Abstract Design Views and Design Patterns

Marcus E. Markiewicz
e-mail: mem@acm.org

Carlos J. P. Lucena
e-mail: lucena@inf.puc-rio.br

Donald D. Cowan
e-mail: dcowan@csg.uwaterloo.ca
Computer Science Department and
Computer Systems Group University of Waterloo, Canada

PUC-RioInf.MCC46/00 December, 2000

Abstract: Software reuse can be defined as the creation of new software systems using old artifacts. The Abstract Design Views model was created with reuse in mind, allowing the designer to apply separation of concerns from the design to the implementation. Using a reuse taxonomy, this model will be analyzed and categorized. Even further, using design patterns it will be shown how Abstract Design Views can be realized and implemented.

Keywords: reuse, Abstract Design Views, software engineering, design patterns

Resumo: Reuso de software pode ser definido como a criação de novos sistemas de software usando artefatos antigos. O modelo Abstract Design Views foi criado com reuso em mente, permitindo que projetistas de sistemas aplicassem *separation of concerns* do projeto à implementação. Utilizando uma taxonomia de reuso, este modelo será analizado e categorizado neste documento. Além disso, utilizando *design patterns* será mostrado como Abstract Design Views pode ser realizado e implementado.

Palavras-chave: reuso, Abstract Design Views, engenharia de software, design patterns

Sponsored by IBM Brazil

Introduction

Software reuse is perhaps the Holy Grail of software engineering. In a nutshell, software reuse can be defined as the use of existing software artifacts during the construction of a new system. These artifacts include design elements, documentation, specification, source code and many other items in a software engineering project.

For more than 20 years many models and approaches were introduced, but none with complete success or mass adoption. Perhaps this comes as a consequence from design models that don't deal with reuse beforehand.

Abstract Design Views [IEEE95], on the other hand, were created with reuse and separation of concerns in mind. ADVs provide a formal design model that allows the designer to clearly separate the interface from the application, and to keep this separation in the implementation phase.

In this paper we will analyze the Abstract Design View model using a proper taxonomy for software reuse [Krueger92]. 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. Using previous applications of this model, we will show how it successfully improves reuse from the design level to the implementation.

The Abstract Design Views model

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. 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. This way, the first property of this model is devised:

Property #1	**Visibility Property** - An ADO is only accessible through one or more ADV. Thus, only one or more ADVs have references to an ADO at any given time. Also, it is possible for an ADV to have references for one or many ADOs.

This property can be found applied in many papers on the ADV/ADO concept [Formal95][IEEE95][Theory94][Theory94a][Tool95], but it has never been properly formalized.

The separation between views and objects allows us to associate many ADVs to a single ADO. In this case, as the state of an ADO changes, the ADVs connected must be consistent with that change. Using morphisms or mappings defined between the ADV and the ADO, this invariant is expressed, as in [Formal95].

The consistency among ADVs is called *horizontal consistency*, while the consistency between ADOs and ADVs is called *vertical consistency*. Therefore, we have the second and third property of this model.

Property #2	**Horizontal Consistency Property** – Each one of the ADVs related to an ADO must be consistent between 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 [Formal95][IEEE95][Theory94][Theory94a][Tool95]. In these articles, the properties are already called by these names.

In Figure 1 we have an ADV-ADO interaction model. In this figure the horizontal and vertical consistencies are clear, as well as how ADVs act as points of entry to ADOs.

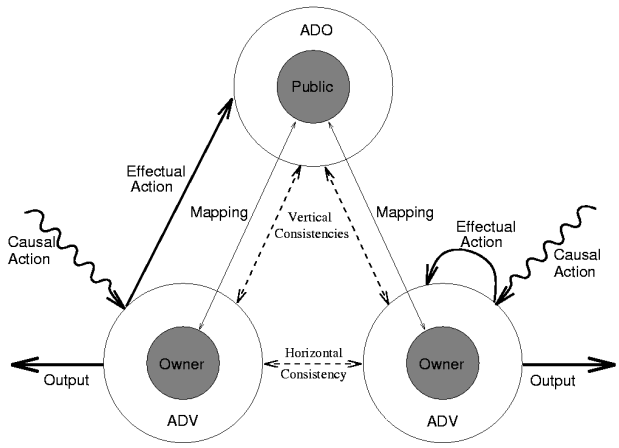

Figure 1: An ADV-ADO Interaction Model

In the Abstract Design View model, it is possible for ADVs and ADOs to nest objects. By nesting we imply that objects are composed or aggregated. The enclosing ADV or ADO knows the identity of its constituents, but the contrary is not true. The details of the nesting of objects are declared through morphisms [Formal95], specifying the relationships between the enclosing and enclosed objects. For example, in Figure 2, we have a composite ADV. In this case, the ADV_{ij} and ADV_{ji} are enclosed within the ADV_{ij}^{b}. Thus, we have the fourth property of the Abstract Design Views model. A more formal definition of this property can be found in [Semantics94][Theory94][Theory94a].

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

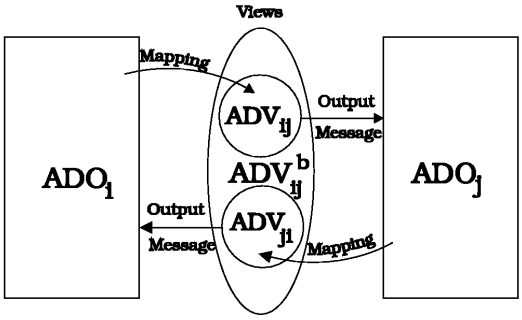

Figure 2: A Composite ADV

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 [Markiewicz00], and is not present in previous articles.

When two ADVs are connected to a 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 of it. Even further, ADV_5 is both a direct viewer of ADV_2 and ADV_4 and an indirect viewer of ADO_x.

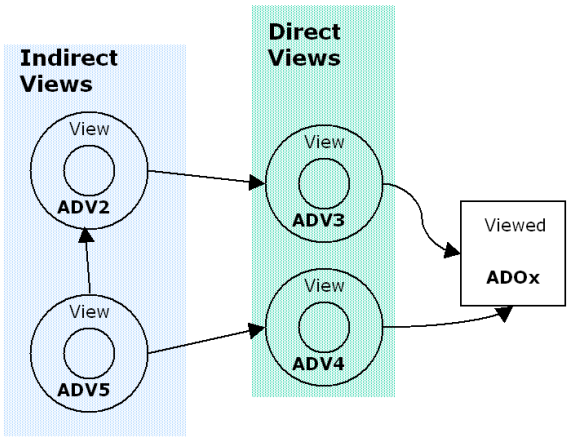

Figure 3: Direct and Indirect Viewers of an ADO

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 ADV, 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 [FME96][IEEE95]. 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 bi-directional manner. If unidirectional, the ADV will map actions directly into other ADOs. If bi-directional, each ADV will map the actions into the other ADO, performing a duplex communication mapping between the ADOs, such as in Figure 2, for example. It is important to notice that the interface ADV here can be used to make translations and adaptations in order to convert the output of one ADO to the format of the other ADO's input. The roles of ADOs and ADVs are explained below.

View of an ADO

According to the visibility property, the ADVs of an ADO are its points of entry. If we consider that each ADV introduces a "view" of an ADO, the ADV acts as a viewer of it. This way, an ADV observes the ADO, mapping the inputs onto the ADO, and relaying the output to the user of the ADV. This architecture is shown in Figure 4.

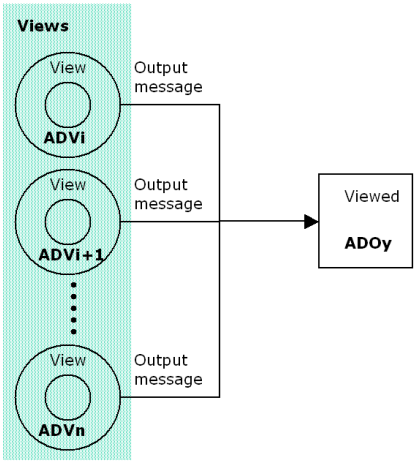

Figure 4: ADVs acting as viewers for an ADO

It is important to notice that it is also possible to have single composite ADVs 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 [Markiewicz00] for access control purposes.

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 there is a representation of this architecture.

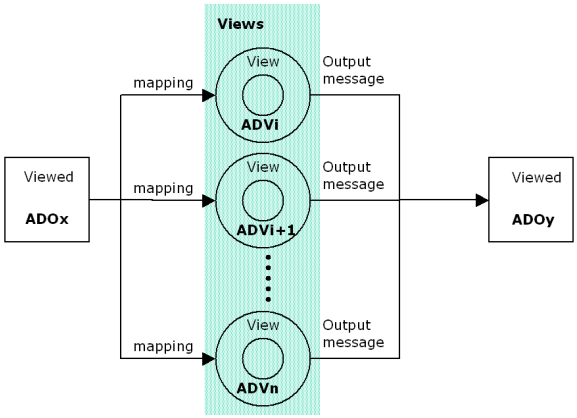

Figure 5: Two ADVs serving as unidirectional interfaces for two ADOs

Bi-directional Interface of two media

Extending the previous role, it is possible that ADVs provide bi-directional 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. In Figure 6 this architecture is represented.

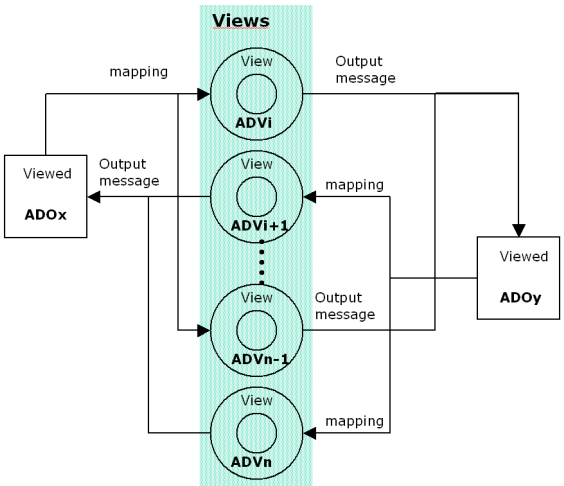

Figure 6: ADVs serving as bi-directional interfaces for two ADOs

Facilitator of n-media

Finally, it is possible to extrapolate the previous roles using multiple ADOs and ADVs. This way, 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 ADVi that is the facilitator of ADOs x,y,z and w.

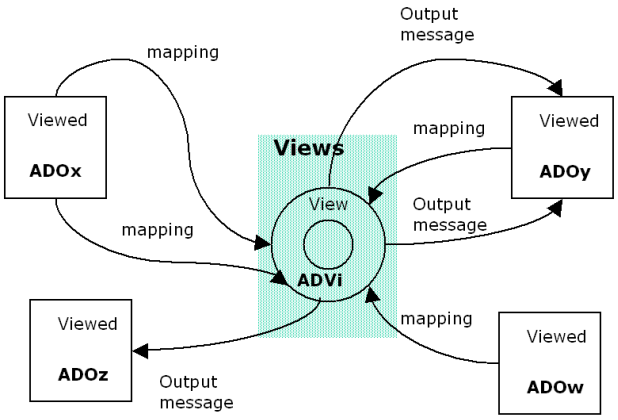

Figure 7: An ADV as a facilitator of n-media

Analyzing the Abstract Design Views model

Using the taxonomy introduced by [Krueger92], it is possible to 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 [Krueger92] of ADVs and ADOs from the software system designer.

Abstraction

Every software component is created having some form of abstraction in mind. This abstraction allows component reusers to figure out what every artifact does without having to pry into 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 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. This relationship is shown in Figure 8.

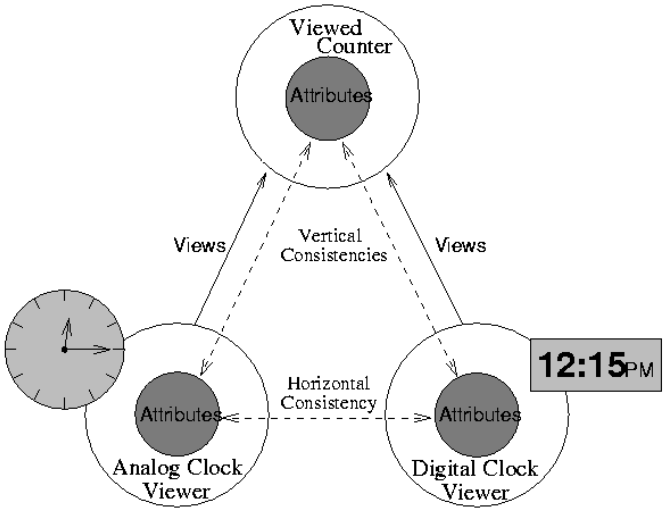

Figure 8: Two different clocks from a single source

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.

Selection

It is very important for the reuser 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 that is usable. 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.

Specialization

Many reuse technologies use generic artifacts that are instantiated by parameters or even inheritance. These artifacts are refined before its use, allowing the reuse of the generic artifact in many solutions.

Besides the possibility of nesting ADVs and ADOs as explained in the last section, their specifications can be reused and combined using mechanisms such as composition, inheritance, sets and sequences. This way, ADVs can be specialized or incremented over time. It is important to notice that these relationships are reflected on

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.

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 interconnection 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.

Conclusion

In Figure 9 we have 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.

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 reuser 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. This way, ADVs can be specialized or incremented over time.
Integration	ADVs can be nested, and ADVs can act as interconnections between ADOs. This way, 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 user 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.

Figure 9: Reuse in the Abstract Design Views model

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 located at a higher design level than the implementation, their realization is possible using distinct architectural approaches.

Realization using Design Patterns

One possible approach for realizing ADVs and ADOs is using design patterns [Gamma95]. However, many design patterns can be used for that purpose and their selection can be based on ADV's particular aspects and properties.

Using the Observer Design Pattern

The most obvious approach to realize ADVs and ADOs is the observer design pattern [Gamma95]. The main objective of this pattern is to define a one-to-many dependency between objects, so that the dependent objects can monitor changes in one object. In this architecture, represented in Figure 10, ADVs correspond to the observers, while the ADOs are the subjects.

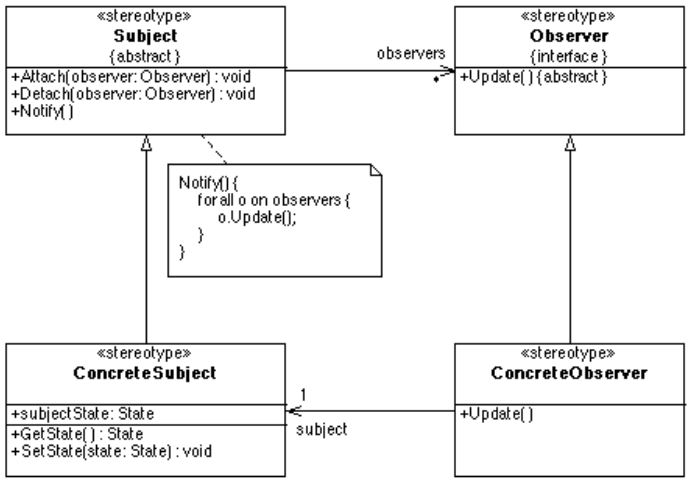

Figure 10: The Observer Design Pattern

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.

The Callback vs. 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.

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, in [Tool95] 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. 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.

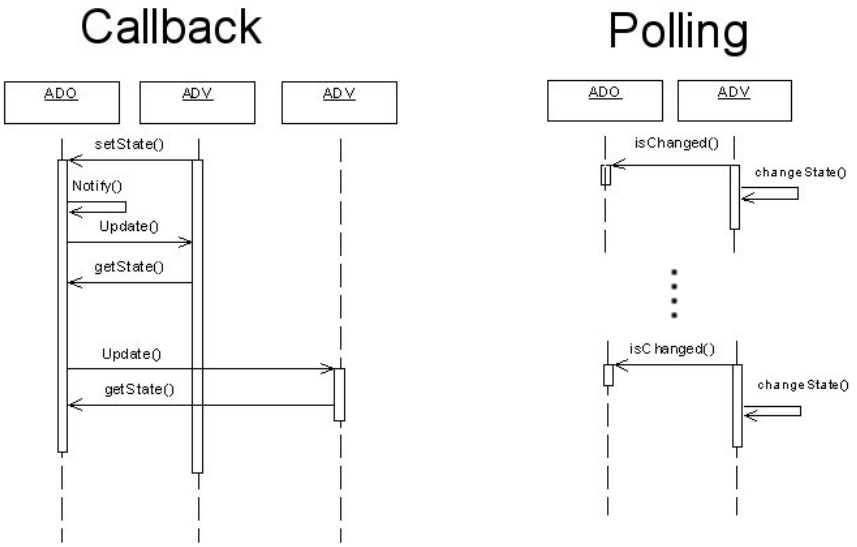

Figure 11: Callback versus Polling dynamics

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 [Tool95] 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 11. In the callback, the notify() action taken by the 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 vs. callback solution before implementing the vertical consistency of ADVs and ADOs. One must trade run-time overhead versus promptly update.

Using the Proxy Design Pattern

The proxy design pattern [Gamma95] can also be used to realize ADVs and ADOs. In this architecture, the RealSubject is the ADO, and the Proxy the ADV. This is represented in Figure 12.

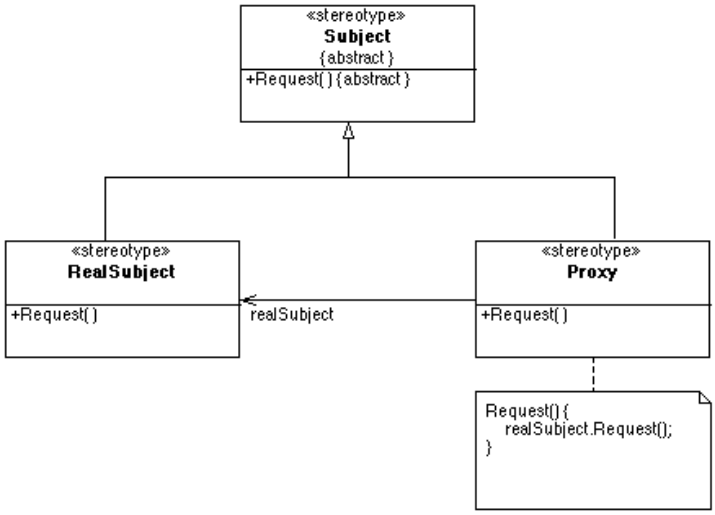

Figure 12: The Proxy Design Pattern

It is important to notice that the use of this pattern allows for the ADV and ADO to be binded 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 ancestral. Thus, by belonging to a same inheritance hierarchy, ADVs can act as ADOs object without any overhead.

Using the Adapter Design Pattern

Another possible design pattern that can realize ADVs and ADOs is the adapter design pattern [Gamma95].

In this architecture, the client ADO is the target object, the component ADO the adaptee, and the view (ADV) the adapter. This architecture is used in [Tool95]. It is represented in Figure 13.

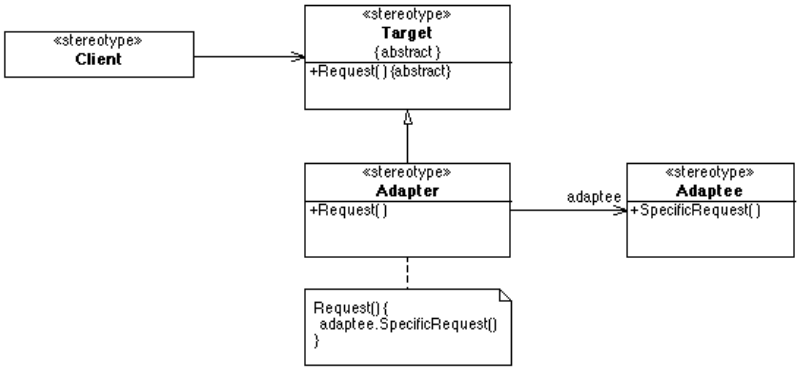

Figure 13: The Adapter Design Pattern

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 making the ADV inherit the ADO interface.

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 [Gamma95]. This design pattern is represented in Figure 14.

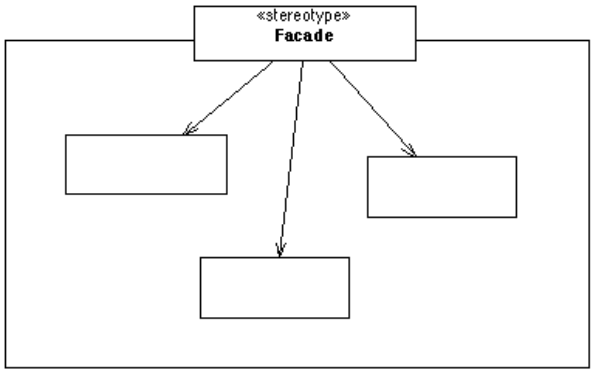

Figure 14: The Façade Design Pattern

For this architecture, 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 the whole component.

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, thus not acting bidirectionally.

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

For example, in Figure 15 we have two classes. The Vector class implements a vector with a method that returns the value of the nth 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.

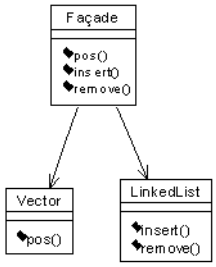

Figure 15: Semantically Incoherent Façade

By creating a façade for these two classes, we have a semantically incoherent façade, as it can be misused due to 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.

Using the Pipes and Filters Design Pattern

The pipes and filters design pattern [Buschmann96] 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 [Buschmann96]. In this architecture, represented in Figure 16, the ADOs are the data source and data sink, as the ADVs are the filter objects.

Figure 16: The Pipes and Filters Design Pattern

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

Using the Master-Slave Design Pattern

The Master-Slave design pattern [Buschmann96] 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 [Buschmann96]. In this design pattern, ADVs are the master objects, while the ADOs are the slaves. In Figure 17 we have the class diagram of this pattern.

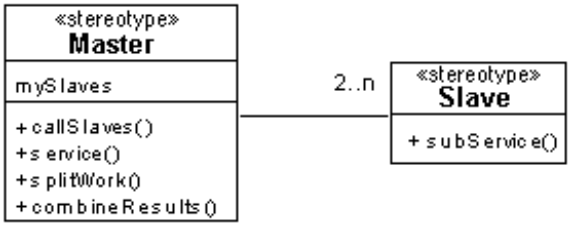

Figure 17: The Master-Slave Design Pattern

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 n equal method (subservice()), that must receive the same number and type of parameters. In this architecture there is a single ADV for many ADOs (more than two). Thus, its use is limited.

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

Using the Blackboard Design Pattern

The blackboard design pattern [Buschmann96] 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 subsystems assemble their knowledge to build a possibly partial or approximate solution [Buschmann96]. In this architecture, represented in Figure 18, the ADOs and ADVs do not map directly into the pattern objects.

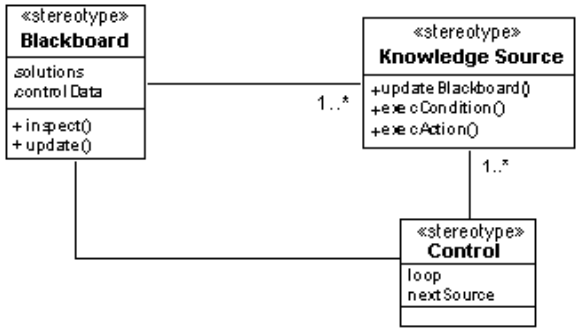

Figure 18: The Blackboard Design Pattern

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.

Using the Mediator Design Pattern

Another possible design pattern that can be used to realize ADVs and ADOs is the mediator design pattern [Gamma95]. In this architecture, the ADVs are represented as the mediator objects, and the ADOs as the colleague derivations. This pattern is represented in Figure 19.

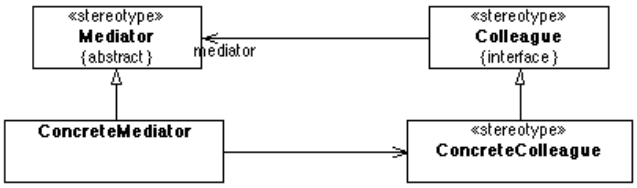

Figure 19: The Mediator Design Pattern

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

Using the View Handler Design Pattern

The View Handler design pattern [Buschmann96] 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 [Buschmann96]. This pattern is represented in Figure 20.

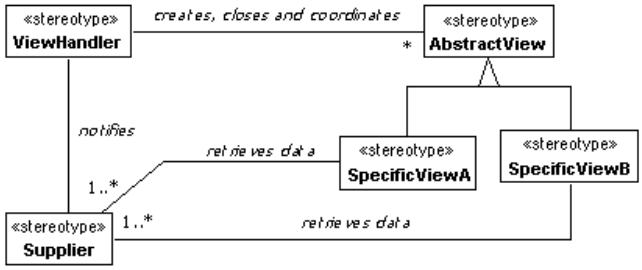

Figure 20: The View Handler Design Pattern

In this architecture, 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 (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.

Using the Fowarder-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 decouple peers from the underlying communication mechanisms [Buschmann96]. This pattern is represented in Figure 21.

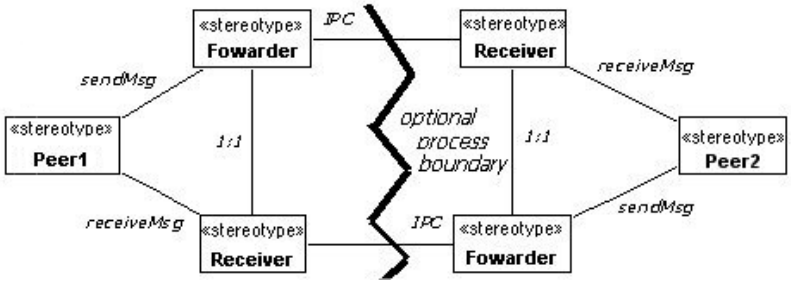

Figure 21: The Forwarder-Receiver Design Pattern

In this architecture, 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 bi-directional communication between the two ADO objects.

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 [Buschmann96]. This pattern is presented in Figure 22.

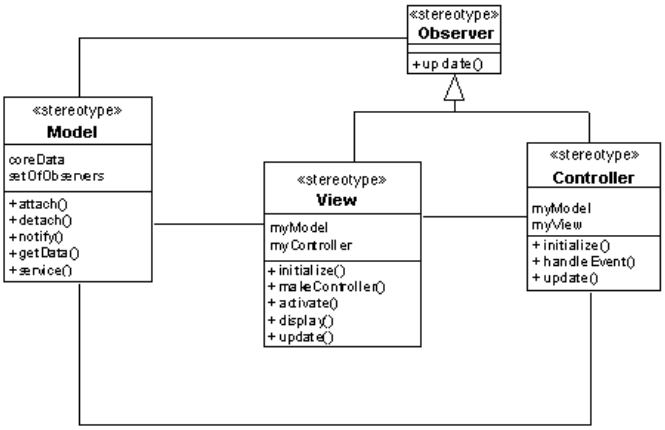

Figure 22: The Model-View-Controller Design Pattern

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.

Compositions

It is possible to realize ADVs and ADOs using not only the design patterns shown above, but also combinations of them. It is possible, for example, to combine the fowarder-receiver design pattern with the observer design pattern, creating a subscription to a bi-directional channel of communication for objects.

The composition approach often will be present, and should be used carefully. The use of complex realizations will 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.

Summing up

In the previous sections we have presented ten design patterns that can be used to realize ADVs and ADOs relationships. Since the Abstract Design Views model resides on 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 fFgure $FIG we have the pros, cons and comments of each possible realization.

It is important to notice that the possibilities presented here are not at all exhaustive, and the composition of patterns will happen often.

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 milder 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 binded to the ADO contract ➢ Easy composition of ADOs with ADVs as simple filters	➢ ADVs are not binded 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 that all ADOs have a common method signature. ➢ 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 functionalities 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.	Introduces little or no reuse since any common hierarchy between two mediators is not necessary.
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 bi-directional interface of two media.	➢ Difficult to apply to different ADVs cardinalities.	Recommended for realization of bi-directional interfaces of two media.
Model-View-Controller	➢ Long established pattern.	➢ A tertiary element is introduced. ➢ The visibility property is endangered.	None.

Figure 23: Possible Design Pattern Realizations for ADVs

Another issue that one must deal with is the relationship between the realizations and the properties and roles that ADVs and ADOs have. Many realizations will not enforce Abstract Design Views properties, but none of these break the properties. This is shown in Figure $FIG. For example, the proxy pattern will not enforce the horizontal consistency property, leaving it to be checked externally, but does not break it at all.

Design Pattern	Properties					Best Realizes ADV-ADO Relationship			
	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		√	√	√	√	√	√	√	

Figure 24: Realizations and ADV's Properties

An Example: An e-Commerce System

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

In Figure $FIG 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 subsystems of the e-commerce system, by separating viewed and viewed sub-units of the whole diagram.

In the example shown in Figure $FIG, there are six subsystems: payment, database, checkingOut, inventory, catalog and browsing. The payment subsystem is responsible for the sensitive information needed to buy products. The Database subsystem provides consistency in order to log all purchases. The checkingOut subsystem models the process of purchasing and the steps necessary to achieve this purpose. The inventory subsystem deals with the information and processes needed for the maintenance of the items being sold, its characteristics and status. Finally, the catalog subsystem 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.

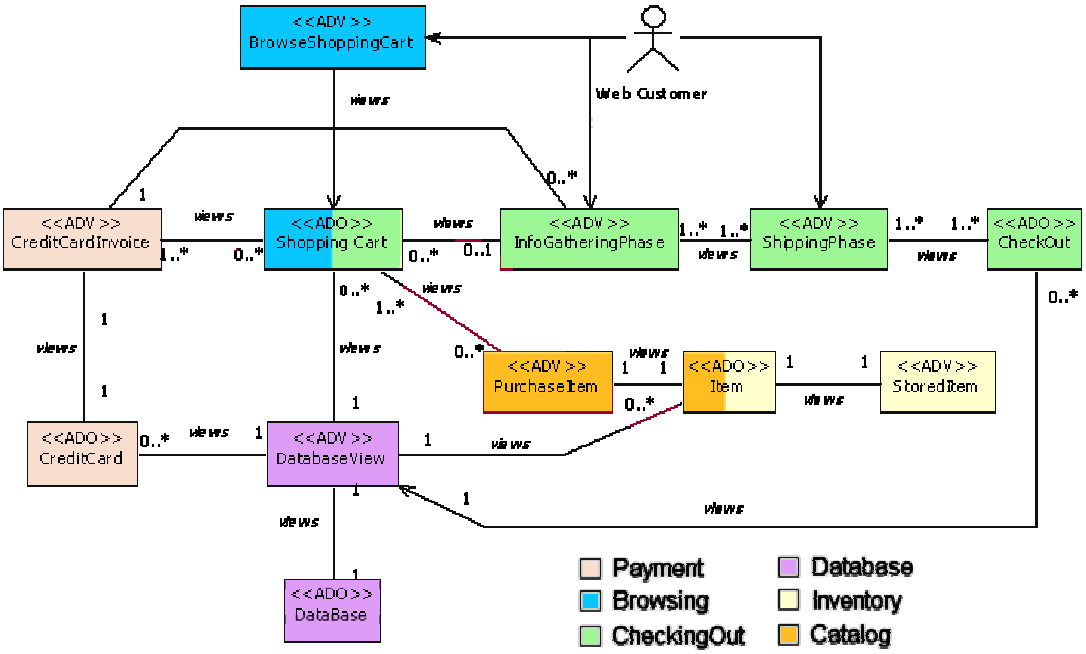

Figure 25: Modeling an e-commerce System using ADVs

It is important to notice that some ADVs will belong to more than one ADO, thus being necessary to be merged later on.

Payment Subsystem

In this subsystem, the credit card class has some of its information hidden in order to ensure security. The shopping cart must only be aware of some details of the credit card information. Thus, there must be an access of 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 functionalities, 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 $FIG.

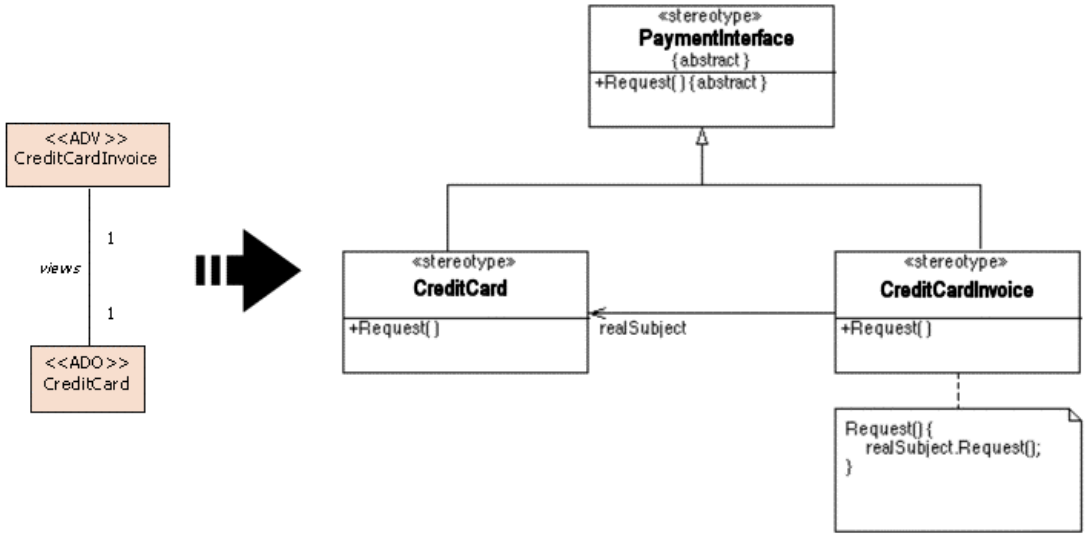

Figure 26: Realizing the Payment Subsystem

In this case there is no need to update the CreditCardInvoice object, since it is "stateless" in a 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.

Browsing Subsystem

The browsing subsystem is the viewable functionalities of the shopping carts, being the point of entry of the customers in the e-commerce system. Much like the payment subsystem, the shopping cart visible to the user is a restricted one, perhaps one that has no direct information regarding the credit card number. Since it is a restriction of access functionalities, once more the proxy pattern is appropriate. The resulting realization is shown in Figure $FIG.

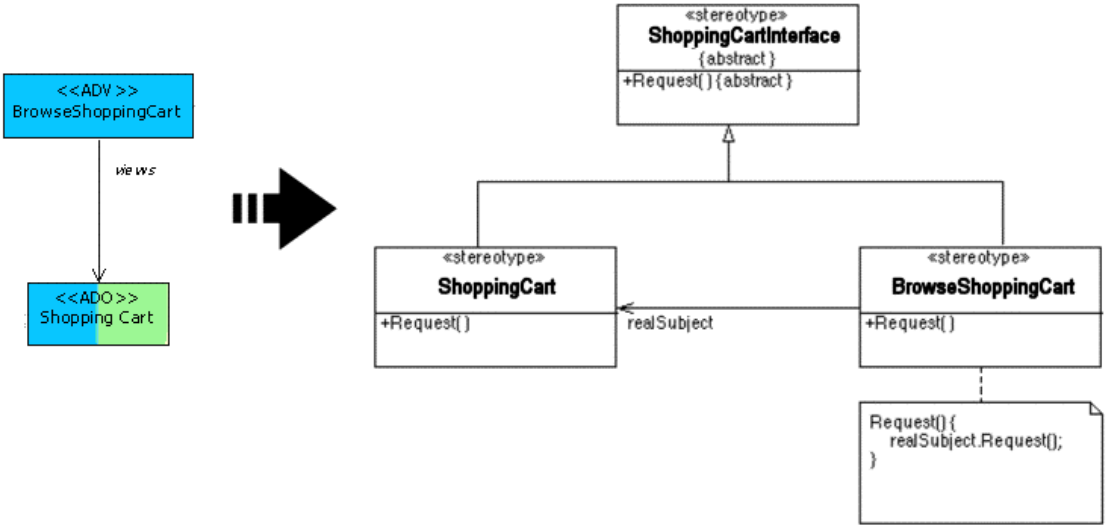

Figure 27: Realizing the Browsing Subsystem

CheckingOut Subsystem

The checkingOut subsystem 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 of 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 $FIG.

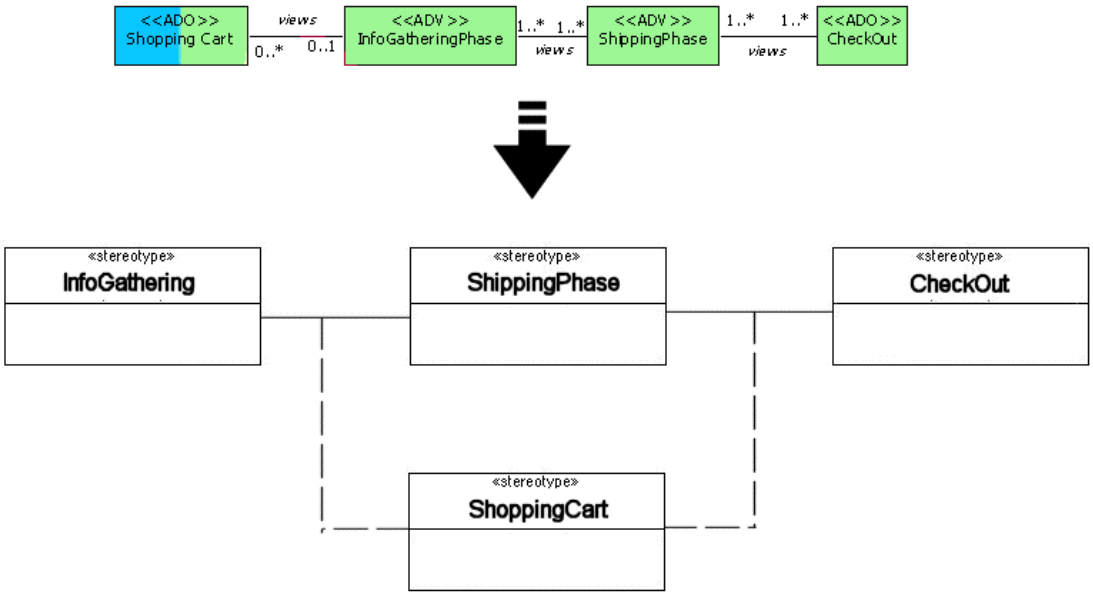

Figure 28: Realizing the CheckingOut Subsystem

Database Subsystem

This subsystem 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 subsystem is the façade design pattern, as shown in Figure $FIG.

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.

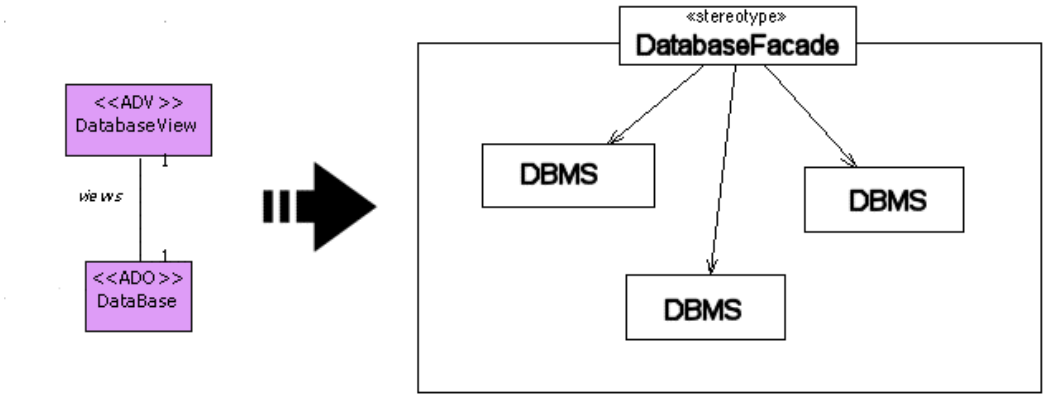

Figure 29: Realizing the Database Subsystem

Inventory Subsystem

This subsystem 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 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 $FIG.

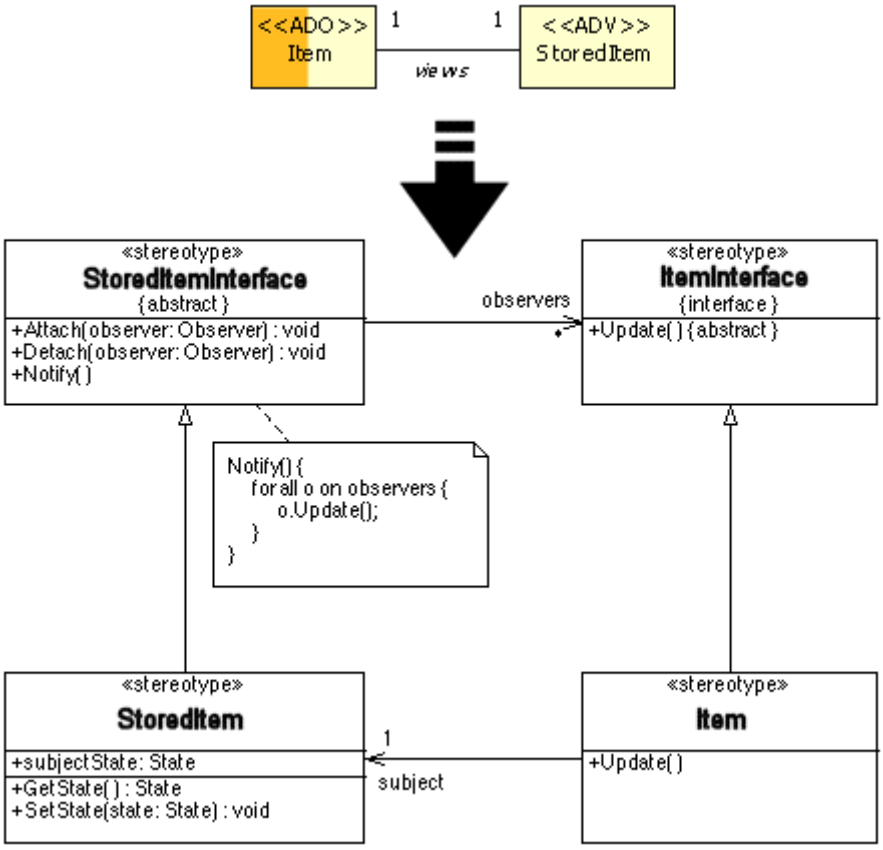

Figure 30: Realizing the Inventory Subsystem

Catalog Subsystem

Much like the payment and browsing subsystems, 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 $FIG.

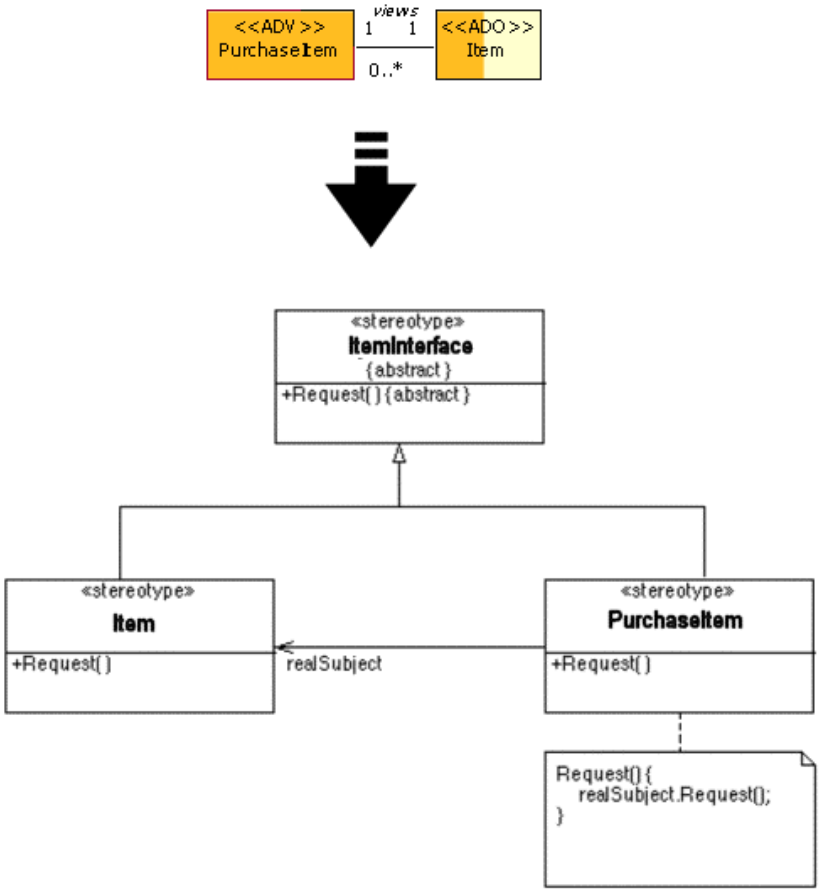

Figure 31: Realizing the Catalog Subsystem

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 subsystems that were not yet dealt with.

Thus, the next step is to integrate the subsystems. In the cases where there is name collision (ShoppingCart) or even similar behaviors (Item class of both the inventory and catalog subsystems), it is necessary to apply refactoring. One can use the refactorings presented in [Fowler99] in order to achieve a unified class diagram.

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 subsystems. The unified class diagram is shown in Figure $FIG.

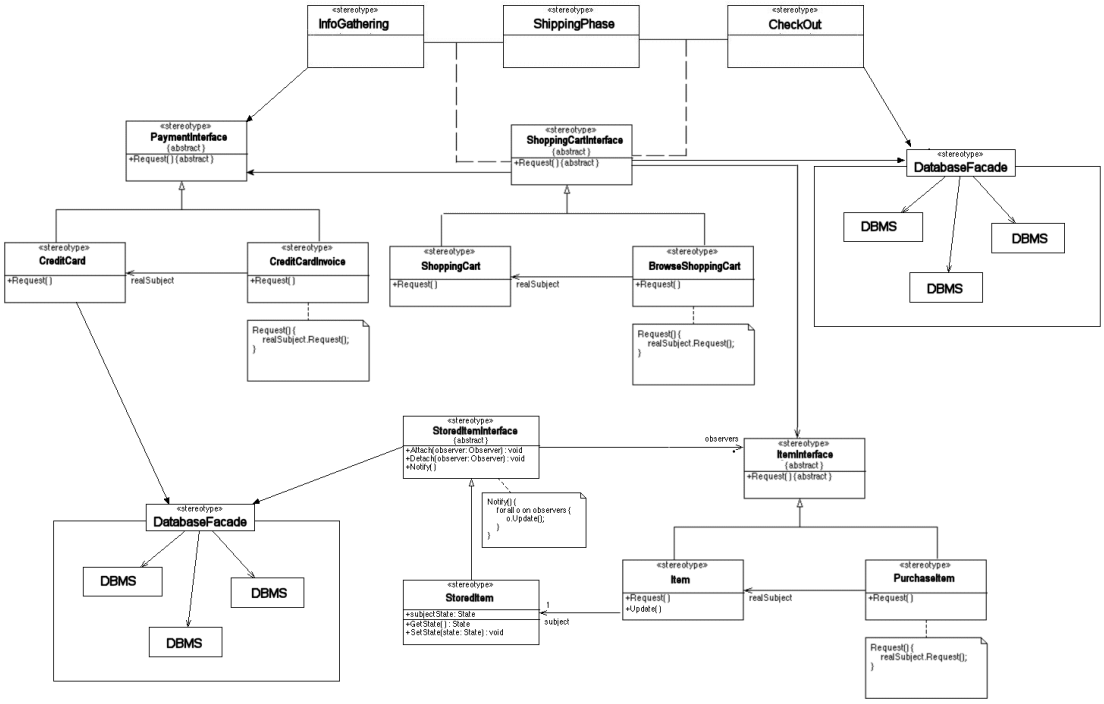

Figure 32: The Integrated Realization

Conclusion

In this paper the reuse of the Abstract Design Views model was assessed. As seen above, this model allows reuse of design and implementation through the realization of the ADVs and ADOs relationships using design patterns.

Due to the many possible relationships and relationship cardinalities of ADVs and ADOs, there are many possible mappings from the design to the implementation. In this paper we presented ten design patterns that can be used in this task. However, as shown, it is important to notice that each possible realization has strong and weak points that must be considered. Even further, there are design patterns that only realize a subset of the ADVs and ADOs relationships.

Note to the Reader

This work is part of an IBM Brazil project at the TecComm/LES project (http://www.teccomm.les.inf.puc-rio.br) at PUC-Rio, Brazil.

Many of the technical reports mentioned in this paper are available via anonymous ftp form 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”.

At the Teccomm/LES ftp site ftp.teccomm.les.inf.puc-rio.br there is a mirror of the University of Waterloo ftp site, with all the above-mentioned technical reports. The files are located in the same directory path: “pub/ADV”.

Bibliography

[Buschmann96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. "**A System of Patterns: Pattern-Oriented Software Architecture**", Wiley, 1996.

[FME96] Alencar, P.S.C.; Cowan, D.D.; Lucena, C.J.P. "**A Formal Approach to Architectural Design Patterns**", Technical Report, Computer Science Group at the University of Waterloo, 1996.

[Formal95] Alencar, P.S.C.; Cowan, D.D.; Lichtner, K.J.; Lucena, C.J.P.; Nova, L.C.M. "**Tool Support for Formal Design Patterns**", Technical Report, Computer Science Group at the University of Waterloo, August, 1995.

[Fowler99] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. "**Refactoring : Improving the Design of Existing Code**", Addison-Wesley, 1999.

[Gamma95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. "**Design Patterns: Elements of Reusable Object-Oriented Software**", Addison-Wesley, 1995.

[IEEE95] Cowan, D.D.; Lucena, C.J.P. "**Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse**", IEEE Transactions on Software Engineering (TSE), Vol. 21, No. 3, March, 1995.

[Krueger92] Krueger, C. W. "**Software Reuse**", ACM Computing Surveys, Vol. 24, No. 2, June 1992.

[Markiewicz00] Markiewicz, M.E.; Lucena, C.J.P.; Cowan, D.D. "**Taming Access Control Security using the "Views" Relationship**", Technical Report MCC19/00, PUC-Rio, Brazil, May 2000.

[Semantics94] Alencar, P.; Carneiro-Coffin, L.; Cowan, D.D.; Lucena, C.J.P. "**The Semantics of Abstract Data Views: A design concept to support reuse-in-the-large**", Proc. Colloquium Object-Orientation in Databases and Software Eng., Kluwer Press, May 1994.

[Theory94] Alencar, P.; Carneiro-Coffin, L.; Cowan, D.D.; Lucena, C.J.P. "**Toward a Formal Theory of Abstract Data Views**", Technical Report, Computer Science Group, University of Waterloo, Waterloo, Ontario, Canada, April 1994.

[Theory94a] Alencar, P.; Carneiro-Coffin, L.; Cowan, D.D.; Lucena, C.J.P. "**Toward a Logical Theory of ADV's**" Proc. Workshop Logic, Found. Object. Orient. Programm. ECOOP94, July 1994.

[Tool95] Alencar, P.S.C.; Cowan, D.D.; Lichtner, K.J.; Lucena, C.J.P.; Nova, L.C.M. "**Tool Support for Formal Design Patterns**", Technical Report, Computer Science Group at the University of Waterloo, Canada, August, 1995.