Rendering Patterns for Adaptive Object-Models

León Welicki
ONO (Cableuropa S.A.)
Basauri, 7-9
28023, Madrid, Spain
+34 637 879 258
lwelicki@acm.org

Joseph W. Yoder
The Refactory, Inc.
7 Florida Drive
Urbana, Illinois USA 61801
1-217-344-4847
joe@refactory.com

Rebecca Wirfs-Brock
Wirfs-Brock Associates
24003 S.W. Baker Road
Sherwood, Oregon USA
1-503-625-9529
rebecca@wirfs-brock.com

ABSTRACT
An Adaptive Object-Model is an instance-based software system that represents domain-specific classes, attributes, relationships, and behavior using metadata.. This paper presents three patterns for visually presenting and manipulating AOM domain entity objects.

Categories and Subject Descriptors
D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 **[Design Tools and Techniques**]: Object-oriented design methods, User Interfaces; D.2.11 **[Software Architectures]**: Patterns

General Terms
Design

Keywords
Visual Rendering, Adaptive Object-Models, Patterns

1. INTRODUCTION
An Adaptive Object-Model is an instance-based software system that represents domain-specific classes, attributes, relationships, and behavior using metadata [19, 20]. Typically in an Adaptive Object-Model, metadata descriptions are stored in a database and interpreted at runtime. This is similar to a UML Virtual Machine described by [12]. The object model is adaptable and tools are often provided with AOM systems that allow end users or domain experts to edit and change these metadata descriptions. So when changing requirements cause the domain model to be updated, end users edit metadata. These changes can immediately be reflected in the running system without any software program changes.

In contrast, in a typical object-oriented program, classes are designed to represent domain entities and their attributes. A change in requirements that results in changes to the domain model causes developers to modify and/or add new classes, leading to a new application version.

Adaptive Object-Model architectures are typically made up of several interrelated patterns. TYPE OBJECT [8] is used to define a domain entity. An entity has attributes, which are represented using the PROPERTY pattern [5]. The TYPE OBJECT pattern is used again to define the legal types of attributes, called PropertyTypes. Thus Entity, EntityType, Property, and PropertyType are the core set of constructs used to represent Adaptive Object-Models [13].

An Adaptive Object-Model expresses relationships between entities using metadata. Any rules and constraints governing these relationships can also be described with metadata. In contrast, with traditional object-oriented programs, relationships between domain entity objects are implemented via a direct reference or an appropriate structuring object (e.g. hash table or a collection). Constraints on relationships are implemented by methods in related classes.

In an Adaptive Object-Model, the STRATEGY pattern [6] can be used to define the behavior of EntityTypes. If behavior is complex, instead of using Strategies, an interpreted rule-based language can be defined. In contrast, with a typical object-oriented programming language implementation of an entity, simple behavior is typically implemented in class methods.

The above core AOM patterns have been described previously. One area that has not been described are how to implement the user interface in an AOM system. Since an AOM is instance based rather than class based and has metadata which drives domain entity behavior, interpretation of the entities needs to be considered when constructing a user interface. This paper describes patterns for dynamically building the GUI layer which supports the modification and visualization of AOM domain objects.

2. TOWARDS AN ADAPTIVE OBJECT-MODEL PATTERN LANGUAGE
Adaptive Object-Model architectures are usually made up of several smaller patterns. In the existing literature they are documented by the patterns TYPE OBJECT, ATTRIBUTES, PROPERTY LIST, TYPE SQUARE, ACCOUNTABILITY (Entity-Relationship), STRATEGY, RULE OBJECTS, COMPOSITE, BUILDER, and INTERPRETER.

Besides these patterns, less widely-known patterns are often used in AOM systems. In the AOM current literature descriptions of these other patterns are scattered among a number of different

PLoP'07, September 5-8, Monticello, IL, U.S.A..

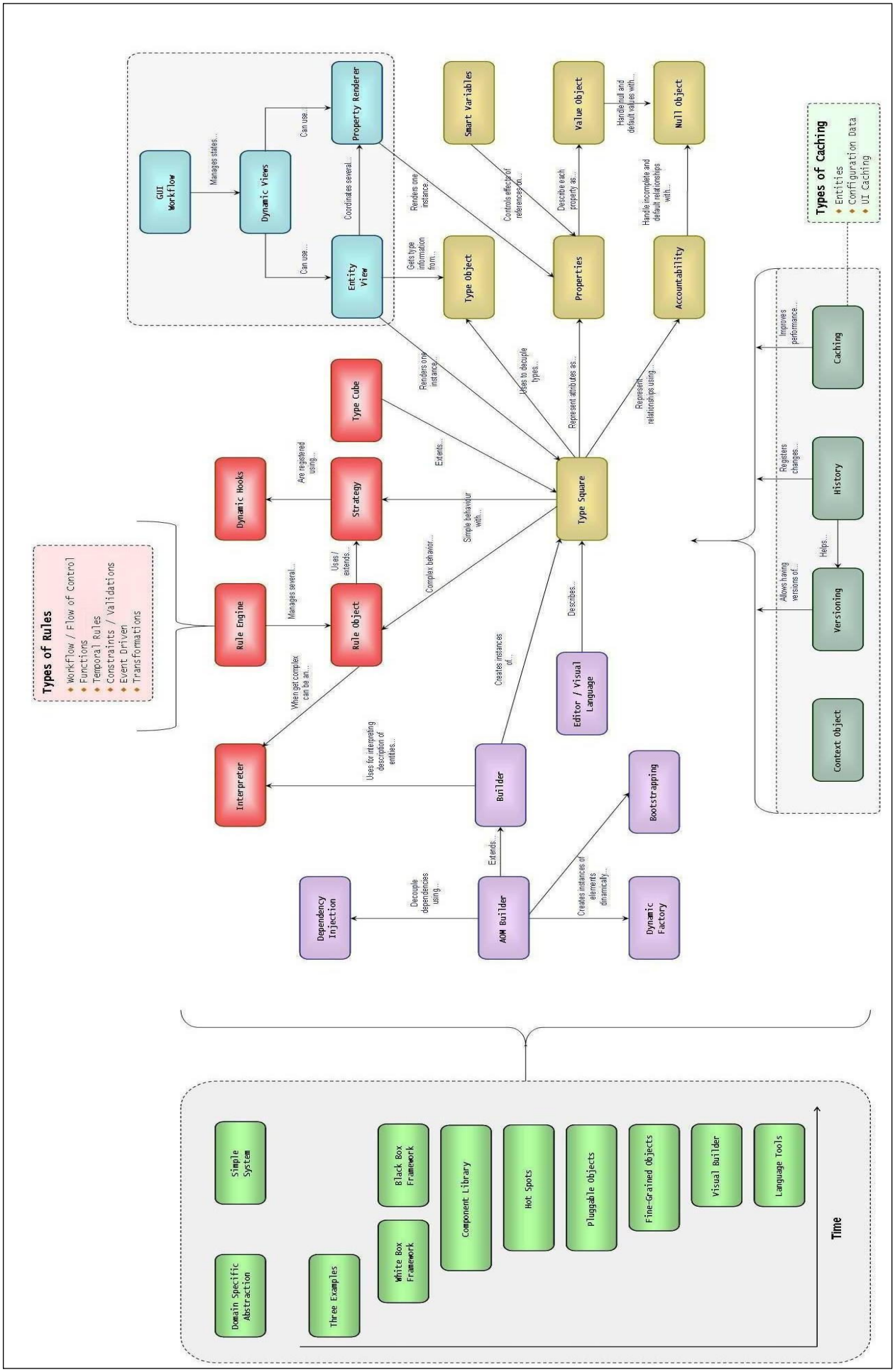

Figure 1 - AOM Pattern Language Map

patterns papers with different templates and styles. Additionally, not all these papers work through similar examples. Some patterns have not been updated to reflect implementation trends, new programming language environments or development platforms. We ultimately see the patterns described in this paper as part of a more complete pattern language for building Adaptive Object-Models. Patterns in this pattern language are organized into these categories:

Core: patterns that represent the basic implementation of AOM entities, their behavior and relationships. They are the ones that govern this architectural style.

Process: patterns that describe how to create and evolve AOM systems. They establish guidelines and advice on when and where the use of meta-description based approaches is warranted.

Presentation: patterns which describe how to visually represent and manipulate objects representing domain entities, their attributes and relationships to other objects in an AOM.

Creational: patterns for creating instances of AOMs

Behavioral: patterns for dynamically adding, removing or modifying AOM system behavior.

Miscellaneous: patterns for instrumentation, usage, and version control of AOMs. These patterns also provide mechanisms which support non-functional requirements such as performance or auditing.

Figure 1 from [19] is a map of our AOM pattern language as presented at the OOPSLA 2007 Poster Session.

2.1 THE AOM VISUALIZATION LAYER

In the existing literature [5, 14, 20, 21, 22], the core architecture of AOMs is represented by two different levels:

Knowledge Level: which defines the general rules that govern the behavior [4] and the structure of domain entities (TypeObjects, PropertyTypes).

Operational Level: which contains instances of the domain (in our case, instances of entities and properties for representing values) whose behavior is governed by associated objects in the knowledge level [4]

Because of the way objects are represented in the operational level, a specialized rendering layer is almost always needed. It consists of "instructions" for how to construct the UI which presents AOM domain objects for viewing and modification. This visualization behavior is embodied in rendering components which can be composed at runtime (from configuration information) and combined dynamically (and adaptively) to generate complex views of AOM domain objects. Separating this behavior into a rendering layer allows us to abstract and encapsulate presentation issues [17].

3. AOM RENDERING PATTERNS

This paper contains the following patterns:

Property Renderer: describes how to render the UI code for instances of specific properties using their data.

Entity View: describes the coordination of several property renderers to produce more complex UI fragments for an entity (rendered from descriptive data from the type objects).

Entity-Group View: given a set of entities, renders UI code (including layout issues). Several different views can exist for the same set of entities and they can be linked dynamically at run-time (useful to present a set of entities).

The patterns presented in this paper are interrelated. While they can be used individually, more commonly they are used in

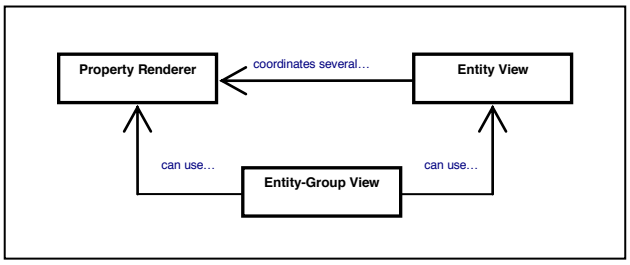

Figure 2 - Rendering Patterns Map

combination. Figure 2 shows the relationships between these rendering patterns. The most fine-grained pattern is PROPERTY RENDERER, which renders individual property instances. It is connected with all the other patterns: an ENTITY VIEW coordinates several PROPERTY RENDERERS to generate a fragment of a UI for an entity, and the ENTITY-GROUP VIEW uses these elements to render a set of entities. The ENTITY-GROUP VIEW is coarser-grained, since it generates a coherent UI for related entities. To implement its behavior it can either use the other patterns or be hand coded.

Patterns presented in this paper discuss presentation concerns which arise when working with AOMs. Therefore, any developer working with this kind of systems (mainly TYPE OBJECT, PROPERTIES [8], and TYPE-SQUARE [20] or DYNAMIC OBJECT MODEL [13] based architectures) can benefit from using these patterns to visually represent and manipulate AOMs.

These rendering patterns may apply also to other rendering scenarios, but our main focus is on AOM-based architectures. We describe these patterns in that context. Use of these patterns in other contexts is outside the scope of this paper.

If you are unfamiliar with Adaptive Object-Model based systems, you will need to become familiar with the core AOM patterns before you can appreciate the rendering patterns described in this paper. An appendix at the end of this paper briefly explains the core concepts and patterns of AOM systems. We invite the reader visit to www.adaptiveobjectmodel.com for additional publications that offer more comprehensive discussions and examples.

3.1 Shared Pattern Context

All the patterns in this paper share the same basic context scenario: *You are creating an application using an Adaptive Object-Model. This model relies on a variant of TYPE SQUARE and therefore you are using a combination of TYPE OBJECT and PROPERTIES patterns*. Each pattern then adds its own issues and forces to this general context and presents a problem accompanied with its respective solution.

3.2 Property Renderer

3.2.1 Context

You want to render the entities in your model using a standardized approach. You want to minimize code redundancy and present a GUI with a consistent look and feel.

3.2.2 Example

Imagine you've created a Content Management System (CMS) using AOM patterns. New content types can be created by end users. A tool allows them to compose several pre-defined 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 "binaryElement" that is of type "binary property". Your system is implemented using common variant of TYPE SQUARE pattern.

You have several applications that rely on this Document entity. Application create instances of entities based on the "Document" type object and present them to users in a UI. Each time you want to render a property, you have to write very similar rendering code. Your code for rendering each type of property may in the best case be duplicated in all your applications. In the worst case, it may be duplicated in each visualization layer as well as additional application code (for example, when rendering the name and description properties of the document two different pieces of very similar code may be invoked).

This redundancy leads to a higher degree of maintenance and potential inconsistency in your UIs. Slightly different approaches may exist in each application for rendering the same type of properties (for example, each application may render differently the "binary" properties).

3.2.3 Problem

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

3.2.4 Forces

- An entity may have several properties of different types and the properties can be attached to and detached from entities at any time.
- You want to ensure UI consistency across applications.
- You want to encapsulate the rendering code.
- You want to avoid rendering code duplication.
- You want rendering aspects to be composable into more complex visual representations.
- You want your UI code to evolve independently of entities.
- You want to vary the way a property is rendered according to its rendering context (e.g., target device, state of the application, client options, etc.).
- You don't want to bloat rendering code with conditional statements to handle each rendering context.

3.2.5 Solution

Create rendering objects that have the responsibility for rendering the UI for a certain type of property within a given context. Each rendering object will encapsulate the way an instance of a property of a concrete type (when TYPE OBJECT pattern is applied) is visualized in a certain context. We call these objects "Property Renderers".

A PROPERTY RENDERER contains code that generates the UI for an instance of a property in a particular context. This renderer is coupled both with the property type (it knows how to handle it) and the target context (it knows how to generate appropriate UI code for it).

To start, provide a default PROPERTY RENDERER implementation that generally knows how to interpret all properties and to generate minimal UI code targeted to a prototypical context. This default implementation may not accurately render the property or generate nice UI code, but allows you to marginally render any property in any context. While this default implementation is likely not suitable for production code, it can be useful when prototyping or evolving an adaptive system.

Then, define individual PROPERTY RENDERERS as needed.

Each PROPERTY RENDERER will be responsible for rendering a concrete visualization of some specific property type, and may be specialized for a concrete context (it may be included in a Web Form, an e-Mail, a report, etc.). Since individual property renderers are specific and "fine-grained" they can be combined to create complex UI visualizations (see ENTITY VIEW and ENTITY-

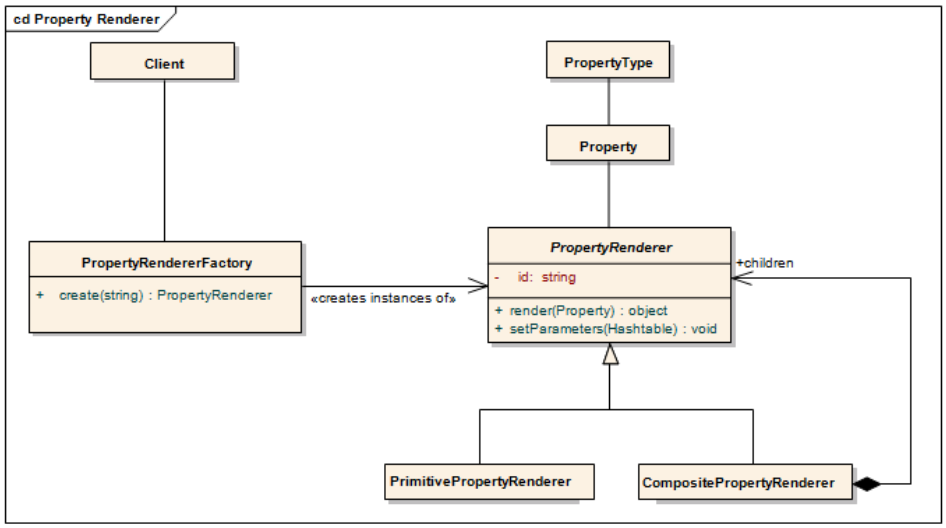

Figure 3 - Property Renderer Structure

GROUP VIEW).

PROPERTY RENDERERS enforce a strong separation between the domain entities and their visualization, isolating all presentation related code into distinct objects.

Figure 3 shows the UML class diagram of the solution. `PropertyRenderer` is the base class for all the renderers and provided default implementations for behaviors that each concrete renderer must redefine. As shown, `PropertyRenderer` has two methods: one for rendering a property, `render`(), and other for receiving sets of parameters `setParameters`() (parameters can be any arbitrary piece of data to be used in the rendering process). Subclasses of `PropertyRenderer` can either be primitive (stand-alone renderers of strings, numbers, dates, etc.) or composite (combining several renderers to create more complex output). Instances of PROPERTY RENDERERS are created using a factory (`PropertyRendererFactory`). Finally, the `Client` uses the PROPERTY RENDERERS to compose the UI.

3.2.6 Example Resolved

Thus you can create a `PropertyRenderer` class for each type of `Property` and use it in all applications. In our example, two renderers may be created: one for the `StringProperty` and other for the `BinaryProperty`. These renderers may be used in all applications, giving consistency to their UIs and simplifying maintenance (the property rendering code is in a single well-known location).

In Figure 4 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, BinaryElement and DateCreated) belong to the `Document` entity type. The `PropertyRenderers` create the appropriate UI elements for the properties. Note that in the example: 1) the UI elements have a standardized look and feel and behavior which provides a consistent user experience; and 2) the property renderers could also contain additional logic which analyzes certain characteristics of the properties used when producing the appropriate UI elements (for example, a string property renderer might analyze the length of the input text and produce an appropriately sized single text box or a text area for data entry).

3.2.7 Resulting Context

- Responsibility for rendering instances of properties of concrete types is assigned to fine-grained rendering objects.
- UI code is separated from entities and encapsulated in specialized property renderers.
- UI code can evolve independently from the model consisting of entities, properties and relationships between them.
- New `PropertyRenderers` can be created, allowing for dynamic change in how instances of property of a specific 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` might exist for each kind of target device).
- Since properties are fine-grained elements with specific responsibilities they can be easily combined to create more complex visual representations.
- The base `PropertyRenderer` class provides a generic implementation that allows for rendering any entity, facilitating prototyping and evolving adaptive systems.
- A `PropertyRenderer` is strongly coupled with its respective `PropertyType`.
- A `PropertyRenderer` is coupled to its rendering context.
- The indirection found in this solution can lead to lower performance than in a non-AOM system.

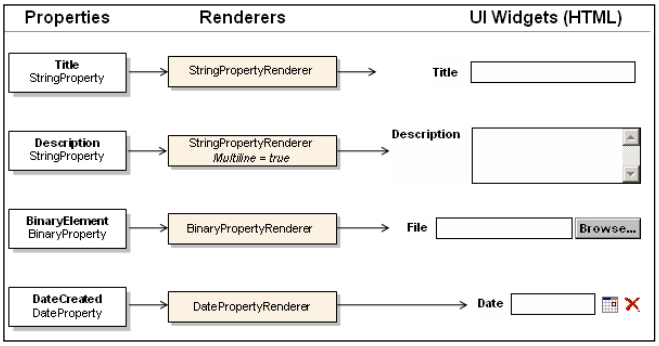

Figure 4 - Example Property Renderers for Generating Data Entry HTML UI Widgets

3.2.8 Related Patterns

PROPERTY RENDERERS are a special type of STRATEGY concerned with the generation of UI code for instances of properties of a given property type.

PROPERTY RENDERERS instances can be created using a FACTORY.

PROPERTY RENDERERS instances can be created using a PRODUCT TRADER. If so, the rules for selecting one renderer or another are not hardcoded in the factory but determined at run-time using Specification objects [3].

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 UI code fragment.

PROPERTY RENDERER performance can be improved using CACHING [11].

PROPERTY RENDERER can be combined with FLYWEIGHT [6] to improve performance and resource utilization of pre-allocated rendering instances.

ANYTHING [15] have a similar abstraction called Renderers, but with a more broad scope. If you want to use this pattern to render ANYTHING instances, the PROPERTY RENDERER can be seen as specialized instance of such renderers.

3.3 Entity View

3.3.1 Context

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

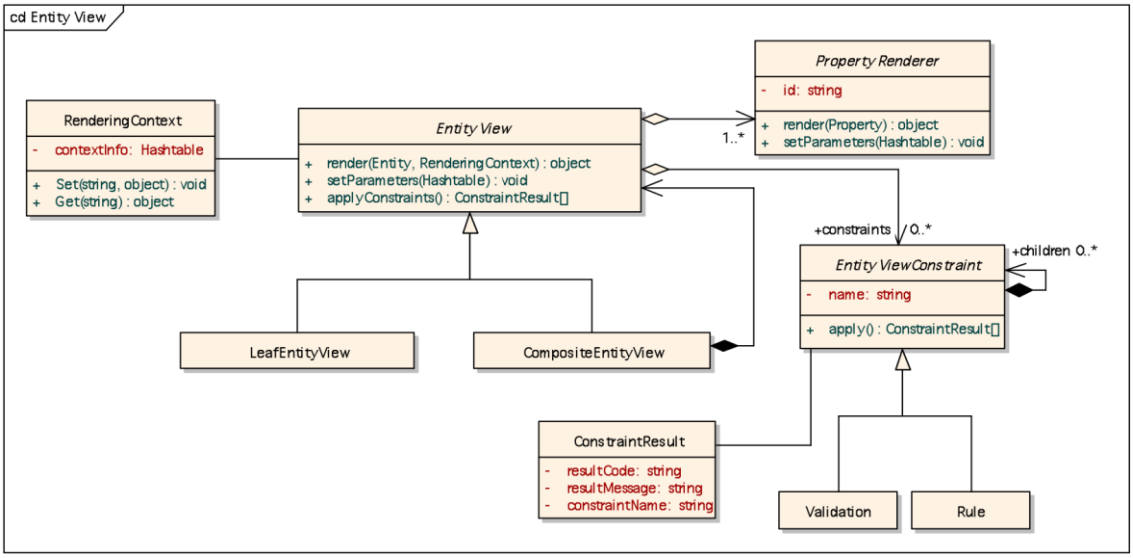

Figure 5 - Entity View Structure

An entity contains one or more properties that need to be rendered and might have different views.

3.3.2 Example

Consider again the CMS example presented previously (see Example section in PROPERTY RENDERER) and the Document entity.

You may want several ways to render the properties for a Document entity. For instance, you may want to render it as a form for editing purposes or as a set of text fields for visualization. You have property renderers for each kind of property, but you will have to coordinate each screen to produce desired behavior. This could result in duplicate code within the same application or lack of consistency across applications.

3.3.3 Problem

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

3.3.4 Forces

- You want to combine several property renderers to produce a complex UI fragment for an entity.
- UI fragments should be easy to change.
- The resulting structure should be easy to change.
- You don't want redundant UI code.
- You may want to use different sets of fragments in different contexts (for example, you may use different renders for a mobile device than for a web browser).

3.3.5 Solution

Create view components which coordinate the presentation of several property renderers of an entity to produce different complex UI fragments. Each property renderer is specialized to generate UI code for instances of a property type in a certain context. A view component will coordinate several fine-grained renderers and produce more complex UI code for an entity.

The sequence and composition of renderers could be specified using source code or with metadata stored in a database or a file. To simplify the coordination of compositions of renderers a Domain Specific Language might be created.

The ENTITY VIEW is aware of its rendering context (target device, state, etc.) and therefore must contain instances of the suitable property renderers for that context. It may also contain additional contextual information used when rendering.

The ENTITY VIEW may have several constraints (such as validations, rules, etc) that are used while rendering an entity. You can create new types of constraints, by creating a new specialization of the abstract class `EntityViewConstraint`, for use in an `EntityView`. When the constraints are applied, a variant of the WARNING MESSAGE ACCUMULATOR pattern [1] can be used and consequently a set of `ConstraintResult` instances may be returned. It is important to stress that the constraints included are focused on UI concerns such as client side data validations. Any other business validation or rule enforcement should be delegated to the domain specific constraints associated with the core AOM instance being rendered and not be located in presentation-layer code.

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

Figure 5 presents the UML class diagram of the solution. The abstract class `EntityView` defines the public interface and basic behavior of all entity views. It also maintains a set of `PropertyRenderer` instances (see the PROPERTY RENDERER pattern in this paper) which are coordinated to generate UI code for an entity instance. The concrete `EntityViews` can be leafs (stand-alone views) or composite (composing several entity views to generate the output). An `EntityView` receives context information from its associated `RenderingContext`. Some constraints 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.

3.3.6 Example Resolved

You can create two different kinds of `EntityViews`: ones for editing and others for visualizing. These views may be used in all

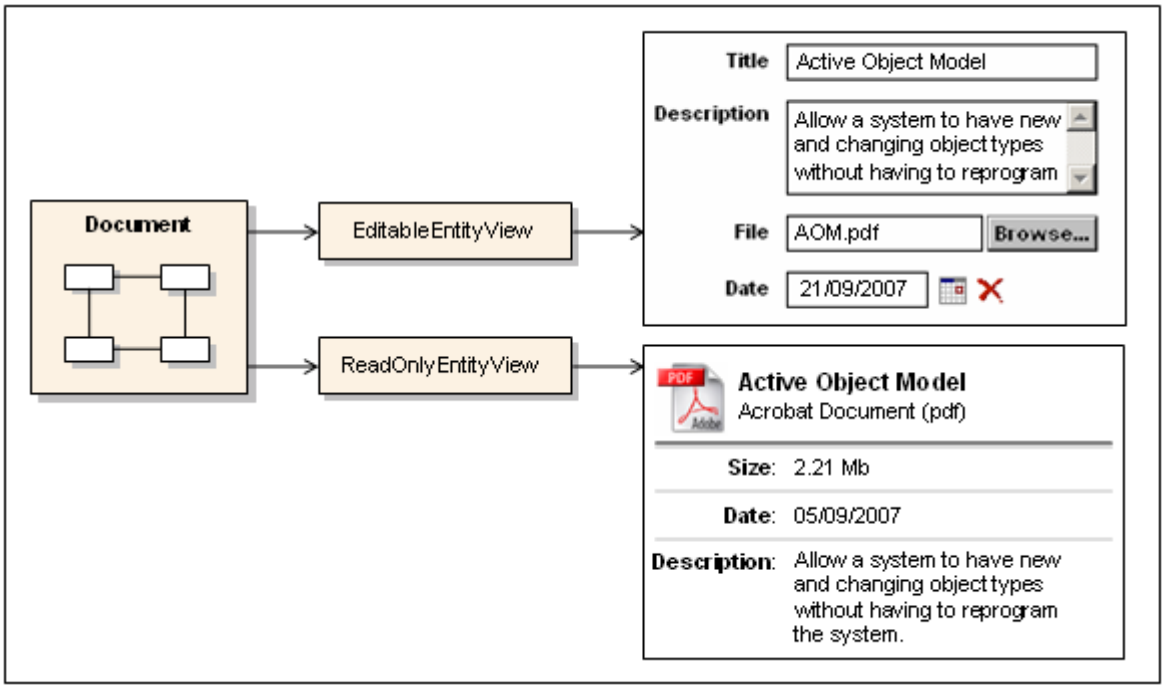

Figure 6 - Entity View Example

applications, giving consistency to their UIs (the same group of elements is rendered in a consistent way in all applications) and simplifying maintenance (the property renderer coordination code is in a single, well-known location).

In Figure 6, two `EntityViews` are shown: the first, called `EditableEntityView`, allows for editing an instance of an entity (in this case to create a new `Document` entity representing the paper "Dynamic Object Model" [13]). Notice how all the editing UI widgets shown are the same as those shown previously in Figure 4 for the PROPERTY RENDERER pattern. The second `EntityView`, called `ReadOnlyEntityView`, in the lower section of the figure renders a read-only representation of the `Document` entity. In this view no `Document` entity properties can be edited. Note that this `EntityView` shows additional `Document` properties.

3.3.7 Resulting Context

- UI composition of rendering entities can be abstracted, encapsulated and easily modified.
- The rules for showing an instance of an AOM entity can be modified dynamically at runtime.
- The rules for showing an instance of an AOM entity can be modified declaratively (when rules are stored as 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 in constructing different visualizations than with hand-coded solutions.
- This introduces more complexity in the form of additional classes and interpretation of metadata.
- The indirection interpretation of metadata found in this solution can lead to lower performance than in a non-AOM system.

3.3.8 Variants

Form Entity View: orchestrates several property renderers to create a form for data input. It may also contain constraints which establish input validations, and rules for showing or hiding groups of renderers, etc.

Table Row Entity View: orchestrates several property renderers to create a table showing an each entity in a row of a grid. To show a full grid this Entity View must be applied to a set of entities in an ENTITY-GROUP 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, you can have several views for a type of entity where each view shows a different subset of entity properties. For example, in case of an entity type "Patient" you 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 more complex entity view selects the property renderers to be used by applying rules. For example, you may have an entity view that shows or hides fields according to profile of the target user.

3.3.9 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; ENTITY-GROUP VIEWS display a set of related entities.

RENDERING ORCHESTRATOR performance can be dramatically enhanced using CACHING [11].

ANYTHING [15] has a similar abstraction called Renderer, but with a broader scope. To use this pattern to render ANYTHING instances, you can construe ENTITY VIEW to be a specialized instance of such renderers.

3.4 Entity-Group View

3.4.1 Context

You want to generate UI code for several 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 for different views of the same entity to be selected dynamically. You want several views applied to the same model and you want to have the possibility of selecting any of them according to arbitrary decisions.

3.4.2 Example

You are developing a Web-based Content Management application (the one quoted in the Property Renderer pattern). You built a Document Management module on top of the CMS engine. This content management module has entities `Document` and `Link` that are contained in `Categories` (a special kind of entity which contains other entities). `Categories` simulate Folders in the document management module.

Whenever a user selects one Folder, its contents (the contained entities) should be displayed in one of several ways depending on the specific context. You want to be able to attach and detach views to the folders. For example, a thumbnails view might only be applied to folders which contain images. Views should be easily linked to and unlinked from categories, allowing users to specify how they want to view folder contents according to their preferences.

Having the UI generation code static on a web page is not a very good idea because it would complicate your abstraction of a rendering algorithm that could be applied to different contexts. Additionally, if you want to reuse the UI generation code for another application you won't be able to, since it would be contained in a page and therefore could not be reusable artifact in another application (in the best case, you might copy the page, but if you want to change a single feature of that "common page", you would need to modify all instances of that page in all client applications).

3.4.3 Problem

How can you abstract the visualization (including the complex layout) of a set of dynamic entities from an AOM so as to decouple this visualization from the model?

3.4.4 Forces

- You want to be able to attach and detach views dynamically to sets of entities.
- You want to abstract layout details.
- You want to render several entities in the same presentation.
- You want to reuse that rendering code in different contexts.
- You don't want redundant UI code.
- You want to have control of all the generated UI code.
- You may not be using PROPERTY RENDERERS or ENTITY VIEWS.
- When using PROPERTY RENDERERS or ENTITY VIEWS you may want to add additional UI code (layout code, glue code to give consistency and context to the renderer properties, or perhaps code unrelated to entities).

3.4.5 Solution

Abstract the UI code generation into a view component that processes a set of entities to produce UI code. The `EntityGroupView` is a component specialized in generating UI code for a set of one or more entities. It will produce the appropriate UI code according to the purpose of the view. As in

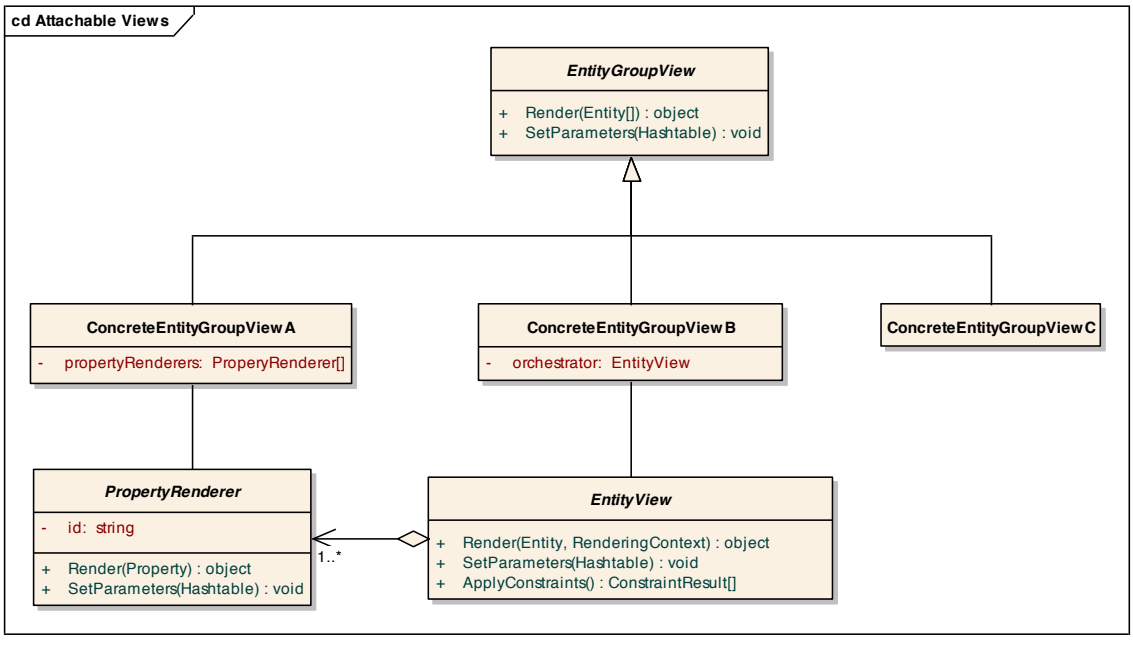

Figure 7 - Entity-Group Views Structure

MVC, the view components present information to the user. Different views can then present the information in the model in different ways.

The `EntityGroupView` can contain complex layout logic. The layout code may even allow dynamic set up and modification of the layout (for example like the models in WinForms [9] or Swing [7]) or may represent in a fixed way a specific set of entities (the layout is hard-coded in the view).

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

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

Figure 7 shows a UML class diagram of the solution. The abstract class `EntityGroupView` defines the public interface and default behavior of all `EntityGroupViews`. Concrete `EntityGroupView` subclasses can generate their output using several approaches: using Property Renderers (`ConcreteViewA`), using Entity Views (`ConcreteViewB`) or generating all UI code themselves (`ConcreteViewC`).

3.4.6 Example Resolved

If the UI rendering code for an `EntityGroupView` is represented as metadata, it can be stored in a views repository. This can then be linked to existing entities in order to generate UI code for them.

In our example, several views are created (e.g. Details View, Icons View and Thumbnails View) and then linked to the categories that represent the folders. When the user selects a Folder and views 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 (delegated to the concrete View) as shown in Figure 8.

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

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

3.4.7 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 they are stored in metadata).
- The rules for showing sets of entities are explicitly stated.
- It is easy to change the way sets of entities are shown.
- Better adaptability to new visualization requirements.
- More flexibility.
- More complexity.
- Lower performance.

3.4.8 Related Patterns

ENTITY-GROUP VIEWS can use several PROPERTY RENDERERS.

ENTITY-GROUP VIEW can use several ENTITY VIEWS.

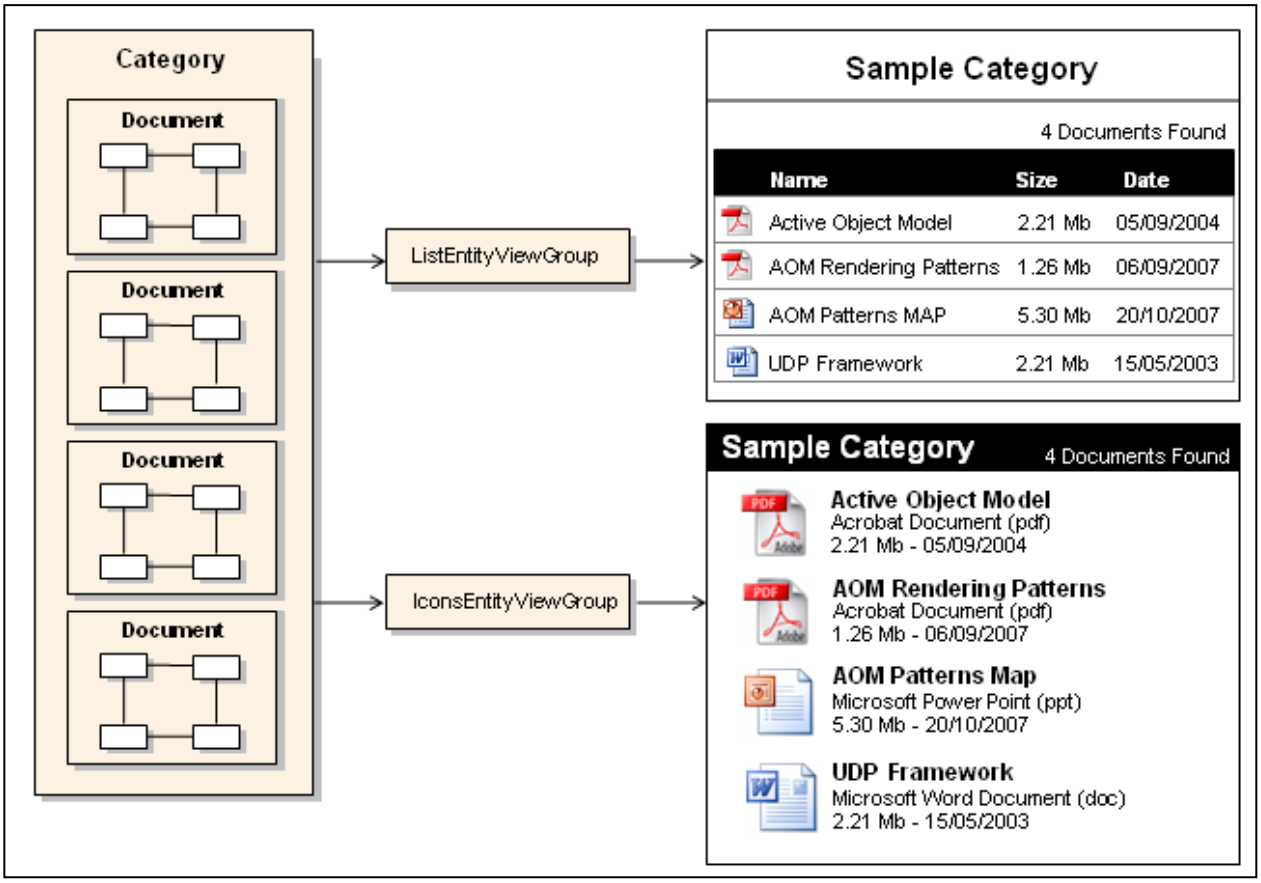

Figure 8 - Several views applied to the same entities.

ENTITY-GROUP VIEW instances should be created using a FACTORY.

ENTITY-GROUP VIEW can be seen as a special type of STRATEGY that is concerned with the generation of UI code for sets of entities.

An ENTITY-GROUP VIEW can be applied in MODEL VIEW CONTROLLER [10] scenarios.

ENTITY-GROUP VIEW performance can be dramatically enhanced using CACHING [11].

4. Putting It All Together

This paper presented a set of patterns for dealing with dynamic presentation of Adaptive Object-Models. Each pattern presented in this paper address the rendering problem at a different level of granularity as shown in Figure 9.

We used as an example building an application on top of a CMS system that is based on an AOM. In our CMS we created a `Document` entity type that contained several properties for storing the title, description, binary element (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 entities (in our case `Document` entities). We wanted a consistent UI decoupled from the application logic that could be easily changed and reused throughout this application or other systems.

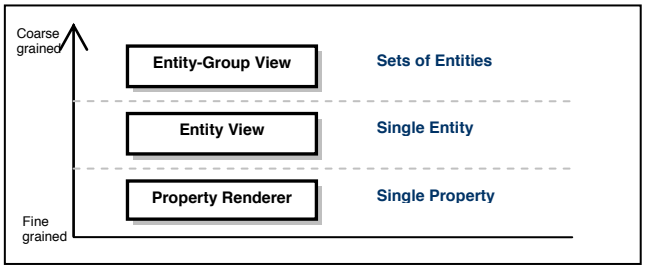

Figure 9 - Granularity level of the patterns in the language.

Since we wanted to render consistently all the properties of similar types, we determined to use the PROPERTY RENDERER pattern to generate the UI widgets for each property type. The first step was to create a `PropertyRenderer` for each `PropertyType` in `Document`: one for strings, another for binaries and one for dates. Thinking more deeply, we quickly realized that this is not enough: in some cases, we need two renderers for each property type, one for editing it and another for visualizing it. Therefore, we created these six property renderers:

- `StringInputPropertyRenderer`
- `FileInputPropertyRenderer`
- `DateInputPropertyRenderer`
- `StringPropertyRenderer`
- `FilePropertyRenderer`
- `DatePropertyRenderer`

After our renderers were created, we needed to establish how to present `Document` entities to end users. We used the ENTITY VIEW pattern 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). These kinds of Entity Views were addressed in the Variants section of the Entity View pattern.

These patterns work together to provide a consistent and reusable way for rendering AOM properties and entities. However, rendering concrete properties or entities is not enough to create the UI for our example document management application. To address this final gap we need to use the ENTITY-GROUP VIEW pattern to create several coherent fragments of UI for entering and retrieving `Document` entity instances. We thus create several `EntityGroupViews` that use the `PropertyRenderers` and `EntityViews` outlined in previous steps. These views can be dynamically linked to sets of `Document` entities to produce fully functioning and consistent UI fragments. The `EntityGroupViews` have content layout code such as in the case of the `DocumentGridDynamicView` which uses several `TableRowDocumentEntityViews` for generating an HTML table of `Document` entities.

There is a very important issue in the solution we present: performance and resource usage can be prohibitive, leading to a poor user experience and degradation of service scenarios (especially for web applications). To address these problems we propose the careful use of CACHING [11, 15]. 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 ENTITY-GROUP VIEW) [18]. The decision on how to apply caching should be carefully considered, keeping in mind that caching, too, adds considerable complexity to an application. Additionally, we might enhance the performance and resource usage of the application by applying other patterns (like POOLING, LAZY ACQUISITION, etc. [11]).

There are also several 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 at a later date.

5. ACKNOWLEDGEMENTS

We would like to thank our shepherd Dirk Riehle for his great help and advice for improving the contents of this paper. We would also like to gratefully thank to the participants of the PLoP 2007 "Sun Singer" Writers Workshop (Richard Gabriel, Ricardo Lopez, Jason Yip, Christian Kohls, Scott Henninger, Avraham Zilverman, and Vibhu Mohindra) and to the OOPSLA 2007 Mini-PLoP writers workshop participants (Peter Sommerlad, Ademar Aguiar, and Andre Santos).

6. REFERENCES

[1] Ahluwalia, K. Warning Message Accumulator Pattern. 13th Pattern Language of Programs Conference (PLoP 2005), Monticello, Illinois, USA, 2005.

[2] Adaptive Object-Models. http://www.adaptiveobjectmodel.com

[3] Bäumer, D ; D. Riehle. Product Trader. Pattern Languages of Program Design 3. Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998.

[4] Fowler, M. Analysis Patterns: Reusable Object Models. Addison-Wesley, 1997.

[5] 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.

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

[7] Trail: Creating a GUI with JFC/Swing. http://java.sun.com/docs/books/tutorial/uiswing/

[8] Johnson, R., R. Wolf. Type Object. Pattern Languages of Program Design 3. Addison-Wesley, 1998.

[9] Microsoft .NET Framework. http://www.microsoft.com/net/

[10] Buschman, F. et al. Pattern Oriented Software Architecture, Volume 1: A System of Patterns. Wiley & Sons. 1996

[11] Kircher, M.; P. Jain. Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management. Wiley & Sons, 2004.

[12] 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.

[13] 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.

[14] 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.

[15] Sommerlad, P.; M. Rüedi. Do-it-yourself Reflection. European Conference on Pattern Languages of Programs (EuroPLoP 98), Irsee, Germany, July 1998.

[16] Welicki, L.. The Configuration Data Caching Pattern. 14th Pattern Language of Programs Conference (PLoP 2006), Portland, Oregon, USA, 2006.

[17] Welicki, L; J. Cueva Lovelle; L. Joyanes Aguilar. Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models. European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany, July 2006.

[18] Welicki L; O. Sanjuan Martinez. 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.

[19] Welicki L; J. Yoder; R. Wirfs-Brock; R. Johnson. Towards a Pattern Language for Adaptive Object Models. Companion of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2007), Montreal, Quebec, Canada, 2007.

[20] 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.

[21] 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

[22] Yoder, J.; R. Razavi. Metadata and Adaptive Object-Models. ECOOP Workshops (ECOOP 2000), Cannes, France, 2000.

APPENDIX- A BRIEF SUMMARY OF THE ARCHITECTURAL STYLE OF AOMS

Important Notice: *This section is a summary extracted from [21] and [20] and has been included to help readers unfamiliar with the AOM architectural style. To get a more complete view we recommend the reader read the original papers found at www.adaptiveobjectmodel.com.*

The design of Adaptive Object-Models differs from most object-oriented designs. Normally, object-oriented designs have classes which model the different types of business entities and associate 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) which are interpreted at run-time. Thus, whenever a business change is needed, these descriptions are changed, and can be immediately reflected in a running application.

Adaptive Object-Model architectures are usually made up of several smaller patterns. TYPE OBJECT [8] provides a way to dynamically define new business entities for the system. TYPE OBJECT is used to separate an `Entity` from an `EntityType`. Entities have `Attributes`, which are implemented using the PROPERTY pattern [5]. The TYPE OBJECT pattern is used a second time in order to define the legal types of `Attributes`, called `AttributeTypes`. As is common in Entity-Relationship modeling, an Adaptive Object-Model usually separates attributes from relationships.

The STRATEGY pattern [6] can be used to define the behavior of `EntityTypes`. These strategies can evolve, if needed into a rule-based language that gets interpreted at runtime. Finally, there is usually an interface for non-programmers which allows them to define the new types of objects, attributes and behaviors needed for the specified domain.

Therefore, we can say that the core patterns that may help to describe the AOM architectural style are:

- TYPE OBJECT
- PROPERTY
- ENTITY-RELATIONSHIP / ACCOUNTABILITY
- STRATEGY / RULE OBJECT
- INTERPRETER (of Metadata)

Adaptive Object-Models are usually built from applying one or more of the above patterns in conjunction with other design patterns such as COMPOSITE, INTERPRETER, and BUILDER [6].

COMPOSITE is used for building dynamic tree structure types or rules. For example, if the entities need to be composed in a dynamic tree like structure, the COMPOSITE pattern is applied. BUILDERS and INTERPRETERS are commonly used for building the structures from the meta-model or interpreting the results.

But, these are just patterns; they are not a framework for building Adaptive Object-Models. Every Adaptive Object-Model is a framework of a sort but there is currently no generic framework for building them. A generic framework for building the TypeObjects, Properties, and their respective relationships could probably be built, but these are fairly easy to define and the hard work is generally associated with rules described by the business language. These are usually very domain-specific and varied from application to application.

Type Square

In most Adaptive Object Models, TYPE OBJECT is used twice: once before using the PROPERTY pattern, and once after it. TYPE OBJECT divides the system into `Entities` and `EntityTypes`. `Entities` have attributes that can be defined using `Properties`. Each `Property` has a type, called `PropertyType`, and each `EntityType` can then specify the types of the properties for its entities. Figure 10 represents the resulting architecture after applying these two patterns, which we call TYPE SQUARE [20].

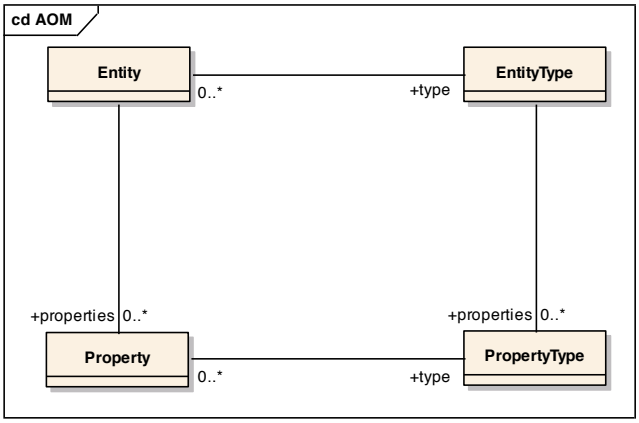

Figure 10. The Type Square.

TYPE SQUARE often keeps track of the name of the property and whether the value of the property is a number, a date, a string, etc. The result is an object model similar to the following: Sometimes objects differ only in having different properties. For example, a system that just reads and writes a database can use a Record with a set of Properties to represent a single record, and can use `RecordType` and `PropertyType` to represent a table.