and ADV approaches

## 5 The ADV Model and Design Patterns

In addition to providing reusable components to build a system, we also require a reusable process to guide the designer in assembling those components into a maintainable design. Recent work [GHJV95] has produced a catalog of design patterns for composing elementary objects into application structures that are amenable to modification. These guidelines could be used in the context of subject-oriented programming if the design patterns were organized in the context of ADVs and ADOs, that is within the framework of reusable components.

In addition to recasting design patterns into the realm of reusable components, we have formalized the patterns through the use of a process programming language. This formalization helps to eliminate any ambiguities in the process of design pattern instantiation, and should lead to some automation of code production. We first provide an overview of the formalization and then show the formal description of some of the patterns in [GHJV95].

### 5.1 Design Pattern Constructors

We formalize design patterns by introducing development constructors 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 [LS94]. The development constructor structure is shown in Figure 7 where the purpose of each component of the structure is presented.

| | |
|---|---|
| *Operator* **Pattern Name** | |
| *Objective* | Description of the intent of the pattern |
| *Parameters* | External elements used in the pattern definition |
| *Subtasks* | Description of pattern in terms of primitive constructors |
| *Consequences* | How the pattern supports its objective |
| *Product Text* | Language-dependent specification of pattern |
| *End Operator* | |

Figure 7: Development constructor structure for a design pattern

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 language-dependent part of the pattern constructor 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 is a helpful instrument in guiding the user to a particular specification. Such templates are illustrated using the pattern development constructors and the formalism of ADV/ADO schematic representations described in Section 3.

```
Operator Design Pattern Facade
    Objective       Provide a unified interface to a set of interfaces in a subsystem
    Parameters      Objects: SubSys1, ..., SubSysN;
    Subtasks        1 - Create Object: → FACADE
                    2 - Compose Objects: SubSys1, ..., SubSys, FACADE → FACADE

    Consequences    FACADE is a higher-level interface object shielding many other subsystem interfaces

    Product Text    ADV/ADO FACADE
                        Declarations
                        ...
                          Nested ADVs/ADOs
                              Compose SubSys1, ..., SubSysN;
                        ...
                    End FACADE
End Operator
```

Figure 8: Specification of facade pattern constructor

## 5.2 Examples of Design Pattern Constructors

In this section we describe how to specify the facade and composite design patterns based on the pattern constructor described in Figure 7. We have chosen these two patterns since they support the example in Section 6. Several other examples of design patterns from the three categories in [GHJV95] are presented in [ACL95a]. There is a substantial difference between the pattern specifications in [GHJV95], and our specifications. The patterns in [GHJV95] are based on OMT diagrams and C++ templates, while our version of the patterns is based on a process language and the formalism of the ADV design approach.

The development operators in the following examples are sequentially numbered in a section called *Subtasks*, while the object schemas are defined in the language-dependent section called *Product Text*. Other sections complete the design pattern specifications by providing additional information. The *Subtasks* which specify how to instantiate a pattern are given in a task or function notation of the form: f: x → 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.

The *facade* pattern, shown in Figure 8, is a pattern that composes many interface modules into a single one. The pattern has two subtasks: "Create Object" which returns a copy of the FACADE *Product Text* and the function "Compose Objects" which takes the the N+1 arguments SubSys1 ..., SubSysN and FACADE and returns the modified *Product Text* for FACADE.

The current specification approach differs from the one proposed in [GHJV95], in that the link between a view and its application object is represented by the ADV *mapping* mechanism, which was explained in Section 2. This approach does not describe the implementation of the link, but indicates a morphism between elements of the objects involved. In contrast, the design described in [GHJV95] proposes a design technique that is closer to implementation than the proposed *mapping*. However, experiments with the ADV approach [CL95] indicate that efficient implementations of this mapping are possible.

```
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 of ComponentType;
                                Inherit Component;
                        ...
                    End COMPOSITE

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

Figure 9: Specification of composite pattern constructor

Figure 9 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.

## 6 Integrating Subjects and Design Patterns — A Case Study

In this section we present a partial case study based on the editor described in [GHJV95]. Some of the design patterns used in this case study are described in Section 5. The structure of the system is described with an OMT diagram notation [Rum91] extended for the ADV approach. The objects in this notation are represented by ADV boxes or ADO boxes, while dashed arrows represent the mapping between an ADV and its associated ADO. The lozenge, triangle and black circle in the OMT diagram represent aggregation, inheritance and zero or more associations respectively. Thus, in Figure 10, *Document* is an aggregation of zero or more *Element(s)* and *Composite View* inherits from *Document View.*

After the description of the editor OMT diagrams, each of the objects composing the design is specified by means of the schemas introduced in Section 3. The specifications are somewhat simplified, so that the concepts are not obscured with detail.

The first OMT diagram presents the basic data structure. This structure is defined using the *composite* design pattern, and is illustrated in Figure 10. This structure is used to build a document and its view from basic elements, which in our example are characters and pictures. While the application objects (ADOs), defined in Figure 11, are oriented toward sequencing of document elements, the interface objects (ADVs), specified in Figure 12, represent the user views of the document, consisting of columns and lines of text.

The sequence of elements is stored in the *Document* ADO, while the organization of the text in columns and lines is defined by the composite design pattern, which consists of ADV objects. Character and picture views are represented by *CharView* and *PictureView* ADVs, respectively, while the document hierarchy is defined by *CompositeView* ADVs. All of these ADVs have interfaces defined by the *GlyphInterface* abstract class.

The command pattern defines how the document editor functions associated with the user interface buttons are encapsulated inside objects. While the *ButtonsView* ADV composes a set of *Button* ADVs to create menus or any other user interface objects, the *Command* ADV defines an abstract class that specifies the interface of the objects that will implement the respective button operations. Additionally, the *Button* ADV composes objects that inherit the *Command* interface in order to call the functions associated with these objects.

In Figure 13 we show the OMT diagram containing the application of the *command* design pattern to the case study. The diagram shows the objects described in the previous paragraph together with two examples of interface objects inheriting from the *Command* ADV, namely *ChangeFont* and *CutText*. While the former ADV is related to the *CharView* ADV, from which it triggers

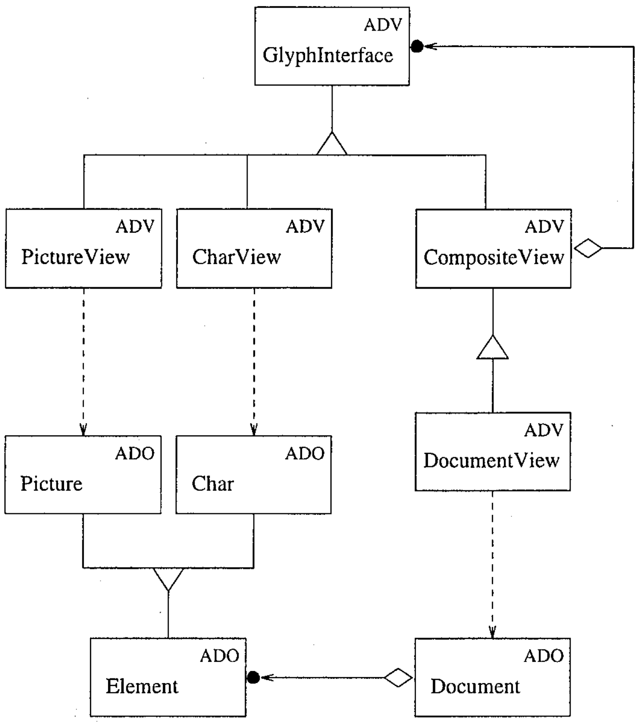


**Figure 10: The document editor structure**

```
ADO Document
    Declarations
        Attributes
            DocType: string;
        Effectual Actions
            InsertElement: position, ADO Element;
            DeleteElement: position;
        Nested ADOs
            Sequence of Element;
End Document

ADO Element
    Declarations
        Attributes
            ASCII: nat;
            Font: string;
            File: string;
            Format: string;
        Effectual Actions
            SetValue: nat;
            SetForm: string;
            SetFileName: string;
            SetFormat: string;
End Element

ADO Char
    Declarations
        Attributes
            Element.ASCII: nat;
            Element.Font: string;
        Effectual Actions
            Element.SetValue: nat;
            Element.SetForm: string;
        Nested ADOs
            Inherit Element;
End Char

ADO Picture
    Declarations
        Attributes
            Element.File: string;
            Element.Format: string;
        Effectual Actions
            Element.SetFileName: string;
            Element.SetFormat: string;
        Nested ADOs
            Inherit Element;
End Picture
```

**Figure 11: Basic ADOs of the document editor**

```
ADV DocumentView ForADO Document
    Declarations
        Attributes
            CompositeView.GlyphInterface.Child: ADV GlyphInterface;
        Effectual Actions
            CompositeView.GlyphInterface.CreateChild: ADV GlyphInterface, position;
            CompositeView.GlyphInterface.RemoveChild: position;
            CompositeView.GlyphInterface.Draw: coordinates;
            CreateElement: position, ADV GlyphInterface, ADO Element;
            RemoveElement: position;
        Nested ADVs
            Inherit CompositeView;
End DocumentView
```

```
ADV GlyphInterface
    Declarations
        Attributes
            Size: nat;
            Child: ADV GlyphInterface;
        Effectual Actions
            Draw: coordinates;
            CreateChild: ADV GlyphInterface;
            RemoveChild: ADV GlyphInterface;
End GlyphInterface
```

```
ADV CharView ForADO Char
    Declarations
        Attributes
            GlyphInterface.Size: nat;
        Effectual Actions
            GlyphInterface.Draw: coordinates;
        Nested ADVs
            Inherit GlyphInterface;
End CharView
```

```
ADV PictureView ForADO Picture
    Declarations
        Attributes
            GlyphInterface.Size: nat;
        Effectual Actions
            GlyphInterface.Draw: coordinates;
        Nested ADVs
            Inherit GlyphInterface;
End PictureView
```

```
ADV CompositeView
    Declarations
        Attributes
            GlyphInterface.Child: ADV GlyphInterface;
        Effectual Actions
            GlyphInterface.CreateChild: ADV GlyphInterface;
            GlyphInterface.RemoveChild: ADV GlyphInterface;
        Nested ADVs
            Inherit GlyphInterface;
            Sequence of GlyphInterface.Child;
End CompositeView
```

**Figure 12: ADVs forming the composite pattern**

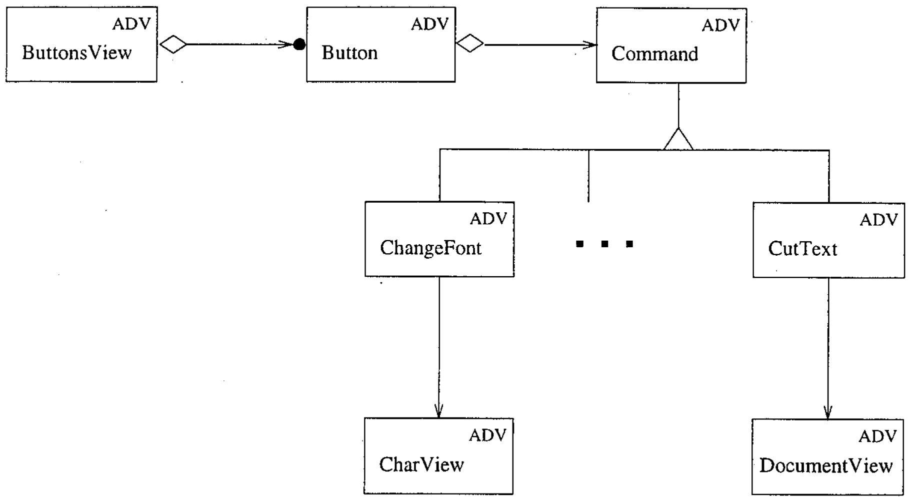


Figure 13: The command design pattern application

methods that changes the character font, the *CutText* ADV performs its operations based on its relation to the *DocumentView* ADV. All of these objects have their specification schemas described in Figure 14.

The facade pattern is the third pattern used in our specification and it has a very simple role. It composes the functions of the *DocumentView* and *ButtonsView* ADVs in order to present the user interface as a single structure. Such a pattern and the ADV it introduces are shown in Figure 15.

## 7 Conclusions

This paper contains the description of an integrated formal design model that supports both subject-oriented programming and design patterns. The basis for the model is the division of a software system into objects (Abstract Data Objects or ADOs), and object views (Abstract Data Views or ADVs). The object views are not just observers of objects; they can also change an object's state. Furthermore, object views are the design analog of subjects. Using the ADV design approach we believe we have been able to address many of the design issues of subject-oriented programming. Although the model is at a higher level of abstraction we have been able to map our designs systematically into object-oriented implementations of software systems. Of course if subject-oriented languages and systems were available, then this mapping process would be much easier.

The Abstract Data View Model was first created to separate the specification of the user interface components (ADVs) from the application components (ADOs) in an application, and was initially described in [CILS93a, CILS93b]. One such implementation was the MVC model, although several other implementation strategies have been satisfactorily explored. Subsequent work [CL95] extended the model to include subjects and subject activations which we called interfaces. The work on subject-oriented programming caused us to view our formal models in this context, and

```
ADV ButtonsView
    Declarations
        Attributes
            CommandName: ADV Command;
        Effectual Actions
            AssociateCommToButton: ADV Command;
        Nested ADVs
            Set of Button;
    Dynamic Properties
        Interconnection
            WithADV Button
                SetCommand ⟼ AssociateCommToButton;
End ButtonsView
```

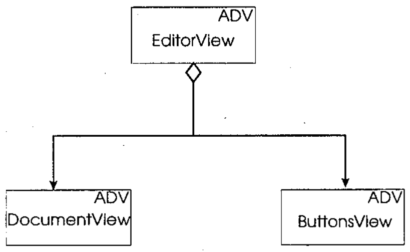
```
ADV Button
    Declarations
        Attributes
            ButtonName: string;
            Position: coordinates;
            CommandName: ADV Command;
        Causal Actions
            Pressed: boolean;
        Effectual Actions
            SetCommand: ADV Command;
        Nested ADVs
            Compose CommandName;
    Dynamic Properties
        Interconnection
            WithADV Command
                Execute ⟼ Pressed;
End Button
```

```
ADV Command
    Declarations
        Effectual Actions
            Execute;
End Command
```

```
ADV ChangeFont OnADO Char
    Declarations
        Attributes
            FontName: string;
        Effectual Actions
            Change: string;
            Command.Execute;
        Nested ADVs
            Inherit Command;
    Dynamic Properties
        Interconnection
            WithADO Char
                SetForm ⟼ Change;
End ChangeFont
```

```
ADV CutText
    Declarations
        Attributes
            PosInit: nat;
            PosEnd: nat;
        Effectual Actions
            Remove: position;
            Command.Execute;
        Nested ADVs
            Inherit Command;
    Dynamic Properties
        Interconnection
            WithADV DocumentView
                RemoveElement ⟼ Remove;
End CutText
```

Figure 14: Objects of the command design pattern application



```
ADV EditorView
    Declarations
        Nested ADVs
            Compose DocumentView, ButtonsView;
End EditorView
```

Figure 15: The facade design pattern

inspired the production of this paper.

We view ADVs and ADOs as the basic building blocks of software design, where the ADV approach encompasses at least two basic activities. ADV-oriented design must:

1. identify the ADOs and ADVs or subjects related to an application and determine the way the views are related to the intrinsic ADO objects, and

2. define approaches to assemble these basic building blocks into a software system.

Both these activities can be performed at various levels of abstraction and can be more or less formal. Subject-oriented programming currently appears to take an implementation-oriented perspective in which it does not appear possible to reason about the designs. We are tackling these problems from a more abstract or design level, and because of our formulation of the design we can support formal reasoning [ACL95b, BACL95].

In order to assemble the basic building blocks we need various techniques. Our approach supports primitive constructors such as composition, and acquaintance (interconnection) at one level, and then allows us to construct components in the form of design patterns, thus, reusing the work reported in [GHJV95]. Because we have a formal method of expressing design patterns we are able to map many of the patterns into code schemas that can be completed interactively. We are currently building tools to support this activity.

Object-oriented analysis and design methods (OOADMs) [Boo91, CY91a, CY91b, dCLF93, MO92, SM88, WBWW90] do not appear to support subject-oriented programming directly. However, we have observed that a design level approach as typified by the ADV formalism can be easily incorporated into most of the well known methods. This observation is supported by the work reported in [HvdGB93]. Incorporating the ADV approach into one of the common OOADMs would allow the creation of a subject-oriented method without the necessity of creating a whole new approach.

## 8 Note to the Reader

Many of the technical reports mentioned in this paper are available via anonymous ftp from [csg.uwaterloo.ca] at the University of Waterloo. The names of the technical reports are in the file "pub/ADV/README" and electronic copies of the reports in postscript format are in the directories "pub/ADV/demo," "pub/ADV/theory," and "pub/ADV/tools." These reports are also on the World Wide Web at "http://csg.uwaterloo.ca:80/ADV.html" or "ftp://csg.uwaterloo.ca/pub/."

## References

[ACL95a] P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. Integrating Design Patterns and Subject-Oriented Programming within the ADV Framework. Technical Report CS-95-33, University of Waterloo, Waterloo, Ontario, Canada, July 1995.

[ACL95b] 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. submitted to IEEE Transactions on Software Engineering.

[ACLN95] 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.

[BACL95] 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.

[Boo91] G. Booch. *Object-Oriented Design With Applications.* The Benjamin/Cummings, Redwood City, 1991.

[CGH92] Stefan Conrad, Martin Gogolla, and Rudolf Herzig. TROLL light: A core language for specifying objects. Technical Report 92-02, Technische Universitat Braunschweig, December 1992.

[CILS93a] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, January 1993.

[CILS93b] 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.

[CL95] 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.

[CY91a] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press, Prentice-Hall, 1991.

[CY91b] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice-Hall, Englewood Cliffs, 1991.

[dCLF93] Dennis de Champeaux, Douglas Lea, and Penelope Faure. *Object-Oriented System Development.* Addison-Wesley, Reading, Massachusetts, 1993.

[GCH93] 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.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[GVH$^+$94] M. Gogolla, N. Vlachantonis, R. Herzig, G. Denker, S. Conrad, and H.D. Ehrich. The KORSO approach to the development of reliable information systems. Technical Report 94-06, Technische Universitat Braunschweig, June 1994.

[HO93] William Harrison and Harold Ossher. Subject-Oriented programming (A Critique of Pure Objects). In *OOPLSA'93*. ACM, 1993.

[HOSD94] William Harrison, Harold Ossher, Randal B. Smith, and Ungar David. Subjectivity in Object-Oriented Systems, Workshop Summary. In *OOPLSA'94*. ACM, 1994.

[HvdGB93] Shuguang Hong, Geert van den Goor, and Sjaak Brinkkemper. A Formal Approach to the Comparison of Object-Oriented Analysis and Design Methodologies. In *Proceedings of HICSS-26*, January 1993.

[LS94] N. Levy and G. Smith. A Language Independent Approach to Specification Construction. In *Proceedings of the SIGSOFT'94*, New Orleans, LA, USA, December 1994.

[MO92] J. Martin and J. Odell. *Object-Oriented Analysis and Design.* Prentice-Hall, Englewood Cliffs, 1992.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive systems: Specification.* Springer-Verlag, 1992.

[Rum91] James Rumbaugh. *Object-oriented modeling and design.* Prentice Hall, 1991.

[SM88] S. Shlaer and S. J. Mellor. *Object-Oriented System Analysis.* Prentice-Hall, New York, 1988.

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.