An Architecture for Dynamic Web Services that Integrates Adaptive Object Models with Existing Frameworks

Antonio Dias
National Institute for Space Research INPE
São José dos Campos, São Paulo
Brazil
antoniodiasabc@gmail.com

Eduardo Guerra
National Institute for Space Research INPE
São José dos Campos, São Paulo
Brazil
guerraem@gmail.com

Phyllipe Lima
National Institute for Space Research - INPE
São José dos Campos, São Paulo
Brazil
phyllipe_slf@yahoo.com.br

Abstract

Nowadays, web services became one of the main alternatives for communication between software systems and even inside the components of the same application. In some domains, the change of requirements happens frequently, demanding flexibility from the architecture of the applications, and consequently also in the web services that they provide. In this context, the goal of this work is to provide an architecture that can be used for dynamic web services, allowing services to be changed and introduced at runtime. To fulfill these requirements this work proposes the usage of Adaptive Object Models (AOM) combined with existing web service framework, using dynamic adapters to integrate and decouple them. The framework Esfinge AOM Role Mapper received features to implement the behavioral part of the AOM model and to map the AOM rule objects with metadata to methods with code annotations in the dynamic adapters. The proposed architecture passes the adapters generated at runtime to the existing framework, which provide the web service based on its methods and code annotations. An evaluation based on a case study performed scenario-based tests to verify the architecture capability to create and change dynamic web services. Additionally, a modularity analysis verified the coupling between classes that use both frameworks. As a result, the proposed architectural solution was able to implement the dynamic web services in all the scenarios keeping the classes that handle the AOM model decoupled from the classes responsible for providing the web services.

CCS Concepts

• **Software and its engineering → Object oriented frameworks**.

Keywords

Adaptive Object Model, Software Engineering, Framework, Metaprogramming, Web Services, Reuse

ACM Reference Format:
Antonio Dias, Eduardo Guerra, and Phyllipe Lima. 2019. An Architecture for Dynamic Web Services that Integrates Adaptive Object Models with Existing Frameworks. In *XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357141.3357602

1 Introduction

In the development of complex systems, design rarely meets all architectural needs, as new attributes may arise in an entity or a new method to compose important functionality that was not identified in the initial phases of a project. An alternative to solving these issues is the use of a more flexible and dynamic architectural style, which allows adaptation to requirements changes at runtime. The architectural style called the Adaptive Object Model (AOM) [16] defines entity types, equivalent to classes, as instances based on the metadata that is read at runtime. This architecture style enables entity types to be changed dynamically by changing their metadata, which as a consequence should modify the application behavior.

Web Services are one of the most popular approaches to exchange information between applications, and even between components of the same application. A recent study indicates that changes in web services can be frequent [1, 3], and especially more granular micro-services would benefit from an adaptive architecture, with capabilities to create new services and change existing ones.

The creation of web services in applications usually rely on the use of frameworks. Those frameworks, that we refer as "traditional frameworks", usually use reflection and annotations to access the structure of static classes. The main difficulty to use AOM to create dynamic web services is because existing frameworks do not work with AOM entities, since they have a different structure.

The goal of this work is to propose an architecture that can integrate existing traditional frameworks for web services with AOM entities, in a way that changing the entity type structure enables the creation of new services and change the existing ones. To achieve this, the architecture uses Esfinge AOM Role Mapper[1], which is an AOM framework that can create dynamic adapters [12] that wraps the AOM entity in an interface that follows the Java

[1] esfinge.sourceforge.net/AOM.html

Beans standard and can be consumed by traditional frameworks. New features were added in Esfinge AOM Role Mapper during this work to introduce the dynamic behavior to the AOM model and implement its mapping to static methods that could be consumed by frameworks. The proposed architecture was evaluated in a case study that assesses if the requirements for changing web services behaviors are being achieved and to evaluate the coupling between the AOM framework and the web services framework.

2 Adaptive Object Model

Adaptive Object Model is an architectural style for systems in which classes, attributes, relationships, and behaviors are represented as metadata consumed at run time, and the same code can be used to process different classes that are not known at compile time. This enables systems built with this architecture to be flexible and can be modified at runtime, not only by programmers but also by business analyst or users, allowing changes to be done and made available quickly. This flexibility allows the domain to evolve as part of the business.

Fig. 1 presents the first model of the AOM architecture proposed by [18]. The entity metadata is stored in the XML data entry, that passes in the XML Parser and in the Metadata Interpreter before being stored at runtime in the metadata repository. This metadata is used to create domain objects that follow the AOM structure. The domain objects have their persistence mechanisms that can be relational databases or XML/XMI files.

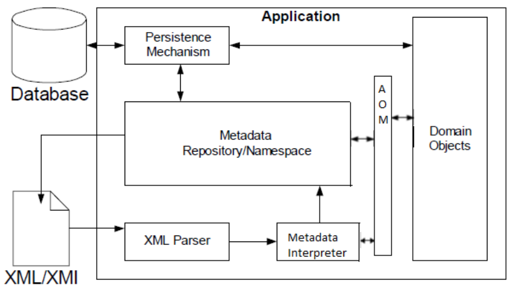

Figure 1: Adaptive Object Model [17]

AOM architecture is comprised of the following core patterns: Type Object, Property, Type Square, Accountability and Rule Object.

2.1 AOM Core Patterns

The Type Object pattern [9] is used in situations where the number of sub-classes that a class may need can not be determined during the system development. This pattern solves this situation by representing the sub-classes that are not known at development time as instances of a generic class that represents the type of an object.

The Property pattern [6] is applied in situations where instances of the same class can have different types of properties. The Property pattern solves this problem by representing the properties of an entity using a class and causing that entity to have a collection of instances of that class.

In the AOM architectural style, Type Object and Property patterns are used combined, resulting in the Type Square pattern [16]. In this one, the Type Object is used twice - once to represent the entities and entity types of the system and once to represent the properties and property types. Using Type Square pattern, new types of entities with different types of properties can be created. Likewise, existing entity types can be changed at runtime, since modeling is done at the instance level.

The Accountability [16] pattern allows the relationship between entities to be represented by an object (usually an instance of an Accountability class). Each Accountability object is associated with an Accountability Type object, which represents the relationship type. Because the associations between entities are represented at the level of instances, the types of relationships between entities can be created and modified at runtime.

Rule Object is a pattern used to create abstractions for both simple and complex business rules [17]. The Rule Object defines a default interface for a family of algorithms so that clients can work with any of them. If the behavior of an object is defined by one or more instances of Rule Object, then it can be easily changed by changing its instance. In the AOM architectural style, classes that use the Rule Object pattern are often associated with entity types, where they implement operations on the methods of a class.

The core AOM structure, composed by the patterns Type Object, Property, Type Square, Accountability and Rule Object, is presented in Figure 2. The operational level is used to represent the domain objects instances, which contains the information that is of interest to the application. The knowledge level represents the application metadata, which describes the application entities. Relating to an object-oriented language, the operational level is similar to objects while the knowledge level is similar to classes.

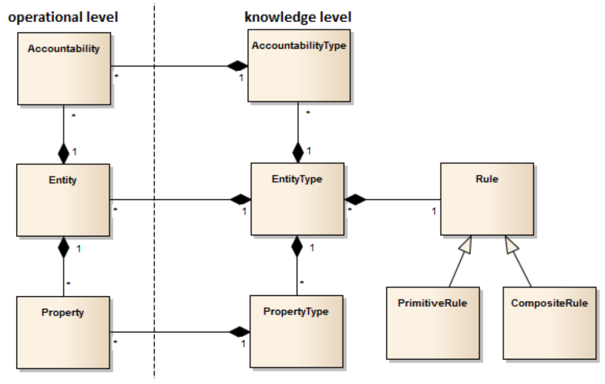

Figure 2: Core AOM structure, based on [15]

2.2 AOM Frameworks

AOM Applications usually implement the patterns in classes that are coupled with the application domain. These are called domain-specific AOM models. Since traditional frameworks do not support the structure of AOM entities, the application developers usually create their own components that are specific for its AOM models, such as for persistence and graphical interface. Since the application AOM model and its respective components depend on the domain, it is hard to reuse them in other AOM applications. This makes AOM applications expensive to develop.

Focusing on the optimization of the development time, some AOM frameworks, such as Oghma [5] and Ink [8], provide AOM classes that do not depend on the domain. These are called domain-independent AOM frameworks. Due to the high level of abstraction, they provide a very complete structure, but they require various configurations for its use. In addition to providing an AOM structure, they also include components, such as for the persistence of entities. Even though this solution allows some kind of reuse, it is not compatible with the usage of traditional frameworks, and the application is limited to the framework proprietary components.

The Java framework Esfinge AOM Role Mapper proposes a distinct approach. It provides an API based on a domain-independent AOM model, however, it provides adapters that can be used to recognize a domain-specific AOM model based on its code annotations. It also provides dynamic adapters [12] that generate classes based on the Java Beans standard that wraps the AOM entities. Since the dynamically generated adapters provide an API similar to static classes, the AOM entities can be used and consumed by traditional frameworks, increasing the reuse possibilities. The Esfinge AOM Role Mapper framework is described with more details in section 3.

Prior to this work, the Esfinge AOM Role Mapper framework did not implement the behavioral model of an AOM structure, usually implemented with the Rule Object pattern. The Rule Object is used to represent behavior in an entity, just as methods represent it in a class. The behavioral model of the AOM frameworks represents the logic linked to the entities, being used to create and change business rules in systems that have frequently changing requirements. To enable the dynamic definition of web services, this work implemented the behavioral model the Esfinge AOM Role Mapper framework, mapping the Rule Objects to methods in the Java Beans dynamic adapters. Section 4 presents this implementation with more details.

3 Esfinge AOM Role Mapper

Esfinge AOM Role Mapper is the AOM framework used as the basis by this work, implemented in Java, and available open-source [2]. It is part of a larger project called Esfinge that embrace several innovative metadata-based frameworks.

The Esfinge Role Mapper AOM framework works with three different types of models. The domain-specific AOM models that can be provided by the application, a proprietary domain-independent model, and dynamically generated Java Beans model. The main functionality of the framework is to map AOM structures from a domain-specific to a domain-independent AOM structure and from the domain-independent AOM structure to dynamically generated Java Beans model. Code annotations and descriptor files are used to provide additional information for creating the adapters. The idea is to make possible a hybrid model, which can be composed of entities defined by each of the approaches. For example, an entity defined as Java Bean could be added as a property in an AOM entity. This increases the possibility of reuse of these entities and gives more freedom to AOM application developers. A graphical representation of these mappings is shown in Figure 3.

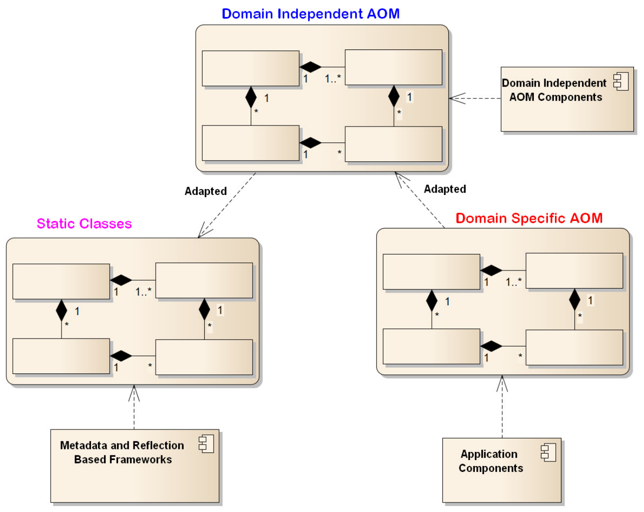

Figure 3: AOM Framework Model Representation

In real applications that use AOM as the architectural style, the AOM model is usually implemented only in entities where flexibility is a requirement, using only the required patterns [7]. Other entities also follow a static model adopted by the target programming language, containing fixed attributes and access methods. Even in classes representing AOM entities, for example, there might be static properties that represent peculiarities of the application domain [12].

To address these two issues, the Esfinge Role Mapper AOM framework provides a solution that meets the following requirements:

(1) Allow the use of components made for a domain-independent AOM application for entities from a domain-specific AOM model;
(2) Allow the use of traditional frameworks based on reflection and code annotations in domain-independent and domain-specific AOM entities.

4 Dynamic Behaviour on AOM

This section describes the following improvements developed on Esfinge AOM Role Mapper to support dynamic behaviour:

(1) Implementation of the Rule Object pattern in the framework's domain-independent model;
(2) Mapping the Rule Object pattern from a domain-specific AOM model to the domain-independent AOM model of the framework;
(3) Mapping of the Rule Objects from the domain-independent AOM model to methods on static class methods (Java Beans);
(4) Metadata mapping in AOM Rule Objects for annotations in methods.

The first step was to create classes that implement the Rule Object in the domain-independent AOM model provided by the

[2]esfinge.sourceforge.net/AOM.html - documentation available in portuguese

framework. The adapters were then updated so that domain-specific AOM objects could be mapped to this structure. From this mapping, the class that creates adapters for the domain-specific AOM entities can recognize Rule Object implementations and map behavior to the framework structure. Finally, the behavioral model was also added in the generation of adapters for the static classes API (Java Beans), allowing the use of these objects adapted with traditional frameworks that work with the invocation of methods by reflection. In this mapping, Rule Objects metadata becomes annotations in their adapters methods. This feature is important because web service frameworks use annotations to map methods to services.

4.1 Adding Rule Object in Domain-independent AOM Model

This section describes how the behavioral model was implemented in the domain-independent AOM model of Esfinge AOM Role Mapper. The first change was the creation of the `RuleObject` interface, establishing a contract for classes that implement entity behavior. The `RuleObject` interface extends the `HasProperties` interface to be able to receive additional metadata, as the `IEntity` and `IEntityType` interfaces. Next, we created the `BasicRuleObject` class that implements the `RuleObject` interface and inherits the `ThingWithProperties` class so that the behavioral model has its properties, according to the diagram shown in Figure 4.

The `RuleObject` interface should be implemented by classes that represent the behavior of AOM entities. The new behavior should be implemented in the `execute()` method. This method receives two parameters, the first parameter being the entity in which the method is executed, and the second parameter an array of Objects which represent the parameters that the method receives for its execution.

To allow the addition of a `RuleObject` to an AOM entity type, the `addOperation()` method has been added in the `IEntityType` interface. It receives as parameters the name of the rule and an instance of a class that implements the `RuleObject` interface. The `RuleObject` is internally stored on a map, and should be located using its name.

The method `executeOperation()` was added to the `IEntity` interface. It is used to execute the operation on a given entity. It has two parameters, which are the rule name that will be executed and an array of Object with the parameters expected by the Rule Object. The rule name is the same name that was used to add the operation on the `IEntityType` interface.

The `GenericEntity` and `AdapterEntity` classes implement the `IEntity` interface, and use the method `executeOperation()` to obtain the `RuleObject` of the corresponding entity type and invoke the `execute()` method with its parameters.

4.2 Usage Example of Domain-independent API

This section presents an example of how the domain-independent API is used to create an Entity Type with a Rule Object and execute its respective logic. The first step is to create the object of type `IEntityType`. The Listing 1 shows the creation of an `IEntityType` named product with the attributes `creationDate` of type `Date`, and `name` of type `String`. Both attributes are mapped as required.

Listing 1: Creating the Entity Type with Properties and a Rule Object

```java
//create entity type instance
IEntityType productType = new GenericEntityType("Product");

//creating property types
GenericPropertyType creationDatePropertyType = new
      GenericPropertyType("creationDate", Date.class);
creationDatePropertyType.setProperty("notempty", true);
GenericPropertyType namePropertyType = new GenericPropertyType("name",
      String.class);
namePropertyType.setProperty("notempty", true);

//adding property types to the entity type
productType.addPropertyType(creationDatePropertyType);
productType.addPropertyType(namePropertyType);

//adding a rule object in the entity type
productType.addOperation("creationTime", new CalculateTime("creationDate"));
```

The last line of code from Listing 1 shows the creation of the `RuleObject` and its addition to the entity type. The `CalculateTime` class extends the `BasicRuleObject` class that implements the interfce `RuleObject` with its `execute` method. Its constructor receives a String parameter that should have the name of an attribute of type `Date` from the entity type. The implementation of `CalculateTime` is not presented but it aims to calculate the number of years from the date contained in the attribute until now.

As can be seen in that example, the `CalculateTime` class could be reused for another operation in the same entity type referring to others attributes. The rule object implementations can receive parameters to configure its execution, enabling several different behaviors being executed by the same class.

Listing 2 shows the creation of an entity from the `Product` entity type. It sets values for the properties and executes the operation. As can be seen in Listing, the method `executeOperation()` was invoked passing the String added with the rule object. Since the method return is of type `Object`, it should be cast for a specific class.

Listing 2: Creating an Entity and executing its operation

```java
//create entity
IEntity product = productType.createNewEntity();

//adding property values
product.setProperty("name", "Notebook");
product.setProperty("creationDate", new GregorianCalendar(2010, 09 ,
      23).getTime());

//executing operation
int years = (Integer) product.executeOperation("creationTime");
assertTrue("Considering SBCARS 2019 date", 9, years)
```

4.3 Rule Object Based in Expression Language

Expression Language (EL)[3] is a language used in Java projects where you create simple expressions that have direct access to the attributes of objects considering the Java Beans standard. To allow the definition of logic based on expressions defined in the entity type metadata, the framework Esfinge AOM Role Mapper provide in its class library an implementation of the `RuleObject` interface that executes an Expression Language passed in its constructor. The use of this type of Rule Object allows the configuration of expressions that need to be changed at runtime. By having operations based

[3]docs.oracle.com/javaee/6/tutorial/doc/gjddd.html

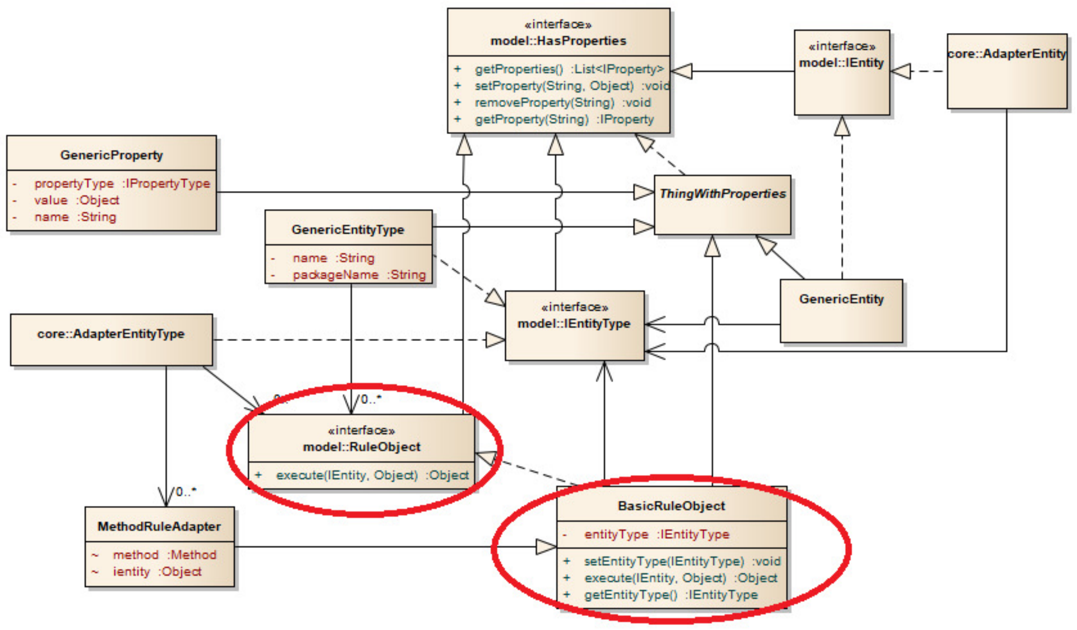
«interface» model::HasProperties
+ getProperties() :List<IProperty>
+ setProperty(String, Object) :void
+ removeProperty(String) :void
+ getProperty(String) :IProperty
«interface» model::IEntity
core::AdapterEntity
GenericProperty
- propertyType :IPropertyType
- value :Object
- name :String
ThingWithProperties
GenericEntityType
- name :String
- packageName :String
«interface» model::IEntityType
GenericEntity
core::AdapterEntityType
0..*
0..*
«interface» model::RuleObject
+ execute(IEntity, Object) :Object
0..*
MethodRuleAdapter
~ method :Method
~ ientity :IObject
BasicRuleObject
- entityType :IEntityType
+ setEntityType(IEntityType) :void
+ execute(IEntity, Object) :Object
+ getEntityType() :IEntityType

Figure 4: AOM RoleMapper after Behavioral Model

on a configured expression, the logic can be easily modified by changing this expression.

For this, the ELContextAOM class was created to represent the context where the EL will be executed. This class has three attributes. The first one, functionMapper, map the EL functions. The second one, variableMapper, map variables, and the third one, compositeELResolve, interpret the variables and solve their value. Listing 3 displays the ELContextAOM class.

Listing 3: Class ELContextAOM for Expression Language execution

```
public class ELContextAOM extends ELContext {
   private FunctionMapper functionMapper;
   private VariableMapper variableMapper;
   private CompositeELResolver elResolver;

   public ELContextAOM(FunctionMapper functionMapper, VariableMapper
          variableMapper, ELResolver... resolvers) {
      this.functionMapper = functionMapper;
      this.variableMapper = variableMapper;
      elResolver = new CompositeELResolver();
      for (ELResolver resolver : resolvers) {
         elResolver.add(resolver);
      }
   }
   public static EvaluationContext createContext(Class<?> functionClass,
          Map<String, Object> attributeMap) {
      VariableMapper vMapper = mapVariables(attributeMap);
      FunctionMapper fMapper = mapFunctions(functionClass);
      ELContextAOM context = new ELContextAOM(fMapper, vMapper, new
             ArrayELResolver(), new ListELResolver(),
             new MapELResolver(), new BeanELResolver());
      return new EvaluationContext(context, fMapper, vMapper);
   }
   public static Object execute(String expr, Class<? extends Object>
          objectClass, Map<String, Object> map) {
      EvaluationContext ec = ELContextAOM.createContext(objectClass, map);
      ValueExpression result = new
             ExpressionFactoryImpl().createValueExpression(ec, expr,
             Object.class);
      return result.getValue(ec);
   }}
```

To create an operation based on EL rules the ExpLangRuleObject class was created. It implements the RuleObject interface to have the same abstractions as rule objects and receives the target expression in its constructor. Its execute() method uses the class ELContextAOM to add the AOM attributes as EL variables and execute the expression. After the addition in the entity type, the rule object can be executed as any other.

4.4 Mapping Rules from a Domain-specific AOM

Esfinge AOM Role Map supports domain-specific AOMs created by applications. It uses code annotations to map the AOM elements. Through them, the framework identifies and creates the adapters that make the connections between the specific domain entity and the independent domain entities.

This section focus on the annotations created to map methods and rule objects. Information about other mappings, such as for entities, entity types and properties, can be obtained in previous publications [12]. To perform the behavior mapping, three annotations were created:

- @RuleClass - Configure the interfaces used for Rule Objects in the target domain-specific model;
- @RuleMap - Configure an attribute in the Entity Type that contains a map of Rule Objects. This map should have as generic types a String as the key and an interface with @RuleClass as the entry;
- @RuleMethod - Used to configure methods with behavior in two distinct scenarios. It can configure a method of an interface with @RuleClass as the method that represents the operation, as well as fixed methods in the entity type

class that should be mapped as Rule Objects in the domain-independent model;

Listing 4 presents an example of mapping from an interface that represents a Rule Objects in a domain-specific AOM model. It can be observed that the interface is mapped with the `@RuleClass` annotation and the `executeLogic()` method is mapped with the `@RuleMethod` annotation.

Listing 4: Domain-specific Rule Object interface

```
@RuleClass
public interface SensorLogic{
   @RuleMethod
   Object executeLogic(Sensor s, Object... params);
}
```

Listing 5 displays the domain-specific entity type mapping, named `SensorType`. This class is used to create dynamically types of sensor, which can have different attributes and operations. The instances of these types should have the class `Sensor`, which is not presented in the paper. To map this class to the domain-independent entity the `@EntityType`, `@PropertyType` and `@CreateEntityMethod` annotations were used. They map the code elements to its respective role in the AOM model, but since they are not the focus of this paper, they will be not further explained.

Listing 5: Map a RuleObject

```
@EntityType
public class SensorType {

   @RuleMap
   private Map<String, SensorLogic> operations = new HashMap<>();

   @PropertyType
   private Set<SensorPropertyType> propertyTypes = new
        HashSet<SensorPropertyType>();

   public void addPropertyTypes(SensorPropertyType propertyType) {
      propertyTypes.add(propertyType);
   }

   @RuleMethod
   public int miliUnitValue(Sensor s, String property){
      return entity.getProperty("property") * 1000;
   }

   @CreateEntityMethod
   public Sensor createSensor() {
      Sensor sensor = new Sensor();
      sensor.setSensorType(this);
      if (operations == null) {
         operations = new HashMap<>();
      }
      return sensor;
   }
}
```

The attribute operations receive the annotation `@RuleMap`, which maps the the attribute used to map the entity operations. The annotation `@RuleMethod` is used on the method `miliUnitValue()`, configuring that this fixed method should be included in the list of Rule Objects in the domain-independent model.

It is very common for entities and entity type classes in a domain-specific AOM to have common methods that are available to all types. Since these methods are not available in a domain-independent model, they should be mapped to Rule Objects.

Listing 6 presents a code example that creates the adapters based on the domain-independent API from the domain-specific entity Sensor. In the presented source code, an entity type and an entity instance is created based on the domain-specific model, and further, this entity is adapted to the domain-independent API, which is used to invoke the rule objects. As can be seen, the rule object added dynamically and the one based on an existing method on the entity type class are invoked the same way.

Listing 6: Invoking the RuleObject in the adapted entity

```
//Create entity type with 2 property types
SensorType sensorType = new SensorType();
sensorType.addLogic("convertUnits", new ConvertUnits());

SensorPropertyType prop1 = new SensorPropertyType();
prop1.setName("unit");
prop1.setPropertyType(String.class);
sensorType.addPropertyTypes(prop1);

SensorPropertyType prop2 = new SensorPropertyType();
prop2.setName("value");
prop2.setPropertyType(Integer.class);
sensorType.addPropertyTypes(prop2);

//create entity
Sensor sensor = sensorType.createSensor()

//adapt entity to domain-independent model and set values
AdapterEntityType adaptedEntityType = AdapterEntityType.getAdapter(sensorType);
AdapterEntity entity = AdapterEntity.getAdapter(adaptedEntityType, sensor);
entity.setProperty("unit", "m");
entity.setProperty("value", 20);

// execute the Rule Object added
Object result1 = entity.executeOperation("convertUnits", "value", "feet");

// execute mapped fixed method
Object result2 = entity.executeOperation("miliUnitValue", "value");
```

4.5 Rule Objects to Methods in the Java Beans Adapter

The Esfinge AOM Role mapper has a feature to generate dynamically Java Beans adapters from entities from the domain-independent model. Another feature developed for the behavior model in these adapters was the insertion a method for each Rule Object. To generate the adapter class bytecode at runtime, it uses the ASM framework[4]. Each method generated to represent a Rule Object has the behavior to invoke it in the encapsulated entity.

In the example of Listing 7 a Java Bean adapter is created based on an entity and the method that represents a Rule Object in the adapter is invoked using reflection. Passing the Java Bean adapter to a traditional framework it would be able to invoke its method by reflection as it does in a regular class.

Listing 7: Entity Generation with RuleObject

```
//create entity type and entity
IEntityType productType = new GenericEntityType("Product");
productType.addOperation("operation", new ExampleRuleObject());
IEntity product = productType.createNewEntity();

//create Java Bean adapter
AdapterFactory af = AdapterFactory.getInstance("mapping.json");
Object beanAdapter = af.generate(product);

//invoke adapted method by reflection
Method m = beanAdapter.getClass().getDeclaredMethod("operation",
      Object[].class);
Object result = m.invoke(personAdapter, null);
```

Esfinge AOM Role Mapper domain-independent model also provides methods to add custom metadata to the elements. Based on a

[4]https://asm.ow2.io/

mapping defined in a JSON file, when generating the Java beans adapter, the framework add code annotations to these elements. So, the metadata added to the `RuleObject` is adapted to annotations on the target method. An example of the JSON file is presented in Listing 8. In this listing, three types of metadata are mapped to annotations from the Spring framework [5].

Listing 8: JSON file example to map adapted object

```
{
  "restcontroller":[
    {"target":"class"},
   {"annotationPath":"org.springframework.web.bind.annotation.RestController"}
  ],
  "autowired":[
    {"target":"attribute"},
   {"annotationPath":"org.springframework.beans.factory.annotation.Autowired"}
  ],
  "nomeendpoint":[
    {"target":"method"},
   {"annotationPath":"org.springframework.web.bind.annotation.RequestMapping"},
   {"parameter_1": "value"}
  ]
}
```

This feature is important to enable the usage of the adapters by existing frameworks, since many of them locate the method to be invoked by the presence of annotations. For instance, web services methods usually uses annotations to identify methods to be used as web services endpoints.

5 Proposed Architecture for Dynamic Web Services

This section presents the architecture being proposed for dynamic web services. Its main requirements are to enable the creation and modification of web services at runtime. Another important requirement is to reuse existing web services frameworks, which is important to enable the introduction of dynamic services in existing applications.

The main idea for the architecture to generate web services is following these steps: (a) an AOM entity is created dynamically based on the metadata provided; (b) an adapter based on static classes is generated with the annotations mapped to its metadata; and (c) a web service framework loads the generated classes to provide services based on its annotated methods. So, providing new metadata or changing the existing data it is possible to respectively create new web services or change the behavior or contract from the existing ones. A concrete architecture using Esfinge AOM Role Mapper for AOM and Spring as the web services framework was defined to implement the proposed solution. Figure 5 presents the structure created for the architecture.

As shown in Figure 5, the Application class runs the application and instantiates the BeanRegistration class, which is responsible for registering objects created with the AOM architecture. It instantiates the BeanFactory class where AOM-adapted objects will be created. The BeanFactory class instantiates the EntityFactory class and creates the AOM entity, which according to the type of configuration of the AOM entity type that can be an instance of EntityType or AdapterEntytiType. It then returns this created object to the BeanFactory class that creates the dynamic adapter with the respective methods and annotations.

The Esfinge AOM Role Mapper uses the properties contained in the RuleObject itself to map the metadata into annotations during adapter generation. Then, the BeanRegistration records this object in Spring, which recognizes the object through the @RequestController annotation and the method mapped with the @RequestMapping annotation. After that, it makes the new web service available to meet client requests through the HTTP protocol.

6 Case Study

To evaluate if the proposed architecture for web services reaches its goals presented in the previous section, a case study was elaborated. Its focus is to verify how the creation of web services at runtime is accomplished through the addition of behavior in AOM entities by the framework Esfinge AOM Role Mapper reusing functionality from the framework Spring, designed to work with static classes. For this assessment to be considered successful, from the behavior inserted in the entities should be possible to use the functionality of the Spring framework to generate web services without a direct coupling between the code that works with the AOM entities and the code that use Spring framework to generate the services.

Following Basili's Goal-Question-Metric-GQM [2], the purpose of this case study is:

> *To analyze the use of the AOM Role Mapper framework with the purpose of using reflection-based frameworks and metadata for AOM applications, regarding the invocation of behavior of AOM entities, from the researcher's point of view, in the context of applications that provide the web services that must be created and modified dynamically.*

The case study is guided by the following research questions:

- Q1 - *Is it possible to use the web services framework functionalities that invoke methods by reflection for AOM entities, adding services and modifying behavior from changes in the entity?*
- Q2 - *Is it possible to decouple the code that generates and manipulates the AOM entities of code that invokes the features of the web services framework?*

To answer the research questions an application with the architecture presented in Section 5 was created[6]. It uses Esfinge AOM ROle mapper as the AOM frameworks and Spring as the web services framework.

In order to answer the research question Q1, we executed the application with metadata to create web services. We simulated different scenarios based on the creation of dynamic web services and verified if the solution was capable to behave as expected. To evaluate the functional part related to the dynamic creation and change of web services (Q1), we used test scenarios based on [10], which contemplates a scenario-based evaluation of given software architecture. Four scenarios different scenarios were designer, executed and evaluated.

To answer the research question Q2, a dependency analysis was performed in the code created for the case study. A Design Structure Matrix (DSM) [14] was used to evaluate the coupling

[5]spring.io

[6]github.com/antoniodiasabc/dynamicwebservice

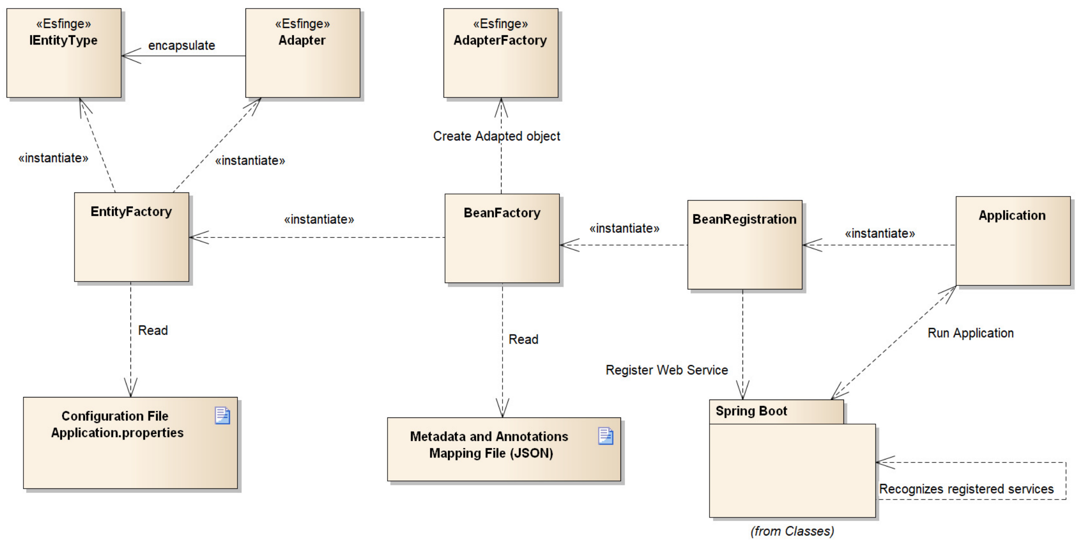

Figure 5: Architecture of the solution developed for the case study

between components that use the Spring and Esfinge AOM Role Mapper frameworks. A DSM is a square matrix where rows and columns represent the elements of a system and the cells represent the relationships between the elements in its row and column. DSM is used to evaluate the coupling between software components in many recent research papers, such as [13] and [4].

6.1 Scenario-based Evaluation

The tests developed to use the capabilities of creating adapters for AOM entities from the Esfinge AOM Role Mapper framework. These entities have metadata attached to the Rule Objects that are mapped to generate Spring framework annotations on the adapter. In this way, the framework must recognize the annotations to create web services, which will invoke the Rule Objects to respond to the requests of the clients.

Entities and entity types are created from a configuration file, named application.properties. A service of the Spring framework reads the properties of this file to create the entity from the RoleMapper AOM domain-independent model, encapsulates the entity in the adapter with static classes API, and from the reading of the annotations of the class by reflection makes its methods such as web services. The name of the web service that serves this request is also informed in the configuration file. It is also metadata that is annotated and read by the Spring framework. After these mappings, values are entered in the entity properties that are used in the method execution. The following four scenarios were created to verify the architecture behavior:

- Create a dynamic web service that accesses the value from an entity property;
- Create a dynamic web service that executes a formula defined in expression language (EL) that use values from entity properties;
- Create a dynamic web service based on a Rule Object that receives parameters;
- Create a new dynamic web service based on data entered on an HTML page form;

All scenarios were structured and executed, and the architecture was able to provide the expected dynamic behavior for each of them. Due to space restrictions, it is not possible to present the detailed information about all scenarios, but as an example, the next section presents the steps to execute one of them.

6.2 Creating Dynamic Web Service

This scenario is intended to test the creation of a service without using parameters, from a Rule Object added at runtime in the entity.

Test Procedure: To run the tests the following steps will be performed:

(1) Change the configuration of the application.properties file;
(2) Run the unit test running Spring:
  - Create web service;
  - Publish the web service;
  - Execute the client requesting the web service;
  - Get the response to the request;
  - Compare the result with the expected value and for Spring execution.
(3) Check the Spring log to confirm that the service has been correctly mapped.

Configuration: Listing 9 displays the configuration used to create the entity that has mapped behavior method to create the magnetometer/intensity name web service to run this test scenario.

Listing 9: Configuration file for this scenario

```
entitytype.name = Magnetometro
entitytype.properties = direcao;sentido;intensidade
entitytype.ruleobject.rulename = retornaIntensidade
entitytype.ruleobject.class = org.inpe.RetornaIntensidade
entitytype.ruleobject.metadata.nomeendpoint = magnetometro/intensidade
entity.direcao.param = 350
entity.sentido = N
entity.intensidade = 50
```

Test Source code: The Listing 10 presents the test developed for this scenario. This test is executed as an integration test that invokes the web service from the outside.

Listing 10: Creating object directly mapping dynamic method as web service

```
@Test
public void testMapDynamicMethod() {
   beanRegistration.handleResults("magnetometro");
   try {
      this.mockMvc.perform(get("/magnetometro/intensidade")).andDo(print())
              .andExpect(status().isOk())
            .andExpect(content().string("50"));
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Result: As a result, during Spring execution, the log displays the magnetometer/intensity web service recognition, showing the name of the generated method, as presented in Listing 11. The test was run successfully and returned the expected result.

Listing 11: Spring Execution Log Mapping a method to web service

```
RequestMappingHandlerMapping : Mapped "{[/magnetometro/intensidade]}" onto
       public java.lang.Object
       magnetometroAOMBeanAdapter.retornaIntensidade(java.lang.String,
    java.lang.Object...) throws org.esfinge.aom.exceptions.EsfingeAOMException
```

6.3 Modularity Analysis

The coupling analysis between the classes developed to perform this activity was done using the Design Structure Matrix (DSM) generated by the JArchitect tool. The generated DSM shows the dependencies between the classes created in the case study for creating web services and the RoleMapper and Spring AOM frameworks. The objective is to verify the low coupling between these classes and the frameworks used, confirming that no class of the case study depends at the same time of the Spring framework and the Esfinge AOM Role Mapper framework. Figure 6 presents the DSM between the classes developed in the case study and the two frameworks used.

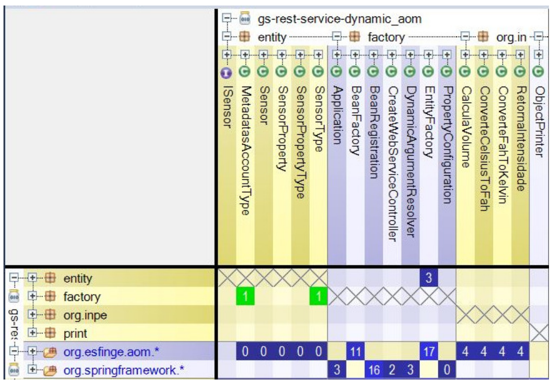

Figure 6: Dependency of the classes for creating web services at run time with the Spring and Esfinge frameworks

By analyzing the dependencies between the case study classes and the AOM Role Mapper framework, 17 dependencies can be identified between the EntityFactory class and the AOM Role Mapper Esfinge framework. The BeanFactory class presents 11 dependencies with the Esfinge AOM Role Mapper framework. Neither of the two classes that have dependencies with the Spring framework.

Analyzing the classes of the case study that interact with the Spring framework, it is verified that: the Application class has three dependencies; the `BeanRegistration` class has 16 dependencies; the `CreateWebServiceController` class has two dependencies, and the `DynamicArgumentResolver` class has three dependencies. None of them depend on AOM Role Mapper. The rule classes used in the test scenarios present four dependencies each with the RoleMapper AOM framework. This dependency is with the `BasicRuleObject` class that is extended to for the class to implements the Rule Object API.

6.4 Analysis

The analysis of the results of this case study is focused on answering the research questions.

Q1: *Is it possible to use the web services framework functionalities that invoke methods by reflection for AOM entities, adding services and modifying behavior from changes in the entity?*

Four test scenarios with different characteristics were designed and executed successfully showing that the proposed architecture is capable of implement web services that can be dynamically created and changed. Spring framework was able to successfully consume the dynamically generated Java Beans adapter created based on the AOM entity properties and Rule Objects. The web services were created based on the adapter methods and annotations, which were generated based on the mapping with AOM custom metadata. All web services worked as expected in all test scenarios presented.

Q2: *Is it possible to decouple the code that generates and manipulates the AOM entities of code that invokes the features of the web services framework?*

The modularity analysis performed based on the DSM has shown that it is possible to structure that architecture in a way that classes that works with the Esfinge AOM Role Mapper framework do not depend on Spring and vice-versa. The adapters dynamically generated by the AOM framework are loaded by Spring as a regular application class and all the AOM structure is wrapped inside that adapter. Even the adapter generator uses an external file to map custom metadata to Spring annotations and does not have a direct dependence on them. With this decoupling, the introduction of dynamic web services would not change the structure of an application that already have static ones implemented.

The decoupling between the web services functionality and the generation of dynamic entities is an evidence that it would be possible to change these frameworks by others that presents similar characteristics. To support AOM, we do not find another framework that generates the same kind of dynamic adapter, however for web services there are other frameworks that provide web services based on method annotations.

7 Related Work

In this section we present previous work performed by other researchers and developers regarding dynamic web services generation and composition.

Web service composition is a much explored field, that aims to create value-added services by integrating available services. According to [19], there is a mass of web services with the same functionalities, but with different Qos (Quality of Services) values and uncertainties of the services' application environment. To overcome this, the authors proposed a a two-stage approach solution. In the first stage, the top K-Web service composition schemes based on each services' historical QoS values are selected. In the second stage, before selecting the best service for each task, QoS values of each candidate service are predicted based on the improved case-based reasoning, and the best service is selected according to the predicted QoS values. Through QoS prediction,the reliability of the composite Web service can be greatly enhanced.

The work performed by [11] deals with automatically generating abstract web services. This a concept used to specify the functionality of certain types of Web services, that significantly benefits their discovery and composition. Usually, the approach to generate abstract services is a manual process that demands intensive human intervention. The authors proposed a process to automate this by forming a service community, which is a functionality-based service organization that groups together services providing similar functionalities. From this community it is possible to extract common functional features of services, generating the maximum number of abstract services as possible. To demonstrate the efficiency, the authors performed a comprehensive experimental study on real world web service data.

We did not find in the literature any work for an architecture that aims to provide a web services infrastructure capable of adding and changing the behavior of existing services at runtime as it is proposed by this work.

8 Conclusion

This paper proposed an architecture for the dynamic creation of web services, using AOM and existing traditional frameworks. The base AOM solution was provided by Esfinge AOM Role Mapper, which received new features to implement the behavioral part of the AOM model. This framework was chosen because it generates Java Beans adapters based on the AOM entities. Spring framework was chosen as the web services framework for its adoption in the software development industry.

A case study was designed and executed to evaluate the target architecture. For the functional point of view, four test scenarios were executed successfully showing that the proposed architecture is capable to provide functionality to enable the dynamic creation and change of web services. A modularity analysis revealed that it is possible to decouple the usage of the AOM frameworks from the usage of the web services framework, presenting an evidence that this solution can work with other frameworks and that a structure of an application that provide web services might be maintained to introduce the proposed solution. One limitation of the case study is that it was developed using only the Spring framework. Although it is widely used in the web development industry, it is possible that other frameworks may have details that are not implemented by the current stage of the solution.

As future works, the following research can be considered: extend Esfinge AOM Role Mapper framework to create new mapping possibilities; verify the viability of the solution for other frameworks; verify the impact of this solution in other quality attributes, such as performance and reliability; and, experiment this approach in a real application.

References

[1] D. B. F. C. de Almeida and E. M. Guerra. 2016. Evolution of XSD Documents and Their Variability During Project Life Cycle: A Preliminary Study. In *Computational Science and Its Applications – ICCSA 2016*. Springer International Publishing, Cham, 392–406.

[2] V. R. Basili and H. D. Rombach. 1988. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering* 14, 6 (1988), 758–773.

[3] D. Benincasa. 2015. *Evaluation of web services contracts and their variability during project's life cycle (Master's thesis).* Master's thesis. Instituto Nacional de Pesquisas Espaciais, São José dos Campos, SP, Brazil.

[4] R. Benkoczi, D. Gaur, S. Hossain, and M. A. Khan. 2018. A design structure matrix approach for measuring co-change-modularity of software products. In *Proceedings of 15th International Conference on Mining Software Repositories (MSR).* IEEE, 331–335.

[5] H. S. Ferreira, F. F. Correia, and A. Aguiar. 2009. Design for an adaptive object-model framework. In *Proceedings of the 2009 International Conference on Model Driven Engineering Languages and Systems (MoDELS'09).* IEEE.

[6] M. Fowler. 1996. *Analysis patterns: reusable object models.* Addison-Wesley.

[7] E. Guerra and A. Aguar. 2014. Support for Refactoring an Application towards an Adaptive Object Model. In *Computational Science and Its Applications : ICCSA 2014.* Springer, Berlin, 73–89.

[8] A. Hen-Tov, L. Nikolaev, L. Schachter, R. Wirfs-Brock, and J. W. Yoder. 2010. Adaptive object-model evolution patterns. In *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs.* Latin American Conference on Pattern Languages of Programs, ACM, 5.

[9] R. Johnson and B. Woolf. 1997. Type Object. In *Pattern Languages of Program Design.* Vol. 3. Addison-Wesley, 47–65.

[10] Rick Kazman, Mark Klein, and Paul Clements. 2000. *ATAM: method for architecture evaluation.* Technical Report. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

[11] X. Liu and H. Liu. 2012. Automatic Abstract Service Generation from Web Service Communities. In *2012 IEEE 19th International Conference on Web Services.* 154–161. https://doi.org/10.1109/ICWS.2012.41

[12] Patricia Megumi Matsumoto and Eduardo Guerra. 2014. An Approach for Mapping Domain-Specific AOM Applications to a General Model. *Journal of Universal Computer Science* 20, 4 (2014), 534–560.

[13] Roozbeh Sanaei, Kevin Otto, Katja Hölttä-Otto, and Jianxi Luo. 2015. Trade-off analysis of system architecture modularity using design structure matrix. In *Proceedings of ASME 2015 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference.* V02BT03A037.

[14] A. Yassine. 2004. An introduction to modeling and analyzing complex product development processes using the design structure matrix (DSM) method. *Urbana* 51, 9 (2004), 1–17.

[15] J. W. Yoder, F. Balaguer, and R. Johnson. 2001. Adaptive object-models for implementing business rules. *Urbana* 51 (2001), 61801.

[16] J. W. Yoder, F. Balaguer, and R. Johnson. 2001. Architecture and design of adaptive object-models. *ACM Sigplan Notices* 36, 12 (2001), 50–60.

[17] J. W. Yoder and R. Johnson. 2002. The adaptive object-model architectural style. In *Software architecture*, Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela (Eds.). Springer, 3–27.

[18] J. W. Yoder and R. Razavi. 2000. Metadata and Adaptive Object-Models. In *Object-Oriented Technology.* Springer Berlin Heidelberg, Berlin, Heidelberg, 104–112.

[19] Z.Liu, D. Chu, Z. Jia, J. Shen, and L. Wang. 2016. Two-stage approach for reliable dynamic Web service composition. *Knowledge-Based Systems* 97 (2016), 123 – 143. https://doi.org/10.1016/j.knosys.2016.01.010