

A Pattern Language for Adaptive Object Models: Part I - Rendering Patterns

León Welicki
ONO (Cableuropa S.A.)
lwelicki@acm.org

Joseph W. Yoder
The Refactory, Inc.
joe@refactory.com

Rebecca Wirfs-Brock
Wirfs-Brock Associates
rebecca@wirfs-brock.com

Introduction

An Adaptive Object-Model is a system that represents user-defined classes, attributes, relationships, and behavior as metadata [YBJ01; YJ02]. The system is a model based on instances rather than classes. Users change the metadata (object model) to reflect changes in the domain. These changes modify the system's behavior. In other words, an AOM stores its Object Model in a database and interprets it. Consequently, the object model is adaptable; when the descriptive information is modified, the system immediately reflects those changes similar to a UML Virtual Machine described by Riehle et. al [RFBO01].

The design of Adaptive Object-Models differ from most object-oriented designs. Normally, object-oriented design would have classes for describing the different types of business entities and associates attributes and methods with them. The classes model the business, so a change in the business causes a change to the code, which leads to a new version of the application. An Adaptive Object-Model does not model these business entities as classes. Rather, they are modeled by descriptions (metadata) that are interpreted at run-time. Thus, whenever a business change is needed, these descriptions are changed which are then immediately reflected in the running application. So in a sense, what we normally would model as a class is now being modeled by metadata which is being interpreted by the AOM.

Adaptive Object-Model architectures are usually made up of several smaller patterns. TypeObject [JW98] provides a way to dynamically define new business entities for the system. TypeObject is used to separate an Entity from an EntityType. Entities have Attributes, which are implemented with the Property pattern [FY98]. The TypeObject pattern is used a second time in order to define the legal types of Attributes, called AttributeTypes. This core set of patterns working together is very common to most AOM architectures as described by Dynamic Object Models [RTJ05]. These Entities and Properties with their valid types are what the user thinks of. So if the user is selling products, the AOM will describe different types of Entities to represent their different types of products. Non-AOM systems would model these with different product classes.

As is common in Entity-Relationship modeling, an Adaptive Object-Model usually separates attributes from relationships. In usual OO design relationships are implemented through an attribute as a pointer or direct reference to the related objects and methods are used to implement any rules about the relationship. However in AOMs these relationships are reified thus enabling a way to describe new types of relationships

and rules governing the relationships via metadata. The Strategy pattern [GoF95] is used to define the behavior of EntityTypes. These strategies can evolve into a rule-based language that gets interpreted at runtime. Finally, there is usually an interface for non-programmers to define the new types of objects, attributes and behaviors needed for the specified domain. This can even include ways to define subtypes and relationships between objects.

The above set of core patterns has been well described in quite a few papers but one area that has not been described is the GUI side of AOMs. In a non AOM system you model the domain level which may include a persistent mapping to a database. This domain model will include the core business rules. Then as described in MVC or three-tiered (multi-tiered) systems you will build a GUI layer on top of the domain model. This GUI layer may include some validation checks [Cunningham] but the core business rules are mostly implemented in the domain model.

However, an AOM does not include domain objects which the GUI layer maps the domain values that are being interacted with by the user. Rather, AOM's have a meta layer that is instance based rather than class based to model domain objects. Therefore, special consideration needs to be given to this meta level in order to build a GUI layer that maps to an AOMs operational layer. These patterns focus on this area of AOMs describing the visualization layer in AOMs and how to dynamically build the GUI layer from the AOM knowledge layer.

The Visualization Level in AOMs

In the existing literature [FY98] [YR00] [RY01] [YBJ01] [YJ02], the core architecture of AOMs is defined by two core levels:

- **Knowledge Level:** defines the general rules that govern the behavior [Fowler97] and the structure of the domain (TypeObjects, PropertyTypes).
- **Operational Level:** records the values of the domain (in our case the instance of Entities and Properties for collection user values.) [Fowler97]

These two levels contemplate the description of structure and behaviour of the adaptive entities. However a presentation level is almost always needed in addition to these levels. This **Visualization level** is composed of rendering components at different levels of granularity. These components can be combined dynamically (and adaptively) to generate complex output for the AOM, helping to abstract and encapsulate presentation issues [Welicki06b].

A fact that makes the presentation layer so important is that visualization is something that is present in almost all documented AOMs and that presents some hard challenges that have to be overcome; specifically because of the dynamic changing behaviour of AOMs.

In summary, we can say that the Visualization layer contains instructions on how to present the elements of the model. These instructions can be attached and composed at runtime (from configuration information) and are themselves dynamic in behaviour reflecting the adaptive system.

AOM Rendering Patterns

This paper contains the following patterns:

- **Property Renderer:** deals with how to render concrete instances of types of properties.
- **Entity View:** deals with the coordination of several property renderers to produce more complex UI fragments for an entity.
- **Dynamic View:** given a set of entities, renders UI code (including layout issues). Several different views can exist for the same set of entities and the views and can be linked dynamically at run-time.

All the patterns presented in this paper are highly related. They can be used individually, in several combinations or all together. The following patterns map (figure 1) shows the existing relationships between the patterns. The more fine-grained pattern is PROPERTY RENDERER, who deals with rendering individual property instances. It is connected with all the other patterns: the ENTITY VIEW coordinates several Property Renderers to generate a fragment of a UI for an entity and the DYNAMIC VIEW can use these pieces for rendering sets of entities as well. The DYNAMIC VIEW is the more coarse-grained pattern, since generates a coherent piece of UI for sets of entities. To achieve its objective it can use the other patterns or be implemented with customized code.

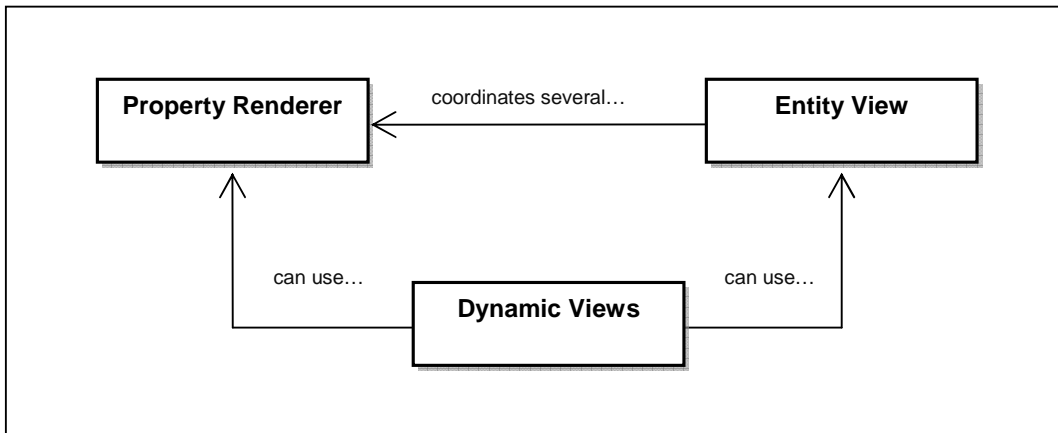


Figure 1 – Rendering patterns map

Target Audience

The patterns presented in this paper deal with the presentation issues that arise when working with AOMs. Therefore, any developer working with this kind of systems (mainly TYPE OBJECT, PROPERTIES [JW98], and TYPE-SQUARE [YBJ01] or DYNAMIC OBJECT MODEL [RTJ05] based architectures) would benefit from using the patterns in this paper to cope with presentation issues.

These patterns may apply also to other rendering scenarios, but their main focus is on rendering AOM-based architectures, so their context is explicitly stated for that. They

have been written thinking in AOMs and therefore some considerations for other kind of systems may not have been taken into account.

We ultimately see these patterns as part of a more complete pattern language for building Adaptive Object-Models which include these six categories.

- **Core:** includes the core patterns that are present in the basic implementation of AOMs. These are the basic patterns and they are the ones that govern this architectural style.
- **Process:** includes the patterns that deal with the process of creating AOMs. They establish guidelines for evolving frameworks and boundaries to avoid going up to the meta-levels far beyond than necessary.
- **Presentation:** includes the patterns that deal with how to present AOMs to end-users in applications.
- **Creational:** includes the patterns that help to create instances of AOMs
- **Behavioral:** includes the patterns for dynamically adding, removing or modifying behavior to the AOMs
- **Miscellaneous:** includes patterns that help to instrument the usage, control, and instrumentation of AOMs. They also help to provide guidelines for non-functional requirements such as performance and auditability.

Property Renderer

Context

You are creating an application using an Adaptive Object-Model. Your model relies on a variant of TYPE SQUARE and therefore you are using a combination of TYPE OBJECT and PROPERTIES patterns.

You want to render the entities in your model following a standardized and unified approach, minimizing code redundancy and with a unified and consistent look and feel.

Example

Imagine a Content Management system, where new types of Contents can be created dynamically by its users composing several primitive property types. For example, an instance of a Content of type Document may be composed of a property called “name” that is of type “string property”, a property called “description” also of type “string property” and another property called “file” that is of type “binary property”. Our system’s design uses a very common variant of TYPE SQUARE pattern.

We have several applications that use our Document abstraction. They create instances of entities based on the “Document” type object and they show them to the users. Each time we want to render each property, we have to repeat the same rendering code. Therefore, the code for rendering each type of property may be (in the best case) duplicated in all our applications. In the worst case, it would be duplicated in all our applications but also duplicated inside the application (for example, when rendering the name and description properties of the document two different pieces code may be used).

This leads to a higher difficulty of maintenance and potential inconsistency in our UIs since different approaches may exist in each application for rendering the same type of elements (for example, each application may render slightly different the “binary” properties).

Problem

How can we encapsulate how the properties of different types are rendered?

Forces

- An entity may have several properties of different types and the properties can be attached and detached to entities at any time
- We want to ensure consistency in the UI of our application
- We want to encapsulate the rendering code
- We want to avoid duplicate rendering code in our application

- We want our rendering pieces to be composable, to create complex output
- We want to separate the UI code from the model
- We want to vary the way a property is rendered according to the rendering context (target device, state of the application, etc.)
- We don't want to bloat our rendering application with lots of conditional statements according to the rendering context

Solution

Create rendering objects with the responsibility of rendering a certain type of property in a certain context. This will encapsulate the way an instance of a property of a concrete type (when TYPE OBJECT pattern is applied) in a certain context. We will call this special kind of objects "Property Renderers".

The PROPERTY RENDERER contains the necessary code for generating UI code of an instance of a property in a certain context. Therefore it has a strong coupling with the property type (it knows how to handle it) and with the target context (it knows how to generate appropriate code for it).

In order to minimize this coupling we can have a PropertyRenderer that knows how to handle a generalization of all properties and to generate minimal UI code for targeting any average device. This code may not render accurately the property and may not generate nice UI code, but will allow us to get a minimal rendering of any property in any context. These may not be suitable for production code, but can be very helpful for architecting, prototyping or developing very flexible and adaptive systems.

The PROPERTY RENDERERS also help to make a stronger separation between the model and the visualization, isolating all presentation related code from the business model itself.

All property renderers are small pieces specialized in rendering a concrete type of property type. They only deal with the rendering of an instance of a property of a concrete type, but are not concerned on the final destination of the UI code (it may be included in a Web Form, an e-Mail, a report, etc.). They implement a common interface that specifies the contracts that the concrete renderers must address. Since the property renderers are so specific and "fine-grained" they can be combined to create complex UI code (see ENTITY VIEW and DYNAMIC VIEW).

In figure 2 the UML class diagram of the solution is presented. The PropertyRenderer class is the base class for all the renderers and defines their interface. In the UML class diagram below the property renderer has two methods: one for rendering a property (Render method) and other for receiving sets of parameters (SetParameters method). The property renderers can be primitives (stand-alone renderers e.g. strings, numbers, dates, etc.) or composite (which combines several renderers to create the output). The instances of the property renderers are created using a factory (PropertyRendererFactory). Finally there is a Client who uses the property renderers to compose the UI.

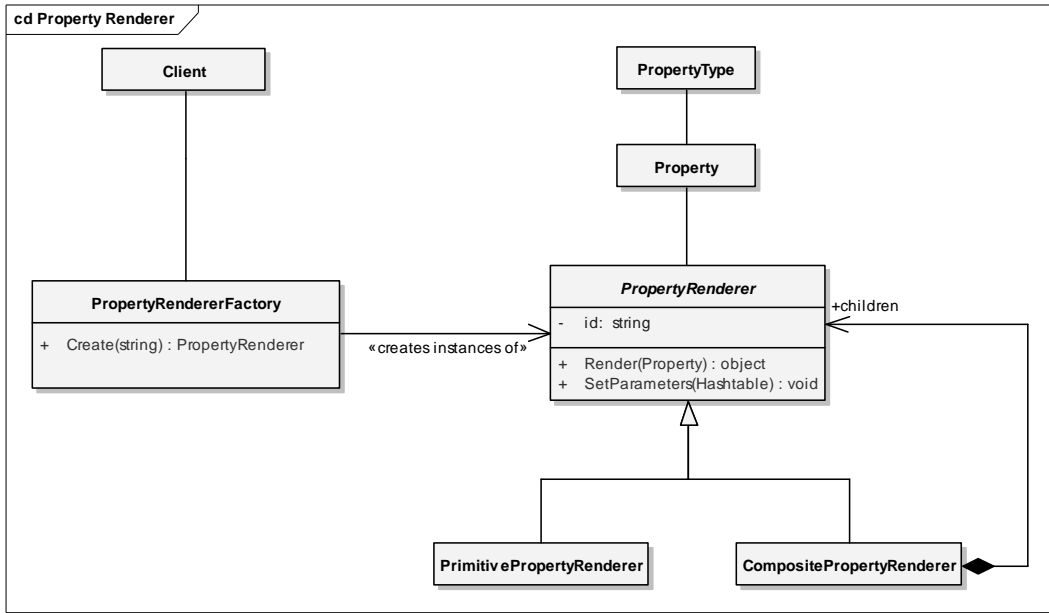


Figure 2 – Property renderer structure

Example Resolved

We can create a Property Renderer for each type of property and use it in all our applications. In our example case, two renderers may be created: one for the “string property” and other for the “binary property”. These renderers may be used in all our applications, giving consistency to our UIs (same elements are rendered in a consistent way in all our applications) and simplifying maintenance (the property rendering code is a single and well-known location).

In figure 3 below some example of property renderers are shown. In the right part of the picture, four property instances are shown (the name of the property is in bold and the property type is in italic below the name). In this example, some property renderers are applied to instances of properties to render data entry UI widgets in a web application. All the properties shown (Title, Description, File and DateCreated) belong to the Document entity type. The Property Renderers create the appropriate UI widgets for the properties. In the figure there are two very interesting facts than can be easily observed: 1) the widgets have a standardized look and feel and behaviour which provides a consistent user experience; and 2) the property renderers can have logic for analyzing certain aspects of the properties and produce the appropriate output (for example, the string property renderer can analyze the length of the text to input and produce a single text box or a text area for data entry).

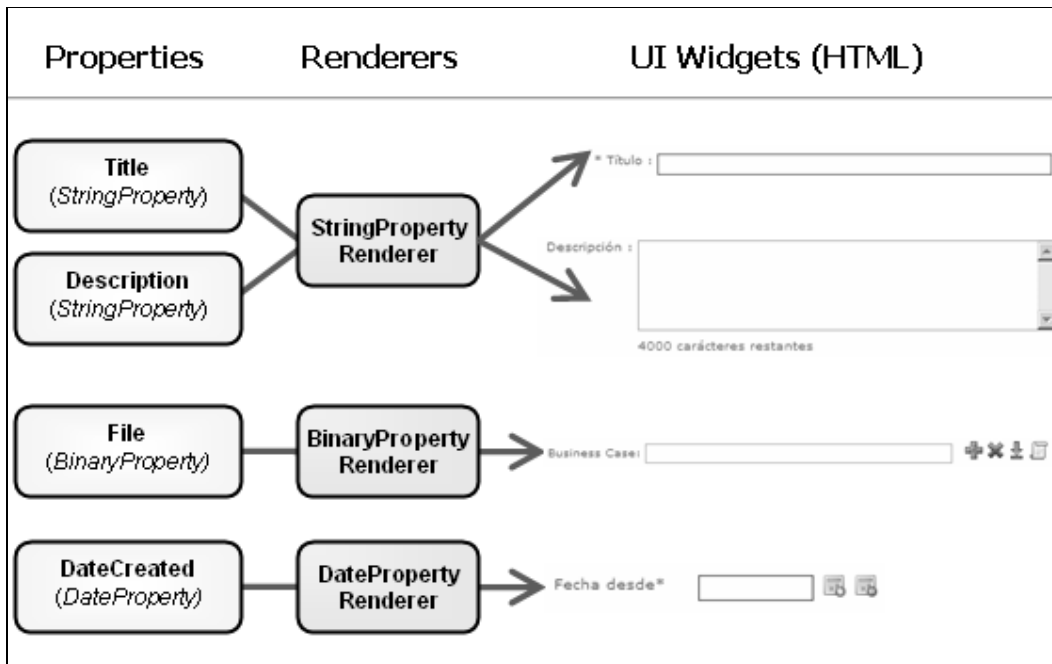


Figure 3 – Sample property renderers for generating data entry HTML UI widgets

Resulting Context

- ✓ Responsibility for rendering instances of properties of concrete types is factored in fine-grained rendering objects
- ✓ UI code is separated from the model (it is encapsulated in the property renderers)
- ✓ UI code can evolve independently from the model
- ✓ New PropertyRenderers can be created allowing dynamic change of how instances of property of a type are rendered
- ✓ PropertyRenderers can contain context-related (target device, purpose, state, etc.) presentation code, eliminating complex conditional code in the UI (e.g.: a different PropertyRenderer may exist for each kind of target device)
- ✓ Since properties are fine-grained elements with very specific responsibilities they can be combined to create complex output
- ✓ The basic PropertyRenderer is a general object that may allow rendering any entity, facilitating prototyping and developing adaptive systems
- ✗ PropertyRenderers have a strong coupling with the property types
- ✗ A PropertyRender has a strong coupling with its rendering context
- ✗ There is more indirection which can lead to lower performance

Related Patterns

PROPERTY RENDERERS can be seen as a special type of STRATEGY (they are only concerned with the generation of UI code for instances of properties of a given property type).

PROPERTY RENDERERS instances should be created using a FACTORY.

PROPERTY RENDERERS instances can be created using a PRODUCT TRADER. In this case, the rules for selecting one renderer or another are not hardcoded in the factory but determined at run-time using Specification objects [BR98].

PROPERTY RENDERERS have code for rendering the PROPERTY TYPES of the PROPERTIES instances when using TYPE SQUARE.

ENTITY VIEW organizes the way several PROPERTY RENDERERS are combined to generate a piece of UI code.

PROPERTY RENDERER performance can be dramatically enhanced using CACHING [POSA3]

Entity View

Context

You are creating an application using an Adaptive Object-Model. Your model relies on a variant of TYPE SQUARE and therefore you are using a combination of TYPE OBJECT and PROPERTIES patterns.

In order to encapsulate and abstract the presentation you are using PROPERTY RENDERER. You have several property renderers and you want to coordinate them for producing a more complex output. This output may be a fragment of the UI or a complete screen.

An Entity contains a set of properties that need to be rendered and might have different views for this Entity.

Example

Consider again our CMS presented previously (see Example section in PROPERTY RENDERER) and the “Document” type entity.

We may want to have several ways of rendering the properties for a Document entity. For instance, we may want to render it as a form (for editing purposes) or as a set of labels (for visualization). We have property renderers for each kind of property, but we will have to coordinate all of them in each screen of our application to produce to desired output.

The above mentioned situation may produced some problems such as a duplicate code (the combination of renderers may be repeated in all client application or in the worst cases several times in the same application) and lack of consistency in client applications (since each application can coordinate the renderers in a different way).

Problem

How can we coordinate several property renderers to render a complex UI fragment for different views of an entity?

Forces

- We have our presentation code encapsulated in property renderers
- We want to combine several property renderers to produce a complex UI fragment for an entity
- The combination of fragments should be easy to change
- The resulting structure should be easy to change

- We don't want to have redundant UI code
- We may want to use different sets of fragments in different contexts (for example, if we are rendering a page to be used in a mobile device we will need to use the appropriate renderers)

Solution

| *Create view components that coordinates the presentation of several property renderers of an entity to produce different complex UI fragments for an entity.* Each property renderer is specialized in the generation of output UI code for instances of a property type in a certain context. This view component will coordinate several fine-grained renderers in order to produce a more complex UI code for an entity.

The sequence and composition of renderers can be specified at source-code level or using metadata (stored in a database or a file). To simplify the coordination of compositions of renderers a Domain Specific Language can be created.

The ENTITY VIEW should be aware of the rendering context (target device, state, etc.) and therefore it must contain instances of the suitable property renderers for a given context. The context may contain additional information to be used in the rendering process.

The ENTITY VIEW may have several constraints (such as validations, rules, etc) that are used in the rendering process of an entity. The developers can create new types of constraints (creating a new specialization of the abstract class EntityViewConstraint) and use them in any EntityView. When the constraints are applied, a variant of the WARNING MESSAGE ACCUMULATOR pattern [Ahluwalia05] can be used and consequently a set of ConstraintResult instances may be returned. It is important to stress that the constraints included are mainly focused on UI issues (like client side validations). Any other business validation or rule enforcement should be delegated in the domain specific constraints in the core AOM instance that is being rendered.

The ENTITY VIEW will be mainly used to generate fragments of the UI for an entity, although it can be used to generate a full page.

In figure 4 the UML class diagram of the solution is presented. The EntityView abstract class defines the public interface and basic behaviour of all entity views. It also contains a set of Property Renderers instances (see the PROPERTY RENDERER pattern in this paper) which are coordinated to generate UI code for an entity instance. The concrete entity views can be leafs (stand-alone views) or composite (they compose several existing entity views to generate the output). The EntityView receives context information through the RenderingContext. There are some Constraints that can be applied to the orchestration process (classes EntityViewConstraint, Validation and Rule). These constraints can be composed to create dynamically complex validation or composition rules.

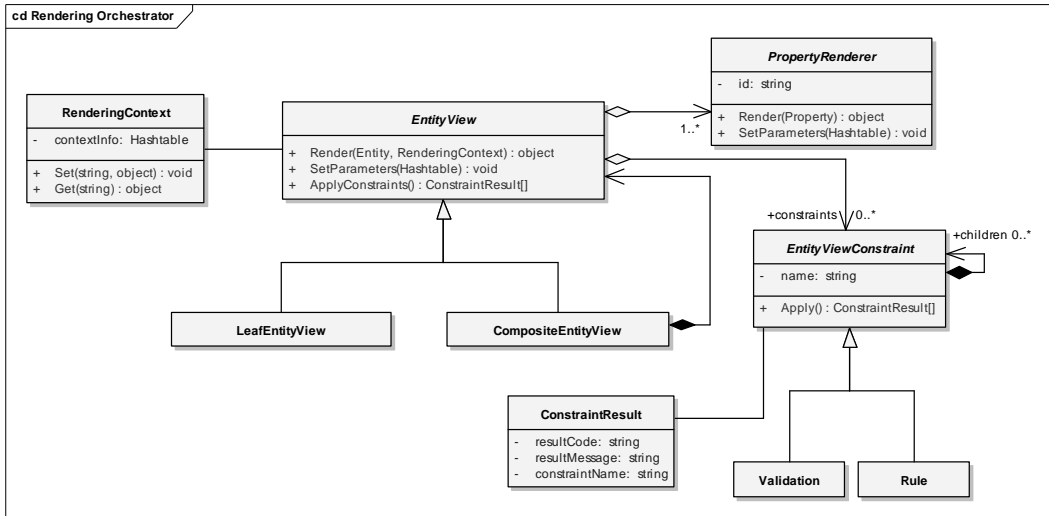


Figure 4 – Entity View Structure

Example Resolved

We can create two Entity Views: one for editing and another for visualizing. These views may be used in all our applications, giving consistency to our UIs (same group of elements are rendered in a consistent way in all our applications) and simplifying maintenance (the property renderers coordination code is a single and well-known location).

In figure 5 below, the two Entity Views are shown: the first one (called EditableEntityView) is used to edit an instance of an entity (in this case we are creating a new Document entity that holds the representation of the paper “Dynamic Object Model” [RTJ05]). Notice how all the editing UI widgets shown in the picture are the same ones shown previously in figure 3 for the PROPERTY RENDERER pattern (this means that this entity view is using the property renderers that are shown in figure 3). The second Entity View (called ReadOnlyEntityView) shown in lower section of the figure renders a read-only representation of the entity, used mainly to display Document entity properties to users. In this case none of the properties of the Document entity can be edited. Notice that this Entity View also shows additional metadata of the entity instance that is presenting.

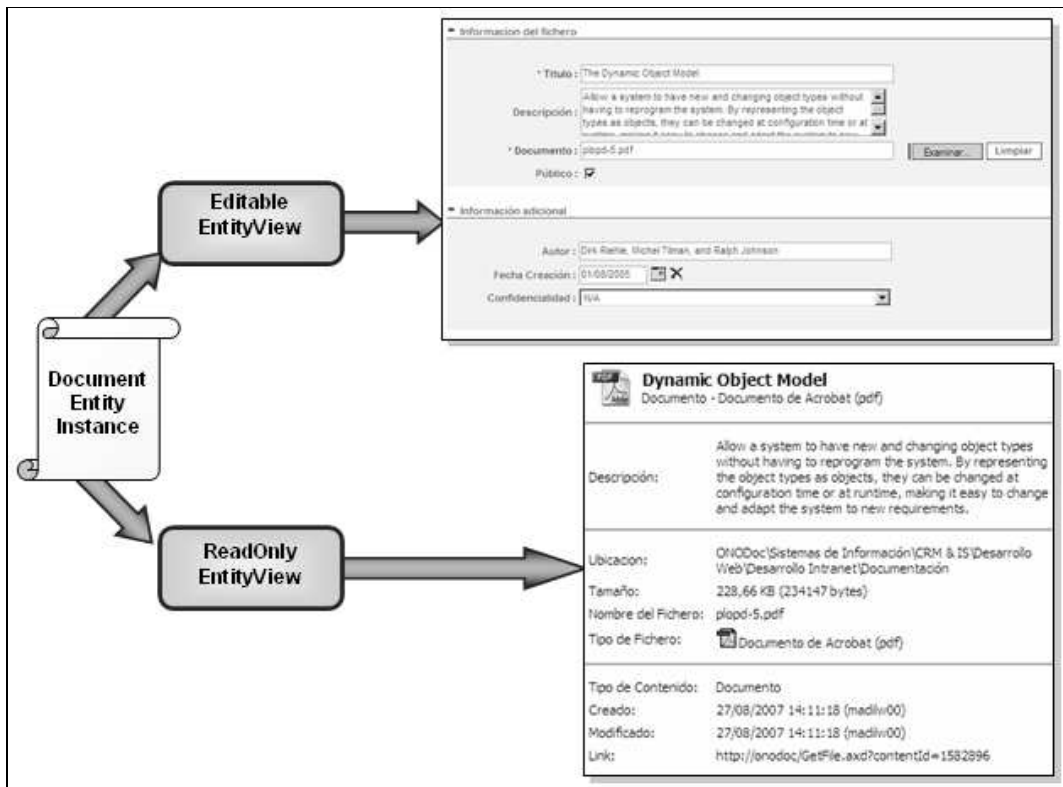


Figure 5 – Entity View example resolved

Resulting Context

- ✓ UI composition for rendering entities can be abstracted, encapsulated and easily modified
- ✓ The rules for showing an entity (an instance of entity in an AOM) can be modified dynamically at runtime
- ✓ The rules for showing an entity (an instance of entity in an AOM) can be modified declaratively (when storing them in metadata)
- ✓ The rules for showing an entity are explicitly stated
- ✓ It is easy to change the way entities are shown
- ✓ Better adaptability to new visualization requirements
- ✓ More Flexibility
- ✗ More complexity
- ✗ Lower performance

Variants

- **Form Entity View:** orchestrates several property renderers to create a form for data input. It may contain also constraints in order to establish validations, rules to show or hide groups of renderers, etc.
- **Table Row Entity View:** orchestrates several property renderers to create a table showing an entity as a row in a grid. In order to show a full grid this Entity View must be applied to a set of entities in a DYNAMIC VIEW.
- **Selection of Fields Entity View:** in this case the view selects a set of the fields of an entity type (or a discrete set of property instances) and generates the output. For example, we can have several views for a type of entity where each view shows a different subset of fields of the entity. For example, in case of an entity type “Patient” we could have an entity renderer that only shows its contact info and another one that shows only the id, the name and the birth date.
- **Full Display Entity View:** this view displays all the fields in the entity type or the provided set of property instances.
- **Rule Based Entity View:** this is a more complex entity view that selects the property renderers to be used using rules. For example, we may have an entity view that shows or hides fields according to profile of the target user.

Related Patterns

An ENTITY VIEW coordinates several PROPERTY RENDERERS.

ENTITY VIEW can be seen as a typed COMPOSITE of PROPERTY RENDERERS for displaying entities.

ENTITY VIEW generates output using PROPERTY RENDERERS; DYNAMIC VIEWS can generate totally custom UI manually or using renderers.

RENDERING ORCHESTRATOR performance can be dramatically enhanced using CACHING [POSA3]

Dynamic View

Context

You are creating an application using an Adaptive Object-Model. Your model relies on a variant of TYPE SQUARE and therefore you are using a combination of TYPE OBJECT and PROPERTIES patterns.

You want to generate UI code for a set of entities but you don't want to have any kind of coupling or to reference the UI in your model. Additionally you may want to attach or detach views to models, allowing different views of the same entity (the views can be selected dynamically). You want to have several views applied to the same model and you want to have the possibility of selecting any of them according to arbitrary decisions.

Example

We are developing a Web-based Content Management application (the one quoted in the Property Renderer pattern). We built a Document Management module on top of our CMS engine. Our content management module has entities of Document and Link type that are contained in Categories (which is a special kind of entity that contains other entities). We will use the categories to simulate Folders in our document management module.

Whenever a user selects one Folder, we want to display its contents (the contained entities) in several ways according to a given context. We also want to be able to attach and detach views to the folders. For example, we can have a thumbnails view that we may only want to apply to folders that contains images. We want to easily link (and unlink) the views to the categories, allowing our users to do this according to their preferences.

Having the UI generation code stale on a web page is not a very good idea. This is primarily due to the fact that it complicates the abstraction of the rendering algorithm in order to reuse it in different contexts. Additionally, if we want to reuse the code for UI generation in another application we won't be able to, since it is contained in a page and therefore cannot be abstracted as a reusable artifact in other applications (in the best case, we can copy the page, but if we want to change a single feature, we will need to modify ALL instances of the page in ALL client applications).

Problem

How can we abstract the visualization (including the complex layout) of a set of dynamic entities from an AOM in a way to decouple the visualization from the model?

Forces

- We want to be able to attach and detach views dynamically to sets of entities

- We want to abstract layout details
- We want to render a set of entities all together
- We want to reuse the rendering code in several contexts
- We don't want to have redundant UI code
- We want to have control of all the generated UI code
- We may not be using PROPERTY RENDERERS or ENTITY VIEWS
- When using PROPERTY RENDERERS or ENTITY VIEWS we may want to add additional UI code (layout code, glue code to give consistency and context to the renderer properties or maybe code not related at all with the entities)

Solution

Abstract the UI code generation in a view component that may receive as input a set of entities and provide as output the UI code.

The dynamic view is a component specialized in generating UI code for a set of entities (the set can also contain one element). It receives as input the set of entities and returns as output the appropriate UI code according to the purpose of the view. As in MVC, the view components present information to the user. Different views present the information in the model in different ways.

The Dynamic View contains code for complex layout purposes. The layout code may allow dynamic set up and modification of the layout (for example like the models in WinForms [MSNET] or Swing [Swing]) or may be code specially written for laying out a set of entities (the layout is hard-coded in the view).

The views can generate all code from scratch or can use PROPERTY RENDERERS and ENTITY VIEWS.

Several views may exist that can be applicable for the same set of entities. The views can be linked to the entities (and entity types) dynamically, allowing easy run-time adaptation and leveraging the creation of multiple-view based interfaces.

In figure 6 the UML class diagram of the solution is presented. The BaseView abstract class defines the public interface and basic behaviour of all Dynamic Views. The concrete Dynamic Views can generate their output using several approaches: using Property Renderers (ConcreteViewA), using Entity Views (ConcreteViewB) or generating all UI code on their own (ConcreteViewC).

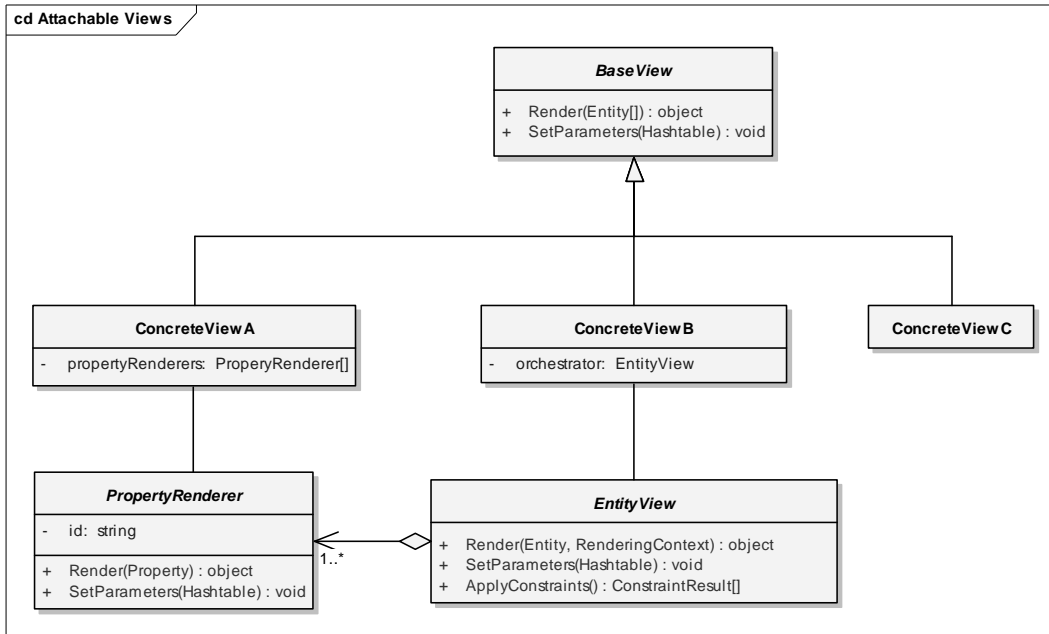


Figure 6 – Dynamic views structure

Example Resolved

We can abstract the UI rendering code in Dynamic Views and store them in a views repository. We can then link the existing views to existing entities in order to generate UI code for them.

In our case, we can create several views (for example, Details View, Icons View and Thumbnails View) and then link them to the categories that represent the folders. When the user selects a Folder to view its contents, it is displayed on a container that allows the user to select any of the views attached to the folder. Whenever the user selects one of them it generates the appropriate UI code (delegating in the concrete View) as shown in figure 7 below.

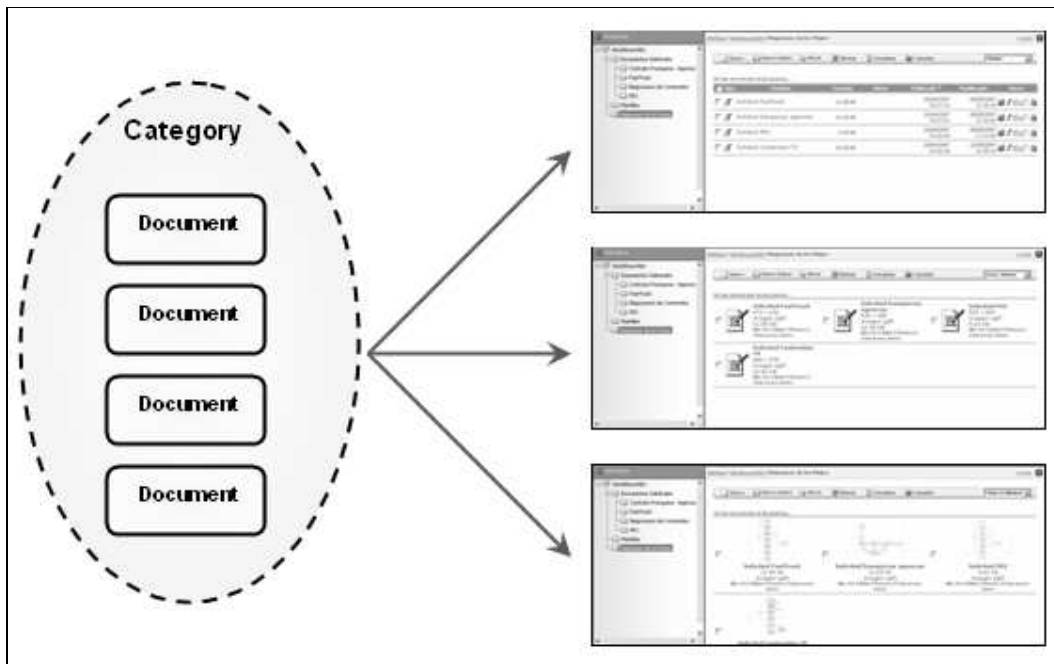


Figure 7 – Several views applied to the same set of entities.

In this example, a set of documents is rendered in several ways (detailed list, big icons, and thumbnails).

We can have more views and attach them to any category. For example, for a particular set of folders we may need to have some special rendering logic such as hiding documents older than three weeks. To achieve this we can create a new view and attach it to the appropriate folders.

Resulting Context

- ✓ UI composition can be abstracted, encapsulated and easily modified
- ✓ The rules for showing sets of entities can be modified dynamically at runtime
- ✓ The rules for showing sets of entities can be modified declaratively (when storing them in metadata)
- ✓ The rules for showing sets of entities are explicitly stated
- ✓ Is very easy to change the way sets of entities are shown
- ✓ Better adaptability to new visualization requirements
- ✓ More flexibility
- ✗ More complexity
- ✗ Lower performance

Related Patterns

DYNAMIC VIEWS can use several PROPERTY RENDERERS.

DYNAMIC VIEW can use several ENTITY VIEWS.

DYNAMIC VIEW instances should be created using a FACTORY.

DYNAMIC VIEW can be seen as a special type of STRATEGY that is only concerned with the generation of UI code for groups of entities.

A DYNAMIC VIEW can be applied in MODEL VIEW CONTROLLER [POSA1] scenarios.

DYNAMIC VIEW performance can be dramatically enhanced using CACHING [POSA3].

Putting It All Together

In this paper we presented a set of patterns for dealing with dynamic presentation issues when building Adaptive Object-Models (AOMs). Each pattern presented in this paper address the rendering problem at a different level of granularity (as shown in figure 8).

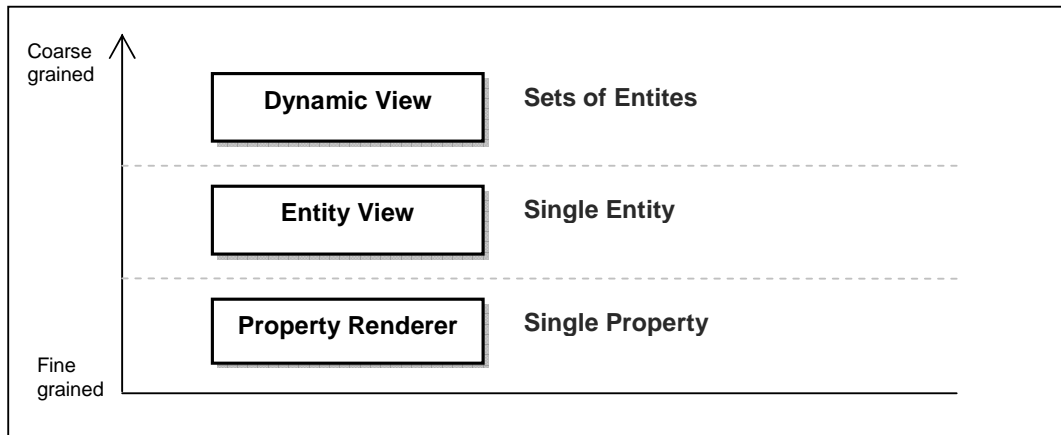


Figure 8 – Granularity level of the patterns in the language

We are building an application on top of a CMS system that is based in an AOM. In our CMS we created a Document entity type that contains several properties for storing the title, description, binary file (e.g. word, pdf, excel, etc.), creation date and author of a document. These Document entity types are stored in Categories, which are abstractions that gather several instances of entites (in our case Document entities). We want to create a consistent UI decoupled from the application logic that can be easily changed and reused through out this application or other systems.

Since we want to render consistently all the properties of similar types, we determined to use the PROPERTY RENDERER pattern (page 5) to generate the UI widgets for each property type. The first step is to create a Property Renderer for each Property Type that we have for the Document: one for strings, another for binaries and one for dates. If we think deeper on this we will quickly realize that this is not enough: actually, in some cases, we need two renderers for each property type, one for editing and one just for visualizing. Therefore, we created six property renderers: (StringInputPropertyRenderer, FileInputPropertyRenderer, DateInputPropertyRenderer, StringPropertyRenderer, FilePropertyRenderer, and DatePropertyRenderer).

After our renderers are created, we need to establish how we are going to present our Document entities to the end users. In this case we used the ENTITY VIEW pattern (page 10) to generate the UI for the entities. We applied the ENTITY VIEW pattern three times to create the following views: FormDocumentEntityView (for creating and editing documents), ReadOnlyEditableEntityView (for viewing instances of Document entities), and TableRowDocumentEntityView (for rendering a row for a table of entities). As can be noticed, the Entity Views named above have been addressed previously in this paper in the Variants section of the Entity View pattern (page 14).

These patterns work together to provide a consistent and reusable way for rendering properties and entities to end users. However, rendering concrete properties or entities is not enough to create the UI for the document management application. In order to fill this final gap we need to use the DYNAMIC VIEW pattern (page 15) to create several coherence pieces of UI for entering and retrieving Document entity instances. To do so we created several Dynamic Views that will use the Property Renderers and Entity Views outlined in the previous steps. These views can be dynamically linked to sets of Document entities to produce full autonomous and consistent UI fragments. The Dynamic Views have content layout code like in the case of the DocumentGridDynamicView (that uses several TableRowDocumentEntityView for generating an HTML table of Document entities).

There is a very important issue in the solution presented here: performance and resource usage can be prohibitive, leading to bad user experience and degradation of service scenarios (especially for web applications). In order to cope with this we propose the careful use of CACHING [POSA3]. We propose several levels of caching according to what we are trying to render: we can have caches for a property type (applied to PROPERTY RENDERER), for an entity (applied to ENTITY VIEW), or for set of entities (applied to DYNAMIC VIEW). The decision on how to apply caching should be taken carefully, keeping always in mind that caching adds considerable complexity to an application. In order to enhance the performance and resource usage of the application other patterns can be applied (like POOLING, LAZY ACQUISITION, etc. [POSA3]).

There are also some other high level patterns for dynamic screen layout of the entities and properties which have not been addressed in this paper. The authors intend on addressing these patterns at a later date.

Acknowledgements

We would like to thank our shepherd Dirk Riehle for his great help and advice to improve the contents of this paper. Leon Welicki wants to thank ONO (Cableuropa S.A.) for their support of this research.

References

- [Ahluwalia05] Ahluwalia, K. *Warning Message Accumulator Pattern*. 13th Pattern Language of Programs Conference (PLoP 2005), Monticello, Illinois, USA, 2005.
- [BR98] Bäumer, D ; D. Riehle. *Product Trader*. Pattern Languages of Program Design 3. Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998.
- [Fowler97] Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley. 1997
- [FY98] Foote B, J. Yoder. *Metadata and Active Object Models*. Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998.
- [GoF95] Gamma, E.; R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 1995.
- [Swing] Trail: Creating a GUI with JFC/Swing
<http://java.sun.com/docs/books/tutorial/uiswing/>

- [JW98] Johnson, R., R. Wolf. *Type Object*. Pattern Languages of Program Design 3. Addison-Wesley, 1998.
- [KSS05] Krishna, A., D.C Schmidt, M Stal. *Context Object: A Design Pattern for Efficient Middleware Request Processing*. 13th Pattern Language of Programs Conference (PLoP 2005), Monticello, Illinois, USA, 2005.
- [MSNET] Microsoft .NET Framework.
<http://www.microsoft.com/net/>
- [POSA1] Buschman, F. et al. *Pattern Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons. 1996
- [POSA3] Kircher, M.; P. Jain. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley & Sons. 2004.
- [RFBO01] Riehle, D., Fraleigh S., Bucka-Lassen D., Omorogbe N. *The Architecture of a UML Virtual Machine*. Proceedings of the 2001 Conference on Object-Oriented Program Systems, Languages and Applications (OOPSLA '01), October 2001
- [RTJ05] Riehle D., M. Tilman, and R. Johnson. "Dynamic Object Model." In *Pattern Languages of Program Design 5*. Edited by Dragos Manolescu, Markus Völter, James Noble. Reading, MA: Addison-Wesley, 2005.
- [RY01] Revault, N, J. Yoder. *Adaptive Object-Models and Metamodeling Techniques Workshop Results*. Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001). Budapest, Hungary. 2001.
- [Welicki06] Welicki, L.. *The Configuration Data Caching Pattern*. 14th Pattern Language of Programs Conference (PLoP 2006), Portland, Oregon, USA, 2006.
- [Welicki06b] Welicki, L. et al. *Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models*. EuroPLoP 2006, Munich, Germany, July 2006.
- [Welicki07] Welicki et Al. *Improving Performance and Server Resource Usage with Page Fragment Caching in Distributed Web Servers*. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2007), Las Vegas, Nevada, June 2007.
- [YBJ01] Yoder, J.; F. Balaguer; R. Johnson. *Architecture and Design of Adaptive Object-Models*. Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001), Tampa, Florida, USA, 2001.
- [YJ02] Yoder, J.; R. Johnson. *The Adaptive Object-Model Architectural Style*. IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (WICSA 2002), Montréal, Québec, Canada, 2002
- [YR00] Yoder, J.; R. Razavi. *Metadata and Adaptive Object-Models*. ECOOP Workshops (ECOOP 2000), Cannes, France, 2000.