Dynamic Generated Adapters from Adaptive Object Models to Static APIs

Eduardo Martins Guerra, INPE, Brazil
Jean Novaes Santos, INPE , Brazil
Ademar Aguiar, FEUP, Portugal
Luiz Gustavo Veras, INPE , Brazil

Abstract: *By using Adaptive Object Models (AOM) it is possible to have a more flexible domain structure in an application, allowing its adaptation at runtime. Patterns for AOMs create a class structure completely different from the static structure that applications and frameworks are used to handling. As a result, AOM applications cannot be integrated with existing frameworks, even the industry standards, which are meant for a static class-based domain model. The Adapter pattern could be used to adapt the AOM structure for the one expected by the framework, however this adapter would need to be manually modified for every change in the AOM structure. This paper proposes a solution to this problem by creating a dynamically generated adapter for the current AOM structure. To exemplify the use of this approach, an implementation was created to adapt the AOM structure from the framework Esfinge AOM to the Java Beans API. Additionally, a framework for instance comparison based on the Java Beans specification was used to compare the adapted AOM entities.*

1. Introduction

The Adaptive Object Model (AOM) [Yoder at. al, 2001; Yoder and Johnson, 2002] is an architectural style, where types are represented by instances and described by metadata. Based on this model, the types can be changed at runtime, allowing the application to be adapted quickly to user needs. This kind of architecture is recommended when the application domain is naturally dynamic, and changing it is part of routine business.

A problem faced in the implementation of AOM applications is that since domain objects are implemented using an AOM-specific structure, it is not possible to easily integrate these objects with existing components or frameworks that rely on implementation conventions. For instance, in the Java language, a static domain model usually follows the Java Bean standard [JavaBeans 2010], that defines a standard way for creating such classes, such as the use of getter and setter methods for accessing properties. A framework that is created to dynamically find the properties of a bean class based on this standard will not be able to process an entity for an AOM because the AOM object has a different implementation structure. Because of that, most of the components for handling an AOM structure need to be created from scratch, increasing the time needed to implement the application, and consequently its cost.

The Adapter pattern [Gamma et al. 1994] could be applied in this context to map calls to a class that fallows the JavaBean standard to the AOM entity, allowing it to be handled by a framework. In this case, for each entity property on the AOM, there would be a respective getter and setter

method that would encapsulate the access to it on the AOM entity, like presented in Fig. 1. However, the code of this adapter would need to change every time that the entity type changes, which would take away the flexibility that motivates the use of an AOM. This scenario is illustrated in Fig. 2.

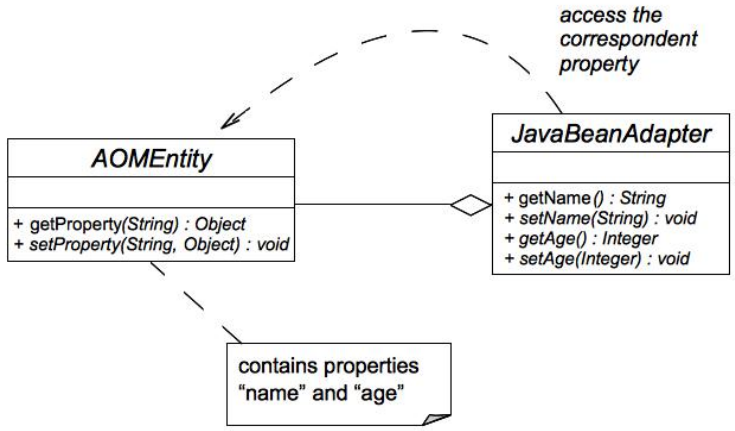

Fig 1 – Example of a Static Java Bean Adapter for an AOM Entity

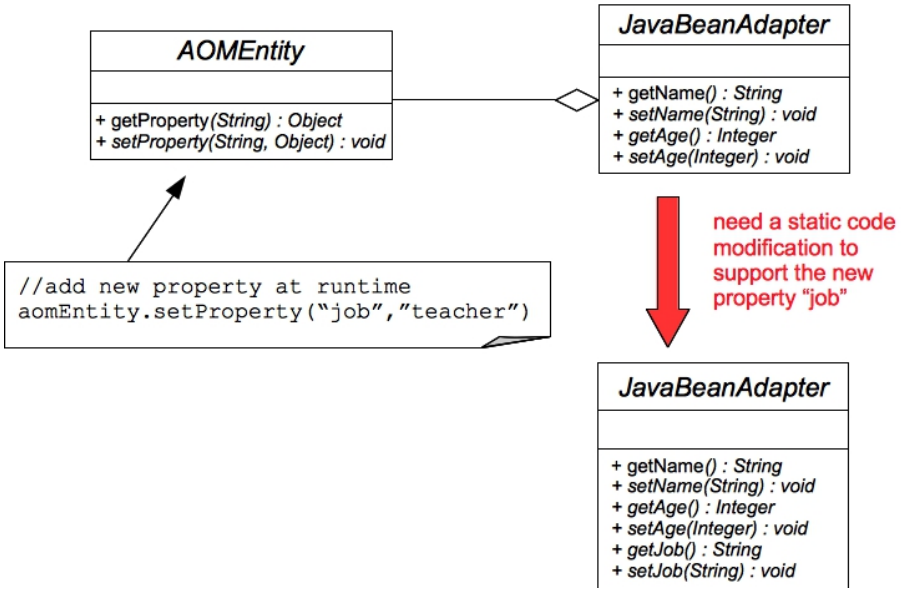

Fig 2 – New properties needs static addition on the static adapter

This paper proposes a solution to this problem, which generates dynamically an adapter class based on the entity type current structure. Since this adapter is generated at runtime, a new one can be easily created based on changes to the AOM entity type. An implementation was created to adapt the AOM structure from the framework Esfinge AOM Role Mapper to classes with getters and setters following the Java Beans standard. This implementation was validated by using these adapters in a framework based on Java beans that finds differences between instances from the same class.

2. Adaptive Object Model Patterns

The AOM architecture style is composed of several patterns, but three patterns can be considered the main ones: Type Object, Property and Type Square. The next subsections describe each one individually.

2.1. Type Object

There are situations in software development cycle where the number of classes may grow too large. This scenario typically happens when there's a necessity to represent many classes that only have minor differences between then. Consider an example of a system where different products need to be registered. One approach for implementing this system would be as shown in Fig. 3. In this implementation several subclasses would inherit a main class but with just a few differences between them.

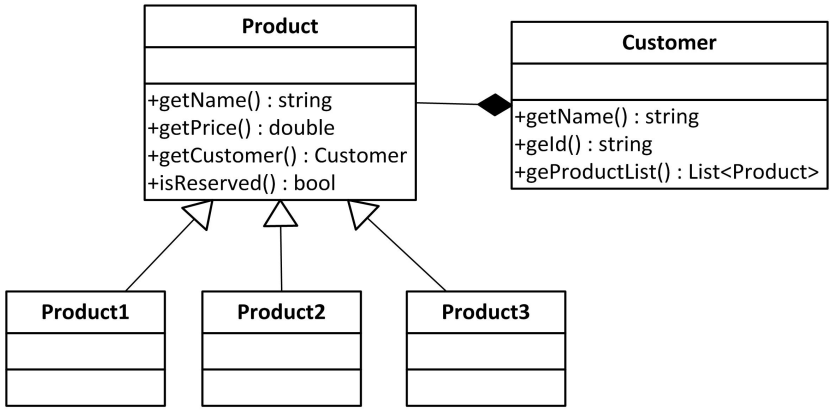

Fig 3 - First Approach to implement a system's products

Introducing a new product using this approach would require creating a new subclass, recompiling the code and updating the system to reflect the addition. The Type Object pattern proposes to represent types as instances that compose the main class, as shown in Fig 4. This solution allows the addition of new types of products dynamically, because the representation is made at the instance level.

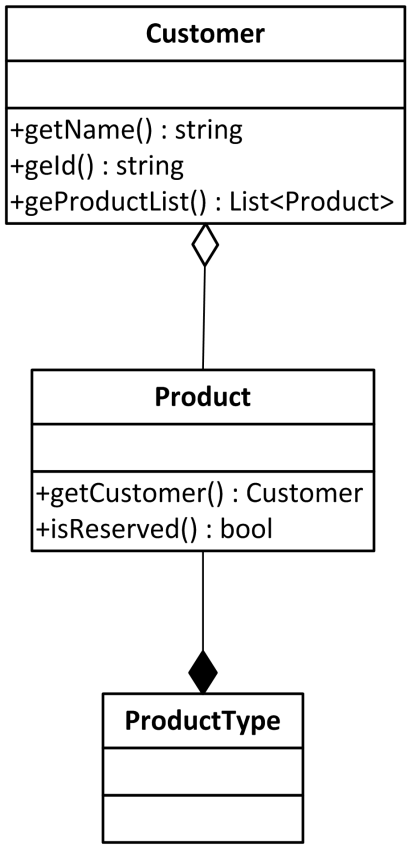

Fig 4 – Product's System using the Type Object design pattern

2.2. Property Pattern

The Property Pattern is suitable for scenarios where objects need to have different properties that can be dynamically added. Considering a system where there is a need to register products with different types of properties. A large number of properties could be created to describe any type of product, but the problem is that some properties that are specific to some products. For instance, a book would have different properties than an electronic device. The property pattern uses a set of objects to represent the instance properties, as shown in Fig 5.

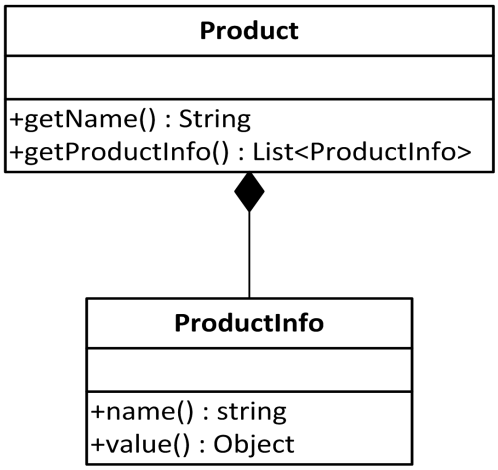

Fig 5 - Representation of a Product class using the Property Pattern.

2.3. Type Square Pattern

The core of an AOM application is usually based on the Type Square pattern, which is a particular way of combining the patterns Type Object and Property. Figure 6 has a representation of this pattern structure. In this model the Type Object pattern is used twice, one time for representing property types of properties, and one more time for representing entities and types of entities.

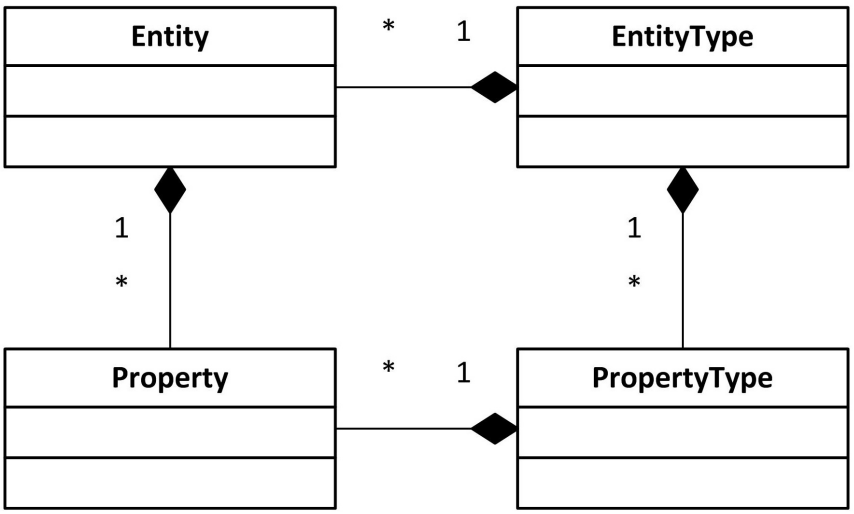

Fig 6 - Type Square Pattern.

3. Esfinge AOM Role Mapper

The framework AOM Role Mapper, one of the components of the Esfinge Project (http://esfinge.sf.net), was developed with the objective to increase the reuse of AOM systems [Matsumoto and Guerra, 2014]. The main role of the framework is to map domain-specific AOM structures to a general AOM structure, creating adapters that can be used by generic frameworks. With this solution, common third party frameworks can add functionality to AOM applications, what would be extremely difficult to achieve with components created for domain-specific AOMs, due to the coupling that the AOM model usually have with application domain. The mapping can be applied to systems with AOMs that are only partially implemented [Guerra and Aguiar, 2014].

To integrate AOM domain-specific applications with generic frameworks the Esfinge AOM Role Mapper framework uses code annotations on the application classes. Listing 1 presents a simple example of using Esfinge AOM Role Mapper annotations. The following describes some annotations with their specific meaning:

- @EntityType: (Class) classes that play the Entity Type role in the AOM architecture ; ((Attribute in classes of Entity Type) Identifies an attribute that references the Entity Type corresponding to an Entity;
- @Entity: (Class) Identifies classes that play the Entity role in the AOM architecture;
- @PropertyType: (Class) Identifies classes that play the Property Type role in the AOM architecture ; (Attribute in classes of Property Type) Identifies the attribute that refers to a Property Type corresponding to a Property;
- @EntityProperties: (Class) Identifies classes that play the Property role in the AOM architecture; (Attribute in classes of Entity Type) Identifies the attribute that refers Properties of an Entity;
- @FixedEntityProperty: (Attribute in classes of Entity Type) (Optional) Identifies attributes corresponding to fixed properties in an Entity class;
- @Name: (Attribute in classes of Entity Type or Property Type) Identifies the attribute containing the name of an Entity Type or a Property Type;
- @PropertyTypeType: (Attribute in classes of Property Type) Identifies the attribute containing the type of a Property Type;
- @PropertyValue: (Attribute in classes of Property Type) Identifies the attribute containing the value of a Property;
- @CreateEntityMethod: (Method in classes of Entity Type) Identifies the method of an Entity Type class that handles with the creation of an Entity with this type. If no method is annotated, the method createNewEntity from interface IEntityType will throw an exception when invoked from the object.

```
@Entity
public class Product {

	@EntityType
	private ProductType type;

	@FixedEntityProperty
	private String productName;

	@EntityProperty
	private List<Information> informations = new ArrayList<>();

	 //accessor methods omitted

}
```

Listing 1 - Example of AOM Role Mapper annotations in an Entity class.

4. Proposed Solution

The solution we propose for adapting an AOM to a static API is to generate dynamically an adapter with a static structure, which access from its methods the properties of the AOM entity.

4.1. Rationale

An adapter needs to support defined interfaces, however in our case the interface is not an abstraction but based on a set of code conventions defined in the Java Bean standard [JavaBeans 2010]. These standards allow the access of a class' properties by using reflection. The existence of many frameworks that rely upon the Java Bean standard makes it valuable to create such an adapter.

Other behavioral AOM patterns, such as Strategy and Rule Object, could be adapted as well, however that would not be useful if they do not conform to a standard that would allow their dynamic identification and invocation. That's why this work chose to focus on the accessor methods based on the Java Bean standard. Section 8 presents how the behavioral part of the AOM structure could be adapted to an interface based on code annotations, however this is not part of the present work.

4.2. Solution Description

To adapt an entity type that implements the Type Square pattern to the Java Bean standard, the adapter needs to have the following methods:

- A constructor that receives the AOM entity to be adapted. This entity should be internally stored to be accessed by the other methods.

- A getter access method for each entity property that the entity type has. This method should receive no parameters and return the target property. Its implementation should retrieve the value from the entity property and return. The return type is the type of the AOM property, unless it is another AOM entity. In this case, the return type is Object, because the entity is also adapted before it is returned.

- A setter access method for each entity property that the entity type has. This method should have no return (void) and receive a parameter with the type of the target property. Its implementation should set the parameter on the entity property. A setter is not generated when the property is another AOM entity.

To create the adapter, our solution proposes the use of an adapter factory. This factory receives the AOM entity that should be adapted and should return the respective adapter encapsulating the entity received. It reads the entity metadata present on the entity type and creates at runtime a new class with the desired API. In Java, for instance, runtime class generation can be done through bytecode manipulation.

Fig. 7 presents a class diagram with the main participants in our solution. The class named Application represents an application that has an instance of an AOM entity and needs to have represented on an API that follows the JavaBean standard. It invokes the AdapterFactory passing the Entity as a parameter. The AdapterFactory accesses the Entity metadata by accessing its EntityType and its respective PropertyTypes. Based on that, it dynamically generates an adapter class and instantiates that class by passing the Entity into the class initialization. The adapter is returned to the application, which can use it with frameworks based on a static API.

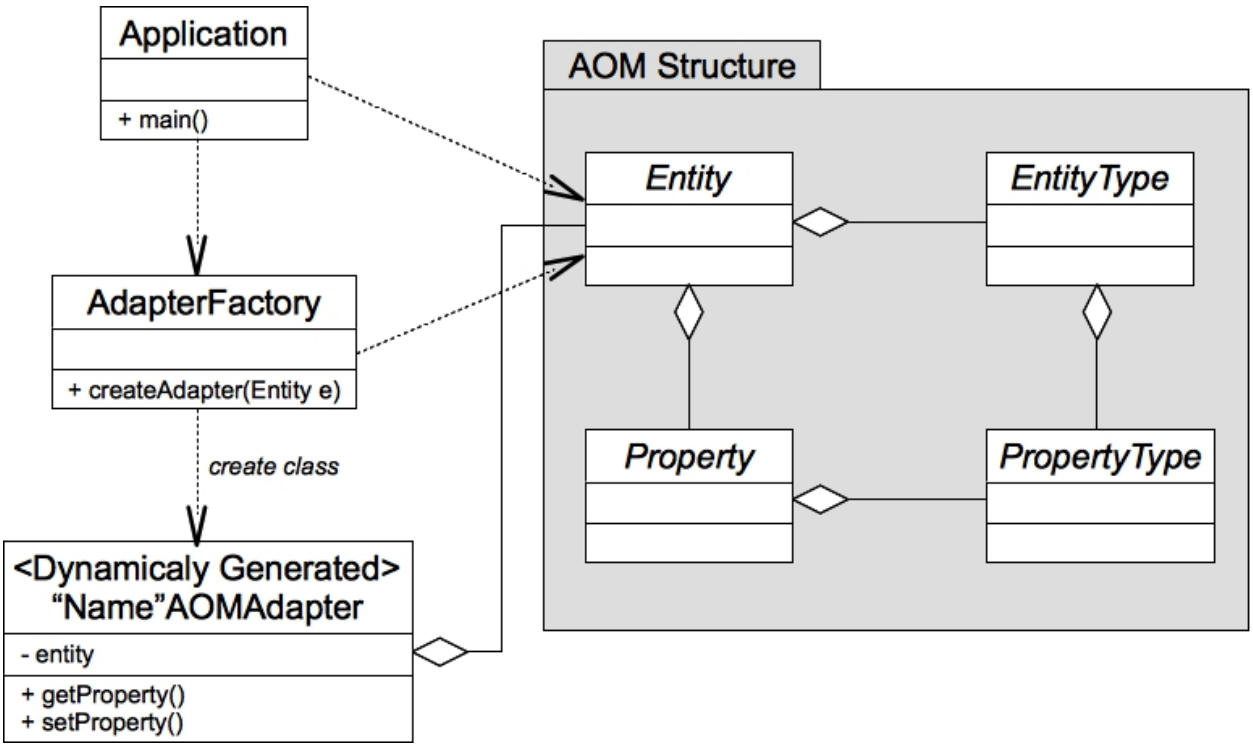

Figure 7 - Representation of the solution with the dynamic adapter

Figure 8 presents a sequence diagram that illustrates the proposed process for adapter creation. The application invokes a method on the Adapter Factory to create an adapter for a given AOM entity. This factory retrieves the entity information, such as its type name and its properties, and uses them as parameters to create a class on a bytecode generation library, represented in the diagram by the class ClassVisitor. After that, the adapter retrieves the generated bytecode and uses a custom class loader to load into the virtual machine the newly generated class. This class is used to instantiate the adapter using reflection, which is populated with the AOM entity that it should wrap and return. The generated classes can be cached and reused for invocations with the same Entity Type.

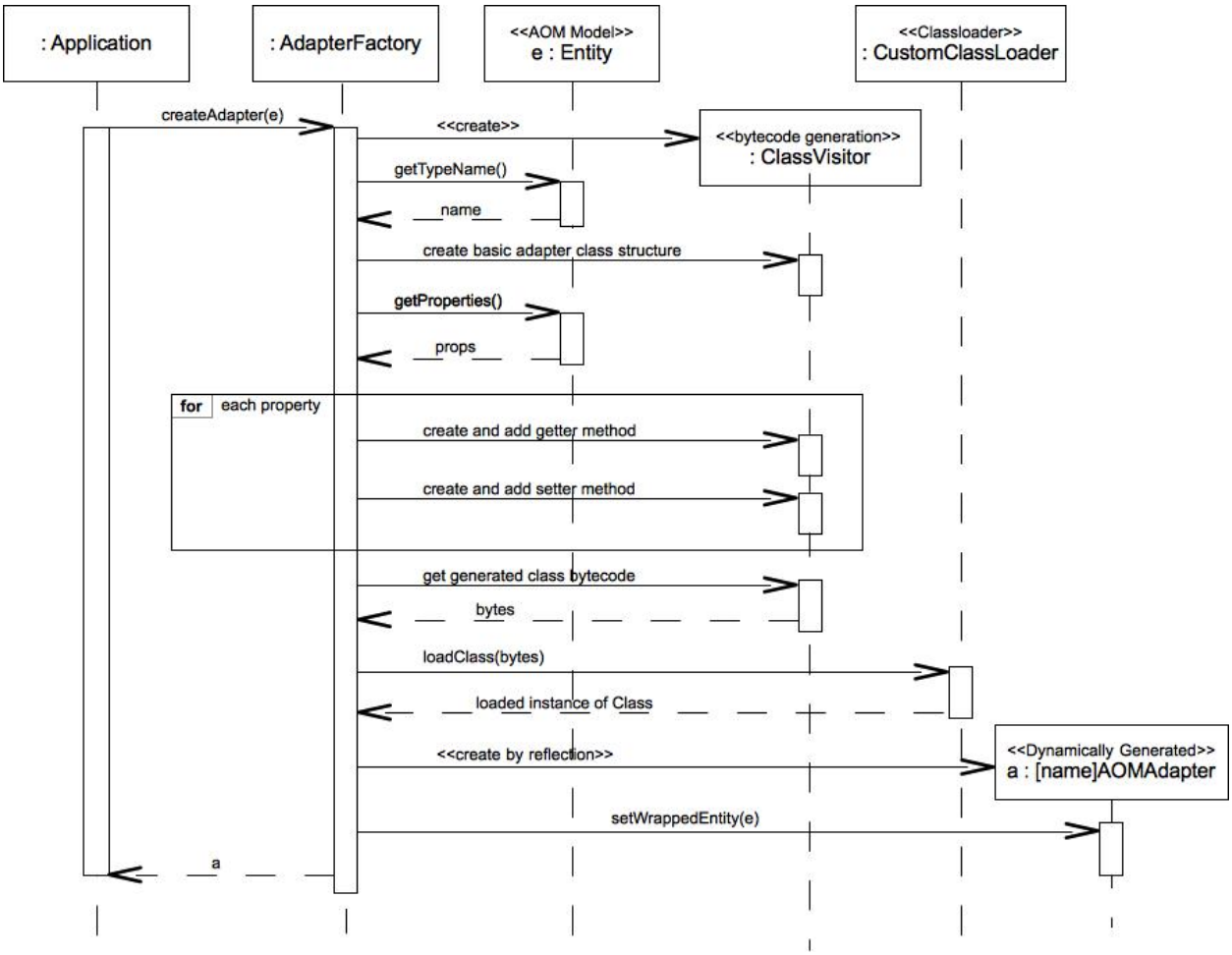

Figure 8 – Sequence diagram for adapter creation

The solution proposed in our work cannot be considered a pattern because it was used only once on Esfinge AOM Role Mapper framework. However, it can be considered a more specific implementation of the Adapter pattern. If this approach is used for other AOM systems in the future, a pattern for it can be written and integrated in the AOM pattern language.

5. Adapter Factory Implementation

This section presents the implementation of the dynamic factory created as an example of our proposed solution. It used the AOM structure from the Esfinge AOM Role Mapper framework to adapt and the ASM framework for bytecode generation. In this initial version, the idea is to provide a minimum proof of concept of how this adapter could be implemented.

The first step was the creation of a static adapter to be used as a model for the bytecode generation process. Listing 2 presents the class that was used as model for the bytecode generation. As presented in the previous section, it has a constructor that receives the entity instance and examples for the creation of the getter and setter methods, which directly access the IEntity instance.

```
public class ExampleAOMToBeanAdapter {

    private IEntity entity;

    public TestClassBeanAdapterV2(IEntity entity) {
        this.entity = entity;
    }
    public Integer getProperty() {
        try {
               return (Integer)entity.getProperty("property").getValue();
        } catch (EsfingeAOMException e) {
               throw new RuntimeException(e);
        }
    }
    public void setProperty(Integer a) {
        try {
               entity.setProperty("property", a);
        } catch (EsfingeAOMException e) {
               throw new RuntimeException(e);
        }
    }
}
```

Listing 2 - Adapter used as example for bytecode generation

The code of the class in Listing 1 was submitted to the ASMifier tool that generates the source code using ASM to generate the target class. This code was parameterized to perform the bytecode generation based on the information of the entity type. The information needed is the class name, the property name and the property types.

Listing 3 presents the base code used on the class AdapterFactory to create the adapter. This class uses a Map called storedClasses to store the class generated based on the entity type name. This class is reused if another adapter for the same entity type is requested from the factory. There is a method, not shown on the listing that removes the existing adapter from the cache and forces the creation of another one. This method is important to force a change to the existing adapter if the entity type changes.

```
public Object generate(IEntity entity) throws Exception {

    Class clazz = null;

    if (storedClasses.containsKey(entity.getEntityType().getName())) {
        clazz = storedClasses.get(entity.getEntityType().getName());
    } else {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        String name = entity.getEntityType().getName() + "AOMBeanAdapter";
        createPrivateAttribute(null, cw);
        createClassAndConstructor(name, cw);
```

```
        for (IProperty p : entity.getProperties()) {
            Class<?> type = Object.class;
            if (p.getPropertyType() != null) {
                type = (Class<?>) p.getPropertyType().getType();
            }
            createGetter(name, cw, p.getName(), type);
            createSetter(name, cw, p.getName(), type);
        }

        DynamicClassLoader cl = new DynamicClassLoader();
        clazz = cl.defineClass(name, cw.toByteArray());
        storedClasses.put(entity.getEntityType().getName(), clazz);
    }

    Object obj = clazz.getConstructor(IEntity.class).newInstance(entity);
    return obj;
}
```

Listing 3 - Base method for generating the adapter

If the respective adapter class is not found in the cache, the method creates it. This method delegates to other helper methods the creation of other parts of the class, such as the attribute and the constructor. It iterates through the properties of the entity, creating respective getters and setters for their access.

After the adapter class is created by using bytecode manipulation, it is loaded by using a custom classloader. After that, the adapter is instantiated using reflection, passing the entity as a parameter to the constructor. The resulting instance is returned by the method generate().

To exemplify the creation by using the adapter factory and the use of the adapter, List. 4 presents a method that creates the adapter for an entity and prints the methods, attributes and the respective return values for getter methods. The code creates an entity type with three properties and an entity of the defined type, defining values for the three properties. After that, an adapter of this entity is created, and a method to print the class metadata in the console is invoked.

```
public static void main(String[] args) throws Exception {
    IEntityType entityType = new GenericEntityType("Person");
    entityType.addPropertyType(new GenericPropertyType("number", Integer.class));
    entityType.addPropertyType(new GenericPropertyType("height",Double.class));
    entityType.addPropertyType(new GenericPropertyType("name", String.class));

    IEntity entity = entityType.createNewEntity();
    entity.setProperty("number", 27);
    entity.setProperty("height", 1.8);
    entity.setProperty("name", "John");

    AOMAdapterFactory adapterFactory = new AOMAdapterFactory();
    Object obj = gc.generate(entity);
```

```
    printObjectAndClass(obj);
}

private static void printObjectAndClass(Object obj) {
    for (Method m : obj.getClass().getMethods()) {
        if (m.getName().startsWith("get")) {
            System.out.print(m + " => ");
            try {
                System.out.println(m.invoke(obj));
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else {
            System.out.println(m);
        }
    }
    for (Field field : obj.getClass().getDeclaredFields()) {
        System.out.println(field);
    }
}
```

Listing 4 - Example of adapter factory usage

The method printObjectAndClass() in List. 4 uses reflection to access object metadata and print it on the console. It prints all the public methods, accessed by getMethods(), and all the attributes declared in the current class independent of its visibility, accessed by getDeclaredFields(). For methods starting with “get”, it also invokes the method on the adapter and prints the returned value.

List. 5 presents what was printed on the console by the execution of the method presented on List. 4. It is possible to see that the class has all the getter and setter methods respective to the entity properties. These methods also use the type of the entity property type as the parameter and the return of such methods. The values of the entity encapsulated on the adapter were also printed correctly on the console. In the end, it is possible to verify that the internal attribute that stores the encapsulated entity was also printed.

```
public java.lang.String PersonAOMBeanAdapter.getName() => John
public void PersonAOMBeanAdapter.setName(java.lang.String)
public java.lang.Integer PersonAOMBeanAdapter.getNumber() => 27
public void PersonAOMBeanAdapter.setNumber(java.lang.Integer)
public java.lang.Double PersonAOMBeanAdapter.getHeight() => 1.8
public void PersonAOMBeanAdapter.setHeight(java.lang.Double)
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws
java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public java.lang.String java.lang.Object.toString()
public native int java.lang.Object.hashCode()
```

```
public final native java.lang.Class java.lang.Object.getClass() => class
PersonAOMBeanAdapter
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
private org.esfinge.aom.api.model.IEntity PersonAOMBeanAdapter.entity
```

Listing 5 - Output generated by the example

The implementation supports complex AOM properties, where the property value is another AOM entity. In this scenario, the entity is also adapted to a Java Bean when it is returned as a property value. For this kind of property, a setter method is not provided, since to change its properties it is expected that the application retrieves its adapter and changes the values directly.

6. Using Java Bean Framework

Since the goal to create an adapter to a static API is to enable the use of frameworks based on it, this section presents the results of an experiment that used the adapters with a framework. The framework used in our experiment is Esfinge Comparison [Guerra et al. 2013], which compares each properties of two Java beans of the same class and return the differences in their properties.

Listing 6 presents the source code that used the adapter to perform the comparison of two entities from the same entity type. An entity type with three property types was created, followed by the creation of two entities with values for all the properties. In order to use the comparison framework, the adapter factory was used to create adapters for the two entities.

```
public static void main(String[] args) throws Exception {
    IEntityType entityType = new GenericEntityType("Car");
    entityType.addPropertyType(new GenericPropertyType("plateNumber",
String.class));
    entityType.addPropertyType(
        new GenericPropertyType("yearOfManufacturing", Integer.class));
    entityType.addPropertyType(new GenericPropertyType("color", String.class));

    IEntity entityA = entityType.createNewEntity();
    entityA.setProperty("plateNumber", "AKZ-3421");
    entityA.setProperty("yearOfManufacturing", 1980);
    entityA.setProperty("color", "yellow");

    IEntity entityB = entityTypeB.createNewEntity();
    entityB.setProperty("plateNumber", "DZZ-3421");
    entityB.setProperty("yearOfManufacturing", 1982);
    entityB.setProperty("color", "yellow");

    AOMAdapterFactory adapterFactory = new AOMAdapterFactory();
    Object objA = adapterFactory.generate(entityA);
    Object objB = adapterFactory.generate(entityB);
```

```
    ComparisonComponent c = new ComparisonComponent();
    List<Difference> difs = c.compare(objA, objB);
    System.out.println("*** Differences: ***");
    for(Difference d : difs) {
        System.out.println(d.toString());
    }
}
```

Listing 6 - Example of code that uses the framework for comparing instances

After the creation of the adapters, the main class of the framework named ComparisonComponent was invoked passing both adapters as parameters. A list with differences was received and printed in the console. Based on the result of the execution, it was possible to verify that the adapters were working as expected.

7. Limitations

The proposed solution can be considered a first step in the direction of defining an AOM adapter that allows the usage of AOM entities by frameworks that need to access objects based on the reflective access to a static API based on the JavaBeans standard. This section presents some limitations that will be targeted by the next steps of our research.

Several frameworks based on the Java bean standard are based on metadata [Guerra et. al 2013], using code annotations to configure constraints on the classes. To use those frameworks full potential, the AOM adapter factory should be able to create code annotations on the generated class based on entity type properties and on property type properties. For instance, a property of a property type that defines the significant decimal digits of its number could be turned into the annotation @Tolerance for the Esfinge Comparison to configure the precision for each number that is compared.

A limitation of our current solution is that despite the fact that new adapters can be created when there are changes to an AOM entity type, this will not change any existing AOM adapters that have already been created. That can be a problem if this adapter is referenced and stored by other classes, and needs to be migrated to a new version.

Problems related to evolution, versioning and migration of an AOM type is addressed by the patterns documented by [Ferreira 2008] and [Hen-Tov et. al 2010]. However these patterns consider that the AOM type can be changed at runtime. However the adapter that represents an AOM entity cannot. Because of this, some of these proposed solutions for AOM evolution cannot be applied to migrate AOM adapters. Since this is still an open issue, our solution is not appropriate for systems where the adapters referenced by other classes need to be migrated due to changes to the AOM entity type.

8. Next Steps

This paper presented the proposed solution for the dynamic adapters for AOM entities and the current status for a minimum proof-of-concept implementation. These results can be considered the first step in a broader study that is being performed on how to increase reuse opportunities for applications with an AOM architectural style. This section describes the next steps intended for continuation of our research.

The first step is to add to the Esfinge AOM model the capability to have properties on every element of the AOM structure. Based on that, the metadata defined for the entity type and property type can be translated to code annotations on the generated adapter. The Java Beans frameworks to handle a property or a class differently can annotations defined on the adapter. Since the methods of a dynamically generated class can only be invoked by reflection, it is important to add more information about them on the AOM adapter to allow it to be fully used by that framework.

After adding the support for generating annotations on the AOM adapters, the model and the implementation will focus on expanding the scope to include some behavioral patterns, such as Strategy and Rule Object. The idea is to map the execution of such objects as methods on the AOM adapter. The metadata added to these methods, as code annotations, should enable their identification and invocation by frameworks. As an example, this could be used is to map an AOM behavior as a callback method invoked by a framework as a response to a given event.

9. Conclusions

This paper presented the proposal of a solution to allow the reuse of regular frameworks for AOM applications. It suggests using a dynamic generated adapter that creates a static class to adapt a dynamic AOM entity. The approach that generates and inserts code to adapt at runtime is not new [Reenskaug et al. 1995], but the contribution of this work is to how to use this approach for AOM architecture. The paper presents the implementation of a proof-of-concept and shows its use to adapt an entity to a Java Bean to be used by a comparison framework.

Acknowledgements

We acknowledge the support of CNPq (grant 445562/2014-5) and FAPESP (grant 2015/16487-1). We also thank a lot our shepherd, Rebecca Wirfs-Brock, for given a valuable and detailed feedback either on the technical ideas and for the paper text.

References

Atzmon Hen-Tov, David H. Lorenz, Lena Nikolaev, Lior Schachter, Rebecca Wirfs-Brock, and Joseph W. Yoder. 2010. Dynamic model evolution. In Proceedings of the 17th Conference on Pattern Languages of Programs (PLOP '10). ACM, New York, NY, USA, , Article 16 , 13 pages.

Gamma, E., Helm, R., Johnson, R., Vlissides, J.: “Design Patterns: elements of reusable object oriented software”; Addison-Wesley (1994).

Guerra, E. M. ; Aguiar, A. . Support for Refactoring an Application towards an Adaptive Object Model. Lecture Notes in Computer Science, v. 8583, p. 73-89, 2014.

Guerra, E., Souza, J. T., Fernandes, C.: “ Pattern Language for the Internal Structure of Metadata-based Frameworks” in Transactions on Pattern Languages of Programming, 3 (2013) 55-110.

Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Leon Welicki. 2008. Patterns for data and metadata evolution in adaptive object-models. In Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP '08). ACM, New York, NY, USA, , Article 5 , 9 pages.

JavaBeans(TM) specification 1.01 Final release, 2010, http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html, accessed on dez/2010.

Matsumoto, P. ; Guerra, E. M. . An Approach for Mapping Domain-Specific AOM Applications to a General Model. Journal of Universal Computer Science (Online), v. 20, p. 534-560, 2014.

Reenskaug, Trygve; Per Wold and Odd Arild Lehne.. Working With Objects: The OOram Software Engineering Method. [S.l.]: Prentice Hall, June 1995.

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).

Yoder, J. W., Johnson, R.: “The Adaptive Object-Model architectural style”; Proc. of 3rd IEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (2002).