Annals of Software Engineering 13, 111–140, 2002

Views and Patterns in E-Commerce Application Design

MARCUS E. MARKIEWICZ and CARLOS J.P. LUCENA mem@acm.org; lucena@inf.puc-rio.br
Computer Science Department, Pontifical University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil 22453-900

PAULO S.C. ALENCAR and DONALD D. COWAN {palencar; dcowan}@csg.uwaterloo.ca
Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

Abstract. Separation of concerns is a well-established principle in software engineering that supports reuse by hiding complexity through abstraction mechanisms. The Abstract Design Views model was created with reuse in mind and allows the designer to apply separation of concerns in a software system from the design to the implementation. In this model, viewed objects represent the basic concern, i.e., the algorithms that provide the essential functionality relevant to an application domain, and viewer objects represent the special concerns related to other software issues, such as user interface presentation, synchronization, and timing. In this paper we use a reuse taxonomy to analyze and validate this model. Using this analysis and the properties of the relationship between viewer and viewed objects, called "views," we also indicate how to map the views-based designs into implementations based on design patterns that satisfy the views properties. Finally, we show how to apply the principles of our approach, using views and the design patterns, to design e-commerce applications.

Keywords: design, separation of concerns, reuse, views, object-oriented relationships, abstract design views, software engineering, design patterns

1. Introduction

Software reuse is the branch of software engineering concerned with using existing software artifacts during the construction of a new system. These artifacts include design elements, documentation, specification, source code and many other aspects involved in a software engineering project. It has been recognized that by introducing reusable artifacts at various levels of abstraction in the software development process, reuse is one approach capable of making substantial improvements in software productivity, development cost, and quality [Mili *et al.* 1995].

However, effective software reuse involves software that is reusable by construction, not by chance, and higher-levels of reuse are obtained with careful architectural and detailed design [Wentzel 1994]. As a reuse technique that has been introduced with this purpose in mind, separation of concerns is a well-established principle in software engineering that hides complexity by means of abstraction mechanisms [Fayad *et al.* 1999]. Separation of concerns can be used, for example, to separate the user interface from the application part of a system or act as an interface for distinct modules of a system. In both cases, reusability is a consequence of the separation of concerns developed in the

structure of the system. By preventing modules of the application from knowing about the surrounding environment, the approach eliminates a significant drawback to reuse.

However, the many facets of separation of concerns are still rarely used in the various phases of the software development lifecycle because of a lack of adequate principles, theories, processes, and tools to support consistent application of these concepts. There is a need for approaches including definitions, models and properties that clearly separate the various functional concerns addressed in a software system and help the developers realize their designs in terms of more detailed ones based, for example, on design patterns.

The Abstract Design Views model [Cowan and Lucena 1995; Alencar *et al.* 2001] was created with reuse in mind and allows the designer to separate the concerns in a software system and to retain this separation in the implementation phase. According to this model, viewed objects represent the basic concern, i.e., the algorithms that provide the essential functionality relevant to an application domain, and viewer objects represent the special concerns related to other software issues, such as user interface presentation, synchronization, and timing. The model also indicates how an object-oriented design can be separated into objects and their corresponding interfaces. In this model objects can be designed so that they are independent of their environment, because adaptation to the environment is the responsibility of the interface or view. Informal versions of the model have already been successfully applied to many operational and commercial software systems [Cowan and Lucena 1995].

In this paper we use a reuse taxonomy to analyze and validate this model. This taxonomy will allow us to characterize the Abstract Design View model in terms of its reusable artifacts and to show how these artifacts are abstracted, selected, specialized and integrated. Based on this analysis and the properties of the relationship between viewer and viewed objects, called "views," we also indicate how to map the designs into implementations based on design patterns that that satisfy the *views* properties. Finally, we show how to apply the principles of our approach on separation of concerns, using views and the design patterns, to design e-commerce applications.

2. Background

2.1. Separation of concerns

According to the Oxford English Dictionary, a concern is a relation of connection or active interest in an act or affair. Alternatively, Czarnecki *et al*. define a concern as a domain used as a decomposition criterion for a system or another domain with that concern [Czarnecki *et al.* 1996].

The term concern has different meanings across software engineering. In some of its connotations, a concern may refer to elements of design that cut across the basic functionality of the system. For instance, memory access patterns may be considered in some cases as one specific concern [Kiczales *et al.* 1997]. Other notions of concern might be related to more general concepts such as performance and quality. However, in this pa-

per we will use the same meaning of a concern as the one given by Fayad *et al*. [1999]. These authors mention that "current frameworks involve a basic concern and a number of special-purpose concerns. The basic concern is represented by [Alencar *et al.* 2001] algorithms that provide the essential functionality relevant to an application domain, and the special purpose concerns relate to other software issues, such as user interface presentation, control, timing, synchronization, distribution, and fault tolerance."

Different concerns can be identified during analysis, design, implementation, and refactoring. Objects with similar concerns are connected by a common interest in a particular domain of the problem description, which may be of structural, functional, or behavioral nature. Distinct concerns should be loosely coupled and as orthogonal as possible. While there are guidelines for separating concerns, the identification of the boundaries of a concern is still an arbitrary task. As Dijkstra states: "The crucial choice is, of course, what aspects to study in isolation, how to disentangle the original amorphous knot of obligations, constraints and goals into a set of concerns that admit a reasonably effective separation" [Dijkstra 1976].

A significant barrier to the reuse of both design and implementation of software objects and modules is the fact that they internalize knowledge about their surrounding environment. For example, a typical module or object in an application often knows about its user interface, specifically details of how its data structures will be displayed, how the user will interact with the application, or what objects on the screen correspond to activations of components of the module. Similarly, a module or object knows too much about the services required from other objects or modules. For example, a module will know too much about naming conventions in a file system, or about the names of modules or functions from which it acquires services. Such depth of specialized knowledge seems counter not only to reuse but to good engineering practice in general [Cowan and Lucena 1995; Fontoura *et al.* 2000].

2.2. The Abstract Design Views model

The Abstract Design Views model [Cowan and Lucena 1995; Alencar *et al.* 2001] was developed in an attempt to overcome the limitations inherent in the separation of concern models, which are based on specific program paradigms. This model is an object-oriented design model which bridges the gap between an internal world of application objects and its requirement for knowledge of the external world [Cowan and Lucena 1995]. The basic constructs of the model are the Abstract Design View (ADV) and the Abstract Design Object (ADO), which represent, respectively, interface objects (views and interactions) and application objects which are independent of the interface. The types of object support a disciplined model to design that attempts to separate concerns.

The separation of concerns introduced by the model divides the "world" into two types of objects. These types are the ADVs and ADOs, and they characterize the concern of an object in a software model as either interface or application. Although we can find many structural similarities in both object concepts, it is important to observe that there is a clear separation between capabilities of ADOs and ADVs. An ADO has no knowledge

of its surrounding environment, thus ensuring independence of the application from its interface. On the other hand, an ADV does know about its associated ADO and can query or modify the ADO's state by means of its public interface or a mapping between the ADV and related ADOs [Alencar *et al.* 1995].

Initially ADVs were used to capture the user interface concerns of interactive software systems. Later, the model was extended to general interfaces that could capture other external concerns such as a timer or a network. A further extension captured other special purpose concerns such as control, timing, and distribution.

ADVs have been used in various software system designs [Cowan and Lucena 1995; Alencar *et al.* 2001], e.g., to support user interfaces for games and a graph editor, to interconnect modules in a user interface design system (UIDS), to support concurrency in a cooperative drawing tool, to design and implement both a ray-tracer in a distributed environment, and to design a scientific visualization system for the Riemann problem. A research prototype of the VX-REXX system [Watcom VX-REXX 1993] was motivated by the idea of composing applications in the ADV/ADO style. In addition, we have shown how ADVs can be used to compose complex applications from simpler ones [Cowan and Lucena 1995] in a style which is similar to some approaches to component-oriented software development and megaprogramming [Wiederhold *et al.* 1992].

2.3. Properties of the views relationship

An Abstract Design Object (or ADO) is a software construct that has no direct contact with the "outside" world. ADOs are only accessible through one or more Abstract Design Views (or ADVs). ADVs are ADOs augmented to support the development of "views" of an ADO, where a view is a simple user interface or an alteration of the ADO's interface, a contract [Helm 1990]. The ADVs affect the ADOs by means of input events at the ADV, which are mapped in the ADO. However, the ADO has no knowledge of the existence of any ADV acting as its intermediary. Thus, by construction, a viewer object is aware of the public interface of the viewed object, however a viewed object should have no knowledge about any internal property of a viewer object. This way, the first property of this model is devised:

Property 1. *Visibility property* – An ADO is accessible through one or more ADVs. Thus, one or more ADVs have references to an ADO at any given time, but the related ADOs do not know about their associated ADVs.

The separation between views and objects allows us to associate many ADVs (viewer objects) to a single ADO (viewed object). In this case, as the state of an ADO changes, the ADVs connected to the ADO must be consistent with that change. Using morphisms or mappings defined between the ADV and the ADO, this invariant is expressed, as in [Alencar *et al.* 1995].

The consistency among ADVs is called *horizontal consistency*, while the consistency between ADOs and ADVs is called *vertical consistency*. The state of ADVs and

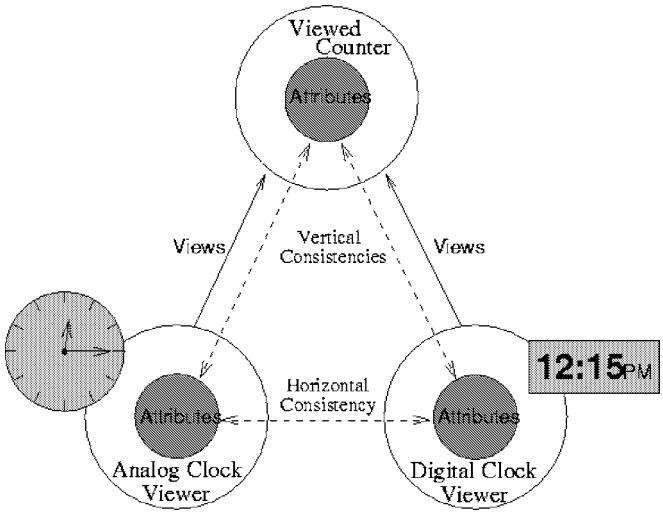

Figure 1. An ADV–ADO interaction model.

ADOs should be consistent at all times by means of a mapping mechanism. Thus, a change in the state of a viewed object will effect corresponding changes in the state of viewer objects related to that object. Therefore, we have a second and a third property of this model.

Property 2. *Horizontal consistency property* – Each ADV related to the same ADO must be consistent among themselves, reflecting any state change of the ADO in a consistent manner.

Property 3. *Vertical consistency property* – Each ADV related to an ADO must change its state consistently with the ADO's state change.

Like the visibility property, the vertical and horizontal consistency properties can be found in many papers, e.g., [Cowan and Lucena 1995; Alencar *et al.* 2001].

In figure 1 we have an ADV–ADO interaction model. In this figure the horizontal and vertical consistencies are illustrated, as well as how ADVs act as points of entry to ADOs. In figure 1, we have a counter that gives us the current time. However, we wish to provide two readings (or clocks) from this source: one digital and one analog. Using ADVs and ADOs, we are able to map these clocks to the source. Each clock is a view from the source, that is, each clock is an ADV, and the source is an ADO. Therefore, in this way we have two different clocks from a single source. Each clock must be consistent with the single source and the two different clocks must also be consistent with each other.

In the Abstract Design Views model, it is possible for ADVs and ADOs to be composed or aggregated. The enclosing ADV or ADO knows the identity of its constituents, but the contrary is not true. The details of the composition of objects are declared through morphisms [Alencar *et al.* 1995, 2001], specifying the relationships between the viewer and viewed objects. For example, in figure 2, we have a composite ADV. In this case,

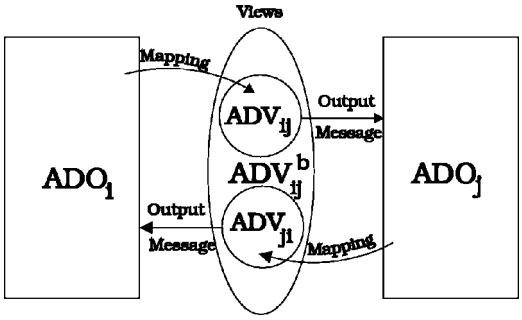

Figure 2. A Composite ADV.

the ADV_{ij} and ADV_{ji} are enclosed within the $\mathrm{ADV}^{\mathrm{b}}_{ij}$. Thus, we have a fourth property of the Abstract Design Views model.

Property 4. *Nesting property* – ADVs and ADOs can have nested objects, but only the enclosing objects have knowledge about the enclosed ones.

It is important to notice that so far only the visibility of ADOs relating to ADVs was discussed. When discussing the visibility of the ADVs themselves, there are two possible approaches: with or without transitivity. In the version without transitivity, ADVs only relate to ADOs. However, where transitivity is present, it is possible for ADVs to pose as "views" of "views," but it is still forbidden for ADVs to communicate in configurations other than this one. Thus, we have the transitivity property of the Abstract Design View model.

Property 5. *Transitivity property* – An ADV may have visibility to another ADV, posing as indirect viewers of an ADO, and direct viewers of another ADV.

The transitivity property has been introduced in a recent paper [Markiewicz *et al.* 2000], and was not described in previous articles. When two ADVs are connected to the same ADO, we consider that they are *direct* viewers of a same ADO. On the other hand, if an ADV is not directly referencing an ADO, but it is another ADV that does it, it is an *indirect* viewer of that ADO. For example, in figure 3, ADV_3 and ADV_4 are direct viewers of ADO_x, while ADV_2 and ADV_5 are indirect viewers. Even further, ADV_5 is both a direct viewer of ADV_2 and ADV_4 and an indirect viewer of ADO_x.

2.4. Roles of ADVs and ADOs

It is possible, considering all the properties shown above, to classify ADVs architectures according to their roles. In prior works on the Abstract Design Views model, the authors considered only two types (or roles) of ADVs: an ADV that acts as an interface between two media, and an ADV that acts as an interface between two ADOs operating in the same medium [Cowan and Lucena 1995; Alencar *et al.* 1996]. Particularly, when an ADV is used as an interface between two such media it can represent a user interface, a

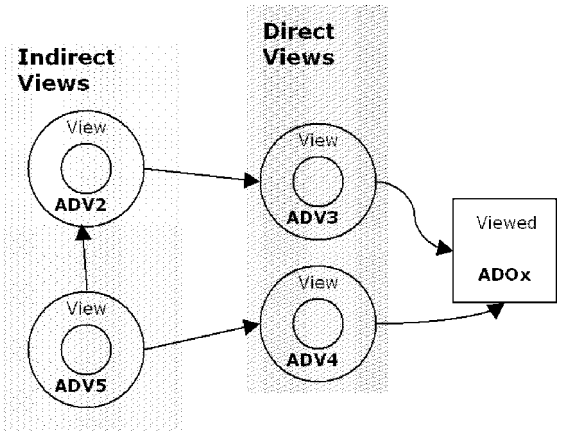

Figure 3. Direct and indirect viewers of an ADO.

network interface, or an external device such as a timer. In this paper we extend these two roles to four possible roles.

According to the visibility property, an ADO can be referenced by more than one ADV at one time. Even further, it is possible to connect two ADOs by a single ADV. This way, ADVs can change roles from simple viewers to full-fledged interfaces between two different media.

This communication can be performed in a unidirectional or bidirectional manner. If unidirectional, the ADV will map actions directly into other ADOs. If bidirectional, each ADV will map the actions into the other ADO, performing a duplex communication mapping between the ADOs, such as ADV^{b}_{ij} in figure 2. It is important to notice that the interface ADV here can be used to perform translations and adaptations to convert one ADO's output to the format of the other ADO's input. The roles of ADOs and ADVs are explained below.

2.4.1. View of an ADO

According to the visibility property, the ADVs of an ADO are its points of entry. An ADV observes an ADO, mapping the inputs onto the ADO, and relaying the output to the user of the ADV. This architecture is shown in figure 4.

It is important to notice that it is possible to have a single composite ADV related to an ADO. This way, many views can be related to an ADO, but there is only a single point of entry for the ADO. This alternative is thoroughly discussed in [Markiewicz *et al.* 2000] for access control purposes.

2.4.2. Unidirectional interface of two media

Another possible role for an ADV is to connect two ADOs, serving as an unidirectional interface. In this role, the ADV serves as a mapper of actions to an ADO, possibly introducing some rationale in this process. Thus, it is possible to translate the output of an ADO to the format of the input of the other ADO. In figure 5 here is a representation of this architecture.

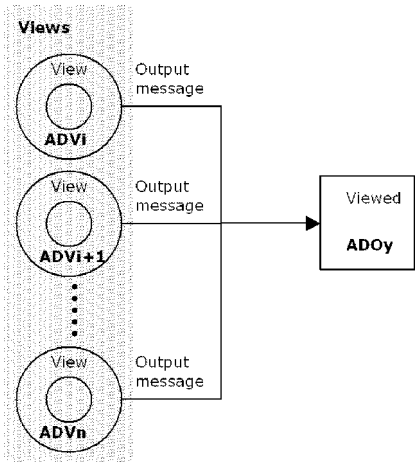

Figure 4. ADVs acting as viewers for an ADO.

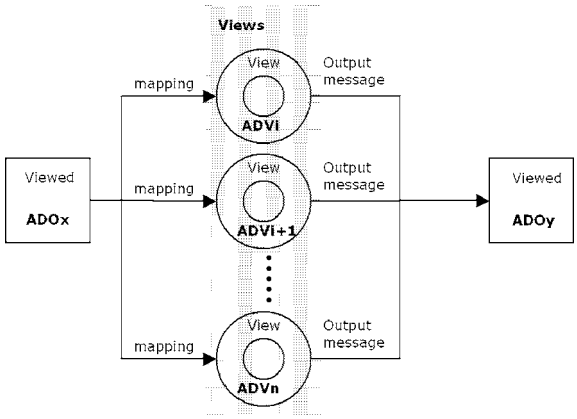

Figure 5. Two ADVs serving as unidirectional interfaces for two ADOs.

2.4.3. Bidirectional interface of two media

Extending the previous role, it is possible that ADVs provide bidirectional communication between two ADOs. In this fashion, ADVs will assume a full-fledged "glue" role between different ADOs, making proper translations and adaptations of the inputs and outputs of the ADOs. This enables designers to compose the ADOs and make them collaborate without altering their original code. This architecture is represented in figure 6.

2.4.4. Facilitator of n-media

Finally, it is possible to extrapolate the previous roles using multiple ADOs and ADVs. ADVs might receive the input of many ADOs and translate that for many ADOs. ADVs this way are turned into shared communication devices, becoming the rendezvous point of many ADOs. In figure 7 we have an ADV_i that is the facilitator of ADOs x, y, z and w.

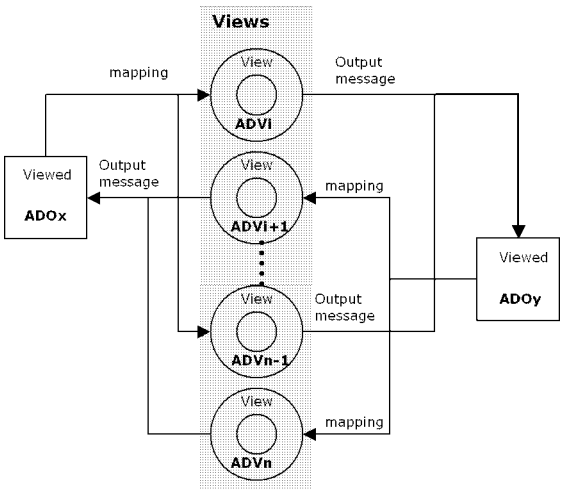

Figure 6. ADVs serving as bidirectional interfaces for two ADOs.

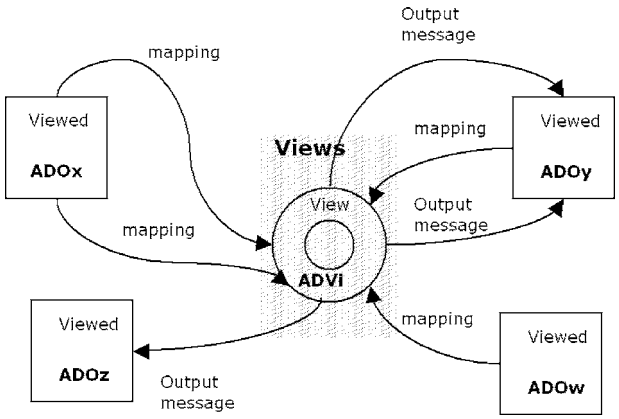

Figure 7. An ADV as a facilitator of n-media.

3. Analyzing the abstract design view model

Using the taxonomy introduced by [Krueger 1992], it is possible to analyze and validate the reuse aspects of the Abstract Design Views model. The taxonomy is based on four dimensions: abstraction, selection, specialization and integration. We will also discuss the cognitive distance [Krueger 1992] of ADVs and ADOs from the software system designer's point of view.

3.1. Abstraction

Every software component is created having some form of abstraction in mind. This abstraction allows component (re-)users to determine the role of every artifact without having to examine its code. Even further, the abstraction allows an easier understanding of

complex components, suppressing irrelevant details. Examples for such abstraction are lists, stacks, trees and other data structures. Computer scientists are able to understand these concepts without having to read every line of code written in their implementation. The Abstract Design Views model introduces the concept of the "view" of an object, allowing the interpretation of an object at a higher abstraction level. This abstraction level is therefore closer to the designer, bridging the cognitive distance.

For example, we presented in figure 1 a counter that gives us the current time. Two readings (or clocks) from this single source: one digital and one analog. Using ADVs and ADOs, we were able to map these clocks to the source. Each clock was represented as a viewer object, i.e., an ADV, of the single source, and the source was represented as an ADO. Therefore, it is possible for the designer to grasp the relationship of consistency between the clocks and the counter without having to grasp details such as shared buffers, message passing or references. These details can be dealt with at a later step, when the ADVs and ADOs are realized. By being able to postpone dealing with these details, the designer is spared from implementation issues that might have been considered beforehand, influencing the design.

3.2. Selection

It is very important for the (re-)user to be able to distinguish components, by browsing and searching through them. By classifying and cataloging components, it is possible to organize a library of reusable artifacts. Thus, the use of each component in this library must be clear and well specified. Otherwise, misuse and improper adaptation of components will surely follow.

3.3. Specialization

Many reuse technologies use generic artifacts that are instantiated by parameters or even inheritance. These artifacts are refined before use, allowing the reuse of the generic artifact in many solutions. In our case, the ADV and ADO specifications can be reused and combined using mechanisms such as composition, inheritance, sets and sequences. This way, ADVs can be specialized or incrementally changed over time. It is important to notice that these relationships are reflected in the ADV's formal specification. Thus, changes and alterations are not introduced in a complete ad hoc manner, requiring some thought and analysis of the alterations.

3.4. Integration

Once components already exist or are being created, it becomes essential to combine these components. This way, complex constructs are made possible through the union of smaller and simpler artifacts. According to the visibility property of the Abstract Design Views model, it is possible to "glue" ADOs or modules using ADVs as interfaces. Thus, ADVs can serve as integration constructs, assembling large and complex architectures from simple ones.

Table 1
Reuse in the Abstract Design Views model.

Abstraction	The Abstract Design Views model introduces the concept of the view of an object, allowing the interpretation of an object in a higher abstraction level.
Selection	The artifacts of this model are ADVs and ADOs. Each class in this model is realized from ADVs or ADOs, so the re-user can always distinguish the view from the application code, and how it reflects on the implementation.
Specialization	ADVs can be combined and reused using mechanisms such as sequences, sets, compositions and inheritance. ADVs can be specialized or incremented over time.
Integration	ADVs can be nested, and ADVs can act as interconnections between ADOs. ADVs can be interconnected by these "glue" components.
Pros	The Abstract Design Views model allows different views, separating concerns and allowing reuse of views throughout the design process. Since these views are at a high level of abstraction, the cognitive distance between the end-user of the software and the design level is minimal.
Cons	The ADVs must be chosen in a way to be semantically coherent with the ADOs, or otherwise their contract might be misused or misunderstood.

3.5. Results of the analysis

In table 1 we present the overall picture of reuse in the Abstract Design Views model. The major advantage of the Abstract Design Views model is the small cognitive distance between the designer and the model, since the abstract level of the ADVs and ADOs constructs are very high. On the other hand, the Abstract Design Views model creates space for loose semantics. One can create ADVs for an ADO that do not relate coherently. This way, the resulting components can be misused or misunderstood.

4. Realizing Abstract Design Views

Once the reuse of ADVs at the design level is clear, it is necessary to investigate if these artifacts also promote reuse at the implementation level. Since Abstract Design Views are considered to be at a higher design level than the implementation, their realization is possible through several distinct forms that will be discussed in this section.

4.1. Realization using design patterns

One possible approach for realizing ADVs and ADOs is through design patterns [Gamma *et al.* 1995]. However, many design patterns can be used for that purpose and their selection can be based on an ADV's particular aspects and properties. All diagrams in this section are in UML [UML 1999].

4.1.1. Using the Observer design pattern

The most obvious approach to realize ADVs and ADOs is the Observer design pattern [Gamma *et al.* 1995]. The main objective of this pattern is to define a one-to-many

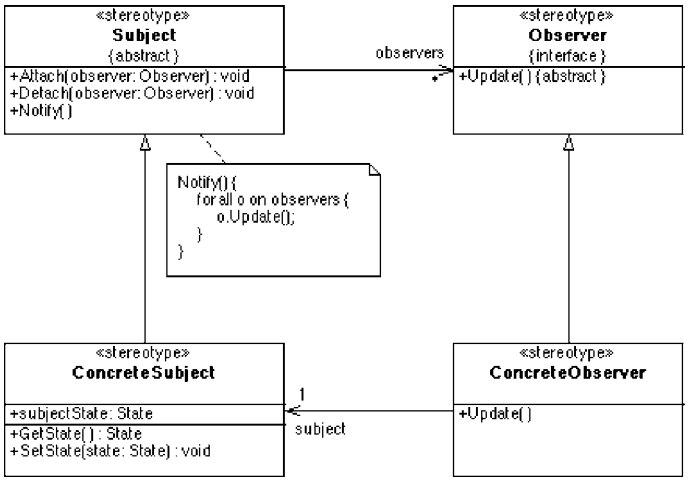

Figure 8. The Observer design pattern.

dependency between objects, so that the dependent objects can monitor changes in one object. In this realization, represented in figure 8, ADVs correspond to the observers, while the ADOs are the subjects.

The ADO base class has references to the ADVs, and invokes the `Notify()` method every time it suffers any state change. This way, ADVs are promptly warned of the ADO's state change.

4.1.2. The Callback versus Polling tradeoff

According to the vertical consistency property, ADVs must keep their state consistent with every state change of the ADOs. This means that if an ADO changes its state, the ADV must somehow notice that change.

The Observer pattern allows the ADV to be updated in the case of state changes by the ADO. This way, the ADO has a list of all objects that need to be warned about its state changes. However, according to the visibility property, no ADO should be able to determine that any ADVs exist.

Thus, the use of the Observer pattern might break the visibility property. However, it is argued that the ADO has only the knowledge that there *may be something* monitoring its internal state that must be notified of a change [Cowan and Lucena 1995]. In this line of reasoning, the ADO has no explicit knowledge of any particular ADV object, hence satisfying the visibility property and so the separation of concerns requirement.

The Observer pattern represents nothing else than a Callback as a solution for the prompt update of the ADV state. This approach is recommended for user interfaces for its timely action. On the other hand, a different approach is for the ADV to poll the ADO for any state change. This polling can take place, for example, every time an action takes place at the ADV and it maps it onto the ADO. This approach has less run-time overhead, but might display incorrect views for large periods. The dynamics of this process are shown in figure 9. In the Callback, the `Notify()` action taken by the

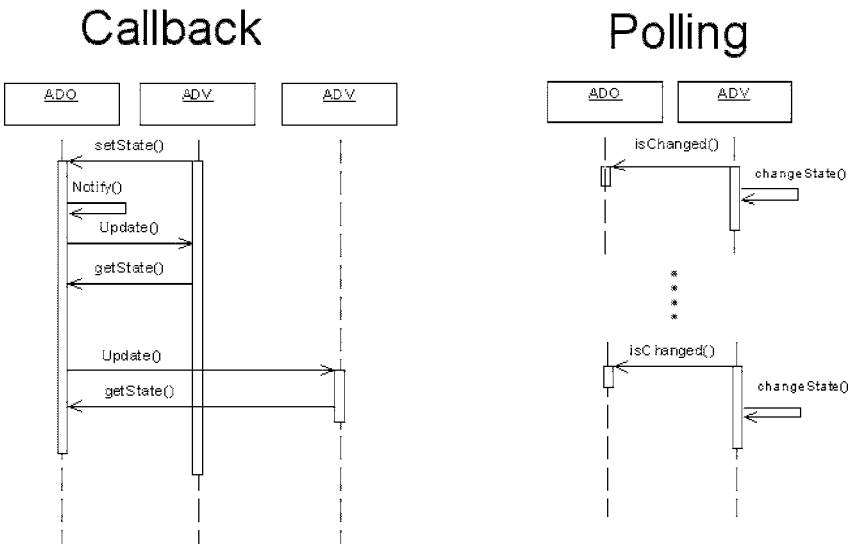

Figure 9. Callback versus Polling dynamics.

ADO will cause all ADVs to be updated (`Update()` method with `getState()`). With Polling, once in a while the ADV will query the ADO's state (`isChanged()` method), and if any change has happened, it will change its state (`changeState()` method).

Therefore, it is important to consider the Polling versus Callback solution before implementing the vertical consistency of ADVs and ADOs. One must trade run-time overhead versus timely update.

4.1.3. Using the Proxy design pattern

The Proxy design pattern [Gamma *et al.* 1995] can also be used to realize ADVs and ADOs. In this realization, the RealSubject is the ADO, and the Proxy the ADV. This is represented in figure 10.

It is important to notice that the use of this pattern allows for the ADV and ADO to be bound to a contract. This particular feature is very useful, as ADVs can be given as references to ADOs seamlessly since ADVs and ADOs are derived from a common ancestor. Thus, by belonging to the same inheritance hierarchy, ADVs can act as an ADO's interface without any overhead.

4.1.4. Using the Adapter design pattern

Another possible design pattern that can realize ADVs and ADOs is the Adapter design pattern [Gamma *et al.* 1995]. In this realization, the client ADO is the target object, the component ADO the adaptee, and the view (ADV) the adapter. It is represented in figure 11.

As in the Proxy design pattern, this approach binds the ADV and ADO to the same contract. However, in this case this is achieved by having the ADV inherit the ADO interface.

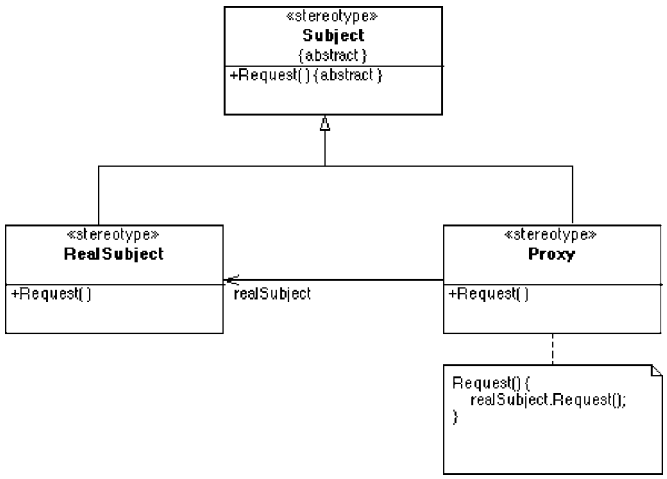

Figure 10. The Proxy design pattern.

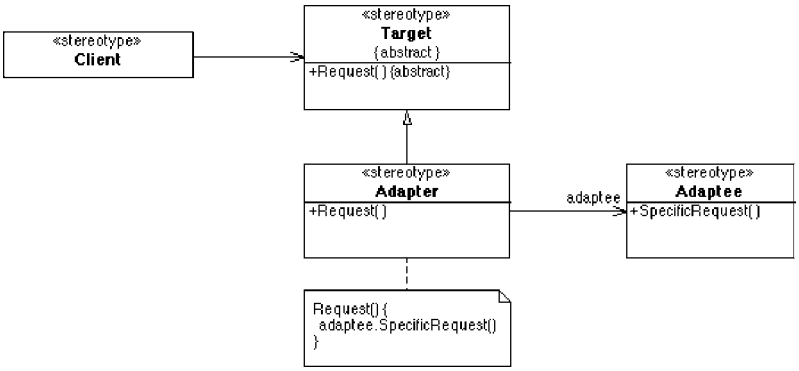

Figure 11. The Adapter design pattern.

4.1.5. Using the Façade design pattern

In the case of the facilitator of n-media shown above, it is possible for an ADV to provide a custom interface (or contract) based on the "clipping" of many ADO interfaces (or contracts). By providing this custom interface, the ADV is encapsulating all the ADO's services, decoupling them from their users. This particular application of the n-media facilitator can be realized using the Façade design pattern [Gamma *et al.* 1995]. This design pattern is represented in figure 12.

For this realization, the ADV would be the Façade class, while all the other classes in the encapsulated module would be the ADOs. It is interesting that in this case the ADV is also serving as "glue" to the ADOs, binding them by providing a unified contract or interface that is the front-end of all the ADOs.

Another important issue is that the façade only allows unidirectional communication between itself and the ADOs. In this pattern, the façade only forwards calls to the ADOs.

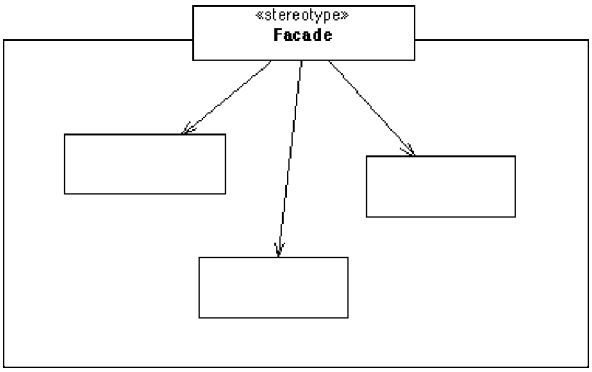

Figure 12. The Façade design pattern.

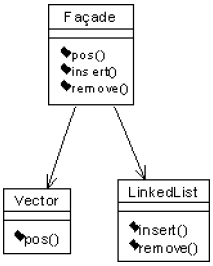

Figure 13. Semantically incoherent façade.

In using this particular realization, one must be cautious about binding incompatible classes in a single interface. If the bound classes have irreconcilable semantic differences, the resulting façade might be counter-productive.

For example, in figure 13 we have two classes. The Vector class implements a vector with a method that returns the value of the *n*th element, passed as a parameter. The LinkedList class allows the insertion and removal of an element. It inserts at the end of the list, and removes an element by receiving its value as a parameter.

By creating a façade for these two classes, we have a semantically incoherent façade, as it can be misused through a misunderstanding. One can assume that the component is itself a structure that has the three services (pos, insert and remove), and not that it is two separate data structures.

4.1.6. Using the Pipes and Filters design pattern

The Pipes and Filters design pattern [Buschmann *et al.* 1996] can be used to realize ADVs and ADOs. The Pipes and Filters pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems [Buschmann *et al.* 1996]. In this realization, represented in figure 14, the ADOs are the data source and data sink, as the ADVs are the filter objects.

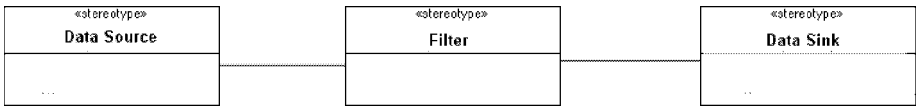

Figure 14. The pipes and filters design pattern.

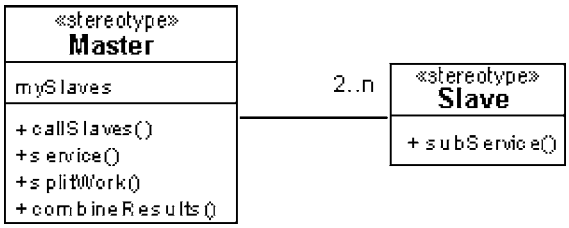

Figure 15. The Master–Slave design pattern.

Unlike the Proxy pattern, this pattern does not bind the ADVs and ADOs to an interface. The ADV can have any desired interface, thus allowing for the use of ADVs without complying with the ADO's interface. This grants re-users the possibility of introducing old code to work with the ADOs without having to tamper with it.

4.1.7. Using the Master–Slave design pattern

The Master–Slave design pattern [Buschmann *et al.* 1996] can also be used to realize ADVs and ADOs. The Master–Slave pattern supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a final result from the results returned by these slaves [Buschmann *et al.* 1996]. In this design pattern, ADVs are the master objects, while the ADOs are the slaves. In figure 15 we have the class diagram of this pattern.

Like the Pipes and Filters design pattern, this realization allows ADOs to have a different interface from the ADVs. However, it is mandatory that all ADOs have a method (`subService()`), that can receive the same number and type of parameters. In this realization there is a single ADV for many ADOs (more than two). Thus, its use is limited.

It is important to notice that in this realization only the ADVs have references to the ADO, and since the master object is simply an object that forwards messages, being "stateless," the Callback versus Polling tradeoff does not take place. If not "stateless," the master object allows for the division of the work between different ADOs.

4.1.8. Using the Blackboard design pattern

The Blackboard design pattern [Buschmann *et al.* 1996] can also be used to realize ADVs and ADOs. The Blackboard pattern is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized components assemble their knowledge to build a partial or approximate solution [Buschmann *et al.* 1996]. In this realization, represented in figure 16, the ADOs and ADVs do not map directly into the pattern objects.

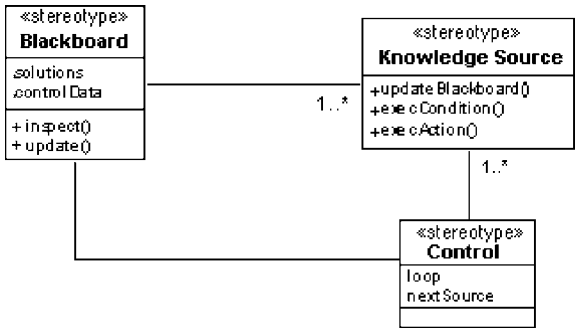

Figure 16. The blackboard design pattern.

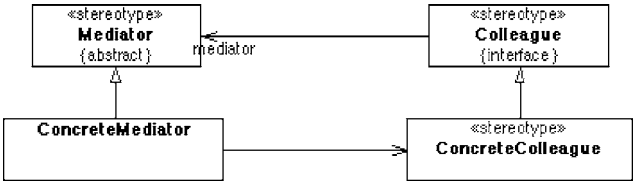

Figure 17. The Mediator design pattern.

In this case, the blackboard poses as the ADV, acting as a facilitator of n-media. Each ADO will contribute to the blackboard, and it will represent a unique shared space, a single view that is the collaboration of all ADOs.

4.1.9. Using the Mediator design pattern

Another possible design pattern that can be used to realize ADVs and ADOs is the mediator design pattern [Gamma *et al.* 1995]. In this realization, the ADVs are represented as the mediator objects, and the ADOs as the colleague derivations. This pattern is represented in figure 17.

It is important to notice that in this pattern the ADOs will have references to the ADOs, being susceptible to the Callback versus Polling tradeoff. It also must be noticed that this design pattern only applies to realizations where there is one ADV to one or more ADOs.

4.1.10. Using the View Handler design pattern

The View Handler design pattern [Buschmann *et al.* 1996] is another design pattern that can be used to realize ADVs and ADOs. The View Handler pattern helps to manage all views that a software system provides. A view handler component allows clients to open, manipulate and dispose of views. It also coordinates dependencies between views and organizes their update [Buschmann *et al.* 1996]. This pattern is represented in figure 18.

In this realization approach, the ADOs are mapped to the supplier objects, and the ADVs to the specific views. It is important to notice that in the original pattern there is a one-to-one relationship between suppliers and specific views. However, shared suppliers

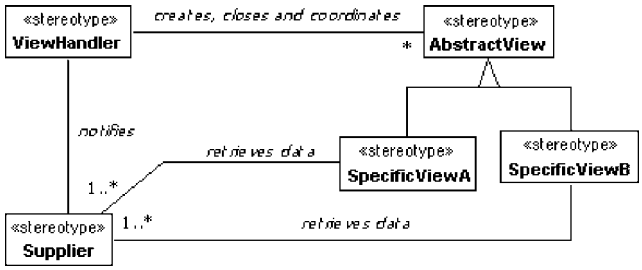

Figure 18. The View Handler design pattern.

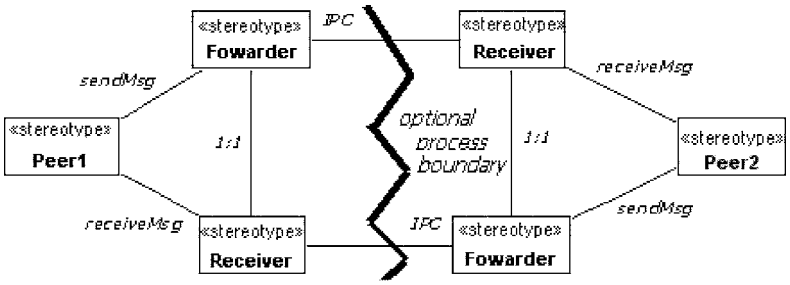

Figure 19. The Forwarder–Receiver design pattern.

(many ADOs to one or many ADV) or shared views (many ADVs to one or many ADO) can be used to accommodate all possible cardinality relationships between ADVs and ADOs. The uniqueness of this approach is the ViewHandler object. This object will act as a director or manager of the ADOs and ADVs, becoming a single point of entry to the entire component.

On the other hand, the presence of an object that has knowledge of both ADVs and ADOs can pose as an inconsistency to the visibility property. Thus, this design pattern should be used with caution so that the ViewHandler is not used outside the "spirit" of the visibility property of the model. We do not consider the ViewHandler to be an ADV or an ADO because, otherwise, the visibility property would be explicitly broken.

4.1.11. Using the Forwarder–Receiver design pattern

The Forwarder–Receiver pattern provides transparent inter-process communication for software systems with a peer-to-peer interaction model. It introduces forwarders and receivers to de-couple peers from the underlying communication mechanisms [Buschmann *et al.* 1996]. This pattern is represented in figure 19.

In this realization, the ADVs are the forwarder and receiver objects, and the ADOs the peer objects. This design pattern allows the realization of the interface of the media role of ADVs and ADOs. In this case, the forwarders and receivers objects will provide the bidirectional communication between the two ADO objects.

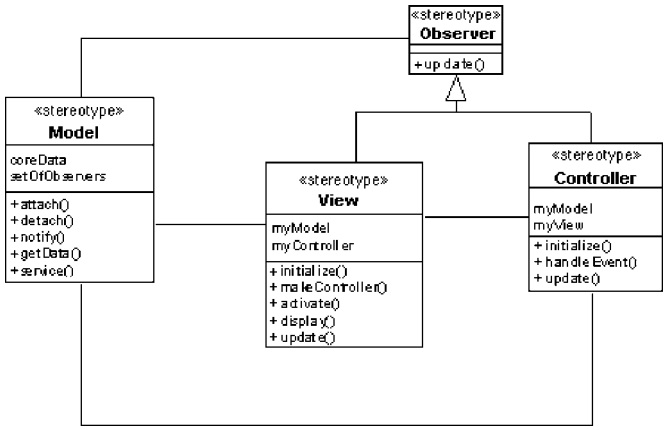

Figure 20. The Model–View–Controller design pattern.

4.1.12. Using the Model–View–Controller design pattern

The Model–View–Controller pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model [Buschmann *et al.* 1996]. This pattern is in figure 20.

Although this pattern guarantees all the ADO and ADV properties, a new artifact is introduced. The ADVs are the view objects, and the ADOs the models. However, the controller is like the ViewHandler object of the View Handler pattern. This way, a new element is introduced that is neither an ADV nor an ADO. We do not consider the Controller to be an ADV or an ADO because, otherwise, the visibility property would be directly broken.

4.2. Composition

It is possible to realize ADVs and ADOs using not only the design patterns shown above, but also combinations of them. For example, the Forwarder–Receiver design pattern can be combined with the Observer design pattern, creating a bidirectional channel of communication for objects.

The composition approach often will be present, and should be used carefully. The use of complex realizations can introduce both a maintenance and run-time overhead. Tangled solutions will be harder to understand and maintain, and will cause unnecessary run-time delays. In our experience, compositions should be avoided, used only when necessary.

4.3. Summing up

In the previous sections we have presented eleven design patterns that can be used to realize ADV and ADO relationships. Since the Abstract Design Views model resides on

Table 2
Possible design pattern realizations for ADVs.

Design patterns	Pros	Cons	Comments
Observer	Prompt refresh of the ADVs keeps vertical consistency at all times.	Profusion of references of ADOs to ADVs causes run-time overhead.	ADVs must keep vertical consistency valid and refreshed constantly. Recommended for user interfaces.
Proxy	Low run-time overhead approach.	Inconsistency might be apparent due to slow refresh synchronization of ADOs and ADVs.	Recommended for uses when vertical consistency needs check with a lower frequency.
Adapter	Low run-time overhead approach.	This approach binds the ADV and ADO to the same contract.	Recommended for uses when vertical consistency needs check with a lower frequency.
Façade	Creates a unified point of entry for a group of classes or component.	Allows semantic binding of incompatible classes.	Recommended for creation of components and de-coupling of sub-systems.
Pipes and Filters	ADVs are not bound to the ADO contract. Easy composition of ADOs with ADVs as simple filters.	ADVs are not bound to the ADO contract	Good solution where simple input-process-output is needed.
Master–Slave	Allows combination of ADOs by splitting work and combining it later.	Enforces a common method signature for all ADOs. Only makes sense for 1-to-n ADV to ADO cardinalities.	Should be used when the ADVs are 'similar' views of an ADO, having common functionality and points of entry.
Blackboard	Allows collaboration that can change dynamically and without a clear set of rules.	Synchronization problems might be present.	Recommended for realization of facilitators of n-media.
Mediator	Simple realization.	None.	Horizontal and vertical consistency properties are not necessarily enforced by this pattern.
View Handler	Different approach to realizing ADVs and ADOs.	A tertiary element is introduced. The visibility property is endangered.	None.
Forwarder–Receiver	Direct implementation of bidirectional interface of two media.	Difficult to apply to different ADV cardinalities.	Recommended for realization of bidirectional interfaces of two media.
Model–View–Controller	Long established pattern.	A tertiary element is introduced. The visibility property is endangered.	None.

Table 3
Realizations and ADV properties.

Design pattern	Properties					Roles of ADVs and ADOs			
	1	2	3	4	5	Views	Uni-int	Bi-int	n-media
Observer	✓		✓	✓	✓	✓	✓		
Proxy	✓			✓	✓	✓	✓		
Adapter	✓			✓	✓	✓			
Façade	✓			✓	✓	✓	✓		
Pipes and Filters	✓			✓	✓	✓	✓		
Master–Slave	✓	✓	✓	✓	✓	✓	✓	✓	✓
Blackboard	✓			✓		✓	✓	✓	✓
Mediator				✓	✓	✓	✓	✓	
View Handler		✓		✓	✓	✓	✓		
Forwarder–Receiver	✓	✓	✓	✓	✓	✓	✓	✓	✓
Model–View–Controller	✓	✓	✓	✓	✓	✓	✓	✓	

a higher abstraction level than classes and objects, its realization through design patterns is not by any means a direct or simple mapping.

The process of translating ADVs and ADOs onto classes must be guided by the semantic meanings attributed to them, thus introducing a human element that cannot be fully controlled or automated. In order to make this point clearer, in table 2 we have the pros, cons and comments on each possible realization. It is important to notice that the possibilities presented here are not exhaustive, and the composition of patterns will happen often.

Another issue is the relationship between the realizations, properties and roles of ADVs and ADOs. Many realizations will not enforce Abstract Design Views properties, but none of these break the properties. This is shown in table 3. For example, the Proxy pattern will not enforce the horizontal consistency property, leaving it to be checked externally.

5. An e-commerce system

For the purpose of illustrating the concepts introduced in this paper, we will show the process of implementation of ADVs and ADOs with design patterns in an e-commerce application.

In figure 21 we have modeled an e-commerce system, from payment to shopping cart systems using ADVs and ADOs. After the system is modeled in this fashion, the next step is to mark the ADVs of ADOs as different groups. By this we mean marking the components of the e-commerce system, by separating viewed and viewed subunits of the whole diagram.

In the example shown in figure 21, there are six components: Payment, Database, CheckOut, Inventory, Catalog and Browsing. The Payment component is responsible for the sensitive information needed to buy products. The Database component provides consistency in order to log all purchases. The CheckOut component models the process

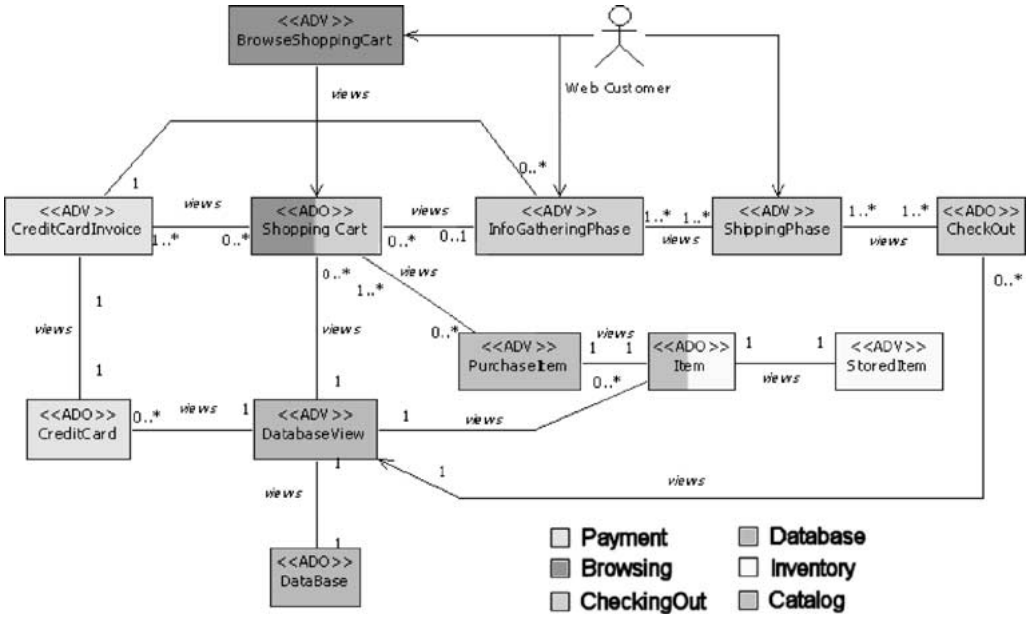

Figure 21. Modeling an e-commerce system using ADVs.

of purchasing and the steps necessary to achieve this purpose. The Inventory component deals with the information and processes needed for the maintenance of the items being sold, its characteristics and status. Finally, the Catalog component is the front-end of all products to the consumer. Each of these systems must be realized using the design patterns shown in the previous sections.

It is important to notice that some ADVs belong to more than one ADO, thus requiring a merge.

5.1. The Payment component

In this component, the credit card class has some of its information hidden in order to ensure security. The shopping cart must only be aware of some credit card details. Thus, there must be an access control object that will act as a view of this class. For this realization the pattern proxy is suitable, since the ADV is only a restricted view of the ADO, only clipping some of its functionality, but adding no extra calculations or actions.

For this reason, the Proxy pattern is introduced, and a common interface class, PaymentInterface, is also created. The result can be seen in figure 22.

In this case there is no need to update the CreditCardInvoice object, since it is "stateless" in the sense that it only forwards calls to the CreditCard object. This way, CreditCardInvoice will not perform any duties other than serve as a restricted functionality proxy.

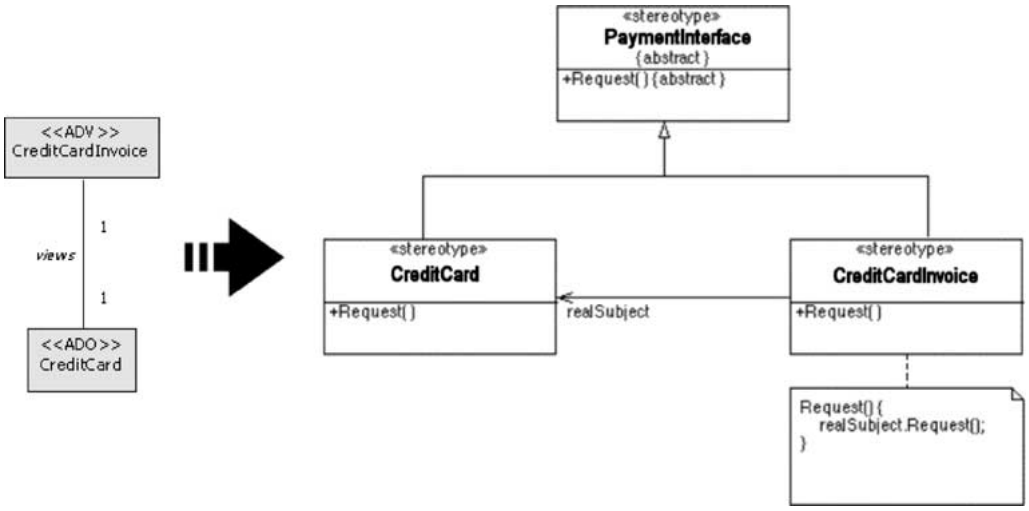

Figure 22. Realizing the payment component.

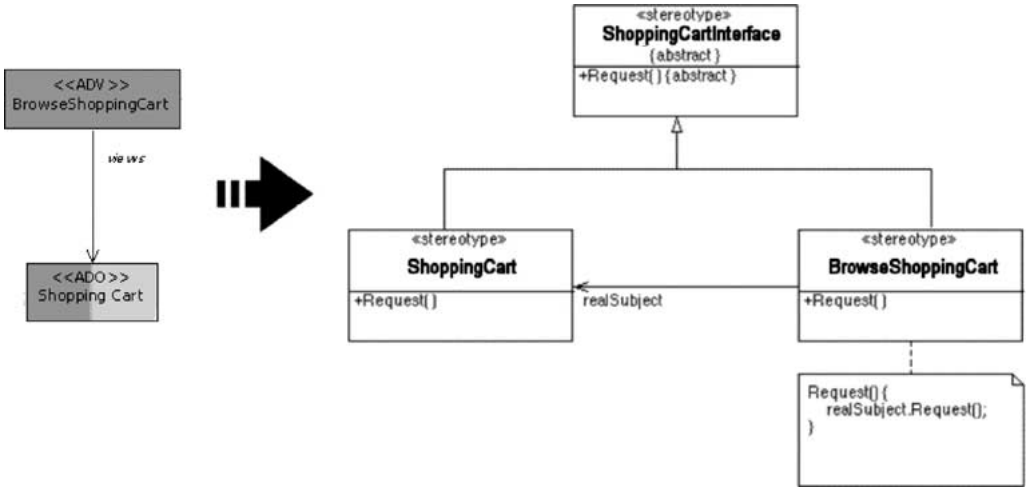

Figure 23. Realizing the browsing component.

5.2. The Browsing component

The Browsing component provides a view of the shopping carts, being the point of entry of the customers to the e-commerce system. Much like the Payment component, the shopping cart is visible to the end user in a restricted sense since there is no direct access regarding the credit card number. Since it is a restriction of access functionality, once more the Proxy pattern is appropriate. The resulting realization is shown in figure 23.

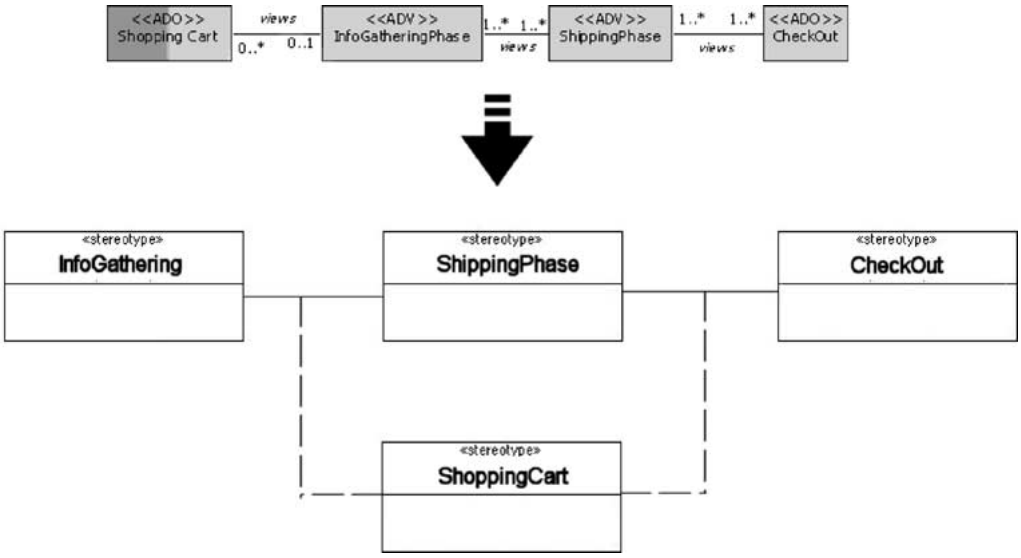

Figure 24. Realizing the CheckOut component.

5.3. The CheckOut component

The CheckOut component models the interaction with the consumer in the buying process after the items to be purchased were chosen. In this case, the ShoppingCart object will act as an association class, being handled by the InfoGathering, ShippingPhase and CheckOut objects. Each of these will receive a ShoppingCart object in one state and release it in another. This way, by the end of these transitions the purchase process will come to its end.

Since this process takes place with layers that perform alterations to its input and have no knowledge of the overall procedure, it is possible to realize it using a Pipes and Filters design pattern. This process is realized in figure 24.

5.4. The Database component

This component will serve as an access layer to the Database Management System (DBMS) program, which is typically a commercial relational database system. One possible realization for this component is the Façade design pattern, as shown in figure 25.

It is important to notice that in this case the ADO Database was not mapped into any direct class, since it is actually the DBMS.

5.5. The Inventory component

This component models the use and maintenance of the item for sale. Since it must be reliable, and shall be used to see the real inventory available at any given time, there must be a prompt update of the state of the items. For this reason, an observer pattern

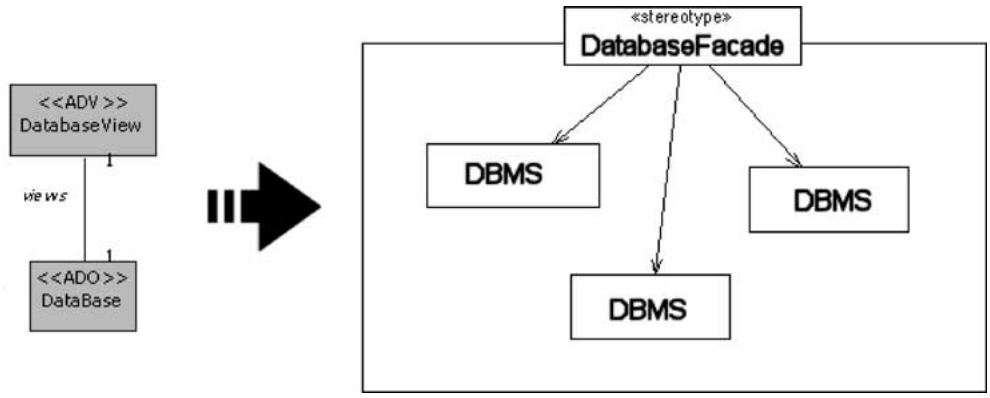

Figure 25. Realizing the database component.

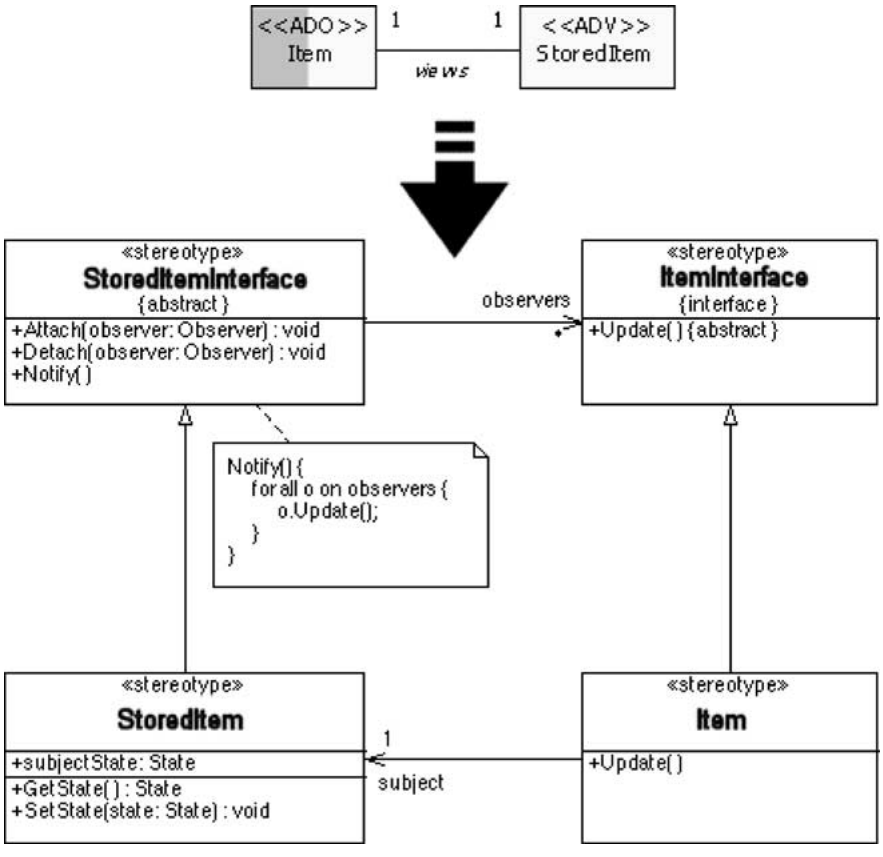

Figure 26. Realizing the Inventory component.

must be introduced, making sure that the inventory is always consistent with the "real" status of the inventory. This realization can be seen in figure 26.

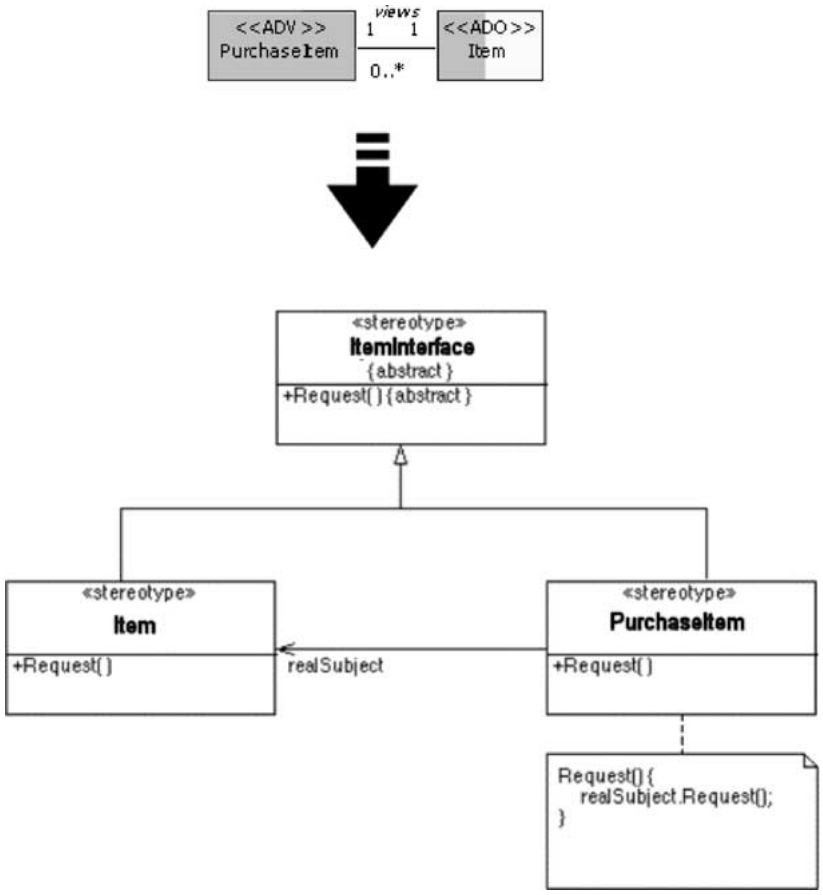

Figure 27. Realizing the Catalog component.

5.6. The Catalog component

Much like the Payment and Browsing components, in this case the ADV acts as a simple proxy of the products, showing only some of its functionality. Thus, the use of a Proxy pattern is appropriate, as is shown in figure 27.

5.7. Integrating the realizations

Once all realizations are completed, it is clear that there are redundant objects, like the ShoppingCart object, for instance. Moreover, there are interconnections between the many components not yet considered.

Thus, the next step is to integrate the components. In the cases where there is name collision (ShoppingCart) or even similar behaviors (Item class of both the Inventory and Catalog components), it is necessary to apply refactoring. One can use the refactorings presented by Fowler *et al.* [1999] in order to achieve a unified class diagram.

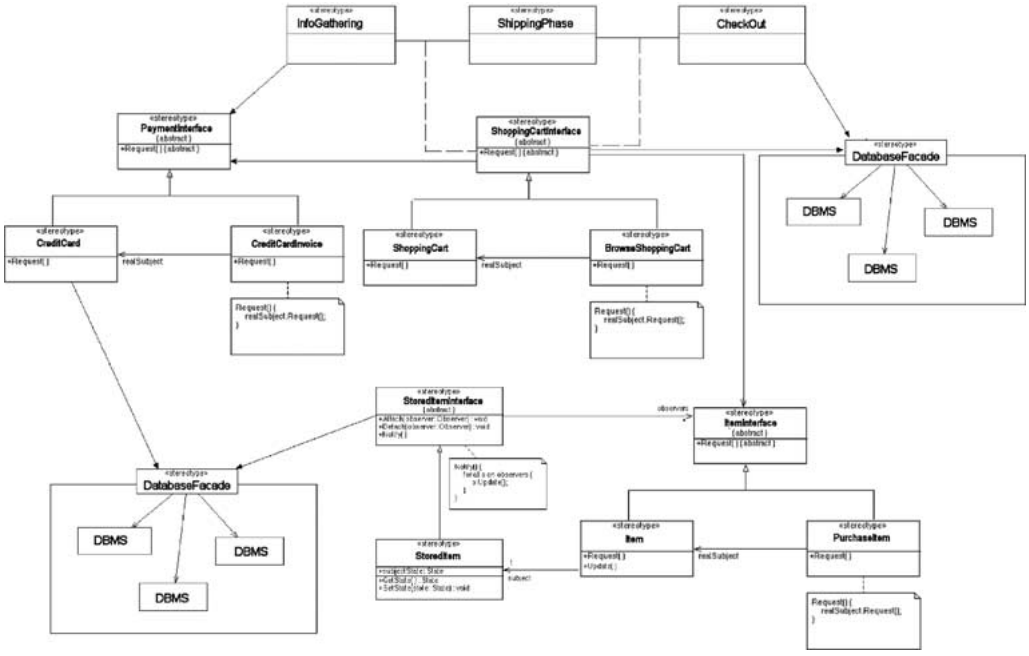

Figure 28. The integrated realization.

For this example, we have resolved behavior and name conflicts by integrating the functionality into a unified class (Item) or by referencing abstract classes that are made concrete according to its use, like the PaymentInterface abstract class that is used by the Payment and CheckingOut components. The unified class diagram is shown in figure 28.

6. Related work

With the concepts of information hiding and the notion of module specification, Parnas [1972] introduced some cornerstones of modern software design. In a sense, the work of Parnas established the roots of our current work. Other early precursors of our ideas were DeRemer and Kron [1976], who defined a module interconnection language to support programming-in-the-large.

In the past few years, a number of architectural models and programming approaches have investigated support to the separation of different concerns in distinct specification modules in order to achieve higher degrees of reuse. Goguen investigated the general interface concept together with the reuse and interconnection of software components [Goguen 1986]. Similar to the views approach, he uses formal languages and mappings of types and operations to interconnect and maintain consistency among objects. He also used category theory to put object theories together [Goguen 1986]. However, Goguen does not define a relationship theory among object specifications, thus making the properties of his design mechanism quite different from our approach.

The Common Object Request Broker Architecture (CORBA), which is supported by the Object Management Group, defined an open standard for application interoperability [Common Object Broker 1991]. This standard is based on a client/server interaction model that separates application interfaces from their implementations. These interfaces are specified in a neutral Interface Definition Language. More recently, Kiczales *et al.* [1997] described a new programming paradigm called Aspect-Oriented Programming (AOP). AOP provides the basis for the identification, isolation, composition and reuse of the several concerns, which are known as aspects, contained in the programs. Nelson *et al.* worked on the specification and validation of models that describe aspects, multi-perspectives, and their composition [Nelson *et al.* 2000].

The MVC model [Krasner and Pope 1988] was one of the first implementations to address separation of concern issues specifically. Currently, several visual development environments [IBM Visual Age 1994; Optima++ 1996] simplify the programming task by making available a library of reusable interface (visual) and application specific (non-visual) objects. The interface objects are interconnected to the application by mechanisms, which are specific to the particular programming paradigm supported by the environment. In addition, the separation of concerns allowed by these mechanisms is mostly directed at the user interface part of the system.

Currently, the Java programming language represents one of the popular paradigms for the development of user interfaces. The Java interface mechanism is event-driven. The event handling model of Java 1.1 is based on the concept of an event listener. An object interested in receiving certain events is called an event listener, while the one generating events is called an event source. This event source object keeps a list of all the listener objects interested in being notified when certain events occur. Such a concept may be very useful in the implementation of a mechanism that maintains the consistency between interface objects and their respective applications. In addition, the AWT class library of Java provides an implementation of the Observer design pattern [Gamma *et al.* 1995], which describes a mechanism for maintaining the consistency between interface and application objects.

Modeling of user interface concepts has been one of the research topics addressing the need for additional and rigorous modeling elements. Other researchers, however, try to improve the expressiveness of modeling languages in other ways. Civello separates roles and meanings of whole-part associations in distinct constructs [Civello 1993]. He argues that the resulting models are easier to understand and maintain with the additional semantics represented. In another attempt to add semantics to modeling elements, Steyaert *et al*. [1996] define reuse contracts based on specialization. These contracts document the way an asset is related to its superclass, thus allowing a better understanding of the circumstances in which an object is specialized.

7. Conclusion

In this paper we have shown how Views and Design patterns can be used to engineer e-commerce applications. This approach allows the reuse of designs and implementa-

tions by indicating how the properties between viewed and viewer objects can be realized through design patterns.

Because of the many possible properties of the *views* relationship, there are various possible mappings from designs that use views or interfaces and their corresponding implementations based on design patterns. In this paper we presented eleven design patterns that can be used to accomplish this purpose. This set of possible realizations should not be seen as a complete set, but as illustration of possible means to implement the *views* properties.

It is important to notice that each of the possible realizations based on design patterns has its strong and weak points that must be considered. In fact, our approach is helpful to clarify important choices and the tradeoffs that are involved in realizing object-oriented designs. These compromises are based on the explicit properties that should be satisfied when design patterns encapsulate different concerns. In this sense, the principles and guidelines presented in this paper constitute a step forward in helping designers to bridge the gap between a design with separation of concerns and its possible realizations.

Acknowledgements

This work is partially supported by NSERC, the Consortium of Software Engineering Research (CSER), CNPq, and by IBM as part of a research project at the TecComm/LES project (`htttp://www.teccomm.les.inf.puc-rio.br`) at PUC-Rio, Brazil.

References

Alencar, P.S.C., D.D. Cowan, and C.J.P. Lucena (2001), "A Logical Theory of Objects and Interfaces," *IEEE Transactions on Software Engineering (TSE)*, to appear.

Alencar, P.S.C., D.D. Cowan, and C.J.P. Lucena (1996), "A Formal Approach to Architectural Design Patterns," In *Proceedings of the Formal Methods Europe Symposium (FME'96),* Springer, Heidelberg, pp. 576–594.

Alencar, P.S.C., D.D. Cowan, and C.J.P. Lucena (1995), "Formal Specification of Reusable Interface Objects," *ACM SIGSOFT Software Engineering Notes 20*, SI, 88–96.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996), *A System of Patterns: Pattern-Oriented Software Architecture*, Wiley Computer Publishing, New York.

Czarnecki, K., U.W. Eisenecker, and P. Steyaert (1996), "Beyond Objects: Generative Programming," In *Proceedings of the ECOOP'97 Workshop on Aspect-Oriented Programming*, Jyväskylä, Finland.

Civello, F. (1993), "Roles for Composite Objects in Object-Oriented Analysis and Design," In *Proceedings of OOPSLA'93*, ACM Press, New York, pp. 376–393.

Common Object Broker (1991), "The Common Object Broker: Architecture and Specification," OMG Document Number 91.12.1, Revision 1.1, Digital Equipment Corporation, Hewlett-Packard Company, Hyperdesk Corporation, NCR Corporation, Object Design Inc. and Sunsoft Inc., OMG Press, Needham, MA.

Cowan, D.D. and C.J.P. Lucena (1995), "Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse," *IEEE Transactions on Software Engineering (TSE) 21*, 3, 229–243.

DeRemer, F. and H. Kron (1976), "Programming-in-the-large Versus Programming-in-the-small," *IEEE Transactions on Software Engineering (TSE) 2*, 2, 80–86.

Dijkstra, E.W. (1976), *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, NJ.

Fayad, M.E., D.C. Schmidt, and R.E. Johnson (1999), *Building Application Frameworks,* Wiley, New York.

Fontoura, M.F., S. Crespo, C.J.P. Lucena, P.S.C. Alencar, and D.D. Cowan (2000), "Using Viewpoints to Derive a Conceptual Model for Web-Based Education Environments," *Journal of Systems and Software (JSS) 54*, 3, 239–257.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts (1999), *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, Reading, MA.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Goguen, J.A. (1986), "Reusing and Interconnecting Software Components," *IEEE Computer 19*, 2, 16–28.

Helm, R., I.M. Holland, and D. Gangopadhyay (1990), "Contracts: Specifying Behavioral Compositions in Object Oriented Systems," In *Proceedings of ECOOP/OOPSLA'90*, Springer, Heidelberg, pp. 169–180.

IBM Visual Age (1994), *Visual Age: Concepts and Features*, IBM Corporation.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and I. Irwin (1997), "Aspect-Oriented Programming," In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'97)*, Springer, Heidelberg, pp. 220–242.

Krasner, G.E. and S.T. Pope (1988), "A Cookbook for Using the Model–View–Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming 3*, 1, 26–49.

Krueger, C.W. (1992), "Software Reuse," *ACM Computing Surveys 24*, 2, 131–183.

Markiewicz, M.E., C.J.P. Lucena, and D.D. Cowan (2000), "Taming Access Control Security Using the "Views" Relationship," Technical Report MCC19/00, Departamento de Informatica, Pontifical Catholic University (PUC-Rio), Rio de Janeiro, Brazil.

Mili, H., F. Mili, and A. Mili (1995), "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering (TSE) 21*, 6, 528–561.

Nelson, T., D.D. Cowan, and P.S.C. Alencar (2000), "A Model for Describing Object-Oriented Systems from Multiple Perspectives," In *Proceedings of the Conference on Foundational Aspects of Software Engineering (FASE2000), ETAPS2000 (the European Joint Conferences on Theory and Practice of Software)*, Technical University of Berlin, Germany, pp. 237–248.

Optima++ (1996), *Sybase Optima++*, Sybase Inc.

Parnas, D.L. (1972), "On Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM 15*, 2, 1053–1058.

Steyaert, P., C. Lucas, T. Mens, and T. D'Hondt (1996), "Reuse Contracts: Managing the Evolution of Reusable Assets," In *Proceedings of OOPSLA'96*, ACM Press, New York, pp. 268–285.

UML (1999), *Unified Modeling Language 1.3*, specification available at `http://www.uml.org`

Watcom VX-REXX (1993), *WATCOM VX-REXX for OS/2 Programmer's Guide and Reference*, Watcom International Corporation.

Wentzel, K.D. (1994), "Software Reuse – Facts and Myths," In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*, ACM Press, New York, pp. 267–268.

Wiederhold, G., P. Wegner, and S. Ceri (1992), "Towards Megaprogramming," *Communications of the ACM 35*, 11, 89–99.