AOM Metadata Extension Points

PATRICIA MEGUMI MATSUMOTO, Instituto Tecnológico de Aeronáutica
FILIPE FIGUEIREDO CORREIA, Departamento de Engenharia Informática, Faculdade de Engenharia, Universidade do Porto
JOSEPH WILLIAM YODER, The Refactory, Inc
EDUARDO GUERRA, Instituto Tecnológico de Aeronáutica
HUGO SERENO FERREIRA, Departamento de Engenharia Informática, Faculdade de Engenharia, Universidade do Porto
ADEMAR AGUIAR, Departamento de Engenharia Informática, Faculdade de Engenharia, Universidade do Porto

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. Frequently, behavior is added to AOM systems by increasingly adding expressiveness to the model. However, this approach can result in a full blown programming language, which is not desirable. This pattern describes a solution for adding behavior to AOM systems by using metadata to identify points in the application where behavior can be dynamically added. This solution may limit the expressive power of the model, but can also simplify it, since points of extension are well defined in the system.

Categories and Subject Descriptors: **D.1.5 [Programming Techniques]**: Object-oriented Programming; **D.2.11 [Software Architectures]**: Patterns

General Terms: Adaptive Object Model

Additional Key Words and Phrases: Adaptive Object Model behavior, metadata, reflection

ACM Reference Format:

Matsumoto, P. M., Correia, F. F., Yoder, J. W., Guerra, E., Ferreira, H. S. and Aguiar, A. 2011. AOM Metadata Extension Points. 18th Conference on Pattern Languages of Programs (PLoP), Portland, Oregon, USA (October 2011), 16 pages.

1. INTRODUCTION

An Adaptive Object Model (AOM) represents classes, attributes, relationships and behaviors as metadata (Yoder et al. 2001, Yoder and Johnson 2002). An AOM system provides great flexibility for applications, allowing relationships, attributes and behaviors to be changed at runtime by programmers, and sometimes by end users. These systems can be adapted more easily in an environment where business rules are rapidly changing.

AOM architectures are usually made up of several smaller patterns (Yoder et al. 2001, Yoder and Johnson 2002), such as TYPE OBJECT, PROPERTY, TYPE SQUARE, COMPOSITE, STRATEGY, RULE OBJECT and ACCOUNTABILITY. Fig. 1 depicts the core design of an AOM. More details on the AOM core design can be found in Appendix 1.

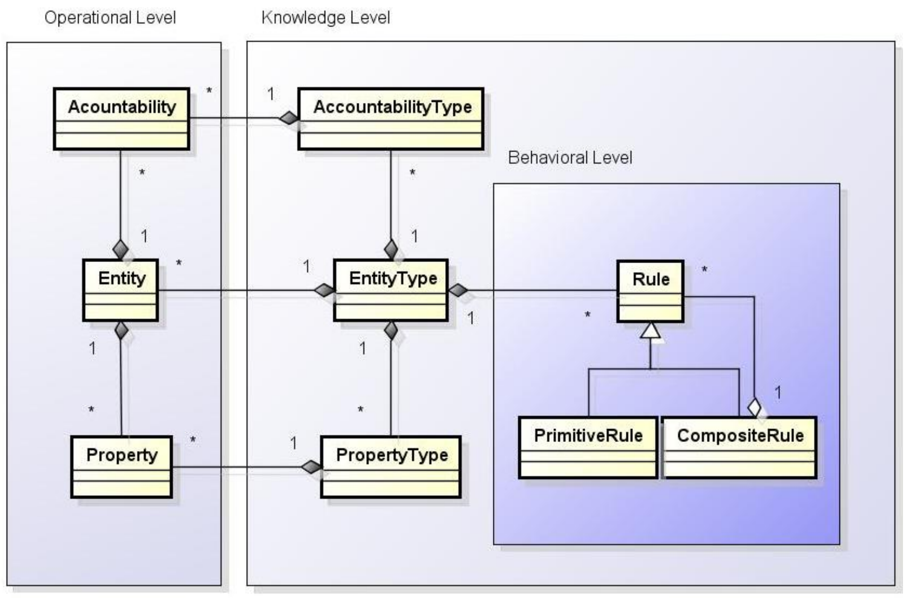

Fig. 1. AOM core design, adapted from (Yoder et al. 2001)

Besides the patterns mentioned above, several other patterns support the development of AOM applications (Welicki et al. 2007b).

An AOM is a metamodeling technique, as it supports the development of systems using models at several levels: instance, model, metamodel, etc. Each of these levels specifies how the level below can be expressed.

Structural flexibility can be achieved with patterns such as EVERYTHING IS A THING and CLOSING THE ROOF (Ferreira et al. 2010), and behavior flexibility can be achieved by extending the RULE OBJECT pattern. This allows both structure and behavior to be changed at runtime by programmers and some end-users. However, depending on the complexity of the system's behavior, it may be difficult to support the expression of a model with enough expressiveness to represent it, without creating a full-blown programming language. The pattern presented in this paper describes how specialized behavior can be easily added to an AOM system through extension points. Although this solution limits the flexibility of the system, it avoids adding unnecessary complexity to the architecture.

Considering an AOM pattern language, this pattern can be categorized in the **Metadata** group, which contains patterns that solve problems related to metadata definition in an AOM.

1.1 Intended Audience

This pattern is primarily intended for those who are building AOM systems and need to add behavior to specific points of the application, without adding too much complexity to the model. This paper assumes the reader is familiar with the basis of AOMs.

2. AOM METADATA EXTENSION POINTS

2.1 Context

AOM applications are software systems with a special focus on flexibility regarding the problem domain. These systems allow users to express domain rules as a model, in such a way that they can easily be adapted, by describing them with metadata. In an AOM, one would usually want everything related to the problem domain to be flexible, so that it can be changed easily. Behavior flexibility is usually added by extending the RULE OBJECT pattern.

Nevertheless, in order to represent complex behaviors in the AOM systems, the expressiveness of the model usually needs to be increased, which leads to increasing the overall complexity of the system. In a worst case scenario, the model supporting an AOM can become a full-blown programming language, which could provide great flexibility, but would bring unnecessary complexity and difficulty for maintaining the system.

Therefore, when developing an AOM system, one should find a balance between the power of flexibility provided by the application and the simplicity of its model. Flexibility should only be added when and where it is needed.

2.2 Problem

How to identify where and how to add new kinds of behavior to an AOM system without increasing the complexity of its model and also maintaining its flexibility?

2.3 Forces

- **Flexibility vs Specialization.** One of the key benefits of AOM systems is how they support domain flexibility. The system should only be as flexible as it needs, which means there's a tradeoff on which parts of the software are to be fixed (source-code) and which ones should be made flexible (metadata). On one hand, we can increase the expressiveness of an AOM, by extending its model for representing domain specific attributes. Generally speaking, these types of extension make an AOM more complex. On the other hand, we could add specialized behavior by simply adding source-code, in which case the flexibility of the system would be a tradeoff (you need to write a lot of source code to adapt to new requirements).
- **Complexity vs Simplicity.** Although an AOM is built to be flexible, adding too much flexibility makes the system be unnecessarily complex. The developers of an AOM application need to find a balance between the power of flexibility, which leads to a greater complexity, and the overall simplicity of the system.
- **Reusability.** The development of AOM applications demands a greater level of abstraction from the developers, compared to traditional approaches, since the class-instance relationship used for the application domain representation is replaced by an instance-instance relationship, which is not straightforward for all developers. The development of AOM applications could be eased if code could be reused among different applications.

2.4 Solution

Use metadata to enable extension points from which specialized behavior can be added to the AOM system. This behavior is expressed using source code and is integrated to the system through a framework that uses inversion of control.

Metadata resources, such as annotations (Java), custom attributes (.NET), XML files, naming conventions and interfaces, can be used to identify specific points where the application can be extended to add specialized behavior expressed using source code.

Because the extension points are hardcoded in the system, this solution provides limited flexibility. However, adding new behavior is easier than extending the RULE OBJECT pattern, since behavior is added programmatically and not by increasing the expressiveness of the model. This also means that an end user will not be able to add or change the behavior without the help of a programmer. Therefore, this solution should be applied in situations in which this kind of flexibility is not needed for the behavior.

Fig. 3 shows an outline of the solution space for the pattern. The core AOM architecture is extended by metadata that describes the extension points. Note that these map to the different possible extensions. These extensions can be of the form: 1) dynamic methods invoked through inversion of control using metadata and 2) TYPE SQUARE extension that utilizes a more intensive meta-architecture for describing your possible extensions.

The first type of extension can be implemented in two different ways: using dynamic methods kept in a single lookup class or using an extension class where the metadata describes the class and method within the extension class. The difference between these methods is that in the first one the metadatas are used to tag different methods in a single class while the second method allows different classes and methods to be tagged by metadatas to be called using inversion of control. Examples of metadata that could be used as extension points for this type of extension are interfaces and XMLs. Interfaces could determine the method signatures for extension classes and XML configurations could determine which extension classes to call on runtime.

In the second type of extension, metadata is used for getting information on the AOM application instead of serving as a point where extensions can be called using inversion of control. More specifically, the extension points metadatas define information such as which application class corresponds to an Entity, which class corresponds to an EntityType and so on. By using this information, a common AOM core structure can adapt domain-specific AOM structures and allow generic AOM frameworks to be applicable to any AOM core structure, regardles of its domain.

Fig. 2 outlines this extension type. In the figure, the annotations in the domain-specific AOM application are the extension points. More details on this type of extension can be obtained in the *Examples* section.

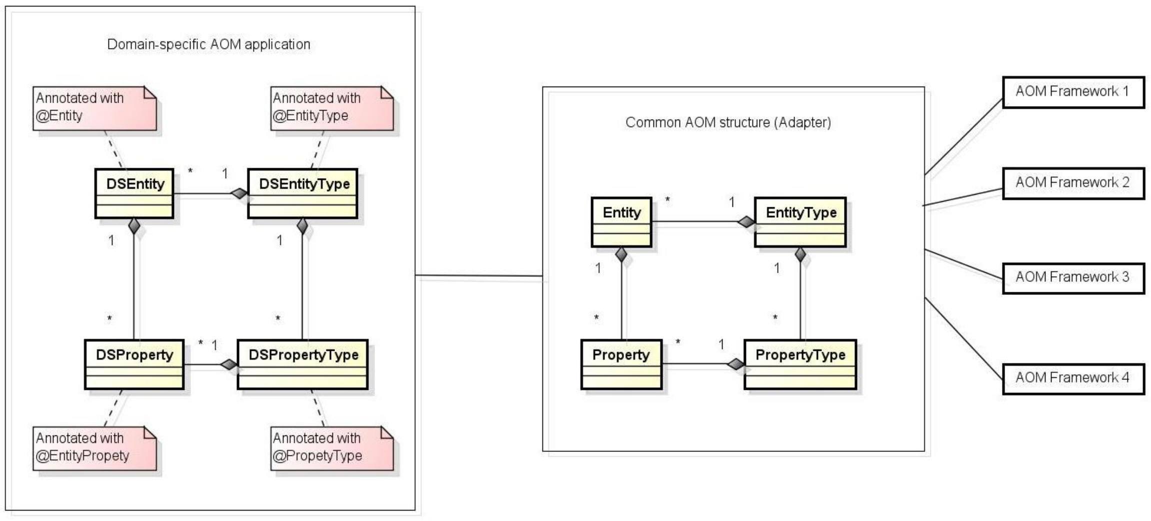

Fig. 2. Outline of the TYPE SQUARE extension type

Although inversion of control and use of metadata are techniques also used by other kinds of dynamic architectures, such as frameworks, this pattern is particularly applicable to AOMs because it is intended to solve a particular issue that is more evident in AOMs due to their highly dynamic nature – the need for a balance between the power of flexibility and simplicity.

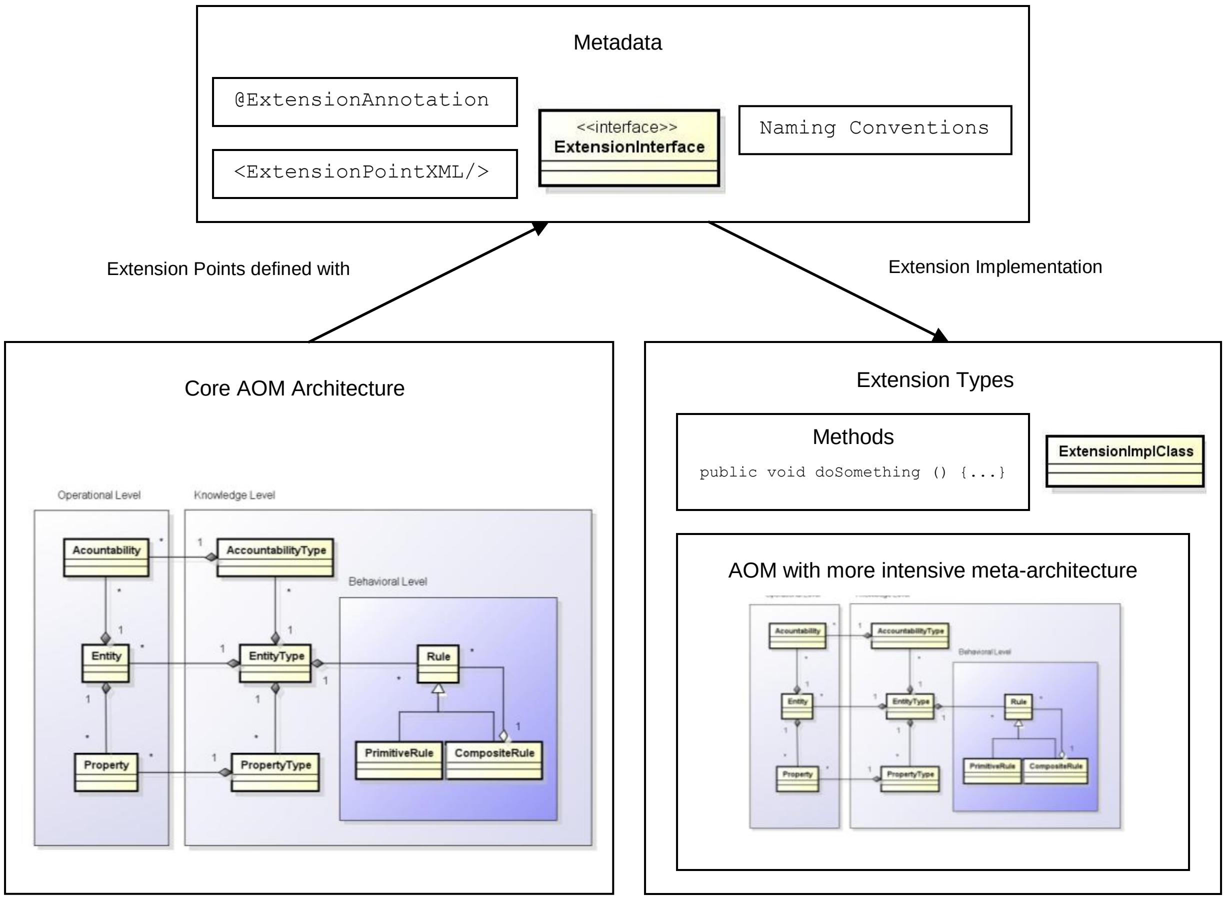

Fig. 3. Blueprint of the solution space for AOM Metadata Extension Points

2.5 Example

Like most software applications, AOM systems usually need to provide persistence mechanisms and a user interface. Besides these, there are additional requirements that are frequently needed due to the adaptive nature of the AOMs, such as version control for model objects, in order to keep the history of changes made in the domain (Ferreira et al, 2008), and end user development tools, which help the end user to make changes in the application's domain.

Additionally, there are patterns, such as the PROPERTY RENDERER (Welicki et al. 2007a), that solve issues found when implementing these requirements in an AOM application. The solution presented in 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 tightly 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 (Yoder et al. 2001, Yoder and Johnson 2002), described in section *2.5.1*; and a banking system for handling customer accounts (Riehle et al. 2000), described in section *2.5.2*. 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.

2.5.1 IDPH Medical Domain Framework

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. 4. 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).

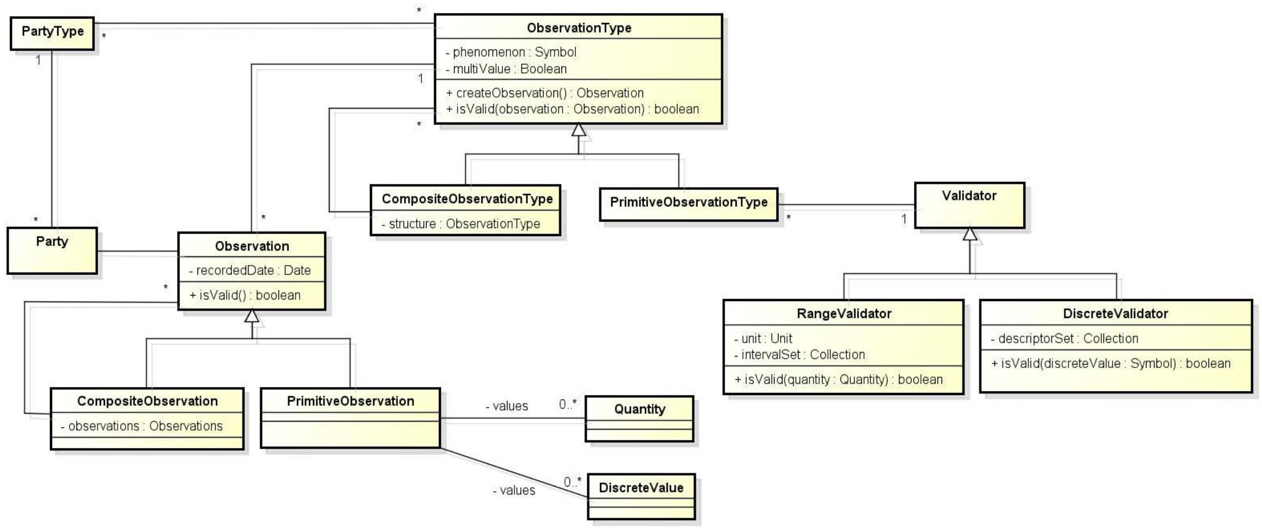

Fig. 4. IDPH Medical Framework design (Yoder et al. 2001)

2.5.2 Banking System

The example given in (Riehle et al. 2000) consists in 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. 5.

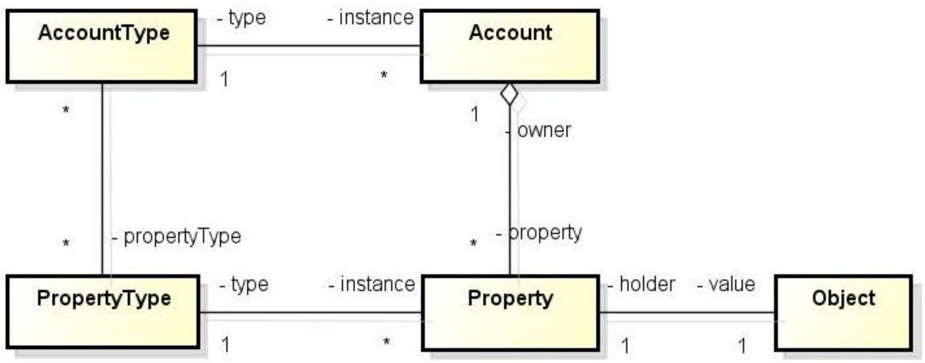

Fig. 5. Basic design for the banking system (Riehle et al. 2000)

2.5.3 Integration Problem

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 by a generic 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.

2.5.4 Example Implementation Details

The use of metadata resources, such as annotations (Java), custom attributes (.NET) or even XML configuration files, allow roles that domain-specific AOM application classes play in the AOM architecture (e.g. Entity, Entity Type, Property, Property Type, Accountability and Accountability Type) to be identified at runtime. These metadata are the extension points of the pattern and are used by classes that participate in the AOM class model that serves as an ADAPTER (Gamma et al. 1995) for the domain-specific class model. This ADAPTER AOM core can be considered an integration framework.

This solution externalizes the core structure of domain-specific AOM applications to a common AOM core structure so that the latter can be used by generic AOM frameworks, what solves the integration issue presented in the previous sections. In order to be integrated with an AOM framework that refers to the common AOM core structure, the domain-specific AOM application only has to identify the roles played by its classes in its core structure. All the responsibility for the integration is left outside the domain-specific application.

In this solution, the domain-specific AOM applications and the AOM frameworks are decoupled and the only external information that the domain-specific applications must have is the metadata to be used for identifying the roles of the classes in their AOM core structure. These metadata (in this case, the annotations @Entity, @EntityType, @EntityProperty and @PropertyType) are the extension points for the TYPE SQUARE extension type.

Fig. 6 shows the solution for the example given in the previous sections. On the left side of the figure the banking system and the IDPH Medical Framework core structures are depicted. The classes that play an AOM role in those applications are annotated so that the common AOM core structure classes (depicted on the right side) can identify these roles and adapt the domain-specific classes. The AOM frameworks use the generic AOM structure for allowing their solutions to work for applications in different domains without the need to know these domains.

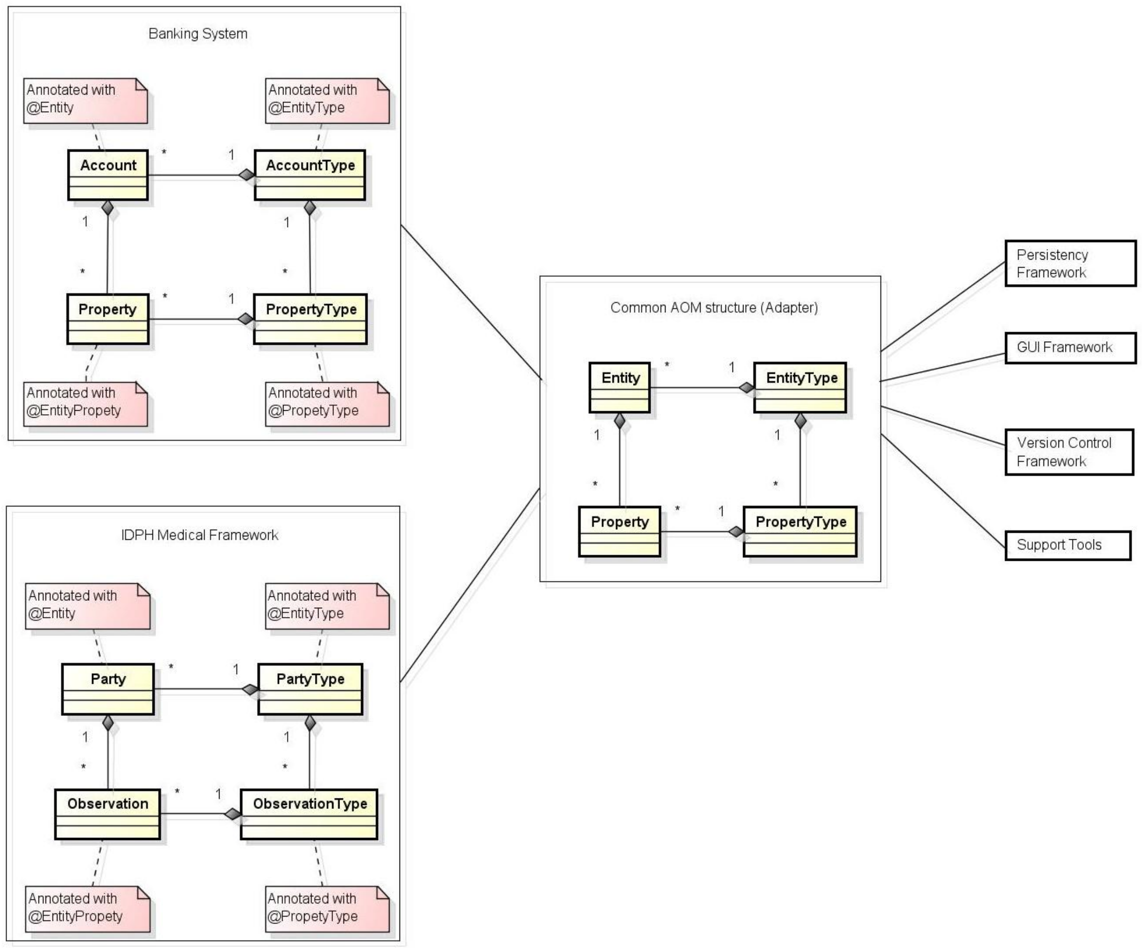

Fig. 6. TYPE SQUARE extension type for the banking system and for the IDPH Medical Framework

The generic model depicted in Fig. 6 could also include other roles, such as Accountability, Accountability Type and Rules, which will not be mentioned in this paper for simplicity (the concept for them is analogous to the Entity, Entity Type, Property and Property Type roles). The incorporation of these roles is made by creating new types of metadata.

Fig. 7 depicts an example of how the Entity Type of an Entity can be obtained by using the metadata information and the integration framework. In this example, the persistency framework queries for the Entity Type of an Entity object in the common AOM core structure. This object contains a reference to an Account object in the banking system and invokes over it a method for getting the account's corresponding AccountType using reflection. The returned object, called *accountType* in this example, is sent as argument to the static method *getEntityType* in the EntityType class of the common AOM core structure. This method uses an EntityTypeObjectMap to check whether an EntityType object which adapts the *accountType* object has already been created. If so, this EntityType object is returned. Otherwise, a new EntityType object which adapts the *accountType* object is created, registered in the EntityTypeObjectMap and returned by the method. Finally, the object returned by the method is sent back to the persistency framework.

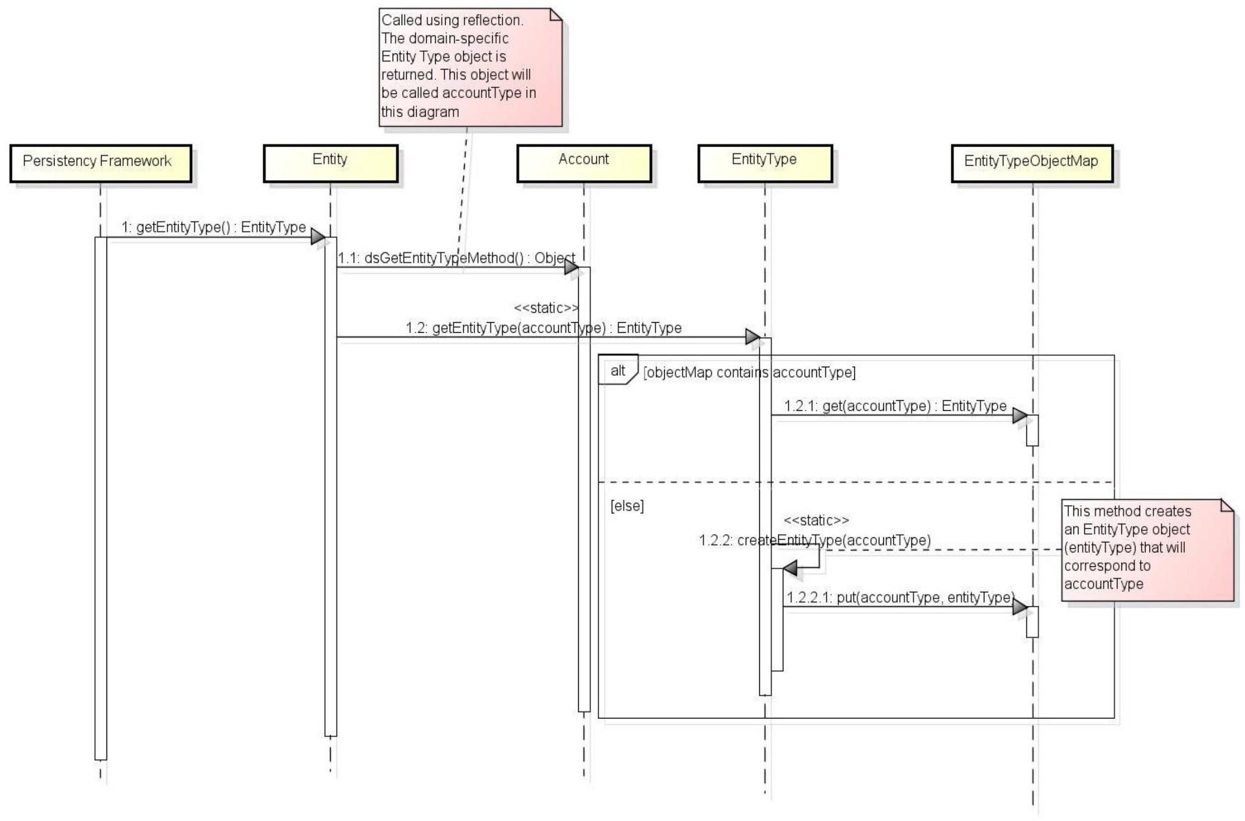
Called using reflection. The domain-specific Entity Type object is returned. This object will be called accountType in this diagram
Persistency Framework
Entity
Account
EntityType
EntityTypeObjectMap
1: getEntityType() : EntityType
1.1: dsGetEntityTypeMethod() : Object
<<static>>
1.2: getEntityType(accountType) : EntityType
alt
[objectMap contains accountType]
1.2.1: get(accountType) : EntityType
[else]
<<static>>
1.2.2: createEntityType(accountType)
This method creates an EntityType object (entityType) that will correspond to accountType
1.2.2.1: put(accountType, entityType)

Fig. 7. Implementation flow for getting the Entity Type

Some examples of annotations that could be created in order to identify the AOM roles played by domain-specific classes are shown below (the examples given in this paper are based on annotations, but role representations are analogous for .NET custom attributes and for XML configuration files):

- @EntityType: used to indicate that a class plays an Entity Type role in the TYPE SQUARE pattern
- @Entity: used to indicate that a class plays an Entity role in the TYPE SQUARE pattern
- @PropertyType: used to indicate that a class plays a Property Type role in the TYPE SQUARE pattern
- @EntityProperty: used to indicate that a class plays a Property role in the TYPE SQUARE pattern

Other kinds of annotations or variations of the above annotations can be created in order to identify methods and fields that contain AOM role information in the common AOM core structure classes.

The situation described above considers two systems, modeled using AOM, which solve different problems in different domains. Following the solution presented above, the *PartyType* and *AccountType* classes can be annotated with @EntityType; the *Party* and *Account* classes can be annotated with @Entity; and so on. Fig. 8 shows simple implementations of the *Party* and *Account* classes with AOM roles annotations being used.

```
@Entity
public class Party {

  @EntityType
  private PartyType partyType;

  @EntityProperty
  private List<Observation> observations;

  public PartyType getPartyType() {
    return partyType;
  }
```

```
@Entity
public class Account {

  @EntityType
  private AccountType accountType;

  @EntityProperty
  private List<Property> properties;

  public AccountType getAccountType() {
    return accountType;
  }
  public void setPartyType(PartyType partyType)
  {
    this.partyType = partyType;
  }

  public List<Observation> getObservations() {
    return observations;
  }

  public void addObservation(Observation
                              observation) {
    observations.add(observation);
  }
}
```

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

  public List<Property> getProperties() {
    return properties;
  }

  public void addProperty(Property property) {
    properties.add(property);
  }

}
```

Fig. 8. Simple implementation of the Party and Account classes that shows AOM role annotations being used

An AOM framework that handles a common requirement in AOM systems, such as persistence, can refer to the common AOM core structure classes. These classes will adapt the classes in the IDPH and the banking systems according to a configuration, making the solution implemented by the AOM framework applicable to both systems, even though the framework does not know any of the domains.

Fig. 9 shows an example for the implementation of the Entity class for the common AOM core structure. This class would be an ADAPTER for the classes annotated with @Entity. The implementation of the classes representing the other AOM roles would be analogous to the implementation shown below.

```
public class Entity {

  // Attribute to store the instance of the domain-specific class
  private Object dsEntity;

  // Method for getting the Entity Type
  private Method getEntityTypeMethod;

  // ... Other attributes, such as method for getting Properties

  private Entity () {}

  public static Entity createEntity (String entityClass)
  {
     // Exception handling code was ommitted

     Entity entity = new Entity();
     Class entityClazz = Class.forName(entityClass);
     entity.setDsEntity(entityClazz.newInstance());
     Field[] fields = entityClazz.getDeclaredFields();
     for (Field f : fields)
     {
        EntityType entityTypeAnnotation = f.getAnnotation(EntityType.class);

        if (entityTypeAnnotation != null)
        {
           // Identifying the method for getting the Entity Type
           String fieldName = Utils.firstLetterInUppercase(f.getName());
           String getEntityTypeMethod = "get" + fieldName;

           Method getMethod = entityClazz.getMethod(getEntityTypeMethod);
           if (getMethod != null)
                entity.setGetEntityTypeMethod(getMethod);

        }

        // ...
     }
     return entity;
  }

  // Method used by the AOM frameworks
  public EntityType getEntityType(){
    // Ommitted exception handling code
    // The getEntityType method returns an EntityType class instance that
    // corresponds to the domain-specific object returned by the
    // getEntityTypeMethod. It guarantees that there is a one-to-one
    // relationship between instances in the generic and domain-specific
    // models
    return EntityType.getEntityType(getEntityTypeMethod.invoke(dsEntity));
  }

  // ...

}
```

Fig. 9. Example of code for the Entity class in the common AOM core structure

This solution eases the process of AOM application development, once it allows generic solutions for AOMs to be adapted to the domain-specific AOM applications. Through the use of metadata, the domain-specific AOM core can serve as an extension point for adding behavior provided by generic AOM frameworks to the application. Different AOM applications can use the AOM Role Mapper framework in order to integrate with generic AOM frameworks, which results in code reuse.

2.6 Consequences

(+) The AOM applications can be easily extended using the metadata to identify possible extension points in the system.

(+) Code for the extended behavior can be reused among different AOM applications if the same integration framework is used.

(+) Extended behavior is decoupled from the system and can be easily changed.

(+) If XML or other external metadata is used for implementing this pattern, the domain-specific application can be decoupled from the extension points metadata.

(-) In order to identify the extension points and/or load the extension code, there is usually a need for reflection, which can impact performance.

(-) Using this pattern, the system can only be extended in the points where the extension metadata were used.

(-) If metadata that is embedded in the code, such as interfaces, annotations or custom attributes, is used for implementing the pattern, the applications become coupled with these metadata, since they must be annotated with them.

2.7 Related Patterns

The PLUGIN (Fowler 2003) pattern can be used for implementing this pattern. The PLUGIN would be a form of extension type that uses inversion of control.

The DYNAMIC HOOK POINTS (Acherkan et al. 2011) can be implemented using this pattern, where the extension points are determined by interfaces and the Dynamic Hook framework mentioned in the paper corresponds to the integration framework.

The METADATA MAPPING (Fowler 2003) pattern uses metadata to map between two different representations (relational and object-oriented). It is similar to this pattern in the sense that this pattern maps between two different representations two: one based on metadata (i.e., the model) and one based on object oriented source code.

When this pattern is implemented for allowing a TYPE SQUARE extension type, the ENTITY MAPPING (Guerra et al. 2010) pattern is used, since the AOM domain-specific application classes are mapped to the common AOM core structure by using metadata.

(Pree 2000) states that flexibility should be injected into frameworks in appropriated doses and adaptation should take place at points of predefined refinement – the *hot spots*. The extension points defined in this pattern can be corresponded to the *hot spots* defined in (Pree 2000).

2.8 Known Uses

Oghma (Correia and Ferreira 2008) is an AOM framework that implements this pattern. On systems developed with Oghma, a great part of the behavior, like persistency, querying and user-interface, is driven by the underlying model. While the default behavior covers a very wide range of needs, it sometimes has to be different for some model elements. In such cases, the developer can create “real” classes, using regular source-code, and bind them to the model elements which behavior needs to be specialized. The way to do this in Oghma is through .NET custom attributes, by annotating classes with the name of the corresponding model elements.

The AOM Role Mapper (AOM Role Mapper Project) is a project under development that implements the pattern with the TYPE SQUARE extension. This framework is being developed in Java, and uses annotations to declare the roles of domain-specific application classes in the AOM model as the extension points (e.g. @Entity and @EntityType, @EntityProperty and @PropertyType). The AOM Role Mapper framework adapts the domain-specific AOM applications’ core structures, marked with the annotations, to a common AOM core structure within the framework. The latter can be used for integrating generic frameworks (e.g. a persistency framework) with the domain-specific AOM applications.

The implementation of the AOM Role Mapper framework is similar to what is described in the *Example Implementation Details* section. In order to provide a flexible implementation for the framework’s metadata handling, the patterns of the pattern language for metadata-based frameworks described in (Guerra et al. 2009) are used.

The Refactory (www.TheRefactory.com) developed various adaptive systems for one of their clients in C#/.NET that used a variation of this pattern implemented through DYNAMIC HOOK POINTS to define known places (hooks) to add new behavior. One dynamic hook point in the Import system was for adding new rules. New rules can be added by creating a DLL, which contains a subclass of ValidationRule. This class will be tagged with the name of the validation rule and have a Validate() method which is invoked during the validation process. By including the DLL in the config file that specifies what will dynamically loaded, you can easily add new rules that can be used by the Import Process. The following is a simplified definition for the InvalidIdValidationRule class. It is for a rule that makes sure invalid ids are not accepted during the import of orders.

```
[ValidationRule("Invalid Id")] public class InvalidIdValidationRule : ValidationRule {
public InvalidIdValidationRule() : base() { }
                public override void Validate(ImportContext context) {
…}
```

Different rules can be invoked based on client-specified values stored in the database. A common ImportContext was passed in that could be used as the context for the new rules. A dynamic “metadata” tag such as "Invalid Id" could be used for associating the rule in the import language to designate the new rule and when to invoke and run the new rule. These hook points are implemented through well defined Metadata Extension Points.

A medical-based AOM system developed by The Refactory for the Illinois Department of Public Health (Yoder and Johnson 2002) is another example of a system that extensively uses DYNAMIC HOOK POINTS. In this system, reflection is also used to dynamically bind hook points. Custom behavior can be described as a dynamic method or strategy associated with new types of objects. Thus a new class can be created, and by using reflection, the new behavior can be dynamically associated with new types of diseases and invoked using stored descriptive information.

Pontis Ltd. (www.pontis.com) is a provider of Online Marketing solutions for Communication Service Providers. Pontis’ Marketing Delivery Platform allows for on-site customization and model evolution by non-programmers. The system is developed using ModelTalk (Hen-Tov et al. 2009) based on AOM patterns. Pontis’ system is deployed in over 20 customer sites including Tier I Telcos. A typical customer system handles tens of millions of transactions a day exhibiting Telco-Grade performance and robustness.

There are also well-known non-AOM uses of metadata extension points such in the Spring, Apache, Eclipse, and NetBeans frameworks. They have different implementations with similar intent - the ability to support the definition of extention points and the ability to dynamically invoke new behavior at certain pre-defined places.

2.9 Summary

This paper presents a pattern for adding new behavior to AOMs by using extension points defined by metadata. This metadata can be in the form of annotations, XML, interfaces and/or naming conventions. This solution adds flexibility in determined points of the system, allowing new behavior to be added without a lot of programming, thus avoiding the creation of a full blown programming language. With this approach, new behavior can be added to an existing AOM by writing the new behavior and linking it in through metadata extension points.

2.10 Acknowledgements

We are deeply grateful to our shepherd, Hans Wegener for his valuable comments and contributions during the PLoP 2011 Shepherding process, and to PLoP 2011 Writers' Workshop group members, Rebecca Wirfs-Brock, Jyothish Maniyath and Sérgio Barbosa, for their helpful feedback.

We would also like to thank the essential support to this research given by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo, Brasil), and by ParadigmaXis and FCT (Fundação para a Ciência e Tecnologia, Portugal), which also partially financed this work through the grant number SFRH/BDE/33883/2009.

REFERENCES

AOM Role Mapper Project. Available at: <http://sourceforge.net/projects/esfinge/>. Accessed in: 2011-11-21.

Acherkan, E., Hen-Tov, A., Schachter, L., Lorenz, D. H., Wirfs-Brock, R., Yoder, J. W. Dynamic Hook Points. In *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs (AsianPLoP2011)*. Tokyo. Japan.

Correia, F. F. and Ferreira, H. S. 2008. Trends on Adaptive Object Models Research. In *Proceedings of the Doctoral Symposium on Informatics Engineering 2008*, Porto, Portugal.

Ferreira, H. S. and Correia F. F., Welicki L. 2008. Patterns for data and metadata evolution in adaptive object-models. *In Proceedings of the 15th Conference on Pattern Languages of Programs*. Nashville, Tennessee, USA.

Ferreira, H. S., Correia, F. F., Yoder, J. W. and Aguiar, A. 2010. Core Patterns of Object-Oriented Meta-Architectures. In *Proceedings of the 17th Conference on Pattern Languages of Programs (PLoP2010)*, Reno, Nevada. USA.

Foote, B. and Yoder, J. W. 1998. Metadata and Active Object Models. In *Proceedings of the 5th Conference on Pattern Languages of Programs (PLoP1998)*, Monticello, Illinois. USA.

Fowler, M. 2003. Patterns of Enterprise Application Architecture. *Addison-Wesley*.

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

Guerra, E., Souza, J. T. and Fernandes, C. 2009. A Pattern Language for Metadata-based Frameworks. In *Proceedings of the 16th Conference on Pattern Languages of Programs (PLoP2009)*, Chicago, Illinois. USA.

Guerra, E., Fernandes, C. and Silveira, F. F. 2010. Architectural Patterns for Metadata-based Frameworks Usage. In *Proceedings of the 17th Conference on Pattern Languages of Programs (PLoP2010)*, Reno, Nevada. USA.

Hen-Tov, A., Lorenz, D. H., Pinhasi, A., Schachter, L. 2009. ModelTalk: When Everything Is a Domain-Specific Language. *IEEE Software*, vol. 26, no. 4, pp. 39-46.

Johnson, R. and Wolf, B. 1998. Type Object. Pattern Languages of Program Design 3. *Addisson-Wesley*.

Pree, W. 2000. Hot-Spot-Driven Framework Development. In: Fayad, R. J. M., Schmidt, D. (Eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design. *Wiley & Sons*.

Riehle, D., Tilman, M. and Johnson, R. 2000. Dynamic Object Model. In *Proceedings of the 2000 Conference on Pattern Languages of Programs (PLoP 2000)*, Washington University Department of Computer Science.

Welicki, L, Yoder, J. W. and Wirfs-Brock, R. 2007. Rendering Patterns for Adaptive Object Models. In *Proceedings of the 14th Pattern Language of Programs Conference (PLoP 2007)*, Monticello, Illinois, USA.

Welicki, L., Yoder, J. W., Wirfs-Brock, R. and Johnson, R. E. 2007. 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, Canada.

Yoder, J. W., Balaguer, F. and Johnson, R. 2001. Architecture and Design of Adaptive Object-Models. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*, Tampa, Florida, USA.

Yoder, J. W. and Johnson, R. 2002. 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.

3. APPENDIX A- A BRIEF SUMMARY OF THE ARCHITECTURAL STYLE OF AOMS

Important Notice: This section is a summary extracted from (Yoder et al. 2001) and (Yoder and Johnson 2002) 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 that 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) that 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 (Johnson and Wolf 1998) 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 (Foote and Yoder 1998).

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.

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. Sometimes objects differ only in having different properties. Fig. 10 represents the resulting architecture after applying these two patterns, which we call TYPE SQUARE (Yoder et al. 2001).

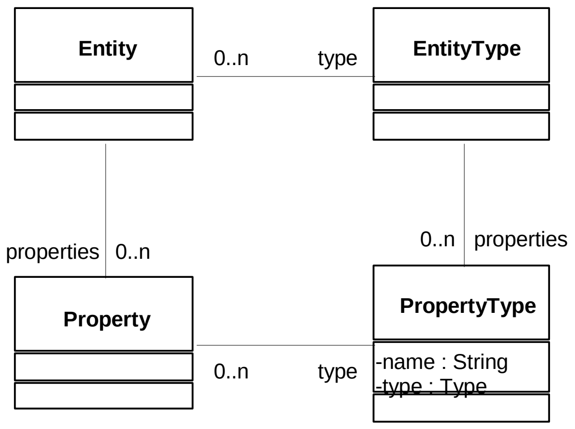

Fig. 10. TYPE SQUARE

As is common in Entity-Relationship modeling, an Adaptive Object-Model usually separates attributes from relationships. In these cases the TYPE OBJECT pattern is applied again to define the legal relationships between types of Entities.

The STRATEGY pattern (Gamma et al. 1995) 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. Adaptive Object-Models are usually built from applying one or more of these patterns in conjunction with other design patterns such as COMPOSITE, INTERPRETER, and BUILDER (Gamma et al. 1995) (see Fig. 11).

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 TYPE OBJECTS, 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.

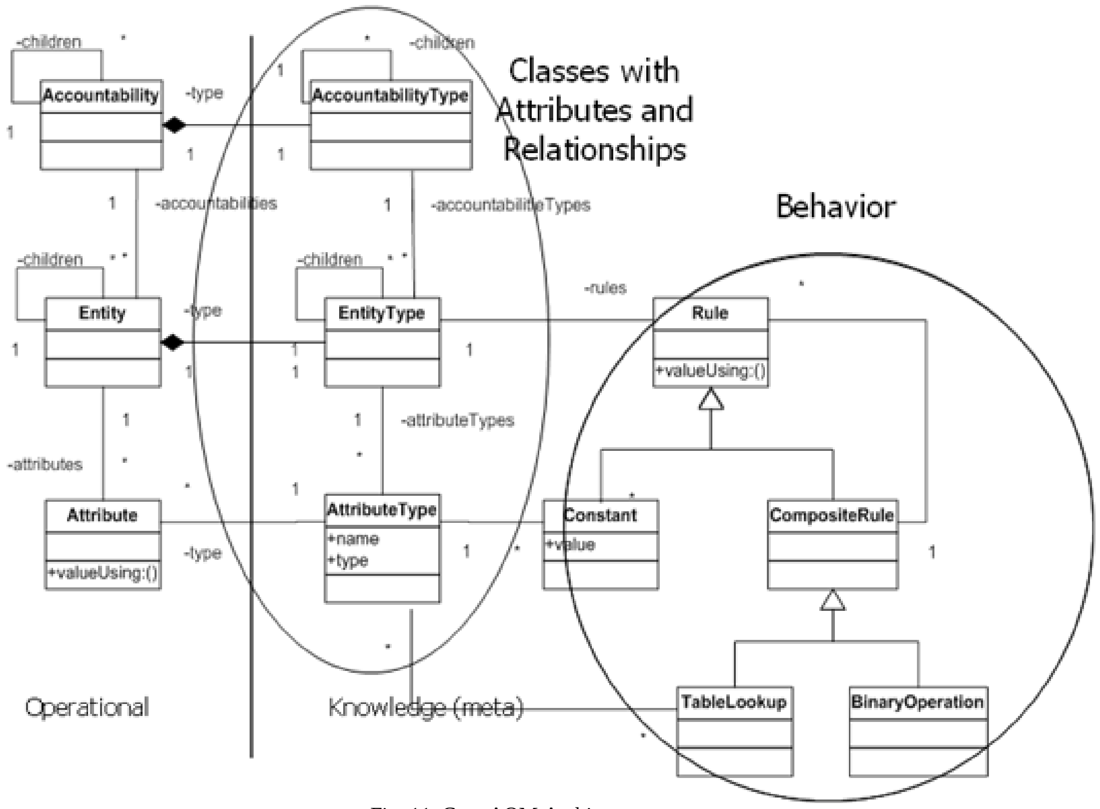

Fig. 11. Core AOM Architecture