An Architectural Model for Adapting Domain-Specific AOM Applications

P. M. Matsumoto, E. Guerra
Departamento de Ciência da Computação
Instituto Tecnológico de Aeronáutica
São José dos Campos, Brazil

***Abstract*—An Adaptive Object Model (AOM) is a common architectural style for systems in which classes, attributes, relationships and behaviors of applications are represented as metadata consumed at runtime. This allows them to be very flexible and changeable at runtime not only by programmers, but also by end users, improving system time-to-market. Nevertheless, this flexibility comes with a cost of a greater complexity when developing the system, and therefore one usually uses a bottom-up approach, adding flexibility only when and where it is needed. As a consequence, many AOM applications are tied to the specific domain to which they were developed and this fact makes it difficult to develop and use generic and reusable AOM frameworks that properly handle specific requirements of the AOM architecture. This work presents an architectural model that aims to adapt domain-specific AOM core structures to a common core structure by identifying AOM roles played by each element through custom metadata configuration. By doing this, this model allows the integration of domain-specific AOM applications and AOM frameworks, making it feasible to develop reusable components for the AOM architecture. This model is evaluated by creating an AOM framework and performing a modularity analysis on a case study based on it.**

Keywords-framework; metadata; modularity; architecture; adaptive system; decoupling; Adaptive Object Model

I. Introduction

An Adaptive Object Model (AOM) is a common architectural style for systems in which classes, attributes, relationships and behaviors of applications are represented as metadata, allowing them to be changed at runtime not only by programmers, but also by end users [1, 2]. AOM systems are able to adapt more easily in an environment where business rules change rapidly.

However, the tradeoff for this flexibility and agility in making changes is the higher complexity when developing an AOM system.

In order to avoid adding unnecessary complexity to the system, developers tend to use bottom-up approaches, adding flexibility only where it is necessary [3]. As a result,, AOM systems tend to be tied to their problem domains instead of being generic.

The fact that systems are usually coupled with their problem domains makes it difficult to develop frameworks for handling common requirements for AOM systems, such as persistency, model version control, user interfaces and end user development tools. Due to the dynamic nature of AOMs, the implementation of these requirements is not trivial and requires solutions that differ from traditional programming approaches. Therefore, the possibility of reusing solutions among different AOM applications would greatly reduce the time and complexity of the system development.

This work presents an architectural model that adapts an AOM core structure coupled with a specific domain to a common AOM core structure by using metadata to identify the AOM roles played by classes, attributes and methods in the domain-specific AOM application. This solution externalizes the AOM core structure from the application domain and provides a common structure to be used by generic frameworks that implement AOM common requirements. The AOM Role Mapper framework [4] was developed based on the proposed model. To evaluate this model, a modularity analysis was performed on a case study application which used the developed framework.

This paper is organized as follows: Section 2 gives an overview of AOMs; Section 3 presents the motivation for the creation of the architectural model presented in this work; Section 4 presents the architectural model proposed in this work; Section 5 presents the AOM Role Mapper framework, which implements the model presented in Section 4; Section 6 presents a case study and the modularity analysis with the developed framework; Section 7 presents related works; and Section 8 presents the main conclusions and future work.

II. Adaptive Object Models

In a scenario in which business rules are constantly changing, implementing up-to-date software requirements has been a challenge. Currently, this kind of scenario has been very common and requirements usually end up changing faster than their implementations, resulting in systems that do not fulfill the customer needs and projects that have high rates of failure.

According to [3], while software engineering methodologies, like Agile Software Development, try to increase the ability of adapting to changes, they consider each outcome of an iteration to be the last one, although it is not. Opposed to this approach, AOMs are developed to be incomplete by design [5].

We thank for the essential support of FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) to this research.

In the AOM architectural style, classes, attributes, relationships and behaviors are represented using metadata [1, 2], and represented at runtime as instances. This allows the model to be changed at runtime and makes it possible to empower end-users to change the system according to their necessities, greatly reducing the time-to-market.

AOM architectures are usually made up of several smaller patterns, such as TYPE OBJECT, PROPERTY LIST, TYPE SQUARE, ACCOUNTABILITY, STRATEGY, RULE OBJECTS, COMPOSITE, BUILDER and INTERPRETER. Besides those, there are many other patterns that are used when creating an AOM application. These patterns form a pattern language for AOMs that is divided into six categories: Core, Process, GUI, Creational, Behavioral and Miscellaneous/Instrumental [6].

The Core category includes patterns that are present in basic implementations of AOMs and guides this architectural style. This work has focused on a subset of the Core category patterns composed by TYPE SQUARE (TYPE OBJECT and PROPERTIES) and ACCOUNTABILITY. These patterns are used for developing the structural part of an AOM core design and are described in the following sections.

A. *Type Object*

The TYPE OBJECT pattern [7] is used in situations in which the number of subclasses that a class may need cannot be determined at development time. This pattern solves the situation by representing the subclasses that are unknown at development time as instances of a generic class that represents the object type.

Fig. 1 depicts the solution of the TYPE OBJECT. The unknown subclasses are represented as instances of the EntityType class. The Entity instances, which represent the actual instances of the system, refer to the EntityType instance that represents their class.

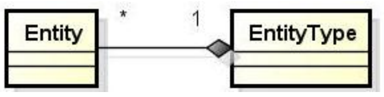


Figure 1. TYPE OBJECT solution

B. *Property*

In situations in which instances of the same class can have different types of properties, to create an attribute to represent each of these properties in the class might not be the best solution. For instance, in a medical system one may create a class called Person to store information on patients, such as height, weight and blood type.

One solution would be to add an attribute to the Person class for each type of information that is necessary for the patient. However, if a hospital has different departments that need different kinds of information, one would probably need a great number of attributes in the Person class and just a few of them would effectively be used by an instance of this class (only those needed by the department in which the patient is being treated).

PROPERTY [8] solves this problem by representing the properties of an entity with a class and making this entity to have a collection of instances of this class. Applying the solution to the example, a Measurement class could be created to represent data from the patient. With this change, the attributes of the Person class could be replaced by one collection of Measurements, which would contain all and only the necessary measurements needed from one patient. Fig. 2 depicts the solution.

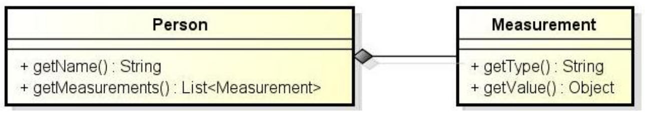


Figure 2. PROPERTY pattern applied to the example (adapted from [8])

C. *Type Square Pattern*

In the AOM architectural style the TYPE OBJECT and PROPERTY patterns are usually used together, resulting in the TYPE SQUARE [1]. In this pattern, the TYPE OBJECT is used twice – once for representing the entities and entity types of the system; and once for representing the properties and property types. Fig. 3 depicts the TYPE SQUARE structure.

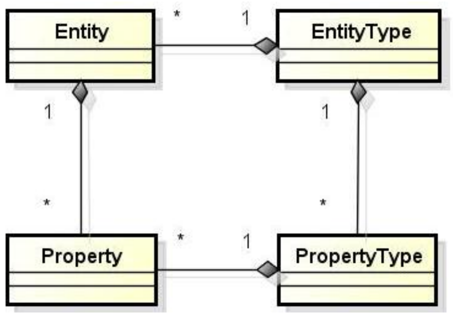


Figure 3. TYPE SQUARE (adapted from [1])

In this pattern, the EntityType and PropertyType classes represent the model and through their association it is possible to determine what kinds of properties are applicable to a given type of entity. The Entity and Property classes are related to the representation of the actual instances of the system. Each instance of Entity refers to an instance of EntityType that represents its type. For each PropertyType in an entity's type, a Property is created to store the value of the property type in the entity.

With the TYPE SQUARE new types of entities with different types of properties can be created. Likewise, existing types of entities can be changed at runtime since modeling is done at instance level.

D. *Relationship Representation*

In an entity there are usually two kinds of properties: those that refer to primitive data types (attributes) and those that refer to relationships between entities (associations). In the AOM architectural style there are different ways to separate attributes from associations [1]: (a) Use PROPERTY twice, once for attributes and once for associations; (b) Make two subclasses of a Property class – Attribute and Association; (c) Check the type of the value of a Property object: a Property whose value is an Entity represents an association, while a Property whose value is a primitive data type is an attribute; (d) Use ACCOUNTABILITY [8] to represent the association.

While any of these options can be used for developing an AOM application, relationships are frequently represented by ACCOUNTABILITY in AOM core design diagrams.

ACCOUNTABILITY [8] allows the relationship between entities to be represented by an object (usually an instance of an Accountability class). Each Accountability object is associated to an AccountabilityType object, which represents the type of the relationship. Since the associations between entities are represented at the instance level, types of entity relationships can be created or modified at runtime, which makes this pattern suitable to the AOM architectural style.

E. The Adaptive Object Model Core Design

The core design of an AOM system is depicted in Fig. 4. The diagram is divided in two parts – the operational level and the knowledge level [8]. The instances of the classes in the operation level store the system's data and day to day events of the domain, while the instances of the classes in the knowledge level contain the representation of the system model. The behavioral level is responsible for handling business rules in the architectural style and usually uses STRATEGY [9] and RULE OBJECT [10]. This level was left out of the scope of this work.

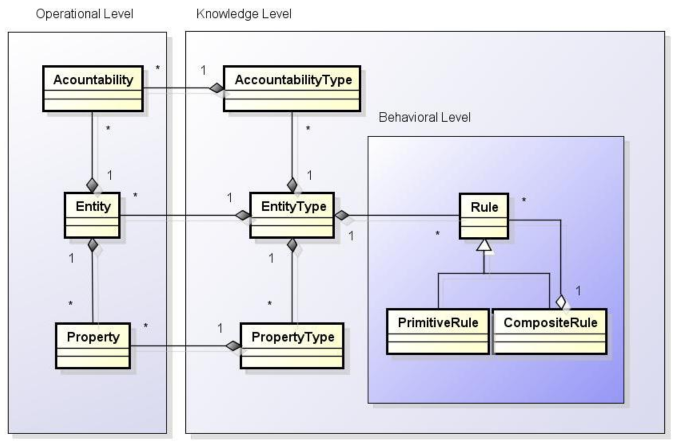


Figure 4. AOM core design, adapted from [1]

F. Concerns when Developing Adaptive Object Models

The flexibility provided by AOMs comes with a cost of a higher complexity when developing the application. Besides the fact that in AOMs metadata is used to represent the actual model of the system, developers also have to consider how to handle some implementation issues [2], such as:

(a) **Persistence:** not only should the actual data of an AOM be persisted, but also the representation of the model (described using metadata) should be stored in a database. The evolutionary nature of this model makes relational databases not the most appropriate type of storage. Another point to consider is how the system will be able to read information stored in the database and populate the AOM with the correct configuration of instances. Patterns like AOM BUILDER [11] should be considered when developing this issue.

(b) **GUI:** due to the dynamic nature of AOMs, user-interfaces have to be developed to be able to automatically adapt to changes in the model. In order to implement that, rendering patterns for AOMs [12] should be considered.

(c) **Model Maintenance Tools:** AOMs generally need tools and support GUIs to define and evolve the types in the system. These tools would be used for describing and maintaining the business rules of the application.

(d) **Version Control:** in order to support the evolution of the model in AOMs there is a need to implement a version control mechanism. Data of objects in the operational level must comply and be consistent with the model in the knowledge level. There is also a need to implement mechanisms to avoid the model to be broken due to partial updates.

Besides the issues presented above, there are many other points to be considered, such as security, instance validation, etc. All implementations, including business rules, related to the application domain must be based on metadata, because the system model is in the instance level and is not available at compile time.

III. MOTIVATION

As mentioned in the previous section, there are some common concerns that should be handled when developing an AOM application. For a great number of these concerns there are patterns that help the development of the system. The solution presented by these patterns usually considers the core structure of AOMs (formed by the patterns TYPE OBJECT, TYPE SQUARE, PROPERTY and ACCOUNTABILITY) and could be implemented with a more generic AOM framework. However, since the core structure of AOM applications is usually coupled with the domain of the problem they solve, applications are not easily integrated with generic AOM frameworks.

In order to illustrate this issue, two systems that were modeled using AOM are considered in this example: the Illinois Department of Public Health (IDPH) Medical Domain Framework [1, 2] and a banking system for handling customer accounts [13]. This example shows that although both systems share some common structures and needs, code cannot be reused among them because their core structures are coupled with their specific domains.

The IDPH Medical Domain Framework was developed in order to manage common information that was shared between applications used by the IDPH. This common information consists of observations made about people and relationships between people and organizations. Examples of these observations are blood pressure, cholesterol, eye color, height and weight.

In order to avoid the need for development and recompilation of the system whenever a business rule changed or a new type of observation was added, the application was developed using AOM. The resulting system model is depicted in Fig. 5. The design considers situations in which one observation is composed by other observations and also considers different types of observations (range values and discrete values).

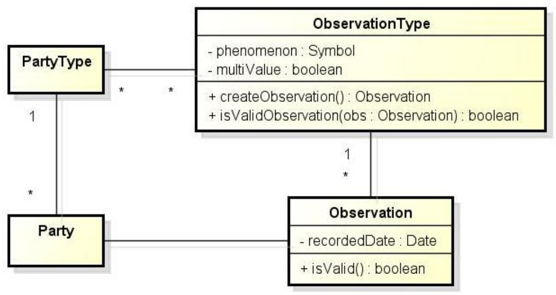


Figure 5. IDPH Medical Framework design (adapted from [1])

The example given in [13] consists of a banking system for handling customer accounts. The fact that the number of types of accounts in the bank can increase significantly is taken into consideration and in order to avoid a subclass and attributes explosion the TYPE SQUARE pattern is used. The basic design for the system is shown in Fig. 6.

Notice the similarities between the structures used in the systems outlined above, such as the usage of the TYPE SQUARE pattern. Both systems present concerns like a persistence mechanism, a GUI, a version control for the object model and support tools for allowing end user development in the systems.

Although the systems share some common core patterns and have common needs, a framework developed for IDPH cannot be used for the banking system and vice versa, because each application is focused on solving the problems in their specific domains. As an example, a persistency framework developed for the IDPH system would be coupled to the medical domain and therefore it could not be used for handling persistency in the banking system.

In this context, if both domain-specific AOM models could be adapted to a common model, the latter could be referred by an AOM framework implementing the common needs of the systems (for instance, persistency), making the solutions reusable between the two applications.

IV. ARCHITECTURAL MODEL FOR AN AOM MAPPING FRAMEWORK

In order to solve the integration problem presented in Section III, this work proposes an architectural model for a metadata-based framework that adapts the domain-specific AOM core structures to a common AOM structure. For the sake of clarity, this framework is referred as the *integration framework* and any client of this framework (i.e. generic AOM frameworks and client applications) is referred as *client*.

In the solution proposed by this work, the integration framework provides a common AOM core structure that can be referred by generic AOM frameworks. The framework also provides classes that adapt the domain-specific AOM core structures to this common AOM structure. Since these classes implement the ADAPTER pattern [9], they are referred as the *adapter classes* in this work.

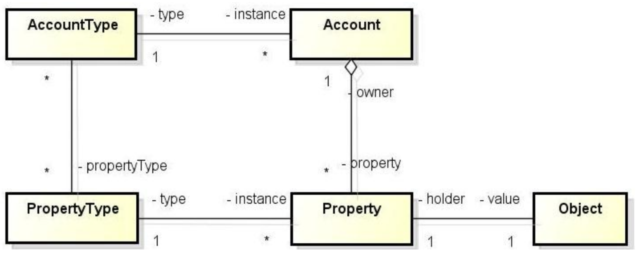


Figure 6. Basic design for the banking system [13]

In order to be able to adapt domain-specific AOM core structures, the integration framework only needs to identify at runtime the roles that classes, methods and attributes of domain-specific classes play in the AOM architecture. Examples of these roles are Entity, Entity Type, Property and Property Type. This identification is accomplished by the use of metadata resources.

Fig. 7 shows the representation of the solution applied to the IDPH and Banking System examples given in Section III. In the figure, the metadata used for identifying the AOM roles are Java annotations. As depicted, the domain-specific classes are marked with specific annotations that are consumed by the adapter classes, which implement the interfaces that define the common AOM core structure. When a specific method is called in an adapter object, it is able to know which exact method to call in the adapted object due to the AOM role annotations. The generic AOM frameworks only need to refer to the interfaces of the common AOM core structure provided by the integration framework.

With this approach, domain-specific applications can be integrated with generic AOM frameworks only by identifying the AOM roles played by its classes, attributes and methods in the AOM core structure, using the metadata provided by the integration framework. All the responsibility for the integration is left outside the domain-specific applications and the generic AOM frameworks. Notice that the only change to the domain-specific applications is the inclusion of metadata to identify the AOM roles. Therefore, the applications can continue using their domain-specific AOM core structure without any change to the solution's architecture.

The use of metadata for identifying the AOM roles allows the integration framework and the domain-specific applications to be completely decoupled if external metadata, like XML, is used or loosely coupled if metadata such as annotations and custom attributes are used. Besides, notice that the domain-specific AOM applications only depend on the metadata provided by the integration framework for identifying the AOM roles of their classes, methods and attributes. Additionally, generic AOM frameworks only have to refer to the common AOM core structure provided by the integration framework.

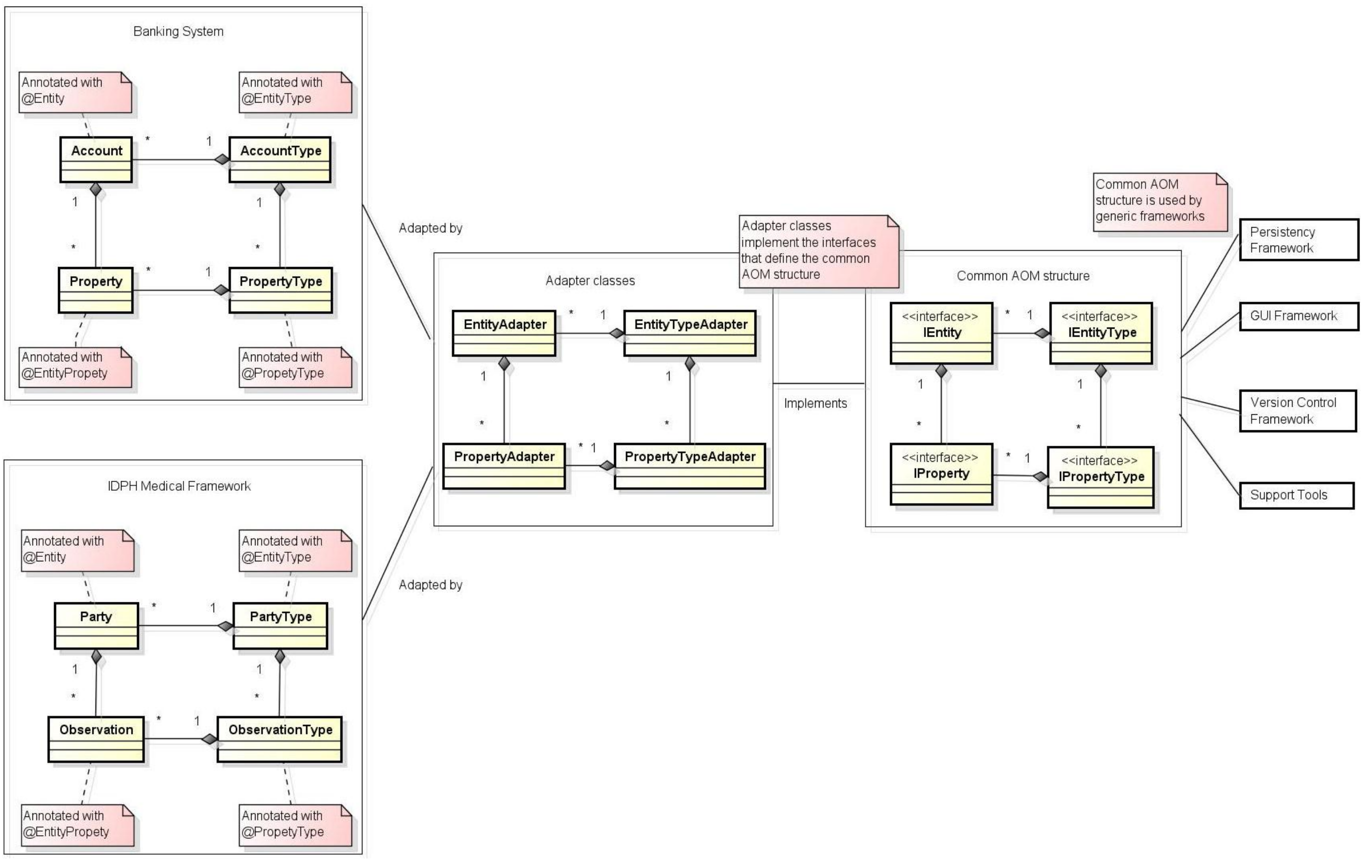


Figure 7. Representation of the solution for the examples given in Section III

A. *Integration Framework Components*

The main components for developing an integration framework are the following: (a) **Metadata Handler** is responsible to retrieve metadata from the application classes. It implements metadata reading patterns [14] to decouple metadata handling operations from the rest of the framework; (b) **AOM Core API** includes a set of interfaces that represent the common AOM core structure provided by the framework; (c) **AOM Core Implementations** contains implementations of the interfaces defined by the AOM Core API component. There are two types of implementations in this component: a basic and general implementation of the AOM core structure and an implementation that adapts domain-specific AOM core structures using the Metadata Handler component; (d) **Model Manager** is responsible to instantiate the model and manage the instances created by the framework.

Fig. 8 depicts the relationship between the main components in the integration framework. In this representation, the component Client represents the clients of the proposed framework.

The client uses the Model Manager component to perform operations over the model, such as loading and saving elements of the architecture. The client and the Model Manager components are able to perform operations on the AOM Core Implementation objects through the AOM Core API component interfaces. When an operation is performed over an adapter object of the AOM Core Implementation component, this object gets information on the domain-specific AOM core structure metadata by using the Metadata Handler component. This way, the adapter object is able to perform the corresponding operation on the adapted domain-specific AOM application object.

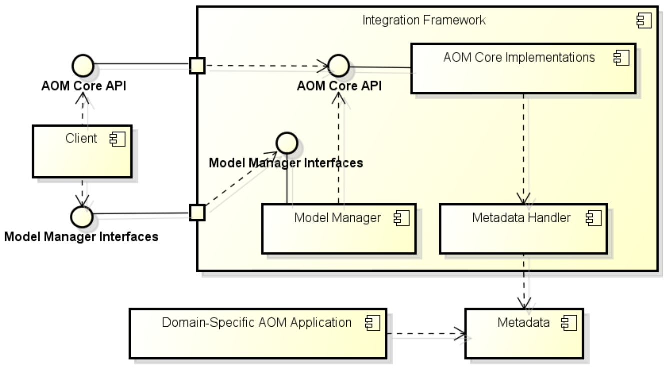


Figure 8. Relationship between the components in the integration framework

B. Analysis of the Model

As shown in the previous sections, the architectural model proposed in this work is able to integrate domain-specific AOM applications and generic AOM frameworks by adapting the domain-specific AOM core structures to a common AOM core structure defined by the AOM Core API component.

The only information needed for adapting the domain-specific AOM core structure is the AOM roles played by the elements in the structure. These roles are identified through metadata that is consumed at runtime by the integration framework, using the Metadata Handler component. Notice that there is only a weak dependency between the domain-specific AOM applications and the integration framework. The applications only have to mark their core structures with the metadata provided by the framework. If external metadata is supported, there is no need of changes in the domain-specific applications' code. Otherwise, only minor changes, such as inserting an annotation to the code, is required in order to make the application be adaptable by the integration framework.

In the generic AOM frameworks perspective, no knowledge related to domain-specific applications is needed. These frameworks simply make use of the components provided by the integration framework (AOM Core API and Model Manager) in order to be applicable to any domain-specific application that is configured to be adaptable by the integration framework.

As a consequence, with this architectural model, generic AOM frameworks and domain-specific AOM applications can be integrated, even though being completely decoupled. Due to this possibility, generic AOM frameworks can be developed without being tied to any specific domain and can be applied to different AOM applications, allowing reuse of code and design.

This model implements the AOM METADATA EXTENSION POINTS pattern [15] by using metadata for getting information on AOM applications in order to provide a TYPE SQUARE extension that can be used for adding specific behaviors to the domain-specific AOM applications.

V. THE AOM ROLE MAPPER FRAMEWORK

The AOM Role Mapper integration framework [4] was developed in order to evaluate the proposed model. Besides the functionality of adapting existing domain-specific AOM core structures to a common structure, the framework also allows the creation of an AOM application from scratch. The following sections give an overview of the framework components.

A. Metadata Handler Component

The Metadata Handler component implements some of the patterns of the pattern language presented in [14] and can be divided in the following parts:

(a) **Descriptors:** implements the METADATA CONTAINER pattern. Each role in an AOM architecture is represented by one descriptor which contains references to get/set/add/remove Method objects for each relevant field

(b) **Metadata Readers:** implements the METADATA READER STRATEGY pattern. Currently, the framework only supports annotations for determining the AOM roles of elements in domain-specific applications, but since the METADATA READER STRATEGY pattern was implemented, it supports extensions related to the support of other types of metadata.

(c) **Metadata Repository:** implements the METADATA REPOSITORY pattern, providing an in-memory cache of the metadata already retrieved.

(d) **Annotations:** contains the Java annotations that allow the identification of the AOM roles of the elements in the domain-specific AOM applications. The names of some annotations created for the framework are similar to some JPA's annotations, but they are completely unrelated. The annotations in AOM Role Mapper are used to map elements of domain-specific AOM applications to the generic AOM core structure.

B. AOM Core API and Implementation Components

The common AOM core structure provided by the framework consists of the following interfaces: IEntityType, IEntity, IPropertyType and IProperty. These interfaces are implemented by classes in two different packages – one that contains implementations related to the adaptation of domain-specific AOM core structures to the common core structure provided by the framework; and another that contains generic AOM classes that can be used for creating a new AOM application using the framework. The framework provides factory classes which are able to decide what class to instantiate according to parameters passed to the creation methods.

Although these two types of core structure implementations are available, this paper focus on the implementation of the ADAPTER classes, which are a differential from the other existing frameworks. The ADAPTER core structure is composed by five classes: AdapterEntityType, AdapterEntity, AdapterPropertyType, AdapterProperty and AdapterFixedProperty. Each of these classes contains an attribute for storing the domain-specific AOM application object that they adapt.

```
@Entity
public class Account {

 @EntityType
 private AccountType accountType;

   ...

 public AccountType getAccountType() {
      return accountType;
 }

 public void setAccountType(AccountType accountType) {
     this.accountType = accountType;
 }
}
```

Figure 9. Example of a domain-specific application class with annotations

By considering the example of a domain-specific class annotated with the metadata provided by the AOM Role Mapper framework depicted in Fig. 9, when an AdapterEntity object is created to adapt an Account object, the Metadata Handler component is used to get the descriptor for the Account class. This descriptor will contain the Method objects for getting and setting the account's account type, among other data. Using the information provided by the descriptor, the AdapterEntity object is able to invoke methods over the Account object.

Fig. 10 shows an example of how the getEntityType() method is adapted by an AdapterEntity object. When the Client calls the getEntityType() method, the AdapterEntity object obtains the Method object that gets the Entity Type of the adapted entity from its metadata descriptor. Then, it invokes this method using reflection and obtains the domain-specific Entity Type instance for the adapted Entity.

The getEntityType() method must return an IEntityType object and, therefore, it calls the getAdapter() static factory method of the AdapterEntityType class, passing the domain-specific Entity Type object as a parameter.

This method queries an internal map in the AdapterEntityType class, which relates a domain-specific object to the AdapterEntityType instance that adapts this object. The use of this map avoids the creation of more than one object to adapt the same domain-specific object – if an AdapterEntityType object was already created for adapting a determined domain-specific Entity Type object, the getAdapter() method only returns the previously created object; otherwise, it creates a new instance of AdapterEntityType, puts it into the map and returns it. Finally, the object returned by the getAdapter() method is returned to the Client.

The adaptation process described above is the overall solution used for the adapter classes of the Core Implementations component (i.e. AdapterEntityType, AdapterEntity, AdapterPropertyType, AdapterProperty).

C. Model Manager Component

The Model Manager component's responsibility is to orchestrate the instances created by the AOM Role Mapper framework. The main class of this component is the ModelManager, whose instance is unique. All the operations involving the manipulation of the model, including model persistency, loading and querying, should be done through this class.

For accessing the database, the ModelManager makes use of the IModelRetriever interface, which can be implemented by persistency frameworks.

In order to get the instance of IModelRetriever to be used, the ModelManager class uses the Service Locator functionality that is available in the standard Java API. The advantage of using Service Locator is that the implementation of the IModelRetriever interface becomes totally decoupled from the framework, allowing great flexibility. It also allows services to be dynamically changed using decentralized configuration.

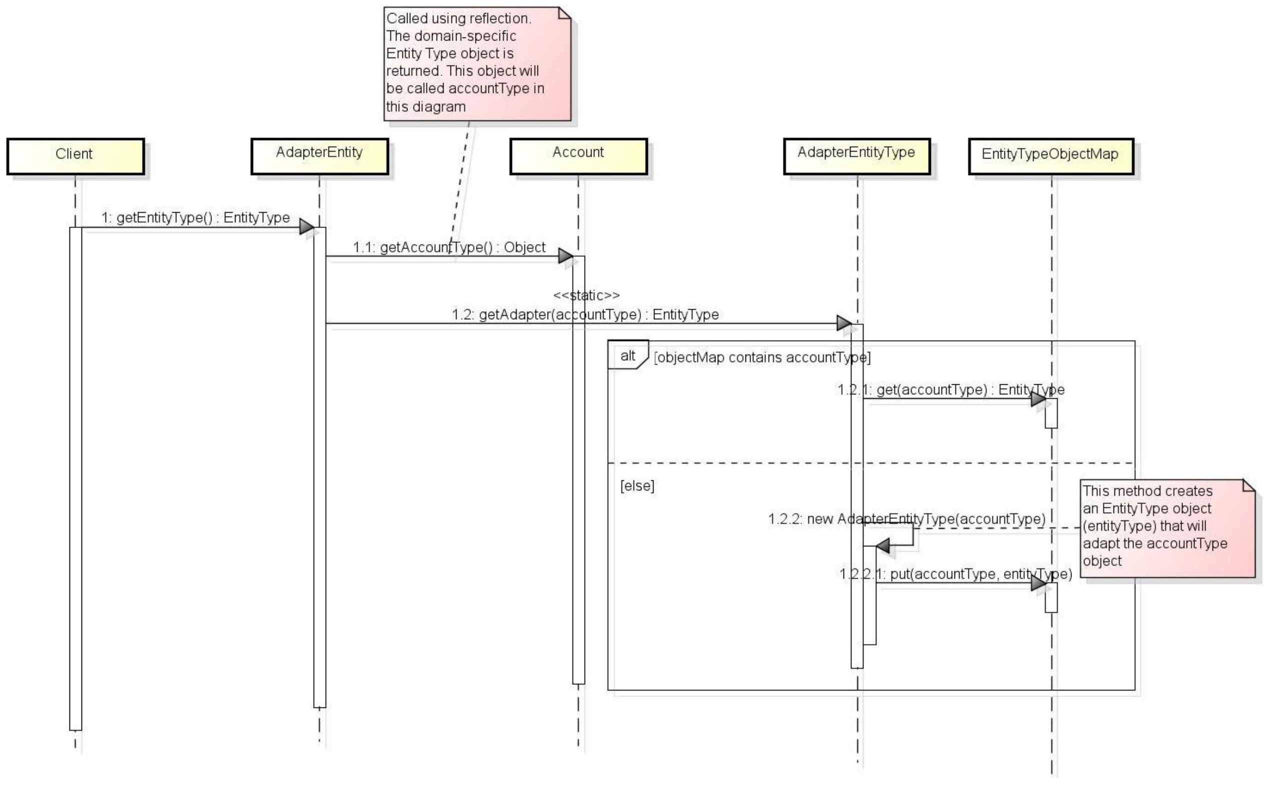

Called using reflection. The domain-specific Entity Type object is returned. This object will be called accountType in this diagram
Client
AdapterEntity
Account
AdapterEntityType
EntityTypeObjectMap
1: getEntityType() : EntityType
1.1: getAccountType() : Object
<<static>>
1.2: getAdapter(accountType) : EntityType
alt
[objectMap contains accountType]
1.2.1: get(accountType) : EntityType
[else]
1.2.2: new AdapterEntityType(accountType)
1.2.2.1: put(accountType, entityType)
This method creates an EntityType object (entityType) that will adapt the accountType object

Figure 10. Sequence diagram showing how domain-specific objects are adapted

One of the main responsibilities of the ModelManager class is to guarantee that a logical element is not instantiated twice in the framework. In order to control that, the ModelManager contains two Map objects – one for storing the loaded Entities by their IDs and one for storing the loaded Entity Types by their IDs. Whenever a method which loads an Entity or an Entity Type is called, the ModelManager checks whether the ID of the instance to be loaded is already found in the corresponding map. If so, it returns the previously loaded object. Otherwise, it calls the IModelRetriever object for loading the object into the memory and saves it into the map.

VI. Modularity Analysis

In order to evaluate the concept of the AOM Role Mapper framework, a case study which included the creation of an AOM persistency framework, a Client application and two domain-specific AOM core structures to be adapted by the framework, is presented:

- *Persistency Framework:* For validating the concept presented in this work, a persistency framework that implements the IModelRetriever interface was developed. This framework is called AOM Mongo Persistence framework. The database used for implementing the persistency framework was MongoDB [16], which is a document-oriented storage and is more suited for dealing with the dynamic nature of AOMs than SQL databases.

- *Domain-Specific AOM Structures:*Two sample domain-specific AOM core structures were developed for the case study: one for a banking system and one for a medical system. It is important to state that these structures are based on examples referenced by other works [13, 1]. The domains of the systems are similar to the examples shown in Section III.

- *Client Application:*The client application developed for this case study is a simple console-based application which shows menus for: loading the model into memory; saving the model; adding / removing / changing Entity Types; adding / removing / changing Entities; and showing Entities and Entity Types. All these operations are performed only by using the Model Manager component and the core structure interfaces.

A. *Case Study*

This case study consisted of properly configuring the client application to make it work with both banking and medical system without the need to change any code. For the case study, the AOM Mongo Persistence framework was used as the persistence framework.

The list below shows the jar files that were used in the analysis: ***(a) aomrolemapper.jar:*** Contains the AOM Role Mapper framework; ***(b) aompersistence.jar:*** Contains the AOM Mongo Persistence framework; ***(c) bankingexample.jar:*** Contains the Banking system; ***(d) medicalexample.jar:*** Contains the Medical system***; (e) aomtest.jar:*** Contains the client application.

When running the client application, the aomrolemapper.jar and aompersistence.jar must be included in the classpath. If the banking system is the one that needs to be adapted, the client application only has to include the bankingexample.jar file into the classpath when it is run. Similarly, if the medical system is the one to be adapted, the client application only has to insert the medicalexample.jar file into the classpath. It is also possible to adapt both systems simultaneously and use the client application without any changes to the code.

B. *Modularity Analysis*

In order to analyze the modularity of the applications and frameworks developed for the case study, a Dependency Structure Matrix [17] that shows the package dependencies in all the jar files involved was generated using the Lattix tool (Fig. 11).

The modules involved in the case study can be easily identified through the different colors. The 'X' character indicates that the module represented in the line depends on the module represented by the column. Any reference from one module to another (e.g. invocation of a method or the use of an annotation) is considered a dependency.

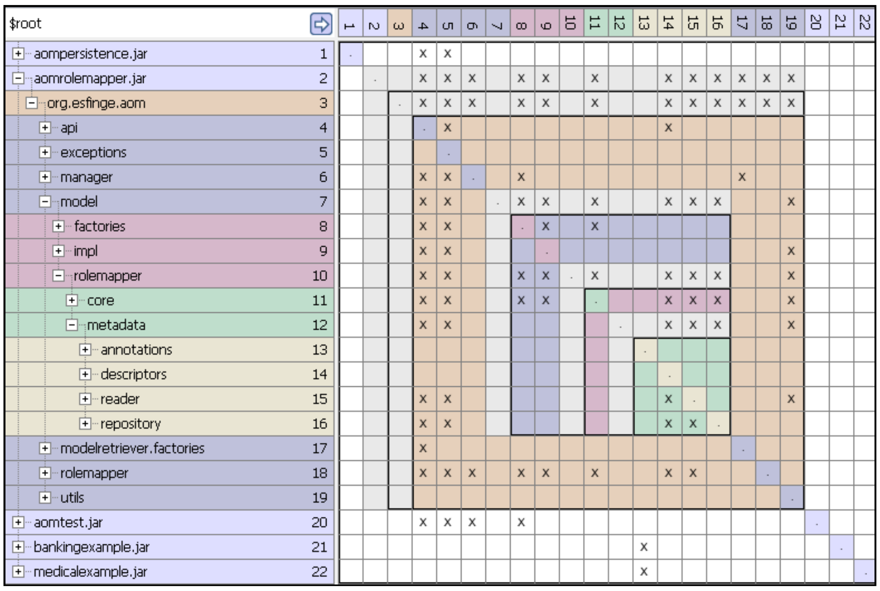
Notice that the domain-specific applications, represented by 21 and 22, only depend on the annotations package of the AOM Role Mapper framework, represented by 13. This package only contains the definitions of the annotations provided by the framework.

Analyzing the dependency of the AOM Mongo Persistence framework package (1), it is possible to observe that the framework depends on the api (4) and exceptions (5) packages of the AOM Role Mapper framework. The first package contains the IModelRetriever interface, which is implemented by the persistence framework; and the Model Manager and AOM Core API component interfaces. The second package contains the exceptions thrown by the AOM Role Mapper framework.

Notice that the persistence framework only depends on information exposed by the AOM Role Mapper framework, not depending on the specific implementation of this framework and also not depending on any information related to the domain-specific applications.

The fact that the persistence framework only depends on the AOM Role Mapper framework makes it applicable to any application that can be adapted by the AOM Role Mapper framework.

Similarly, the client application (20) dependency analysis shows that it only depends on information exposed by the AOM Role Mapper framework. It depends on the api (4), exceptions (5), manager (6), and model.factories (8) packages of the framework. The first two packages contents were explained above. The third package contains the ModelManager class. The forth package contains the factory classes for the common AOM core structure classes.



$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
aompersistence.jar	1	.			x	x																	
aomrolemapper.jar	2		.		x	x	x		x	x		x			x	x	x	x	x	x			
org.esfinge.aom	3			.	x	x	x		x	x		x			x	x	x	x	x	x			
api	4				.	x									x								
exceptions	5					.																	
manager	6				x	x	.		x									x					
model	7				x	x		.	x	x		x			x	x	x			x			
factories	8				x	x			.	x		x											
impl	9				x	x				.										x			
rolemapper	10				x	x			x	x	.	x			x	x	x			x			
core	11				x	x			x	x		.			x	x	x			x			
metadata	12				x	x							.		x	x	x			x			
annotations	13													.									
descriptors	14														.								
reader	15				x	x									x	.				x			
repository	16				x	x									x	x	.						
modelretriever.factories	17				x													.					
rolemapper	18				x	x	x		x	x		x			x	x			.				
utils	19																			.			
aomtest.jar	20				x	x	x		x												.		
bankingexample.jar	21													x								.	
medicalexample.jar	22													x									.

Figure 11. DSM for the case study

C. *Analysis of the Case Study*

In this case study, it was possible to verify that no code changes were needed in order to make the application and frameworks to work with both domain-specific applications. The only actions needed were to change configuration files and to put the proper domain-specific application jar into the classpath.

The metrics depicted by the DSMs shows that the domain-specific applications only depend on the annotations defined by the AOM Role Mapper framework, which means that the domain-specific applications can still be used without the generic AOM frameworks and applications. The metrics also show that the persistency framework and client application only depend on information that is externalized by the AOM Role Mapper framework, which means that they can be reused among different domain-specific AOM core structures.

These results show that the proposed architectural model accomplished its goal to decouple the core structure of AOM applications from their specific domains, providing a way to allow reuse of design and code of generic AOM frameworks and applications among different domain-specific applications.

VII. RELATED WORK

In [2], many examples of systems that use the AOM architectural style are presented. While these systems aim at solving specific issues in specific domains, other frameworks, such as Oghma [18, 19], ModelTalk [20, 21] and its descendant, Ink [22] aim at providing generic AOM frameworks for easing the creation of adaptive systems, mainly through the use of a Domain-Specific Language (DSL).

Oghma is an AOM-based framework written in C#, which aims to address several issues found when building AOM systems, namely: integrity, runtime co-evolution, persistency, user-interface generation, communication and concurrency [18]. The modules that handle each of these concerns reference the AOM core structure of the framework, which was developed to be self-compliant by using the EVERYTHING IS A THING pattern [23].

Oghma allows a client program to instantiate a model for its domain by simply calling the constructor of the MetaModel class of the framework, passing as argument an XML model configuration file that contains the Entities descriptions. After this model is created, the aforementioned AOM requirements implemented by the framework are readily available.

ModelTalk and Ink are AOM frameworks that rely on a DSL interpreter to add adaptability to the model. At runtime, instances of DSL classes are instantiated and used as meta-objects for their corresponding Java instances through a technique called model-driven dependency injection [21]. Developers are able to change the model by editing the ModelTalk/Ink configuration in an Eclipse IDE plug-in specially developed to handle the framework DSL. When changes in the model are saved, the plug-in automatically invokes the framework's DSL analyzer, performing incremental cross-system validation similar to background compilation in Java.

What the AOM Role Mapper framework presented in this work has in common with Oghma and ModelTalk/Ink is the fact that it is an AOM framework not tied to any specific domain and is intended to ease the development of AOM-based systems. However, instead of considering that the entire infrastructure for building AOM systems must be inside the framework, the AOM Role Mapper provides a standard AOM core structure that can be used by different AOM related frameworks, such as persistence, GUI and version control frameworks.

Besides this fact, the AOM Role Mapper framework is able to adapt the core structure of domain-specific AOM applications to the common structure it provides. As a consequence, this framework can be used for integrating generic AOM frameworks to existing domain-specific AOM applications. This functionality is not provided by the Oghma and ModelTalk/Ink frameworks, which requires the applications to be created from scratch, coupling the AOM model to the framework.

Finally, even though the domain-specific core structures are adapted by the AOM Role Mapper, these structures remain logically unchanged and can still be used in the system. This means that behavior can still be added through the application code, which brings simplicity to the system. But with Oghma and ModelTalk/Ink, the development of the system is limited to the expressiveness provided by those frameworks.

VIII. CONCLUSION

The Adaptive Object Model is an architectural style that provides great flexibility by representing classes, attributes, methods and relationships as metadata. The tradeoff of this architectural style is the higher complexity when implementing AOM systems. Therefore, AOM application developers tend to use bottom-up approaches, adding flexibility only where it is necessary. As a consequence, there are many AOM systems that are tied to the specific domain for which they were developed and this makes it difficult to create generic AOM frameworks that can be applied to any AOM application.

This work presented an architectural model to solve this issue by adapting the domain-specific AOM core structures to a common core structure by using metadata to identify AOM roles of elements in the domain-specific application. This common core structure can be used by generic AOM frameworks, allowing the reuse of code and design of these frameworks among different AOM applications even though they are tied to different domains. This work also presented the AOM Roler Mapper framework, which implements the proposed model in Java and uses annotations as metadata. Although the proposed solution API seems similar to other mapping frameworks, such as ORM, the internal solution is very different from these frameworks since it has to cope with two models that can be dynamically changed.

The modularity analysis made over the case study in this work showed that the domain-specific AOM applications have a weak dependency on the AOM Role Mapper framework. The analysis also showed that the AOM generic framework and the Client application created for the case study only depended on the Model Manager component and the common AOM core structure provided by the AOM Role Mapper framework. No information related to the specific implementation of the framework or the domain-specific applications was needed by the generic AOM framework and the Client application.

Although it was possible to show that the issue related to the integration of AOM generic frameworks and domain-specific AOM applications can be solved by the proposed architectural model, there is still a great field of research in this area. This work focused on the creation of the initial version of this integration framework, only supporting the adaptation of a basic AOM core structure. In order to have a framework that fully adapts domain-specific AOM applications, there is still need for research in matters such as inheritance and behavior representation. The analysis of factors such as reuse cost, performance and memory overhead of the solution will also be left for future works.

REFERENCES

[1] J. W. Yoder, F. Balaguer and R. Johnson, “Architecture and design of Adaptive Object-Models,” in Proceedings of the 16th OOPSLA, 2001.

[2] J. W. Yoder and R. Johnson, “The Adaptive Object-Model architectural style,” in Proceedings of the IFIP 17th World Computer Congress – TC2 Stream / 3rd IEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, 2002.

[3] H. S. Ferreira, F. F. Correia, A. Aguiar and J. P. Faria, “Adaptive Object-Models: a research roadmap,” in International Journal on Advance in Software, vol. 3, n. 1, 2010, pp. 70-89.

[4] P. M. Matsumoto, “The AOM Role Mapper framework, “ available on: http://sourceforge.net/projects/esfinge/, accessed in: 12 apr. 2012.

[5] R. Garud, S. Jain and P. Tuertscher, “Incomplete by design and designing for incompleteness,” in Organization studies as a science of design, vol. 29, n. 3, 2008, pp. 351-371.

[6] L. Welicki, J. W. Yoder, R. Wirfs-Brock and R. E. Johnson, “Towards a pattern language for Adaptive Object-Models,” in Proceedings of the 22th OOPSLA, 2007.

[7] R. Johnson and B. Wolf, “Type Object,” in Pattern Languages of Program Design 3. Addison-Wesley, 1997, pp. 47-65.

[8] M. Fowler, Analysis patterns: reusable object models. Addison-Wesley Professional, 1996.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: elements of reusable object oriented software. Addison-Wesley, 1995.

[10] A. Arsanjani, “Rule Object 2001: a pattern language for adaptive and scalable business rule construction,” in Proceedings of the 8th PLoP, 2001.

[11] L. Welicki, J. W. Yoder and R. Wirfs-Brock, “Adaptive Object-Model Builder,” in Proceedings of the 16th PLoP, 2009.

[12] L. Welicki, J. W. Yoder and R. Wirfs-Brock, “A pattern language for Adaptive Object Models: part I - rendering patterns,” in Proceedings of the 14th PLoP, 2007.

[13] D. Riehle, M. Tilman and R. Johnson, “Dynamic Object Model,” in Proceedings of the 7th PLoP, 2000.

[14] E. Guerra, J. T. Souza and C. Fernandes, “A pattern language for metadata-based frameworks,” in Proceedings of the 16th PLoP, 2009.

[15] P. M. Matsumoto et al., “AOM Metadata Extension Points,” in press.

[16] MongoDB, available on: http://www.mongodb.org/, accessed in 4 apr. 2012.

[17] A. A. Yassine, “An introduction to modeling and analyzing complex product development processes using the Design Structure Matrix (DSM) method,” in Quaderni di Management (Italian Management Review), n. 9, 2004.

[18] H. S. Ferreira, F. F. Correia and A. Aguiar, “Design for an Adaptive Object-Model framework: an overview,” in Proceedings of the 4 thWorkshop on Models@Run.Time, 2009.

[19] H. S. Ferreira, “Adaptive-Object Modeling: Patterns, Tools and Applications,” PhD Thesis, Faculdade de Engenharia da Universidade do Porto, 2010.

[20] A. Hen-Tov, D. H. Lorenz and L. Schachter, “Model-Talk: a framework for developing domain-specific executable models,” in Proceedings of the 8th OOPSLA Workshop Domain-Specific Modeling, 2008.

[21] A. Hen-Tov, D. H. Lorenz, A. Pinhasi and L. Schachter and, “ModelTalk: when everything is a domain-specific language,” IEEE Software, vol. 26, n. 4, 2009, pp. 39-46.

[22] E. Acherkan, A. Hen-Tov, D. H. Lorenz and L. Schachter, “The ink language meta-metamodel for adaptive object-model frameworks: [extended abstract],” in Proceedings of the 26th ACM International Conference Companion on OOPSLA Companion, 2011.

[23] H. S. Ferreira, F. F. Correira, J. W. Yoder and A. Aguiar, “Core patterns of object-oriented meta-architectures,” in Proceedings of the 17th PLoP, 2010.