# AOM Metadata Extension Points

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

The 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 added. This solution allows code to be used to express behavior, which simplifies its implementation and also eases the reuse of this new behavior code among different applications.

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

## 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.

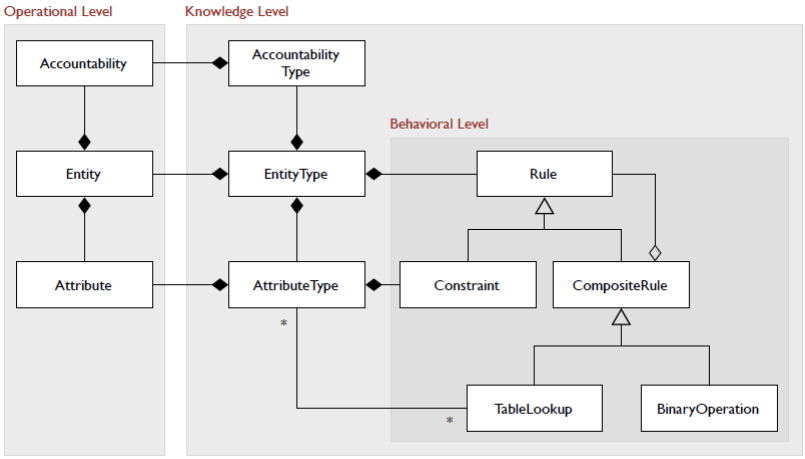


Fig. 1. AOM core design, adapted from (Yoder et al. 2001)

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.

Some patterns, such as EVERYTHING IS A THING and CLOSING THE ROOF (Ferreira et al. 2010) support structural flexibility at the model levels.

A flexible way of adding behavior to AOMs is to extend the RULE OBJECT pattern. This solution allows behavior to be changed at runtime by programmers and end users. However, depending on the complexity of this behavior,it may be difficult to support enough expressiveness to represent it in the model 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 by the use of specific extension points in the system. Although this solution limits the flexibility of the system, it avoids the addition of unnecessary complexity in 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.

## 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 a way that 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. But, sometimes you need more power than what a model expressed by data can provide.

This situation can be observed when adding complex behavior to AOMs. In order to make the model flexible, one could extend the RULE OBJECT pattern in order to add new behavior to the system. However, it may not be easy to add the necessary expressiveness to represent the behavior without creating a full-blown programming language, which is undesirable.

### 2.2 Problem

*How to add new kinds of behavior to an AOM system without increasing the expressiveness of its model and maintaining its flexibility?*

### 2.3 Forces

- **Flexibility vs Specialization.** One of the advantages of AOM systems is flexibility. However, a system should only be as flexible as it needs, otherwise it can become unnecessarily complex and it can result in a full-blown generic programming language. An important tradeoff to an AOM architecture is deciding on which parts of the software is to be fixed (source-code) and which parts of the software can 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).
- **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, since it is done using source code and the deployment of this behavior

is straightforward once the integration framework is implemented. Another advantage of this pattern is that the implementation of some behaviors can be reused among different applications if the same integration framework is used.

Fig. 2 shows an example solution. 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 kept in a single lookup class, 2) extension class where the metadata describes the class and method within the extension class, and 3) TYPE SQUARE extension that utilizes a more intensive meta-architecture for describing your possible extensions.

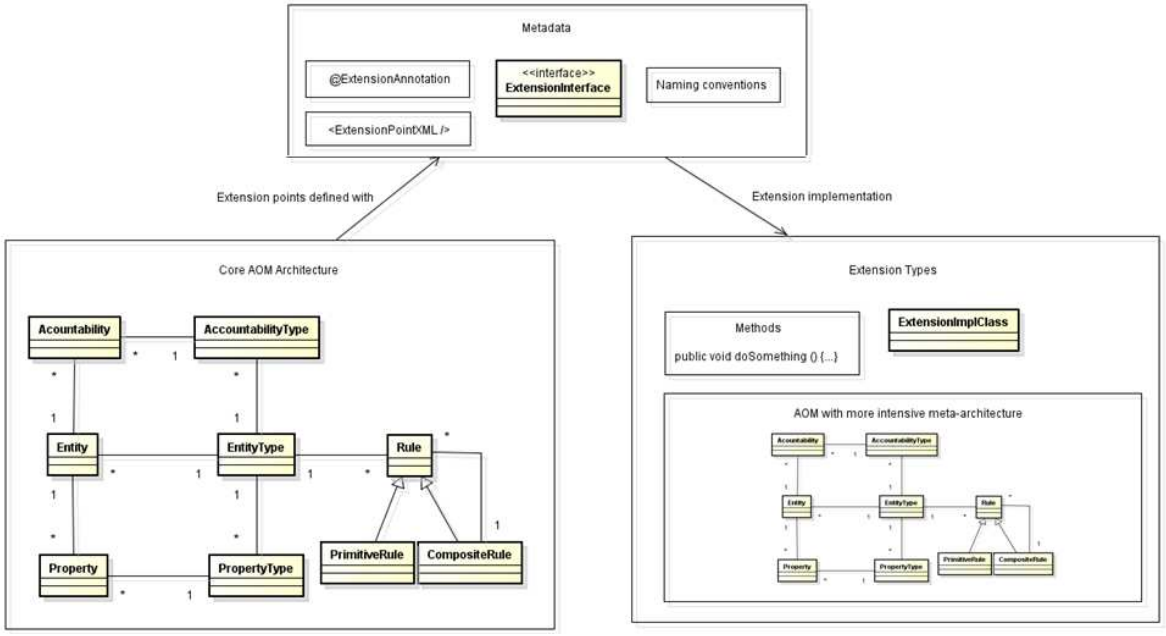


Fig. 2. Example of the solution of the pattern

## 2.5 Example

Like most of the 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 using domain-agnostic AOM frameworks.

In order to illustrate the issue, two systems that were modeled using AOM are considered: the Illinois Department of Public Health (IDPH) Medical Domain Framework (Yoder et al. 2001, Yoder and Johnson 2002) and a banking system for handling customer accounts (Riehle et al. 2000).

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. 3. 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).

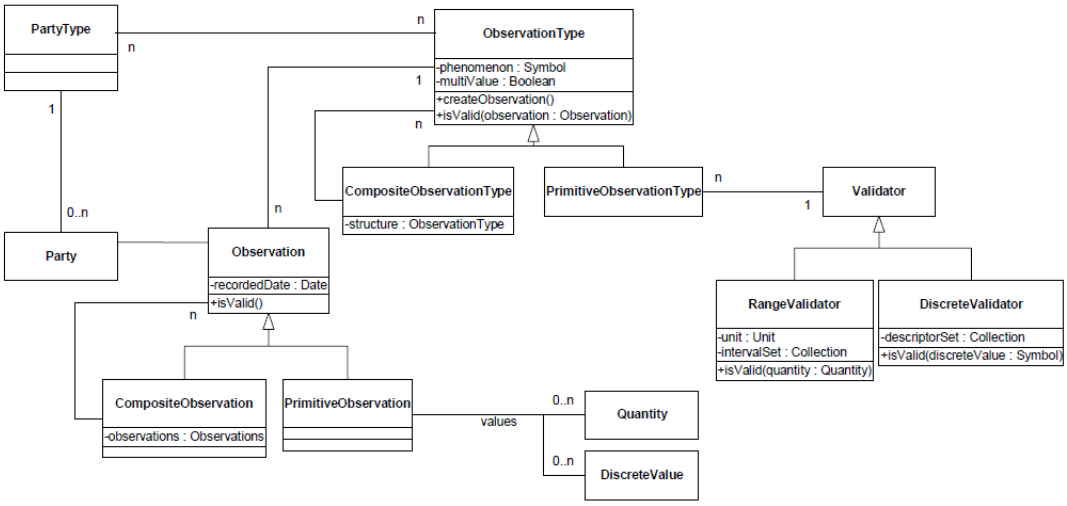


Fig. 3. IDPH Medical Framework design (Yoder et al. 2001)

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. 4.

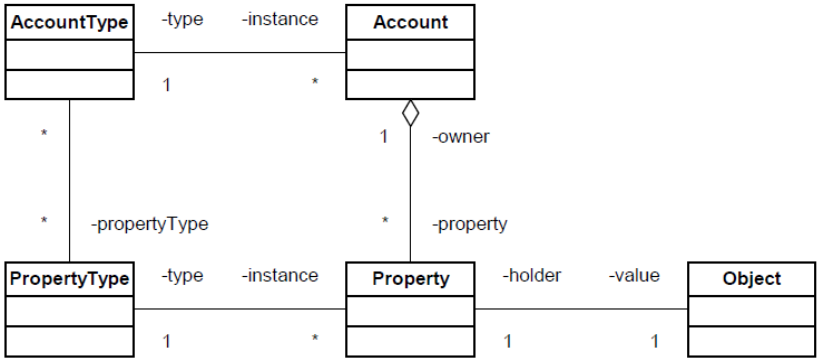


Fig. 4. Basic design for the banking system (Riehle et al. 2000)

Solutions for other kinds of concerns for the system could be described, but this simple model is sufficient for the purpose of this pattern.

Notice the similarities between the structures used in the systems outlined above, such as the usage of the TYPE SQUARE pattern. Despite these similarities, both 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. Additionally the different types of rules for banking are quite distinct from the rules for medical observations.

Although the systems share some common core patterns, a solution 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. 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, making the solutions reusable between the two applications.

### 2.5.1 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 1998) 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 domain-agnostic AOM core structure so that the latter can be used by domain-agnostic 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 domain-agnostic 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.

Fig. 5 shows the solution for a simple domain-specific AOM application. On the left side of the figure the domain-specific AOM application is depicted. The classes that play an AOM role in this application are annotated so that the domain-agnostic 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.

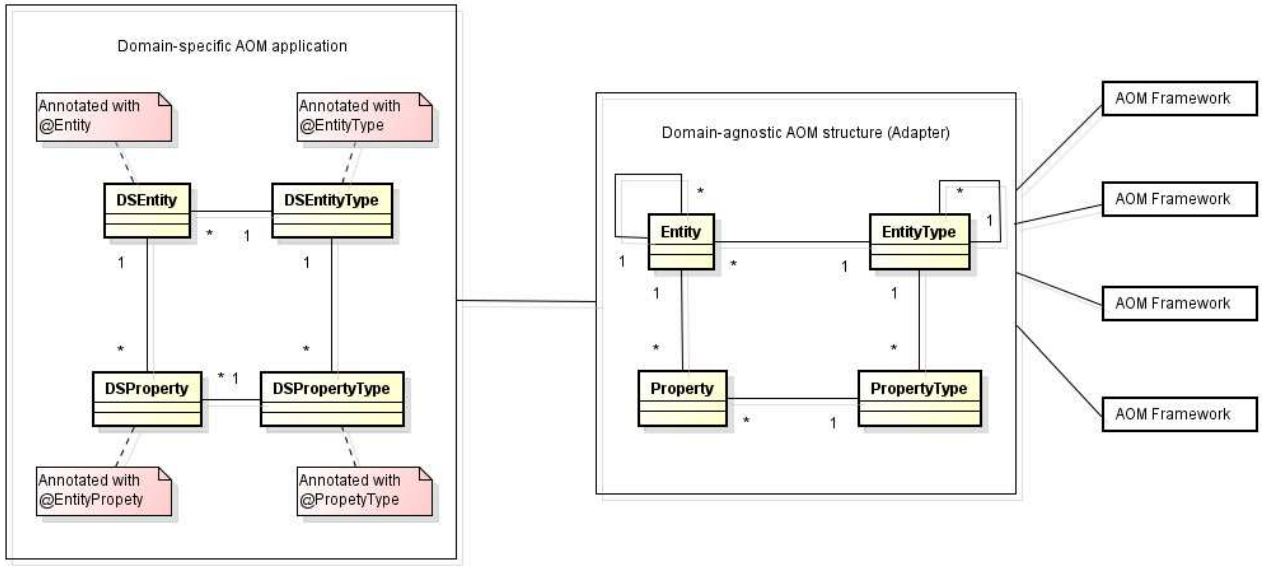


Fig. 5. Extension of a domain-specific AOM application

The generic model depicted in Fig. 5 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. 6 depicts an example of how the Entity Type of an Entity can be obtained by using the metadata information and the integration framework.

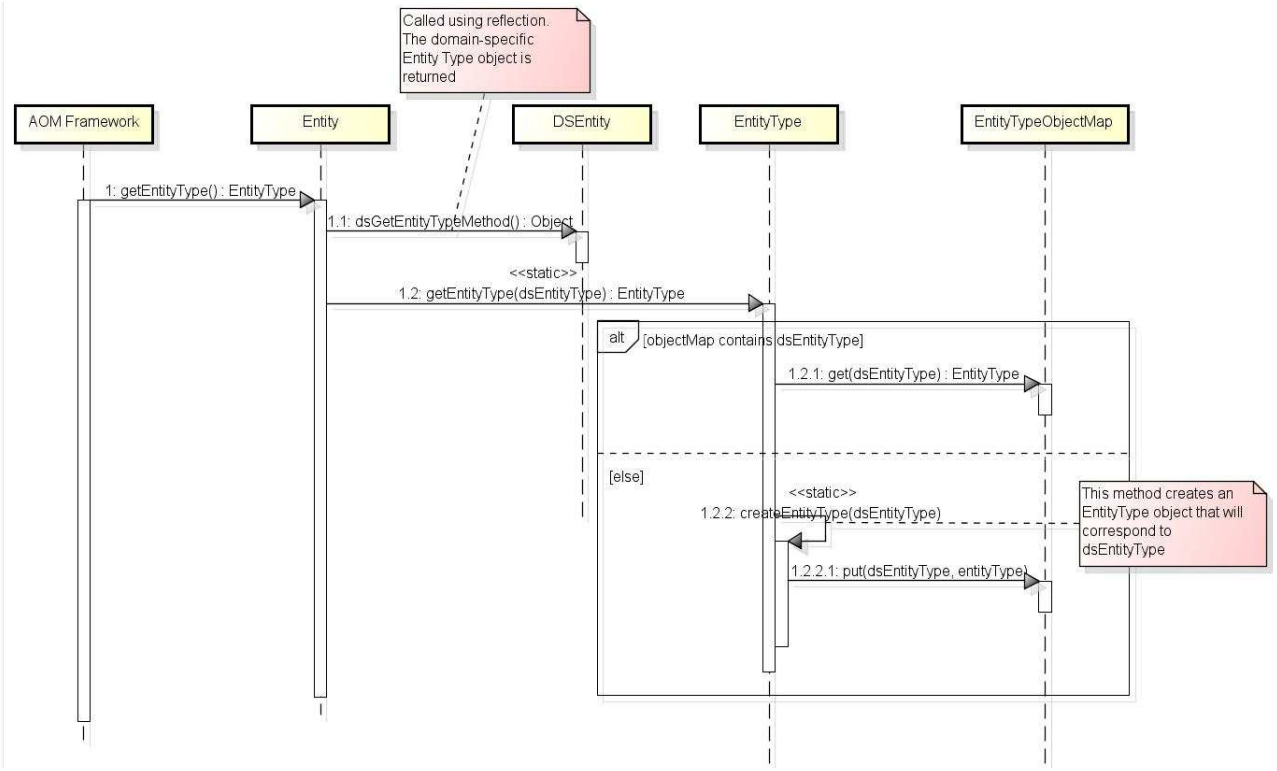


Fig. 6. Implementation flow for getting the Entity Type

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 domain-agnostic 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. 7 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;
  }

  public void setPartyType(PartyType partyType)
  {
```

```
@Entity
public class Account {

  @EntityType
  private AccountType accountType;

  @EntityProperty
  private List<Property> properties;

  public AccountType getAccountType() {
    return accountType;
  }

  public void setAccountType(AccountType
                                accountType) {
```

```
    this.partyType = partyType;
  }

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

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

```
    this.accountType = accountType;
  }

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

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

}
```

Fig. 7. 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 domain-agnostic 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. 8 shows an example for the implementation of the Entity class for the AOM domain-agnostic 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. 8. Example of code for the Entity class in the domain-agnostic 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.

(-) 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 annotations or custom attributes, is used for implementing the pattern, the applications become coupled with these metadata, since they must be annotated with them. This coupling is avoided if interfaces, XML or other external metadata is used for identifying the extension points.

## 2.7 Related Patterns

The PLUGIN (Fowler, M. 2003) pattern can be used in the integration framework.

The DYNAMIC HOOK POINTS (Acherkan et al. 2011) is a specialization of 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 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. (Fowler, M. 2003)

EXTENSION OBJECTS (Gamma, E. 1996) allow a given class to be extended by providing an additional interface.

## 2.8 Known Uses

Oghma (Correia and Ferreira 2008) is an AOM framework that uses this pattern, implementing it with .NET custom attributes.

The AOM Role Mapper (AOM Role Mapper Project) is a project under development that implements the pattern.

There is an Invoicing and Import System developed by The Refactory, which used a variation of this pattern implemented through DYNAMIC HOOK POINTS.

The Illinois Department of Public Health used a variation of this pattern for an AOM implementation for various medical systems including Newborn Screening and a Refugee System.

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.

## 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. Thus 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.

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

REFERENCES

AOM Role Mapper Project. Available at: <https://sourceforge.net/projects/aomrolemapper>. Accessed in: 2011-05-21.

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.

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.

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

Gamma, E. 1996. The Extension Objects Pattern, In *Proceedings of the 3rd Conference on Pattern Languages of Programs (PLoP 1996) PLoP96.*

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1998. Design Patterns: Elements of Reusable Object Oriented Software. *Addison-Wesley*.

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.

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.