Support for Refactoring an Application towards an Adaptive Object Model

Eduardo Guerra[1] and Ademar Aguiar[2]

[1] National Institute for Space Research (INPE) - Laboratory of Computing and Applied Mathematics (LAC) - P.O. Box 515 – 12227-010 - São José dos Campos, SP, Brazil
[2] Departamento de Engenharia Informática, Faculdade de Engenharia Universidade do Porto (FEUP), Rua Dr. Roberto Frias, 4200-465 Porto, Portugal

Abstract. Flexibility requirements can appear in the middle of a software development, perceived by several client requests to change the application. A flexible domain model, usually implemented with using the adaptive object model (AOM) architectural style, required custom-made components to handle the current implementation of the domain entities. The problem is that by evolving an AOM model, the components need to be evolved as well, which generates constant rework. This work studied the possible AOM evolution paths, in order to provide support in the components for model changing. An evolution of the Esfinge AOM RoleMapper framework were developed to provide this functionality, allowing AOM models in different stages to be mapped to a single structure. The study was evaluated using a set of tests that were applied in each possible structure for the model.

Keywords: framework, refactoring, adaptive object model, metadata, software design, software architecture, and software evolution.

1 Introduction

An adaptive object model (AOM) is an architectural style where the types are defined as instances, allowing their change at runtime [1, 2]. Metadata about the types are stored in external sources, such as XML files or databases, and used to create the application domain model. By using this architectural style, new entity types can be created and existing entity types can be changed by application users. This kind of architecture is suitable for systems in which the evolution of the domain model is part of the business processes. There are documented case studies that use AOM in fields like insurance [3], health [2] and archeology [4].

The requirement for domain model flexibility is usually perceived in the middle of a software project. The initial domain model usually reflects the current user needs when the system was requested. However, during the development, when several customer requests are made to change the domain model, some flexibility should be introduced in the entities to allow changes to be made at runtime, without having to change the software code. Following this approach, some information that was defined as domain classes' code is now defined as data that describes the domain entities. This way, it is easier to be changed at runtime.

B. Murgante et al. (Eds.): ICCSA 2014, Part V, LNCS 8583, pp. 73–89, 2014.

When flexibility requirements appear in the middle of the project, the AOM patterns [5] are usually applied gradually through the iterations. A problem that happens with this evolution is that the other application components need to be change accordingly to the domain model implementation. For instance, architectural components, such as for persistence and to manage graphical interfaces, which are created for a fixed domain model, are usually not able to handle a flexible domain model. Even for different degrees of flexibility, the implementation of such components needs to change.

The goal of the research work presented in this paper is to map the paths for refactoring a domain model to an AOM, providing a framework support to allow this evolution without needing to change the existing components. This work is based on the Esfinge AOM RoleMapper framework [6], which uses annotations to map a domain specific AOM implementation to a general AOM implementation. The research work performed included an evolution of this framework to support the mapping of AOM patterns implementations in different stages. Several models on different stages of evolution were mapped using Esfinge AOM RoleMapper to validate the viability of this model implementation.

This paper is organized as follows: section 2 presents the AOM model and the patterns in which it is based on; section 3 talks about metadata-based frameworks, which are used in the framework that implemented the proposed model; section 4 presents the framework Esfinge AOM Role Mapper; section 5 introduces the possible evolution paths that a domain model can have towards an AOM; section 6 presents the support for the evolution path implemented in Esfinge AOM Role Mapper and how each stage can be mapped using metadata; section 7 presents an evaluation of the proposed solution; and, finally, section 8 concludes this paper and presents some future directions.

2 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 [4].

Adaptive Object Model (AOM) is an architectural style where the types, attributes, relationships and behaviors are described as instances, allowing them to be changed at runtime [1, 2]. The information about the types, which is the entities metadata, is stored in an external source, such as a database or an XML file, for the application to be able to easily read and change it. This model is described through a set of patterns, which can be combined in the same application to implement a flexible domain model. An important point is that it is not mandatory to implement always all the patterns, and only the ones that are suitable to the application requirements should be used.

The following describes briefly the core patterns of an AOM architecture. The names highlighted in bold and italic represent important elements of an AOM.

- `Type Object` [7]: This pattern should be used in scenarios where the number of subtypes of an ***Entity*** cannot be determined at development time. This pattern solves this issue by creating a class whose instances represent the subtypes at runtime, which is called the ***Entity Type***. Then, to determine the type, ***Entity*** instances are composed by instances of ***Entity Type***.
- `Properties` [8]: This pattern should be used when different instances of an ***Entity*** can have different kinds of ***Properties***. In this scenario it is not viable to define attributes for all possibilities or subclasses for all possible variations. Implementing this pattern, ***Properties*** are represented as a list of named values in an instance. As a consequence, each instance can have only the necessary ***Properties*** needed, and new ones can be easily added at runtime to an ***Entity***.
- `Type Square` [1, 2]: This pattern join the implementation of the two previous patterns, implementing `Type Object` twice, for the ***Entity*** and for the ***Property***, which now has a ***Property Type***. This pattern should be used when each ***Entity*** can have different ***Properties***, but depending on its ***Entity Type*** there are certain ***Property Types*** allowed. As a consequence, new ***Entity Types*** can be added at runtime, as ***Property Types*** can be added and changed in existing ***Entity Types***. Figure 1 depicts the structure of the `Type Square` pattern.

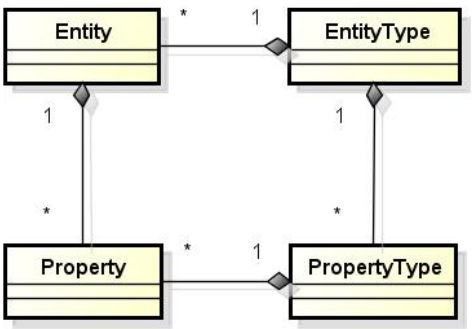

Fig. 1. The structure of the `Type Square` pattern

- `Accountability` [1, 8]: This pattern is used to represent a relationship or an association between Entity Types. Following this approach, an instance of a class is used to represent this ***Accountability***, and an ***Accountability Type*** represents the allowed relationships. Despite this approach is often applied in an AOM architectural style; there are other approaches to represent the relations between ***Entities***. Examples of these other approaches are: to create two subclasses for representing a ***Property***, in which one of them are to

represent associations; or to check the type of the value of a ***Property*** instance, to verify if it is an ***Entity*** or an instance of a language-native class.

The research work presented in this paper focus on these core structural patterns, however there are other patterns to represent the behavioral level of an AOM. Usually, `Strategy` [9] and `Rule Object` [10] are used to represent business rules. The core design of an AOM system is depicted in Fig. 2. The **Operational Level** is used to represent the application instances, which contain the information that is from the direct interest of the application, and the **Knowledge Level** represent the application metadata, which describe the application entities.

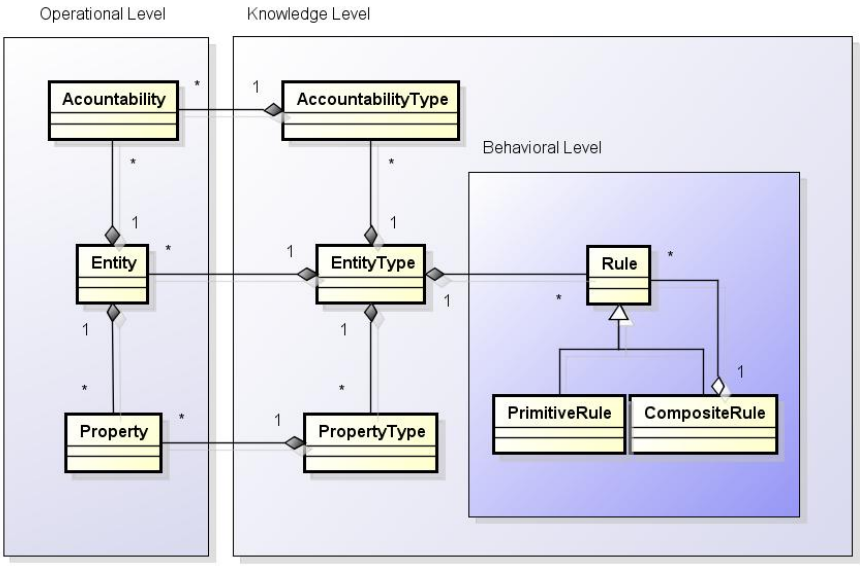

Fig. 2. AOM Core Design, adapted from [1]

The flexibility provided by AOMs comes with a cost of a higher complexity when developing the application. When AOM is used in an application, the other components should be compatible with the implementation of the domain model. For instance, instead of handling fixed properties, the application components need to be able to handle dynamic properties. The implementation of such components is usually coupled with the AOM implementation.

Two concerns that often should be implemented in an AOM application are persistence and graphical interfaces. The persistence should consider, not only the persistence of entities with a variable number of properties and relationships, but also the persistence of the model. The pattern AOM Builder [11] should be considered in this implementation. Graphical interfaces due to the dynamic nature of AOMs, should be able to adapt to changes in the model, and in order to implement it, rendering patterns for AOMs [12] should be implemented. Besides the issues presented, there are many other points to be considered, such as security and instance validation.

3 Metadata-Based Frameworks

A framework can be considered incomplete software with some points that can be specialized to add application-specific behavior, consisting in a set of classes that represents an abstract design for a family of related problems. It provides a set of abstract classes that must be extended and composed with others to create a concrete and executable application. The specialized classes can be application-specific or taken from a class library, usually provided along with the framework [13].

More recent frameworks make use of introspection [14, 15] to access application classes metadata at runtime, like their superclasses, methods and attributes. As a result, it eliminates the coupling between application classes and framework abstract classes and interfaces. For instance, following this approach the framework can search in the class structure for the right method to invoke. The use of this technique provides more flexibility to the application, since the framework reads dynamically the classes structure, allowing them to evolve more easily [16].

When a framework uses reflection [16, 17] to access and find the class elements, sometimes the class intrinsic information is not enough. If framework behavior should differ for different classes, methods or attributes, it is necessary to add a more specific meta-information to enable differentiation. For some domains, it is possible to use marking interfaces, like Serializable in Java Platform, or naming conventions [18], like in Ruby on Rails [19]. But those strategies can be used only for a limited information amount and are not suitable for scenarios that need more data.

Metadata-based frameworks can be defined as frameworks that process their logic based on instances and classes metadata [20]. In those, the developer must define additional domain-specific or application-specific metadata into application classes to be consumed and processed by the framework. The use of metadata changes the way frameworks are build and their usage by software developers [21].

From the developer's perspective in the use of those frameworks, there is a stronger interaction with metadata configuration, than with method invocation or class specialization. In traditional frameworks, the developer must extend its classes, implement its interfaces and create hook classes for behavior adaptation. He also has to create instances of those classes, setting information and hook class instances. Using metadata-based frameworks, programming activity is focused on declarative metadata configuration; and the method invocation in framework classes is smaller and localized.

Despite the use of code conventions, metadata can be defined on external sources like databases and XML files, but another alternative that is becoming popular in the software community is the use of code annotations. Using this technique the developer can add custom metadata elements directly into the class source code, keeping this definition less verbose and closer to the source code. The use of code annotations is called attribute-oriented programing [22]. Some programming languages, such as Java [23] and C# [24], have native support for annotations.

The pattern `Entity Mapping` [25] documented a common usage of metadata-based frameworks, which is to map between different representations of the same entity in a system. For instance, an application class can be mapped to a database to

allow the framework to handle its persistence. Similarly, entities can be mapped to another class schema or to an XML format, and the framework can translate between the two representations based on the class metadata. The idea of metadata mapping is used by the Esfinge AOM RoleMapper Framework, which is presented on the next section.

4 Esfinge AOM RoleMapper Framework

Esfinge AOM RoleMapper framework was created in the context of the Esfinge project (http://esfinge.sf.net), which is an open source project that comprises several metadata based frameworks for different domains. Examples of other frameworks developed in this project were Esfinge QueryBuilder [6] for query generation based on method metadata, Esfinge Guardian [26] for access control, and Esfinge SystemGlue [27] for application integration. Despite each framework provide innovations in its domain, they are also used in a broader research to identify models, patterns and best practices for metadata-based frameworks.

The main goal of this framework is to map domain-specific AOM models to a general AOM model. With this mapping it is possible to have software components developed for the general AOM model, and reused for several specific models. The framework uses adapters that have the general model interfaces, but invoke methods on the application-specific model. The map between the models, the developer should add annotations to the domain-specific AOM model, and use the factory method on the adapter class to encapsulate the original class.

The common AOM core structure provided by the framework consists of the following interfaces: *IEntityType*, *IEntity*, *IPropertyType* and *IProperty*, which represent an API for the `Type Square` pattern. One implementation of these interfaces has a general implementation of an AOM, and other one has adapters to encapsulate the access to a domain-specific AOM model. The interface *IProperty* has two adapter implementations, *AdapterFixedProperty* for fixed entity attributes and *AdapterProperty* for dynamic attributes.

The framework uses code annotations, such as *@Entity*, *@EntityType*, *@PropertyType* and *@EntityProperties* to identify the roles of the classes in the AOM model. These annotations can be added in the class definition and in the attribute the stores such information. For instance, the *@EntityType* annotation should be added in the class that represents it, and in the entity attribute that stores the entity type. There are also other annotations, such as *@Name* and *@PropertyValue* that identifies the attributes that store such properties of the AOM elements.

A class named *ModelManager* manages the AOM instances created by the framework. This class is responsible to all the operations involving the manipulation of the model, including model persistence, loading and querying. For accessing the database, the *ModelManager* makes use of the *IModelRetriever* interface, which should be implemented to allow the persistence of the model metadata. The Service Locator pattern is used to locate the persistence component.

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 that 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.

A problem of the previous implementation of Esfinge AOM RoleMapper is that the mapping is only possible if the domain-specific AOM model implements the Type Square pattern. That can prevent the use of this framework in applications that requires a lower level of flexibility, but still need to use some of the AOM patterns. This scenario usually happens in applications that are evolving from classes with fixed properties towards an AOM model. It would be important for the framework to give support to the AOM model evolution, because the flexibility requirements that drive the design in the direction of an AOM are often discovered in the middle of the project.

5 AOM Evolution Paths

In order to find the evolution paths that the implementation of an AOM can have from a set of fixed classes to the Type Object pattern implementation, an empirical study was performed with developers of AOM systems. In this study were included some papers [1, 3, 4] and case studies that narrates the implementation of an AOM step by step. The goal is to find the intermediate solutions that can be implemented. Figure 3 presents the evolution path that resulted from this study.

The initial point is presented as **Fixed Entity**. In this stage classes with fixed properties compose the application model. To change or add a new property, the class source code should be changed, and to add a new type a new class should be created. In Java language, this kind of entity is usually implemented as JavaBeans [], in which each property is implemented as a private attribute with get/set access methods.

A possible evolution to add flexibility in properties is the model presented as **Property Implementation**. This change is usually motivated by several changes in the entity properties, demanding changes in the source code and in other components that depends on it. Following this model, the entity can still have some fixed properties, but it also has a list where new properties can be included. The class that represents a property usually has a name, a type and a value. The entities with a property list can receive new values as properties with no restrictions.

Another path that can be followed from a class with fixed properties is the model named **Type Object Implementation**. This path is followed when there is an explosion of subclasses to characterize different types, and all existing types can not be predicted at compile time. In this model, the type is represented by an instance that composes the entity. As a consequence, simply creating new instances of the Entity

Type class it is possible to create new types at runtime. Despite flexibility was introduced regarding the type definition, the entity properties are still fixed.

The next step, which can be implemented after the both previous models, is the **Flexible Entity Without Property Enforcement**. This model joins the implementation of Type Object and Properties patterns. New properties can be added in the entity and entity types can also be determined and set at runtime. However, the entity type does not enforce the allowed properties in a given entity. Consequently, properties can be added independent of the entity type.

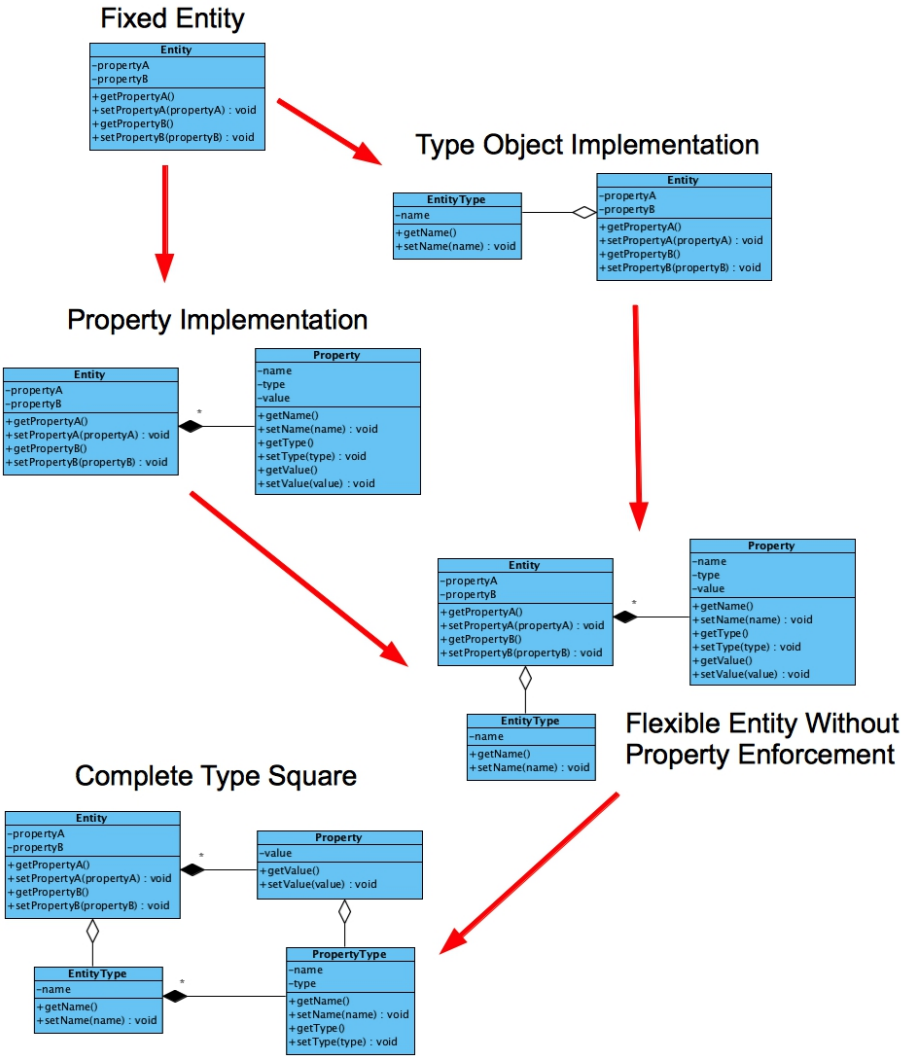

Fig. 3. AOMs evolution path

Finally, the last step covered by this evolution path is the **Complete Type Square**. Now the Type Object pattern is applied twice, one for the Entity, which has the Entity Type, and other for the Property, which have the Property Type. Now, the Entity Type defines the allowed Properties with the respective Property Types. Based on this information, the Entity only accepts Properties whose Property Type exists on its Entity Type. This last evolution does not add flexibility in comparison to the previous model, however it adds mode safety by allowing a more precise definition of the types and validating the Entity according to its Entity Type.

This evolution model considered the structural refactoring path to evolve from a model with fixed entities to a complete Type Square implementation. The other aspects of an AOM related to behavior were considered out of the scope of this research study. Other evolutions after the Type Square implementation are also possible, such as the introduction of inheritance and the introduction of custom metadata types, but they were also out of the scope of this work.

6 Implemented Support to AOM Evolution

After mapping the possible evolution paths towards an AOM model, the next step is to evolve the Esfinge AOM Role Mapper framework [6] to allow mapping for each possible implementation. No additional annotation was introduced to support the intermediary mappings. Most of the effort was employed to handle incomplete models, providing their mapping to a complete generic AOM used as API by the framework.

This section presents how each model stage can be mapped using the framework. Each subsection presents how the metadata should be configured and how the general AOM is expected to work when mapped to that model stage.

6.1 Fixed Entity

The first model to be supported is the fixed entity, which is a class whose properties are represented directly by attributes. Listing 1 presents how this class should be mapped using annotations. The class should receive the *@Entity* annotation and each property should receive the *@EntityProperty* annotation. Despite the annotations are added to the attributes, the access is performed using accessor methods following the JavaBeans standard. These methods can be used to add some additional logic, such as validations or transformation.

```
@Entity
public class Person {

        @EntityProperty private String name;
        @EntityProperty private String lastName;
        @EntityProperty private int age;
        @EntityProperty private Date bithday;
        //getters and setters omitted
}
```

Listing 1. Mapping a Fixed Model

The main restriction of this model is that new properties cannot be added to the entity, since it does not provide this kind of flexibility. When mapped to an AOM Entity by the framework, when the Entity Type is retrieved, it returns an instance of the class *GenericEntityType* created during mapping. It will have the same name as the mapped class and will have a Property Type list that represents the existing attributes in the Entity class.

6.2 Property Implementation

This model adds to the previous model the possibility to add dynamic properties. The entity needs to have a list of a class that represents a property. This list should be annotated with the *@EntityProperties* annotation. Additionally the entity can still have fixed properties, like in the previous model. An example of the resulting mapping can be found on Listing 2.

```
@Entity
public class Person {

        @EntityProperties
        private List<PersonInfo> infos = new ArrayList<>();
        @EntityProperty private String name;
        //getters and setters omitted
}
```

Listing 2. Mapping a model with Properties implementation

The class that represents a property should also be annotated with some metadata to indicate the meaning of the attributes in the AOM context. The class itself should have the *@EntityProperties* annotation, and should have, at least, one attribute with the annotation *@Name* and other with the annotation *@PropertyValue*. These attributes represent respectively the property name and value. A mapping example is presented on Listing 3.

```
@EntityProperties
public class PersonInfo {

        @Name private String name;
        @PropertyValue private Object info
        //getters and setters omitted
}
```

Listing 3. Property representation

Similarly to the previous model, the invocation of the method getEntityType() on a mapped instance of the class Entity, will return an instance of GenericEntityType. There is no restriction to each properties can be added in an entity. Because of that, if an invocation was performed in the entity type to retrieve the property types, an empty list will be returned, unless the entity class has also some fixed property, which should be included on the list.

6.3 Type Object Implementation

This model is an evolution from the model with fixed properties, presented in section 6.1, and it is a parallel evolution to the model presented on section 6.2. This approach introduces the Entity Type as an instance that composes the Entity. As presented on Listing 4, the attribute that represents the Entity Type should receive the annotation *@EntityType*. In this model, the properties are fixed in the Entity, so each one should be annotated with *@EntityProperty*.

Listing 5 presents the class that represents the Entity Type. The annotation *@EntityType* is also used in the class to configure the role it represents. The only required field annotation in this class is *@Name* that is used to differentiate between different types.

The greatest difference from the fixed domain model is that the retrieved Entity Type is an instance of *AdapterEntityType*, which encapsulates an instance of the mapped class. Consequently, while previously the name of the type was always the same, here it depends on the instance configured as the Entity Type.

```
@Entity
public class Person {

        @EntityType private PersonType type;
        @EntityProperty private String name;
        //getters and setters omitted
}
```

Listing 4. Entity type mapping

```
@EntityType
public class PersonType {

       @Name private String typeName;
       //getters and setters omitted
}
```

Listing 5. Class that represents the Entity Type

6.4 Flexible Entity without Property Enforcement

This model joins the independent evolutions of both previous models: the Entity has a list of Properties and a configured Entity Type as an attribute. Listing 6 presents an example of an entity implementation, that contains an attribute annotated with *@EntityType* and another one with *@EntityProperties*. Independent of that, it is still possible to have fixed properties that receive the *@EntityProperty* annotation.

```
@Entity
public class Person {

        @EntityType private PersonType type;
        @EntityProperties
        private List<PersonInfo> infos = new ArrayList<>();
        @EntityProperty private String name;
        //getters and setters omitted
}
```

Listing 6. Entity type and properties mapping

In this implementation, the Property class is similar to the one is Listing 3 and the Entity Type is similar to the one in Listing 5. An important characteristic in this model is that there is no restriction on the properties inserted in the entity, despite is has a defined Entity Type.

6.5 Complete Type Square

In this last model of the studied path there is a complete implementation of the Type Square pattern. The main difference is that the Property also has a defined type, implementing again the Type Object pattern. Listing 7 presents an example of how a class that represents the Property Type should be mapped. The class itself should receive a *@PropertyType* annotation, and additionally it should have attributes to define the property name and the property type respectively annotated with *@Name* and *@PropertyTypeType*. It is important to notice that the type is represented by an *Object*, because it can be an instance of *Class* in case of a simple property or it can be an *EntityType* in case of a relationship.

```
@PropertyType
public class InfoType {

        @Name private String name
        @PropertyTypeType private Object type;
        //getters and setters omitted
}
```

Listing 7. The Property Type mapping

To enable the mapping of this kind of model other classes should also be changed. As presented in Listing 8, the Entity Type should define a list of Property Types that are allowed for its respective Entities. This list should be mapped using the *@PropertyType* annotation. Additionally, as presented in Listing 9, the Property representation instead of being identified by a String with the *@Name* annotation, it is identified by a reference to its property type instance, which should also receive the *@PropertyType* annotation.

```
@EntityType
public class PersonType {

        @Name private String typeName;
        @PropertyType private List<InfoType> list = new ArrayList<>();
        //getters and setters omitted
}
```

Listing 8. Class that represents the Entity Type

```
@EntityProperties
public class PersonInfo {

        @PropertyType private InformationType type;
        @PropertyValue private Object info
        //getters and setters omitted
}
```

Listing 9. Property representation

An important characteristic of this final model is that independent of the mapped implementation, it will validate if the Properties inserted in an Entity actually exist in its Property Type. Consequently, by configuring an Entity Type it is defined which types of properties an Entity can have, in addition to its fixed properties.

7 Evaluation

In order to evaluate if the proposed model fulfill its goals, a small experiment was performed. The aim of this section is to describe this experiment and present the obtained results.

7.1 Experiment Goal

The goal of this experiment is to verify if by using the proposed mapping model it is possible to evolve an AOM model increasing its flexibility without changing the code that handles it. In this context, two hypotheses were formulated:

H1 – The code that handles an AOM model can be decoupled from its domain-specific implementation.
H2 – An AOM model can be evolved without changing the code that handles it.

7.2 Experiment Description

The experiment consists in a set of tests that manipulate the model through the Esfinge AOM Role Mapper API. These tests consist in the execution of operations that access the property list, insert a new property value, access a property and change a property value. The same tests are executed in all the model stages proposed in evolution model.

Figure 4 presents a graphic representation of how the software components were organized to implement the experiment. The tests use an entity factory that return an instance of the interface *IEntity* that represents the entity that should be used for the test. The factory creates the domain specific entities and uses the Esfinge Role Mapper framework to adapt it to the general AOM API, returning it. For each different model, only the Entity Factory and the Domain-specific AOM were changed.

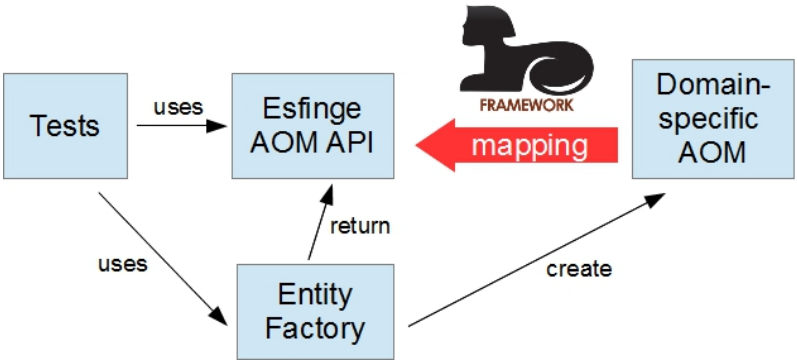

Fig. 4. Experiment representation

7.3 Results and Analysis

The elaborated tests executed successfully in all different model implementations. Based on that it is possible to confirm the first hypothesis H1, because the test code does not depend directly on the domain-specific model and could be reused and executed for different implementations of the same model.

The second hypothesis H2 can also be confirmed to be true, because the test could be executed successfully by evolving the domain-specific model, which vary from the less flexible format with fixed properties to the more flexible format with a complete type square implementation.

8 Related Works

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 [4], ModelTalk [3] and its descendant, Ink [28] 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, persistence, user-interface generation, communication and concurrency [Ferreira et al. 2009]. 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 [29].

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 [Hen-Tov et al. 2009]. 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.

Since these frameworks do not handle domain specific AOM models, it does not make sense to support the evolution of AOMs in different stages. To refactor a system towards the AOM architectural style by using these alternatives, the general AOM model provided by the framework should replace the original application model. Considering this fact, the solution proposed by this work provides a more gentle curve for application refactoring.

9 Conclusions

This work proposes the usage of metadata mapping to support the evolution of an AOM model without changing the code that handles it. By studding different cases in AOM implementation, it was identified the possible evolution paths that the AOM flexibility can be introduced. An implementation using the Esfinge AOM Role Mapper was incremented to enable the mapping between a general AOM Model and each one of the evolution stages. An evaluation was performed in order to verify if the goal to map different stages of the model was achieved.

This work can be considered the first step to achieve the support for mapping hybrid models, which can have different implementations in different stages co-existing on the same model and mapped to the same structure. To achieve this final stage, a further study should focus on the relationship between different entities in different evolution stages. Additionally, another future work could provide the inverse mapping, allowing an AOM model to be mapped to fixed classes with properties in Java Beans style.

We thank for the essential support of Institutional Capacitation Program (PCI-MCTI – modality BSP), which enabled a technical visit when this work was developed.

References

1. Yoder, J.W., Balaguer, F., Johnson, R.: Architecture and design of Adaptive Object-Models. In: Proceedings of the 16th Object-Oriented Programming, Systems, Languages & Applications (2001)
2. Yoder, J.W., Johnson, R.: The Adaptive Object-Model architectural style. In: Proc. of 3rd IEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (2002)
3. Hen-Tov, A., Lorenz, D.H., Pinhasi, A., Schachter, L.: ModelTalk: when everything is a domain-specific language. IEEE Software 26(4), 39–46 (2009)
4. Ferreira, H.S.: Adaptive-Object Modeling: Patterns, Tools and Applications. PhD Thesis, Faculdade de Engenharia da Universidade do Porto (2010)
5. Welicki, L., Yoder, J.W., Wirfs-Brock, R., Johnson, R.E.: Towards a pattern language for Adaptive Object-Models. In: Proc. of 22th Object-Oriented Programming, Systems, Languages & Applications (2007)
6. Matsumoto, P., Guerra, E.M.: An Architectural Model for Adapting Domain-Specific AOM Applications. In: SBCARS- Simpósio Brasileiro de Componentes, Arquitetura e Reutilizacação de Software, Natal (2012)
7. Johnson, R., Wolf, B.: Type Object. In: Pattern Languages of Program Design, vol. 3, pp. 47–65. Addison-Wesley (1997)
8. Fowler, M.: Analysis patterns: reusable object models. Addison-Wesley Professional (1996)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: elements of reusable object oriented software. Addison-Wesley (1994)
10. Arsanjani, A.: Rule Object: a pattern language for adaptive and scalable business rule construction. In: Proc. of 8th Conference on Pattern Languages of Programs (PLoP) (2001)
11. Welicki, L., Yoder, J.W., Wirfs-Brock, R.: Adaptive Object-Model Builder. In: Proceedings of the 16th PLoP (2009)
12. Welicki, L., Yoder, J.W., Wirfs-Brock, R.: A pattern language for Adaptive Object Models - rendering patterns. In: Proc. of 14th Conference on Pattern Languages of Programs, PLoP (2007)
13. Johnson, R., Foote, B.: Designing reusable classes. Journal Of Object-Oriented Programming 1(2), 22–35 (1988)
14. Doucet, F., Shukla, S., Gupta, R.: Introspection in system-level language frameworks: meta-level vs. Integrated. In: Source Design, Automation, and Test in Europe. Proceedings... [S.l.: s.n], pp. 382–387 (2003)
15. Forman, I., Forman, N.: Java reflection in action. Manning Publ., Greenwich (2005)
16. Foote, B., Yoder, J.: Evolution, architecture, and metamorphosis. In: Pattern Languages of Program Design 2, ch. 13, pp. 295–314. Addison-Wesley Longman, Boston (1996)
17. Maes, P.: Concepts and Experiments in Computational Reflection. In: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA 1987, pp. 147–169 (1987)

18. Chen, N.: Convention over configuration (2006), http://softwareengineering.vazexqi.com/files/pattern.htm (accessed on December 17, 2009)
19. Ruby, S., Thomas, D., Hansson, D.: Agile Web Development with Rails, 3rd edn. Pragmatic Bookshelf (2009)
20. Guerra, E., Souza, J., Fernandes, C.: A pattern language for metadata-based frameworks. In: Proceedings of the Conference on Pattern Languages of Programs, Chicago, vol. 16 (2009)
21. O'Brien, L.: Design patterns 15 years later: an interview with Erich Gamma, Richard Helm and Ralph Johnson. InformIT (October 22, 2009), http://www.informit.com/articles/article.aspx?p=1404056 (accessed on December 26, 2009)
22. Miller, J., Ragsdale, S.: Common language infrastructure annotated standard. Addison-Wesley, Boston (2003)
23. Schwarz, D.: Peeking inside the box: attribute-oriented programming with Java 1.5 [S.n.t.] (2004), http://missingmanuals.com/pub/a/onjava/2004/06/30/insidebox1.html (accessed on December 17, 2009)
24. JSR 175: a metadata facility for the java programming language (2003), http://www.jcp.org/en/jsr/detail?id=175 (accessed on December 17, 2009)
25. Guerra, E., Fernandes, C., Silveira, F.: Architectural Patterns for Metadata-based Frameworks Usage. In: Proceedings of Conference on Pattern Languages of Programs, Reno, vol. 17 (2010)
26. Silva, J., Guerra, E., Fernandes, C.: An Extensible and Decoupled Architectural Model for Authorization Frameworks. In: Murgante, B., Misra, S., Carlini, M., Torre, C.M., Nguyen, H.-Q., Taniar, D., Apduhan, B.O., Gervasi, O. (eds.) ICCSA 2013, Part IV. LNCS, vol. 7974, pp. 614–628. Springer, Heidelberg (2013)
27. Guerra, E., Buarque, E., Fernandes, C., Silveira, F.: A Flexible Model for Crosscutting Metadata-Based Frameworks. In: Murgante, B., Misra, S., Carlini, M., Torre, C.M., Nguyen, H.-Q., Taniar, D., Apduhan, B.O., Gervasi, O. (eds.) ICCSA 2013, Part II. LNCS, vol. 7972, pp. 391–407. Springer, Heidelberg (2013)
28. Acherkan, E., Hen-Tov, A., Lorenz, D.H., Schachter, L.: The ink language meta-metamodel for adaptive object-model frameworks. In: Proc. of 26th ACM International Conference Companion on OOPSLA Companion (2011)
29. Ferreira, H.S., Correia, F.F., Yoder, J., Aguiar, A.: Core patterns of object-oriented meta-architectures. In: Proc. of 17th Conference on Pattern Languages of Programs (PLoP) (2010)