PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 28/95

A Formal Approach to Design Pattern Definition and Application

P. S. C. Alencar
D. D. Cowan
D. M. Germán
K. J. Lichtner
L. C. M. Nova
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 28/95
Editor: Carlos J. P. Lucena August, 1995

A Formal Approach to Design Pattern Definition and Application *

P. S. C. Alencar
D. D. Cowan
D. M. Germán
K. J. Lichtner
L. C. M. Nova
C. J. P. Lucena

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

A Formal Approach to Design Pattern Definition & Application[*]

P.S.C. Alencar [†] D.D. Cowan [‡] D.M. Germán [§] K.J. Lichtner [¶] L.C.M. Nova [||]
Computer Science Departament
Computer Systems Group, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
e-mails: alencar@csg.uwaterloo.ca, dcowan@csg.uwaterloo.ca,
dmg@csg.uwaterloo.ca,kurt@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 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

[§]D.M. Germán is a PhD candidate in the Computer Science Dept. at University of Waterloo. dmg@csg.uwaterloo.ca

[¶]K.J. Lichtner is a PhD candidate in the Computer Science Dept. at University of Waterloo and holds a doctoral fellowship from NSERC. kurt@csg.uwaterloo.ca

[||]L.C.M. Nova is a PhD candidate in the Computer Science Department at University of Waterloo and holds a doctoral fellowship from CNPq. lcmnova@neumann.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: In this paper we present a formal approach to define and apply design patterns that is both process- and reuse-oriented. Initially we use a process program based on design pattern primitive tasks or constructors to describe how to instantiate a pattern. As we develop the patterns we introduce a formal model for the interconnected objects that constitute the instantiation. The formal model which is based on Abstract Data Views divides designs into both objects and views in order to maintain a separation of concerns. We have chosen a formal model for pattern definition and application since it allows us to specify the steps in pattern instantiation unambiguously and to reason about the completed design. Furthermore, a formal statement of the application of a design pattern can provide the foundation on which to build tools to assist the programmer in code generation.

Keywords: Object-Oriented Programming, Design Patterns, Reuse, Abstract Data Views, Formal Specification.

Resumo: Neste artigo nós apresentamos um método formal para a definição e aplicação de padrões de design que é tanto orientado ao reuso quanto a processos. Inicialmente nós usamos um programa de processos baseado em tarefas ou construtores primitivos de padrões de design para descrever como instanciar um padrão. Ao apresentarmos os padrões nós introduzimos um modelo formal para a interconexão dos objetos que constituem a instanciação. O modelo formal, que é baseado Visões Abstratas de Dados (ADVs), divide designs em objetos e visões para manter a separação de atribuições. Nós escolhemos um modelo formal para a definição e aplicação de padrões uma vez que este modelo permite especificar os passos da instanciação de padrões não ambiguamente e raciocinar sobre o design completo. Além disso, o enunciado formal da aplicação de um padrão de design pode fornecer a base teórica para a construção de ferramentas para assistir o programador na geração de código.

Palavras-chave: Programação Orientada a Objetos, Padrões de Design, Reuso, Visões Abstratas de Dados, Especificação Formal.

1 Introduction

Design patterns[GHJV95] can be viewed as an approach to encapsulating "good practice" in object-oriented programming, where the patterns indicate to the programmer how carefully engineered collections of objects can be used to produce programs. Design patterns add another reuse dimension to object-oriented programming since they present a way to use repeatedly the experience of the best programmers in structuring programs.

Design patterns were partially inspired as an outgrowth of the MVC model[KP88] which is a programming model to achieve a separation of concerns and allow the definition of the user interface (controller and view) to be separated from the application (model). Even though the MVC model provided the inspiration, some of the examples found in [GHJV95] do not uphold this principle of separation between the model and the controller/view.

Currently design patterns have been grouped into three major categories and been informally described in a catalogue[GHJV95] using text and diagrams. The descriptions can be viewed as an informal recipe or process for producing instantiations of specific patterns in languages such as Smalltalk or C++.

An examination of design patterns prompts the question: "Can we describe design patterns more formally, and is there any advantage in such a description?" There are two aspects to design patterns that could be considered for formalization: the process of producing specific instantiations, and the use of formally defined components to substitute in these instantiations. If the process is defined through a process programming language with formal syntax and semantics, then any ambiguities in the process of design pattern instantiation should be eliminated. Eliminating ambiguity should make it easier to derive code consistently and perhaps even lead to some automation of the production of code for the particular instantiation of a design pattern. 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[ACL95, BACL95] of this type. Recent investigations[BACL95] 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 instantiation of components is based on the Abstract Data View (ADV) approach[CILS93a, CILS93b, CL95] which uses a formal model to achieve separation by dividing designs into two types of components: objects and 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. This creates patterns that are similar but not identical to the ones in[GHJV95]. Each design pattern has an associated process program that describes how to substitute these components to create a specific instantiation.

In this paper we describe the ADV formal model and the associated formal schemas for ADVs and ADOs. Some of the patterns are then used to illustrate a formal approach to the process of constructing instantiations of design patterns. Finally we present a case study based on the editor described in [GHJV95].

2 The Abstract Data View (ADV) and Abstract Data Object (ADO) Concepts

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.

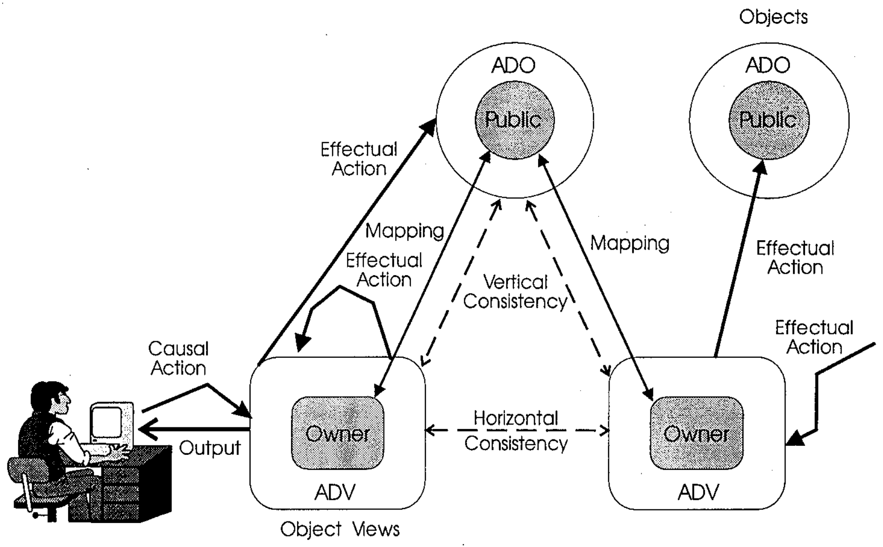


Figure 1: An ADV/ADO interaction model

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. A keystroke or a mouse click are simple examples of input events that are causal actions. Effectual actions are the actions generated directly or indirectly by a causal action, and are supported by both the 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.

In an ADV/ADO configuration only causal actions can change the state of the system. In other words, a causal action can make an effectual action occur, but an effectual action can not make a causal action occur. Since causal actions come from outside the system, causal actions can not cause other causal actions. If we visualize a tree of actions occurring over time, then a causal action can only appear at the root of a tree of action.

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. The form of this association does not violate encapsulation as we compose ADOs and ADVs to make larger systems. The ADV model supports a number of composition mechanisms which are described in detail in [ACL95].

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. In the formal schema for an ADV, the names of the connected ADOs are represented by placeholder variables that are collectively labelled "owner" in Figure 1. This naming convention allows the connection of objects without violating encapsulation.

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 are available 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 is 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 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.

Since ADVs are also objects they can be encapsulated and connected to other ADOs or ADVs by ADVs. Thus, if our views are placed in a distributed system, and become separated from their corresponding objects we can introduce the concept of time delay into the specification through auxiliary ADVs.

3 Specification Schemas for ADVs and ADOs

In this section we describe abstract schemas or classes for the specification of ADVs and ADOs [ACLN95]. These schemas are useful tools for both formal and informal program specifications [CGH92, FHW93], and are based on the ones described in [CGH92, GCH93, GVH$^+$94]. A schema for an object in a system describes how that object modifies its state through its associated actions.

Components of the ADV model are called objects since their schema specifications describe behavior over the lifetime of the object, and involve both static and dynamic properties. Both interface and application components are composed of dynamic properties that specify changes in the attributes representing the state memory of the objects.

```
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.

ADVs and ADOs have distinct roles in a software system and, as a consequence, they are described by different schemas. These schemas are not the actual objects inside a system, but rather provide descriptions of their static and dynamic properties and declarations of entities that are used within the scope of the object. The specification syntax of the whole schema is presented essentially through a temporal logic formalism [MP92]. Every ADV or ADO structure is subdivided into three sections: declarations, static properties, and dynamic properties.

A declaration part contains a description of all the elements that compose the object including sorts, functions, attributes, and actions. They have a public status unless explicitly declared as private. 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 to be 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. The functions, denoting operations over given values, are also part of the data signature section.

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. All objects are components of some composite object; objects which do not belong to a composite object are a component of the "system." Composite objects are responsible for creating the instance of the components. Since the creation action can happen only once in a life of any object, then a component object can never be nested in two distinct composite objects. The rules used here for ADOs also apply to ADVs.

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, how ADVs are associated with an ADO, 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 each other object are specified by expressions of the form $element1 \longmapsto element2$ where $element2$ 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. Most of the object-oriented models represent system behavior by means of transition networks, which are augmented state-machine diagrams [dCLF93]. We have chosen temporal logic formulas to describe the object behavior. However, there are a number of graphical representation languages that we could have chosen instead to describe the *behavior* of objects [CCL93, Har87, CHB92, Che91]. The behavior section defines the sequencing of the actions and changes in the state of the object.

3.2 Schemas for ADVs as User-Interface Views

Figure 3 contains a schema for an ADV, 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 the roles of ADOs and ADVs in the ADV design approach.

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,

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 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.

the header of the ADV schema contains both *ADV_Name* and *ADO_Name* declaring an association 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 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.

3.3 Schemas for ADVs as Views Between ADOs

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* labelled ADO_i views the *component ADO* labelled ADO_j using ADV_{ij} and thus, ADV_{ij} supports the detailed definition of a view between the modules represented by ADO_i and ADO_j.

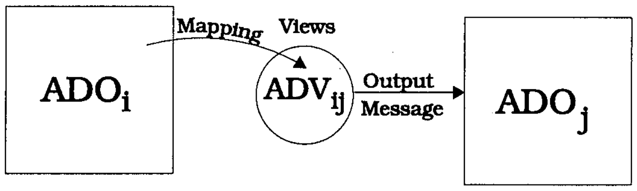


Figure 4: An ADV providing a view between two ADOs

Figure 6 contains a modified version of the ADV schema from Figure 3, and this new definition supports the concept of a view between 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, in the syntax of the modified schema, the representation relation requires that both the client ADO name and the component ADO name be specified in the ADV. The client ADO name should appear in the ADV declaration, and the component ADO name in the output actions or messages.

Whenever the ADO names are declared in both the ADV header and output message declarations, two pseudo-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 pseudo-variables refer to the two ADO instances and allow controlled access and naming.

The pseudo-variable *component_owner* connects an instance of the ADV to an instance of the component ADO and provides access to its view. 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 pseudo-variable *client_owner* associated with an instance of the client ADO is made available inside the ADV. This pseudo-variable refers to the client ADO instance and allows controlled access to the state of the 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 pseudo-variable *client_owner*. The *client_owner* pseudo-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 output message definitions, 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 output messages 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.

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

The view specified in Figures 4 and 6 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 as depicted in Figure 5. The second ADV named ADV_{ji} provides access to the state of ADO_j and communicates with ADO_i through the output message. These two ADVs are composed into a single ADV called ADV_{ij}^{b} and this specification schema is outlined in Figure 7. The associated component schemas ADV_{ij} and ADV_{ji} are described in Figure 8.

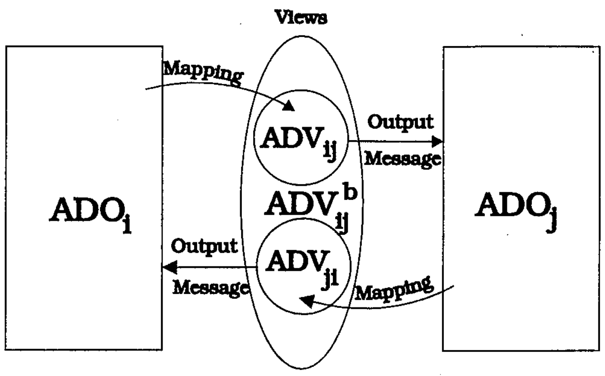


Figure 5: A pair of ADVs providing a bi-directional view between two ADOs

3.4 The specification schemas for ADV and ADO class templates

So far, we have introduced the concept of formal specification schemas or classes for ADVs and ADOs. However, certain of the design patterns are in the creational category and need the specification of meta-level requirements such as a limit to the number of instances produced from a specific schema. For this reason we introduce class templates that reference specific schemas and manage the creation and destruction of objects produced from those schemas. We consider these class templates as meta-objects which support inheritance and nesting of classes.

Class templates support an inheritance structure so that they can be derived from existing class templates. Similarly nesting allows a class template to be composed of other class templates.

The class template for an ADV is described in Figure 9; the schema is similar to the schema for an ADV object except that we are now primarily describing the class specification related to object management. The object specification associated with this class, in this case an ADV, is described in the Class_Member statement. An object is created from within the class template by using the name of the specification for that object (ADV or ADO) and invoking the method Add_Object.

4 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 [GHJV95] provide this form of reuse. Each design pattern is a meta-description of a solution for a small problem that occurs frequently in the design of general systems. The application of the meta-description results in a collection of a few objects that form a specific instantiation of such a design problem.

4.1 Design Pattern Constructors

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

```
ADV ADV_Name For ADO_Map_Name
    Declarations
        Data Signatures
            ...
        Attributes
            ...
        Effectual Actions
            Action_Output_1 For ADO_Output_Names;
            ...
            Action_Output_N For ADO_Output_Names;
    Dynamic Properties
        Interconnection
            ...
        Valuation
            ...
        Behavior
            ...
End ADV_Name
```

Figure 6: An ADV schema for a view.

```
ADV $ADV^b_{ij}$_Name For ADO_Map_Name
    Declarations
        Data Signatures
            ...
        Attributes
            ...
        Nested ADVs
            Compose $ADV_{ij}$ For $ADO_j$
            Compose $ADV_{ji}$ For $ADO_i$
        Effectual Actions
            ...
    Dynamic Properties
        Interconnection
            ...
        Valuation
            ...
        Behavior
            ...
End ADV_Name
```

Figure 7: An ADV specification schema for a bidirectional view.

```
ADV $ADV_{ij}$ [$ADV_{ji}$] For $ADO_i$ [$ADO_j$]
    Declarations
        Data Signatures
            ...
        Attributes
            ...
        Effectual Actions
            Action_Output_1 For $ADO_j$ [$ADO_i$];
            ...
            Action_Output_N For $ADO_j$ [$ADO_i$];
    Dynamic Properties
        Interconnection
            ...
        Valuation
            ...
        Behavior
            ...
End ADV_Name
```

Figure 8: ADV component schemas for a bidirectional view.

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 [LS94].

The language-independent part of the structure should clearly define the characteristics of a design pattern. According to [GHJV95], 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. Such a statement is described in an *Objective* section as shown in the development constructor in Figure 10.

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 the evaluation of 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

```
Class of ADV Class_Name
    Class_Declarations
            Class_Member Object_Name(Number) : ADV_Specification_Name
            Class_ Attributes
                        Number_of_Members
            Class_Actions Create_Class;
                        Add_Object (Number, Name);
                        Remove_Object(Number);
                        Destroy_Class
            Nested ADV Classes
                        allows composition, inheritance, sets, ...
    Class_Static Properties
            Class_Constraints
                        constraints in class attributes values
            Class_Derived Attributes
                        non-primitive class attribute descriptions
    Class_Dynamic Properties
            Class_Initialization
                        list of class initialization actions
            Class_Valuation
                        Create_Class $\rightarrow \bigcirc$ Number_of_members $= 0$
                        Add_Object $\rightarrow \bigcirc$ Number_of_members $=$ Number_of_members $+ 1$
                        Remove_Object $\rightarrow \bigcirc$ Number_of_members $=$ Number_of_members $- 1$
            Class_Interconnection
                        Add_Object (Number, Name) $\longmapsto$ Object_Names(Number)·Create(Number,Name)
                        Remove_Object (Number, Name) $\longmapsto$ Object_Names(Number)·Destroy(Number,Name)
End ADV Class_Name
```

Figure 9: A descriptive schema for classes.

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

Figure 10: The structure of the development constructor for a design pattern.

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 3.

4.2 Examples of Design Pattern Constructors

In this section we describe how to specify the adapter, decorator and composite design patterns based on the pattern constructor described in Figure 10. Several more patterns are described in [ACG+95]. There is a substantial difference between the pattern specifications presented in [GHJV95], and the specifications introduced in this paper. The patterns in [GHJV95] are based on OMT diagrams and C++ templates, and are much closer to the implementation level than the version of the pattern descriptions we present in this paper. Our version of the patterns is based on the specification formalism associated with the ADV design approach.

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 system design. Other sections complete the design pattern specifications by providing additional information.

The first pattern we specify is the *adapter*, and the corresponding specification schema is shown in Figure 11. The purpose of the *adapter* is to modify 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 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 words "inherit from TARGET" inserted in the appropriate location.

The adapter might be seen as an object which is a wrapper for another object, and it can be used for modifications of interface objects (ADVs) as well as application objects (ADOs). An adapter object might be seen as a view (ADV) for an application object (ADO).

Figure 12 describes the specification of the *composite* design pattern. This pattern defines a

```
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: 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
```

Figure 11: Specification of adapter pattern constructor.

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 as well, since the *composite* design pattern might be used to structure both user interface objects and application objects.

Figure 13 illustrates how a design pattern called *decorator* extends the functionality of a module. The extension is made dynamically by concrete objects that inherit their interface from the abstract class called *decorator*, which is also linked to the original functions defined in the *component* objects. The *concrete component* objects, as defined in the pattern constructors, are the ones that will have their responsibilities extended.

Objects derived from the decorator pattern are particularly suitable when subclassing is not a practical option for functional extensions of objects. They are useful elements for both applications and views.

```
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 12: Specification of composite pattern constructor.

```
Operator Design Pattern Decorator
    Objective       Attach additional responsibilities to an object dynamically
    Parameters      Objects: COMPONENT;
    Subtasks        1 - Instantiate Concrete Object: COMPONENT → ConcreteCOMPONENT
                    2 - Create Interface Object: COMPONENT → DECORATOR
                    3 - Instantiate Concrete Object: DECORATOR → ConcreteDECORATOR

    Consequences    You can attach dynamically ConcreteCOMPONENT functionality to ConcreteDECORA-
                    TOR objects

    Product Text    ADV/ADO DECORATOR
                        Declarations
                            ...
                            Attributes
                                ComponentType: ADO COMPONENT;
                            Nested ADVs/ADOs
                                Compose ComponentType;
                                Inherit Component;
                        ...
                    End DECORATOR

                    ADV/ADO ConcreteCOMPONENT
                        Declarations
                            ...
                            Nested ADVs/ADOs
                                Inherit Component;
                        ...
                    End ConcreteCOMPONENT

                    ADV/ADO ConcreteDECORATOR
                        Declarations
                            ...
                            Effectual Actions
                                AttachComponent: ADO COMPONENT;
                            Nested ADVs/ADOs
                                Inherit Decorator;
                        Dynamic Properties
                            ...
                            Valuation
                                AttachComponent(ComponentType) →
                                ○ ComponentType = ProvideInstance(COMPONENT);
                        ...
                    End ConcreteDECORATOR
End Operator
```

Figure 13: Specification of decorator pattern constructor.

5 A Case Study

In this section we present a partial case study based on the editor structure described in [GHJV95]. The design patterns used in this case study are some of the ones described in Section 4, and although similar in intent to the ones used in [GHJV95], we have adapted them to conform to the concepts introduced through the ADV design approach.

The static structure of the system is described in terms of an OMT diagram notation [Rum91] extended to use the ADV approach to design. The objects in this extended notation are represented either by ADV boxes or ADO boxes, while dashed arrows are used to represent the mapping between an ADV and its corresponding ADO. Checks should be applied to designs using this notation to ensure that horizontal and vertical consistency constraints are maintained. The lozenge and the triangle in the OMT diagram represent aggregation and inheritance respectively. The small black circle represents zero or more associations. Thus, in Figure 14 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 schematic structures introduced in Section 3. The case study object specifications are left incomplete and somewhat simplified, so that the basic concepts represented by the ADV approach and the formal expression of design patterns can be easily understood. The first OMT diagram to be introduced in the editor specification is the one describing the basic data structure. This structure is defined using the *composite* design pattern, and is illustrated in Figure 14. 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 15, are oriented toward sequencing of document elements, the interface objects (ADVs), specified in Figure 16, 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.

In this case study, we use the decorator design pattern to add border and scroll bars to the user view of the editor. Using the definition of this pattern in Figure 13 together with the diagrams and schemas shown in Figure 17, we see that the *EnhancedDocumentView* ADV defines the embellished user view of the editor. Such an ADV inherits its interface from the *Decorator* ADV, which has the responsibility of defining links between the *DocumentView* ADV and the *EnhancedDocumentView* ADV.

6 Conclusions

We describe in this paper a formal approach to using "good" programming practice as embodied in design patterns. Design patterns are described using a process programming description which helps in clarifying the intent of each type of pattern. In addition, the patterns have been specified so as to incorporate the concept of objects (ADOs) and views (ADVs).

Our process-oriented approach to design patterns allows us to define primitive design pattern

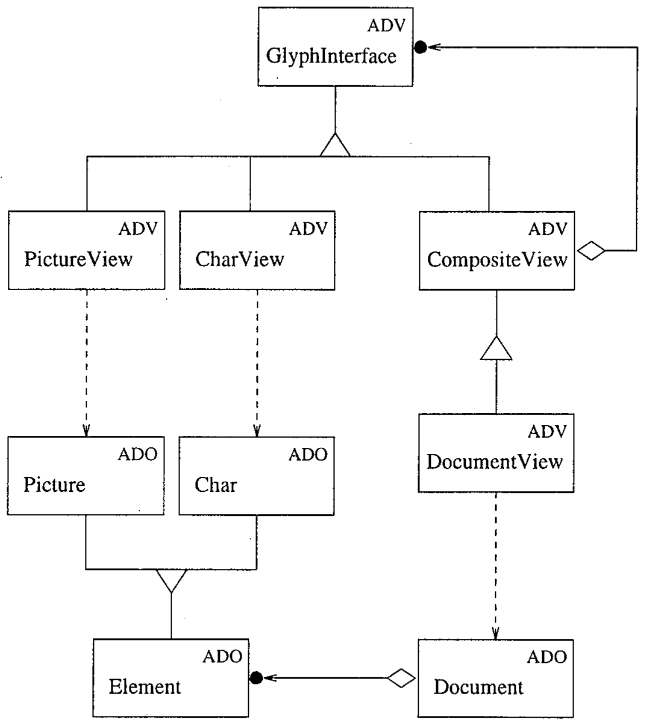


Figure 14: The document editor structure

```
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 15: 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
```

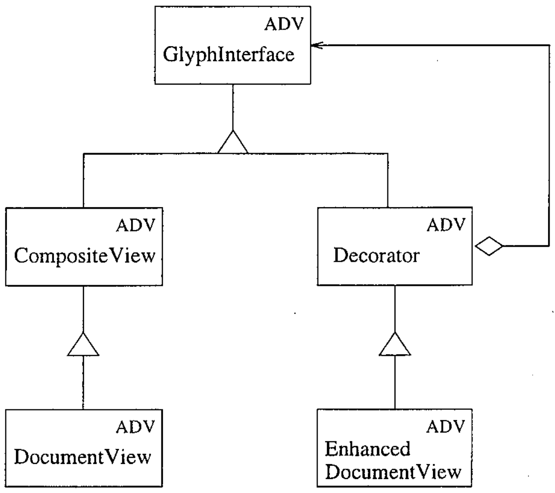
```
ADV GlyphInterface
    Declarations
        Attributes
            Size: nat;
            Child: ADV GlyphInterface;
        Effectual Actions
            Draw: coordinates;
            CreateChild: ADV GlyphInterface;
            RemoveChild: ADV GlyphInterface;
End GlyphInterface
```

```
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 16: ADVs forming the composite pattern



```
ADV Decorator
    Declarations
        Attributes
            Doc: ADV GlyphInterface;
        Effectual Actions
            GlyphInterface.Draw: coordinates;
        Nested ADVs
            Compose Doc;
            Inherit GlyphInterface;
End Decorator


ADV EnhancedDocumentView
    Declarations
        Attributes
            ScrollPosition: position;
            Decorator.Doc: ADV GlyphInterface;
        Effectual Actions
            Decorator.GlyphInterface.Draw: coordinates;
            DrawBorder;
            DrawScroll: position;
        Nested ADVs
            Inherit Decorator;
    Dynamic Properties
        Behavior
            Idle ∧ Decorator.GlyphInterface.Draw → Started;
            Started ∧ Decorator.Doc.Draw
                    ∧ DrawBorder
                    ∧ DrawScroll → Idle;
End EnhancedDocumentView
```

Figure 17: The decorator design pattern

tasks or constructors that can be used in the development of specific instantiations. We believe that this approach clarifies both the application and structure of the design patterns and can help to categorize them in a more formal way.

Using this formal approach including objects and views directs us toward several important results. By using components we are able to reason about the design and prove formally specified properties as shown in [ACL95, BACL95]. 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 [BACL95] by serving as a basis for test case generation [Kor90], or by providing a means for measuring test coverage.

Experiments with the process program description has strongly indicated that design patterns can yield corresponding C++ schemas which can be completed by the designer through an interactive dialogue. We are completing our experiments with C++ code generation and have started work on a tool to fill in and generate the schemas.

The relationship of objects and views to subject-oriented programming [HO93] has been explored in [ACC95], where ADV classes, ADOs and ADVs could be interpreted as formal descriptions of subjects, objects and subject activations respectively. Thus, we believe we can link the reuse approach embodied in subject-oriented programming to the resue of designs as exemplified in design patterns.

7 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".

References

[ACC95] P.S.C. Alencar, D.D. Cowan, and Lucena C.J.P. Abstract Data Views as a Formal Approach to Subject-Oriented Programming. Technical Report CS-95-19, University of Waterloo, Waterloo, Ontario, Canada, May 1995.

[ACG+95] 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-95-29, University of Waterloo, Waterloo, Ontario, Canada, June 1995.

[ACL95] 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.

[CCL93] L. M. F. Carneiro, D. D. Cowan, and C. J. P. Lucena. ADVcharts: a Visual Formalism for Describing Abstract Data Views. Technical Report 93–20, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, 1993.

[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.

[CHB92] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[Che91] George W. Cherry. S-R Machines: a Visual Formalism for Reactive and Interactive Systems. *ACM Software Engineering Notes*, 16(3):52–55, July 1991.

[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.

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

[FHW93] B. Fields, M. Harrison, and P. Wright. From Informal Requirements to Agent-based Specification: an Aircraft Warnings Case Study. Technical report, University of York, August 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.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[KP88] 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 1988.

[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.

[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.