Adaptive Object-Model Evolution Patterns

ATZMON HEN-TOV, Pontis Ltd.
LENA NIKOLAEV, Pontis Ltd.
LIOR SCHACHTER, Open University of Israel
REBECCA WIRFS-BROCK, Wirfs-Brock Associates, Inc.
JOSEPH W. YODER, The Refactory, Inc.

An Adaptive Object-Model (AOM) is a software architecture style that represents user-defined entities, attributes, relationships, and behavior in an object-oriented domain model as metadata. In an AOM implementation, domain entities are constructed from externally stored definitions (metadata) that are interpreted at run-time. Users, who may not be programmers, can change externally stored metadata when they want to change the definitions of domain entities. This paper describes two patterns that help minimize the impacts of evolving Adaptive Object-Model definitions. The first pattern, Break and Correct, describes one way to guarantee consistent changes to an Adaptive Object-Model in an environment where the AOM must run continually (24x7). The second pattern, Evolution Resilient Scripts, enables the refinement of AOM type definitions without having to change scripting code.

Categories and Subject Descriptors: **D.3.3 [Programming Languages]**: Language Constructs and Features – abstract data types, frameworks, polymorphism, patterns, control structures; **D.1.5 [Programming Techniques]:** Object-oriented Programming; **D.2.2 [Design Tools and Techniques]:** Object-oriented design methods; **D.2.11 [Software Architectures]:** Domain-Specific Architectures, Patterns—*Adaptive Object-Model*

General Terms: Design, Language

Additional Key Words and Phrases: Adaptive Object-Models, Adapting, Reflection, Dynamic Hooks, Patterns, Scripts

ACM Reference Format:

Hen-Tov, A., Nikolaev, L., Schachter, L., Wirfs-Brock R., and Yoder, J. W., 2010. Adaptive Object-Model Evolution Patterns. Proceedings of the Latin American Conference on Pattern Languages of Programs (September 2010), 19 pages.

1. INTRODUCTION

An Adaptive Object-Model is an instance-based software system that represents domain-specific entities, attributes, relationships, and behavior using metadata [FY98; YBJ01; YJ02]. Adaptive Object-Model architectures are usually implemented by applying several patterns to represent a domain model and its behavior. For a brief description of the core patterns of this architectural style, see Appendix A.

Typically in an Adaptive Object-Model, metadata descriptions of domain-specific definitions are stored externally and interpreted at runtime. This is similar to a UML Virtual Machine described by [RFBO01]. Users change the metadata (object model) to reflect changes in the domain. As metadata evolves, there may be a mismatch between the domain entities, their properties, and their behavior and other code, such as scripting code that relies on metadata definitions. Also, as one AOM entity is refined, it may require other dependent AOM entity definitions to change also. This happens frequently during the evolution of an AOM system. Therefore, it is important that an AOM system be designed to readily accommodate such changes.

One way to do this is to write migration scripts that evolve all metadata changes in lock step, and force a clean restart of the system after a carefully planned upgrade release. However, sometimes a single logical AOM model change comprises several modifications to different entity types, some of which may temporarily break system consistency. The user then needs assistance for isolating the running system, preventing it from reaching an inconsistent state, and for verifying that through a series of changes, consistency has been regained. One way to support model changes is with an AOM definition editor, similar to the tools provided by

This work is partially supported by the *Israel Science Foundation (ISF)* Grant #926/08.
Author's address: A. Hen-Tov, Pontis Ltd., Glil Yam 46905, Israel; email: atzmon.hentov@gmail.com; L. Nikolaev, Pontis Ltd., Glil Yam 46905, Israel; email: lena@pontis.com; L. Schachter, Open University of Israel, Raanana 43107, Israel; email: liorsav@gmail.com; R. Wirfs-Brock, Wirfs-Brock Associates, Inc., USA; email: rebecca@wirfs-brock.com; J. W. Yoder, The Refactory, Inc., USA; email: joe@refactory.com

an IDE for programmers, which enable users to make and test changes that might make the system invalid. The editor can show the users where the inconsistencies are and let them fix the AOM definitions before committing all changes. Thus this allows the user to "Break" and then "Correct" these inconsistencies as their entities evolve.

Also, different behaviors such as GUI scripts often rely on AOM entity definitions to perform calculations or make decisions based on specific AOM attributes and values. If direct references to these AOM structures are hard coded into the scripting code, it will be difficult to evolve the AOM definitions without having to also change these scripts. Instead of directly referencing AOM structures, if the script code takes simple typed parameters that are automatically extracted from the model definitions, controlled changes to the entities can be made without always having to modify the scripts.

2. BREAK AND CORRECT

2.1 Context

In an AOM environment, the application model may need to be revised on the fly. Sometimes, however, changes in one `EntityType` definition will cause other `EntityTypes` to become invalid and consequently cause the overall model to become inconsistent. If inconsistent model changes are permitted to a running system, they will affect other parts of the architecture, and will impact the design of other parts of the system. For example, a forgiving GUI might raise run-time errors when it detects model inconsistencies while an overly restrictive GUI would have to be in lock step with changes to the model. Assuming the user may wish to make many changes to a model, we want to provide the AOM user, that is the person that defines new entity types and refines existing definitions, with IDE-like usability for making these changes. The user should be able to change parts of the model, be informed of any inconsistencies, and be able to incrementally resolve them.

2.2 Problem

How can we provide a robust process for adapting the application while preserving model consistency?

2.3 Forces

– The system should be designed to adapt at runtime to new user requirements.
– System downtime should be minimized.
– The domain model in the production system should always be consistent.
– AOM `EntityTypes` are interconnected with dependencies.
– New features developed in the AOM should be tested before they are deployed to production.

2.4 Solution

Define a development or testable repository isolated from the production repository, where the user is permitted to incrementally make and test changes to the AOM model. The user is permitted to work on a single `EntityType` at a time. The user is permitted to save changes to an `EntityType` only if there are no validation errors for this `EntityType`; e.g., it is consistent with other `EntityTypes` it references. Provide a Problems window to show validation errors for all `EntityTypes` in the development repository. Only after all validation errors are fixed can the development repository be deployed (manually or automatically) to the running system.

2.5 Dynamics

Figure 1 shows the classes involved in the AOM Editing process. The `AOMEditor` is used to edit the application model by changing `EntityType` instances. Upon saving (e.g., the user pressed the 'save' button) the `AOMEditor` calls the `validate` method on `EntityType` which delegates the call to the associated `Validator` [VAL]. If the `EntityType` is valid (i.e., no errors were reported), it can be saved to the `Repository`, otherwise, the user is informed and should fix the inconsistencies before changes can be saved.

Figure 2 illustrates an editing session for an `EntityType`. The user works on a working copy of the entity type (by calling `EntityType clone` method). An entity type definition can be updated by invoking the `setField` method. Upon saving, the `EntityType` definition is validated and only persisted to the Development repository if it passes all validations. Consequently, other repository elements (`EntityTypes` or instances) may become invalid as a result of changes made to the `EntityType` (e.g., when a new mandatory property is defined in an `EntityType`, instances of this `EntityType` become inconsistent as they don't have any value assigned to the new property). The AOM framework will detect those invalid elements, e.g., by an asynchronous job that is invoked periodically to validate the instances in the development repository and alert the user of those inconsistencies.

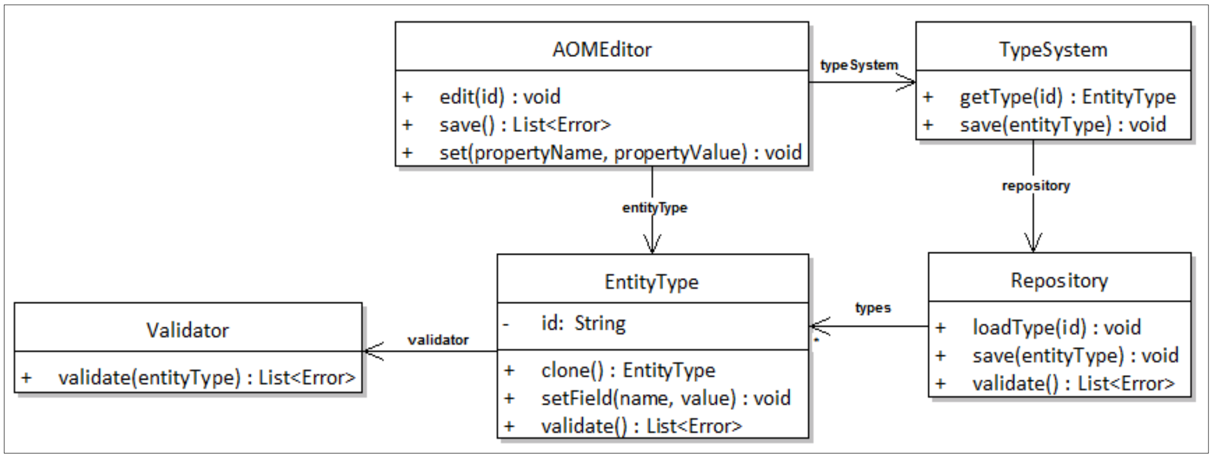

AOMEditor
+ edit(id) : void
+ save() : List<Error>
+ set(propertyName, propertyValue) : void
typeSystem
TypeSystem
+ getType(id) : EntityType
+ save(entityType) : void
repository
entityType
Repository
+ loadType(id) : void
+ save(entityType) : void
+ validate() : List<Error>
types
EntityType
- id: String
+ clone() : EntityType
+ setField(name, value) : void
+ validate() : List<Error>
validator
Validator
+ validate(entityType) : List<Error>

Figure 1. Main Classes in AOM Editing Session

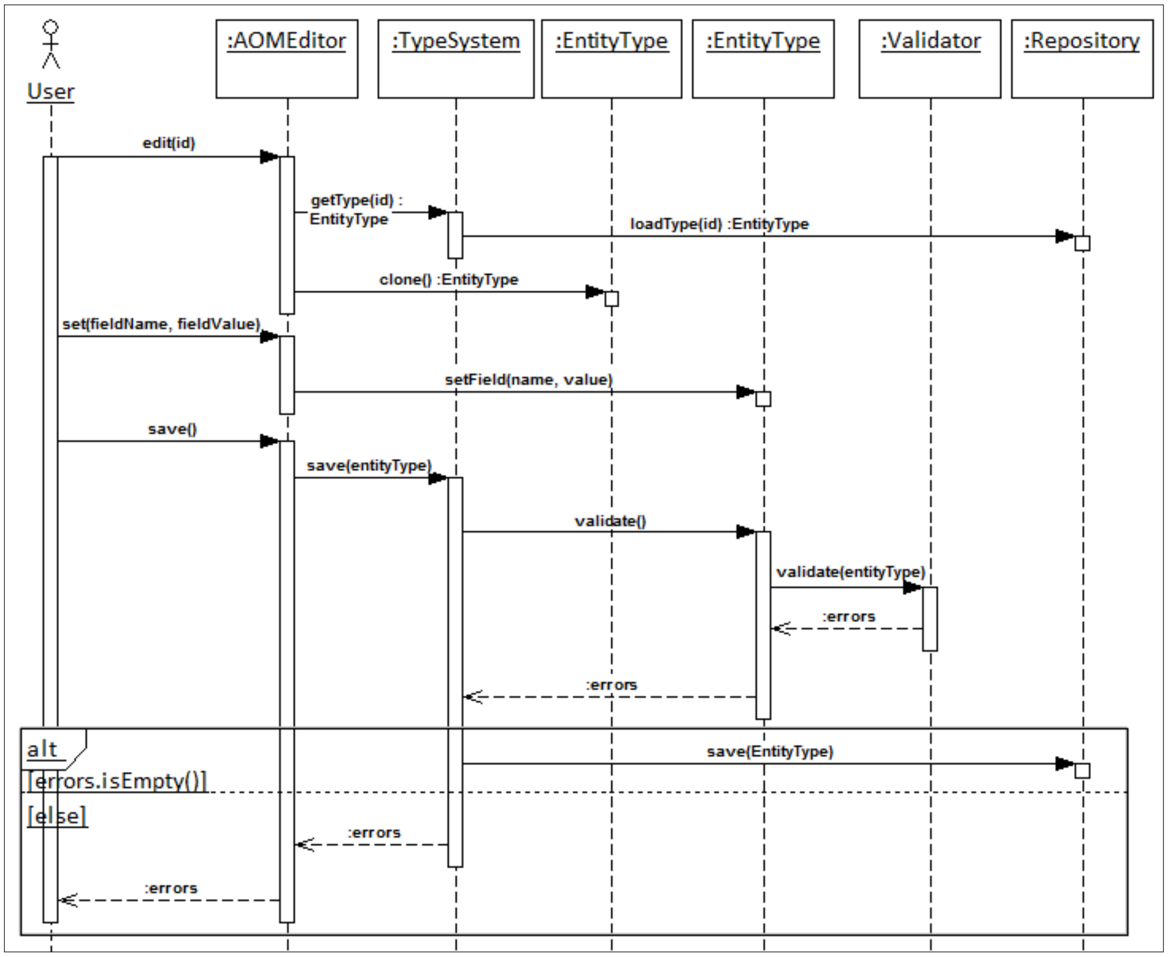


Figure 2. AOM Simple Evolution Scenario

2.6 Implementation

The metatype system performs validations on the *development repository*. This validation process can be executed upon saving the `EntityType` (using dependency analysis like incremental compilation in the IDE), or it can run as a background process, which periodically validates all `EntityTypes` and their instances in the *Development repository*. Only when there are no problems in the repository, is the user able to deploy the *development repository* to the production system which entails:

– Acquiring a system lock: finishing existing client requests in the production system while queuing new requests.
– Switching the production system over to use the new repository version.
– Clearing all caches in the production system.
– Releasing the system lock.
– Creating a new *development repository*; the baseline for the next version.

Like modern IDEs, the AOM framework allows the user to work in a "Break and Correct" mode where changes to some parts of the model break model consistency. An `EntityType` can be committed to the repository as long as it conforms to its own constraints (implemented by the associated `Validator`). Other `EntityTypes` that rely on the edited `EntityType` may become invalid as a result. Thus the user may have to make a series of changes to make the AOM consistent.

2.7 Example

In a telephony Business-Support-System where the user may define new types of events such as voice-call, movie-purchase, ringtone-download, etc., there is a need to define the mapping from external data sources to the new event type. E.g., an EventLoader defines that the `amount` property of Event is populated from an XML file using "/TopUp/@RechargeAmount" xpath expression.

Our AOM model should support defining new `EventTypes` and the associated `EventLoaderTypes`. In Figure 3, a new `TopUpEvent` was defined containing the property `amount`. The `TopUpLoader` is used to load external data to the newly defined event. An `EventLoaderType` contains a list of `FieldMappers` that define how external data should be mapped to the corresponding event. The AOM framework contains a `Validator` for `EventLoaderType`, which verifies that all mappers are valid and each event field has a valid mapper. An `EventLoader` holds the folders location for the incoming files (`inputFolder`) and for the processed files (`archiveFolder`). It also contains the file pattern (in a regular expression format) that identifies the files to be loaded. The framework uses this configuration to load the file, process it, and then to move it into to the archive folder.

Suppose a new requirement specifies that a `TopUpEvent` should also contain the reference to the top-up purchase channel (e.g., store, phone, Internet) in order to migrate subscribers to channels where the Telco pays less commissions. The user then edits the `TopUpEvent` and adds a new property, channel, of type `TopUpChannel`, and saves the `EventType`. The framework then validates the `EntityType` definition (`TopUpEvent`) and reports an error; "Duplicate field name 'channel'". This error is reported because the base `Event` already has a `channel` property indicating the inbound integration channel of the event (e.g., switch, billing-system, CRM, etc.). The user fixes the error by renaming the property to `topUpChannel` and saving the change. This time, the `EntityType` is valid, so it is persisted to the *Development repository*. A few seconds later, our user notes a "problems" indication in the GUI. So the user navigates to the "Problems view" and is informed that there is a problem in the `TopUpLoader`: "A field mapper should be defined for each TopUpEvent field. Missing mapper for field 'topUpChannel'". The user can now edit the `TopUpLoader` to fix that problem.

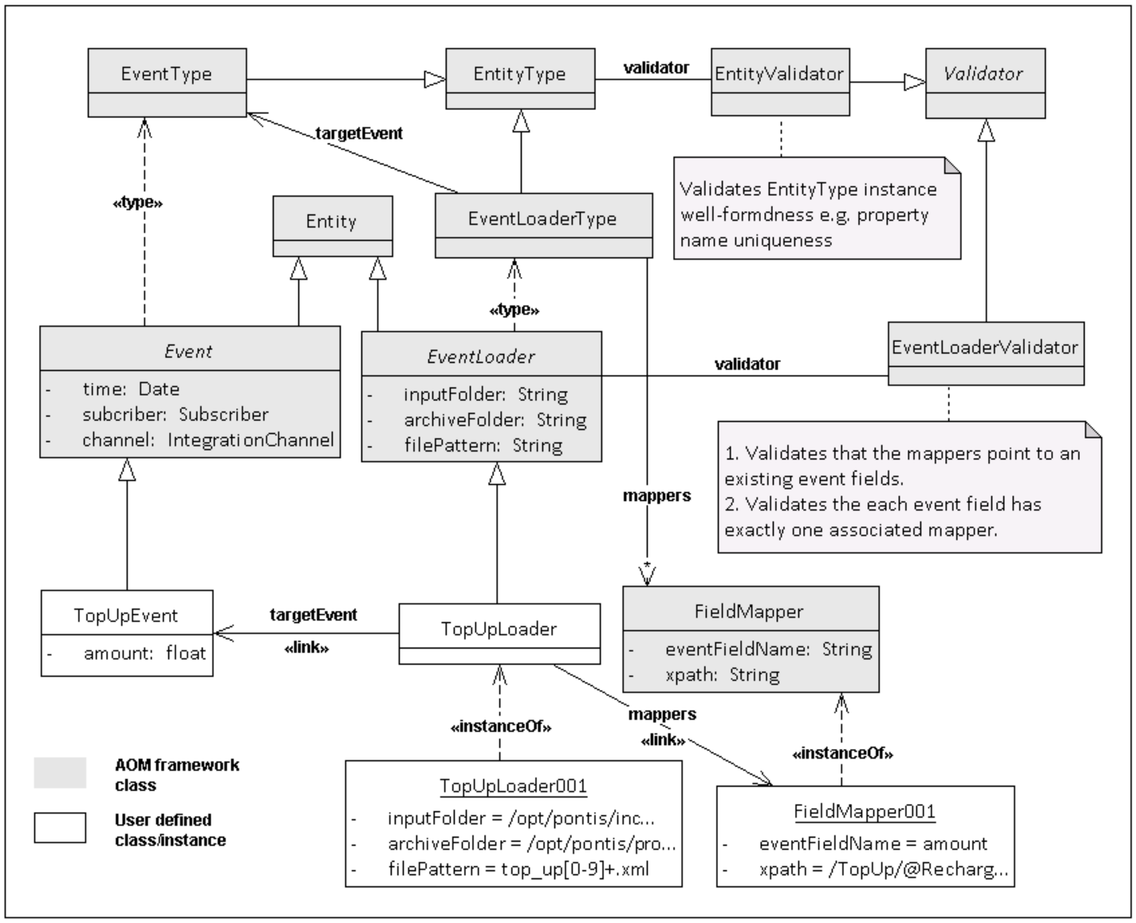


Figure 3. Class Diagram for TopUpEvent Example

2.8 Resulting Context

- The production system is consistent although there can be metadata that evolves just in time as mentioned in [HLN10].
- The user is provided with an IDE-like support for making and verifying changes to complicated AOM models.
- The *development repository* can be used by a different runtime machine to test the new model definition before it is deployed.
- Switching to the next version of the AOM repository minimized possible downtime.
- Switching to the next version of the AOM may take up to few seconds in which service is suspended. For systems that are very sensitive to response time it may result in losing some requests.
- Even minor ("safe") changes should go through the "heavy" process of making changes first in the *development repository*.
- Developing the supporting infrastructure to enhance usability and guarantee "safe" changes is costly.

2.9 Related Patterns

– While dealing with *Break and Correct*, *Evolution Resilient Scripts* can be used to detect inconsistencies in scripts as a result of model changes.
– *Break and Correct* is used to verify the consistency of the metadata upon system upgrade as in *Dynamic Model Evolution* pattern [HLN10].

2.10 Known Uses

The popular Eclipse IDE contains a Problems View that logs errors from various tools and allows the user to navigate to the relevant elements and fix them. For example, when working with the Eclipse JDT and changing a base class to have a new abstract method, errors are reported for all sub-classes of the changed class. Upon fixing each of the sub-classes the relevant error disappears from the Problems View.

Microsoft's Visual-Studio has a Problems View similar to Eclipse's.

Pontis Ltd. (www.pontis.com) is a provider of Online Marketing solutions for Communication Service Providers. Pontis' Marketing Delivery Platform allows for on-site customization and model evolution by non-programmers. The system is developed using ModelTalk [HLPS09] based on AOM patterns. Pontis' ModelTalk AOM GUI environment includes a background job that periodically validates the AOM repository (EntityTypes and instances) and alerts the user with a red marker when errors are found.

The AOM environment includes a "Problems View" page listing all the errors from which the user can navigate to the erroneous elements and fix them.

3. EVOLUTION RESILIENT SCRIPTS

3.1 Context

In an AOM, the model is often revised [YBJ01, YJ02]. As it evolves, new entity types and subtypes may be added or changed. These modifications to the AOM may also require changes in the behavior not anticipated by the AOM developer (e.g., a new formula for calculating the final price of a product). While the proper way to address such changes is to modify the AOM domain entities, this entails new software delivery and redeployment of the system.

Scripting code (e.g., JavaScript) can be dynamically executed (with no compilation) so it can be used as an interim solution. The AOM should have predefined hook points that allow the users to override system behavior by writing scripting code.

3.2 Problem

Evolution of the model can cause scripting code that relies on specific entity definitions to break. How can we detect inconsistencies between scripting code and entity definitions and ensure that the scripting code doesn't become overly dependent on AOM meta-data definitions?

3.3 Forces

- The system should be designed to adapt at runtime to new user requirements by adding new behaviors (or scripts).
- Scripting code typically interacts with entities, and relies on certain metadata to exist as defined in the `EntityType` at the time the script was written.
- As an AOM evolves, new `EntityTypes` and subtypes may be added or changed.
- As the AOM model becomes more refined and evolves, one would like to avoid modifying script code every time the model changes.
- Developing and maintaining a test suite that executes scripts to verify their correct usage of the AOM requires a significant effort and often leads to incomplete coverage.
- Parsing scripting code to validate correct use of AOM entities can be complex.

3.4 Solution

Provide a means of decoupling scripting code from the AOM. Instead of embedding direct references to AOM metadata, specify script hook-points in the AOM. A `ScriptHookPoint` (see figure 4) defines a mapping between AOM entities and script arguments and is responsible for script execution at runtime using a `ScriptEngine`. The mapping is done by specifying a list of model paths (relative to the `ScriptContext`) for each script argument in a `ScriptContract`. Every script argument has an equivalent `ScriptArgument` specification in the `ScriptContract` and eventually all script arguments are of simple types.

To support polymorphism (when the actual entity at runtime may be a subtype of the `EntityType` defined in the script context), a script contract should support defining several model-paths for each script argument. Figure 7 shows an example of such model when there are several types of Subscribers in a Telco business support system.

This solution permits adding or modifying argument definitions without modifying scripting code. The model-paths can be validated generically and alert the user of inconsistencies. At runtime, the `ScriptHookPoint` extracts the script arguments by evaluating the model-paths on the context entities.

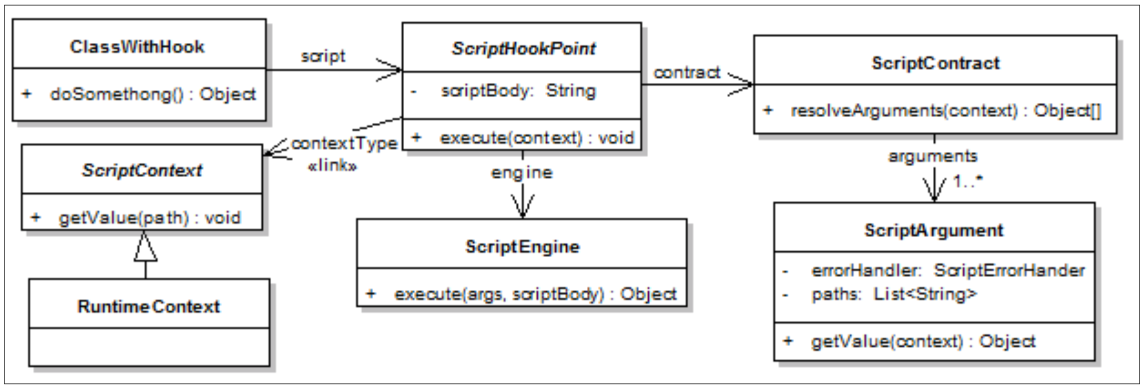


Figure 4. Class Diagram of Script Hook Point

3.5 Dynamics

Figure 5 illustrates the runtime execution of a script hook. When invoking the method `doSomething` on `ClassWithHook` object, a specific `RuntimeContext` is instantiated and populated with the required runtime objects. The newly created context is then used by the `ScriptContract` object to retrieve the script's arguments. This is done by invoking the `getValue` method on each of the `ScriptArguments` defined in the `ScriptContract`. Each `ScriptArgument` contains a list of possible paths to account for different subtypes of the AOM entity. This is needed when the argument is found in different properties in different subclasses of the AOM entity as described above. The `ScriptArgument` iterates over its paths list until a valid model-path is encountered and the actual value is then retrieved. If none of the paths are valid for the actual runtime context, the error handler is activated (not shown in the diagrams). After the arguments are resolved, the script is executed.

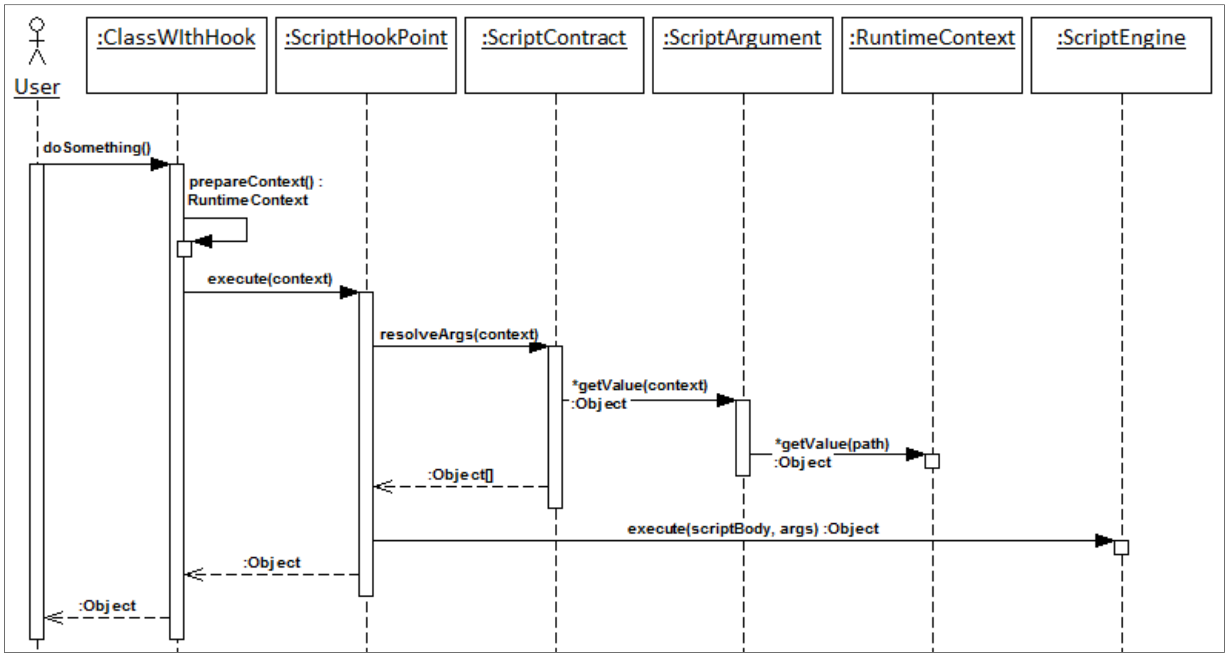


Figure 5. Sequence Diagram for Script Hook Point

3.6 Implementation

ScriptHookPointType (Figure 6) is an abstraction of a specific script hook point. It defines a context type RuntimeContextType, which specifies the types of entities that will be available for script execution. For example, DiscounterScriptHookPoint uses DiscounterRuntimeContext as its context. The DiscounterRuntimeContext defines Event, Request and Subscriber as the available entities at runtime.

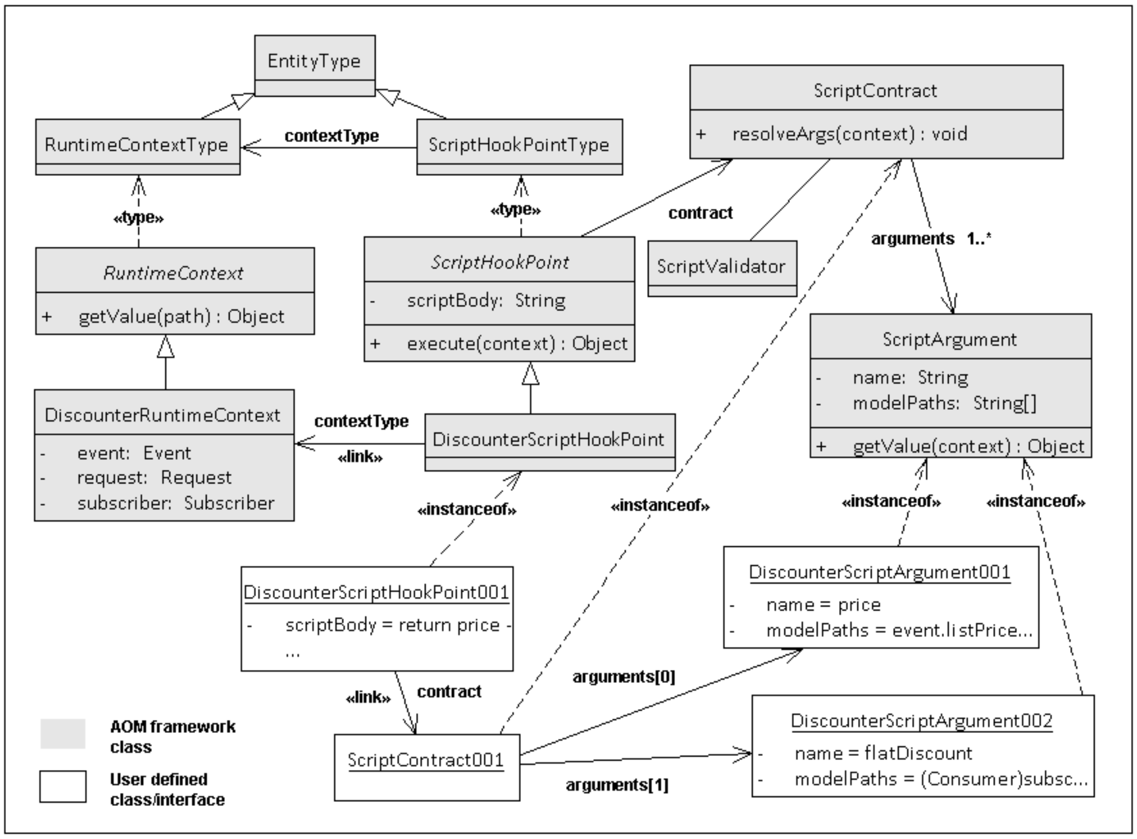


Figure 6. Class diagram of Script Hook Point

A concrete script type inherits ScriptHookPoint, which contains a definition of scriptBody (the script code itself) and ScriptContract. ScriptContract is responsible for supplying arguments to the script instance from the runtime context instance and consists of a list of ScriptArgument. Each of the ScriptArguments has a name (the script code references the argument by the name) and a list of model-paths. These model-paths provide mapping between the AOM entities from the associated script context (RuntimeContext) and the actual script argument. When the ScriptArgument cannot obtain a result from any of its model-paths it invokes its ScriptErrorHandler. ScriptErrorHandler allows the script creator to specify whether this situation will cause a runtime exception or whether default value shall be used. A ScriptValidator verifies that each ScriptArgument in the ScriptContract is valid by validating its model-paths against the specific RuntimeContextType. This way when changing EntityTypes that are being used in the RuntimeContext, a validation process can "catch" the erroneous ScriptArguments and alert the user (see "Break and Correct" pattern above).

The following Java code demonstrates how a hook point works with the Script and the contract to extract the necessary arguments. All classes shown here are parts of the AOM framework.

```
public class ClassWithHook{
      ScriptHookPoint script;

      public Object doSomething(){
            //do some business logic...
                RuntimeContext context = prepareContext();
                return (Float)script.execute(context);
      }
      …
}

public class ScriptHookPoint{
      ScriptEngine engine;
      ScriptContract contract;
      String scriptBody;

      public Object execute(RuntimeContext context) {
                Object[] args = contract.resolveArguments(context);
                return engine.execute(scriptBody, args);
      }
      …
}

public class ScriptContract{
      List<ScriptArgument> arguments;

      public Object[] resolveArguments(RuntimeContext context){
                 Object[] result = new Object[arguments.size()];
                 for(int i=0;i< arguments.size();i++){
                        result[i] = arguments.get(i).getValue(context);
                 }
                 return result;
      }
}

public class ScriptArgument{
      ScriptErrorHander errorHandler;
      List<String> paths;

      public Object getValue(RuntimeContext context) {
                 Object result = null;
                 for(String p : paths){
                           result = context.getValue(p);
                        if(result!=null){
                                                     break;
                                             }
                 }
                 if(result==null){
                           result = errorHandler.resolve(context);
                 }
                 return result;
      }
}
```

3.7 Example

A Telephony online marketing system runs campaigns targeted at different subscriber segments prompting subscriber to purchase new services and encouraging them to increase their consumption by means of benefits (discounts, loyalty-points, etc.). After running a few campaigns, the Telco's marketing department decided to forbid discounts on the full-charge of voice calls as some of the subscribers have flat-discount defined in their price-plan. Instead, the marketing system should calculate the price after flat-discount and apply its discount only on the reduced price. Luckily the system was developed using AOM, hence enabling an on-site solution using scripting.

Figure 7 shows the `VoiceCallEvent` after adding a `reducedPrice` calculated property. Note that `VoiceCallEvent` is of type `PriceableEventType` that has an attribute to define which of the entity's attributes contains the price for discount calculations (`priceAttribute`).

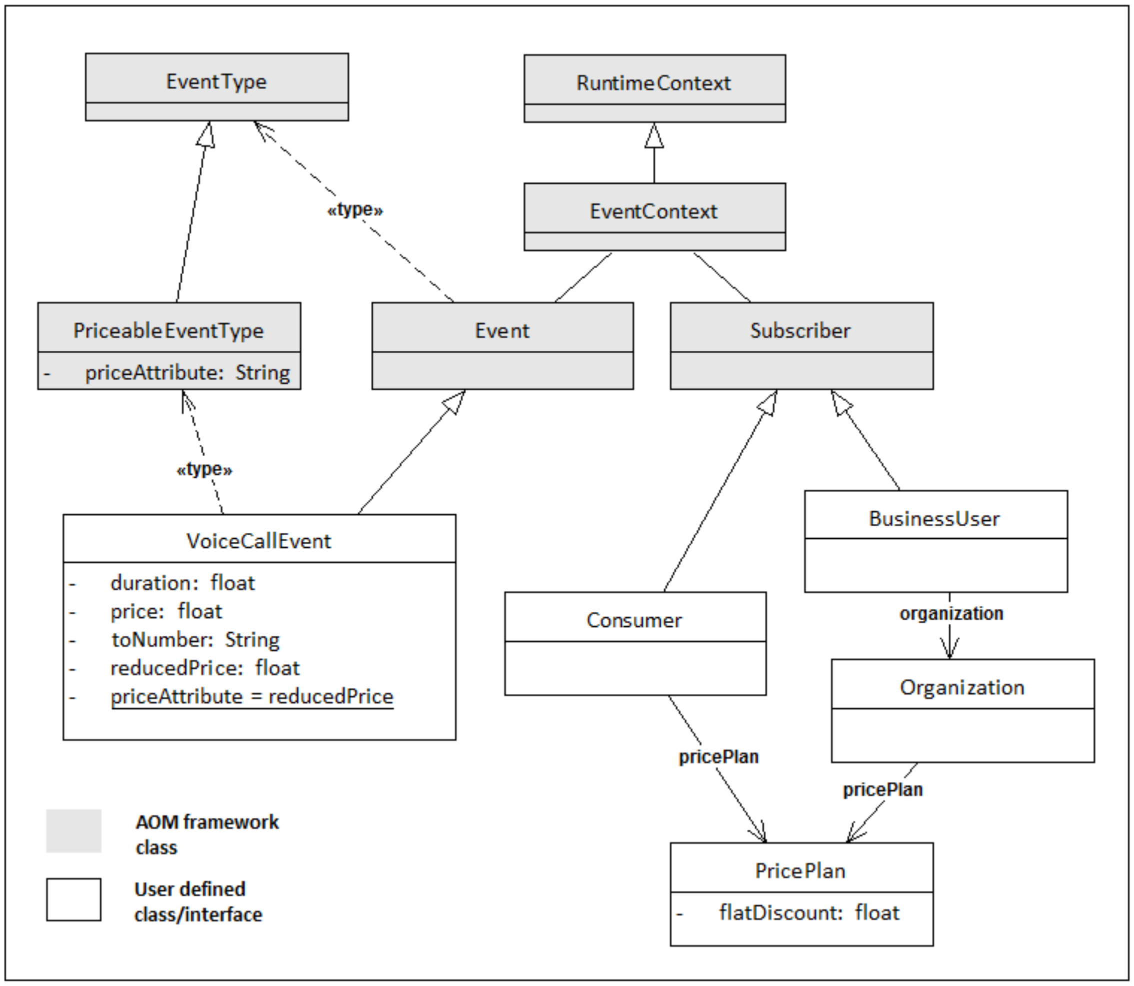


Figure 7. Class Diagram of VoiceCallEvent Example

A calculated property is a `PropertyType` that lets a script calculate the property value on the fly (see Figure 12). The property is defined in the AOM editor using JavaScript for the calculation. The JavaScript will fetch the flat-discount from the subscriber's price plan and will calculate the reduced price based on the flat-discount and the price from the `VoiceCallEvent`. The script author should consider that there are two kinds of AOM subscribers in the system `Consumer` and `Business` User. `Consumer` has a relation to `PricePlan` that contains the `flatDiscount` for the subscriber. `BusinessUser` has a relation to `Organization`, which has a relation to `PricePlan` (containing the `flatDiscount`).

Without using the *Evolution Resilient Scripts* pattern, script code (in JavaScript) for the `reducedPrice` calculation would look similar to the code below (error handling code emitted for brevity):

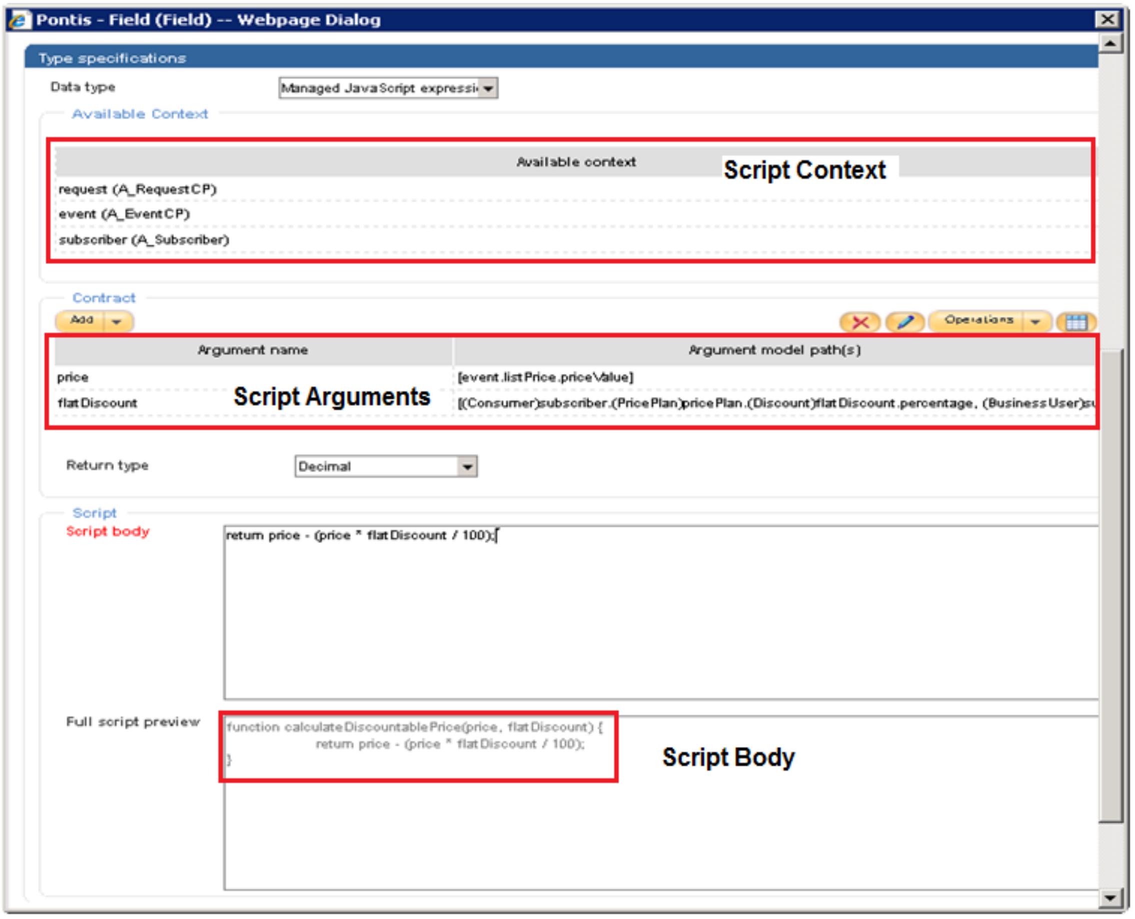
```
function calculate(request, event, subscriber){
  var result = 0;
  var price = 0;
  var flatDiscountPercentage = 0;
  var priceBeforDiscount = event.getPropertyValue("listPrice");
  if(priceBeforDiscount != null){
   priceBeforDiscount=
                 priceBeforDiscount.getPropertyValue("priceValue");
  }
  var pricePlan = null;
  if(subscriber.getType() == "BusinessUser"{
   var organization = subscriber.getPropertyValue("organization");
   if(organization != null){
     pricePlan = organization.getPropertyValue("pricePlan");
   }
  }
  else if(subscriber.getType() == "Consumer"){
   pricePlan = subscriber.getPropertyValue("pricePlan");
  }
  if(pricePlan != null){
   var flatDiscount = pricePlan.getPropertyValue("flatDiscount");
   if(flatDiscount != null){
     flatDiscountPercentage = flatDiscount.percentage;
   }
  }
  return price - (price * flatDiscountPercentage / 100);
}
```

As you can see, much of the scripting code is devoted to retrieving the source values for the calculation. Only the last line contains the business logic for the reduced price calculation. This code is fragile as it is highly dependent on AOM structure, names and attributes definitions. In contrast, when writing a script using a contract, the source values are extracted by the AOM framework and passed along to the script code. This makes the script code much simpler and clearly focused on newly required business behavior (calculating the price). In the following script we can observe that only one line remained from the original script:

```
function calculate(price, flatDiscountPercentage) {
      return price - (price * flatDiscountPercentage / 100);
}
```

Figure 8 is a screenshot of a script hook point taken from an Online Marketing system developed with ModelTalk [HLPS09]. This UI page is used by a script author to define a new script in the system or edit an existing script. The *Script Context* section shows the user the available runtime context over which explicit script parameters can be defined. The *Script Arguments* section is where the author defines the arguments for the script by means of model-paths relative to the script context.

The *Script Body* text-box is where the user enters the script code. The system displays the full script code (the generated method signature plus the user entered script body) in a read-only text-box. Figure 9 shows how a script argument is defined.



Data type: Managed JavaScript expressi

Available context
request (A_RequestCP)
event (A_EventCP)
subscriber (A_Subscriber)

Argument name	Argument model path(s)
price	[event.listPrice.priceValue]
flatDiscount	[(Consumer)subscriber.(PricePlan)pricePlan.(Discount)flatDiscount.percentage, (BusinessUser)su

Return type: Decimal

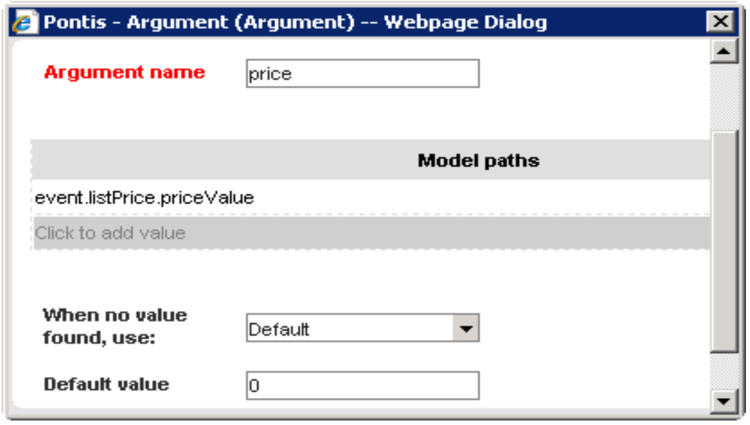
Script body:

```
return price - (price * flatDiscount / 100);
```

Full script preview:

```
function calculateDiscountablePrice(price, flatDiscount) {
        return price - (price * flatDiscount / 100);
}
```

Figure 8. Screenshot of Script Hook Point



Argument name: price

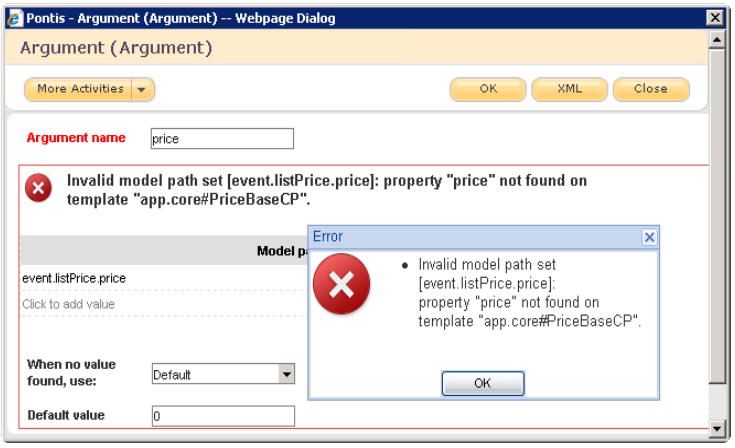
Model paths
event.listPrice.priceValue
Click to add value

When no value found, use: Default

Default value: 0

Figure 9. Screenshot of Simple Argument Definition

The argument definition consists of the argument's name, a list of model-paths to retrieve the argument's value, and the behavior in case the value was not found (return null, default value or raise an exception). The paths are validated upon save and during background validation. Figure 10 shows an error message resulting from such a validation error.



Argument name: price

Invalid model path set [event.listPrice.price]: property "price" not found on template "app.core#PriceBaseCP".

Model p

event.listPrice.price

Click to add value

When no value found, use: Default

Default value: 0

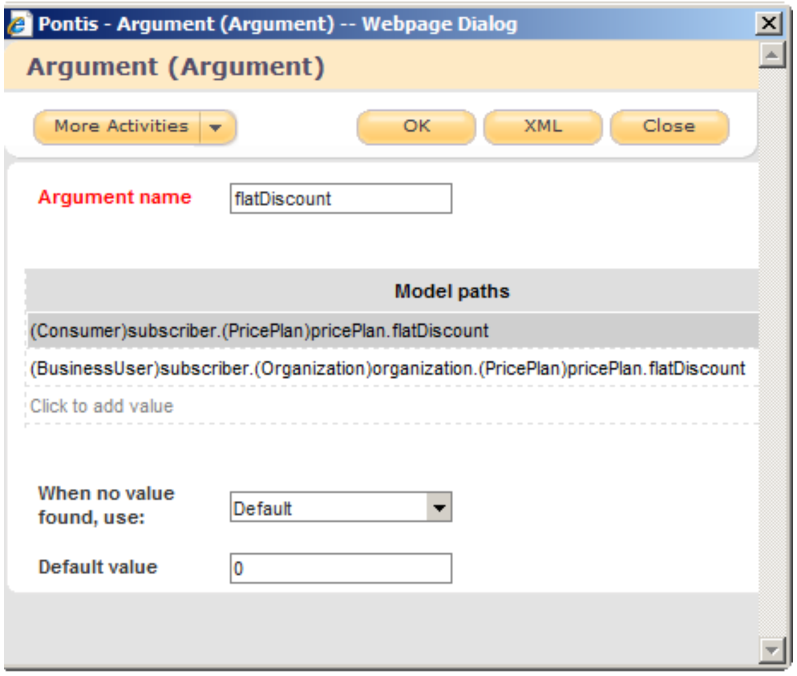
Error

- Invalid model path set [event.listPrice.price]: property "price" not found on template "app.core#PriceBaseCP".

OK

Figure 10. Screenshot of Invalid Argument Definition

Figure 11 contains two different paths to the flat discount percentage for two possible types of subscriber (See `Consumer` and `BusinessUser` in Figure 7).



Argument name: flatDiscount

Model paths

(Consumer)subscriber.(PricePlan)pricePlan.flatDiscount

(BusinessUser)subscriber.(Organization)organization.(PricePlan)pricePlan.flatDiscount

Click to add value

When no value found, use: Default

Default value: 0

Figure 11. Screenshot of Argument Definition Conditioned by EntityType

Figure 12 shows the implementation model of the calculated property feature, using the *Evolution Resilient Scripts* pattern.

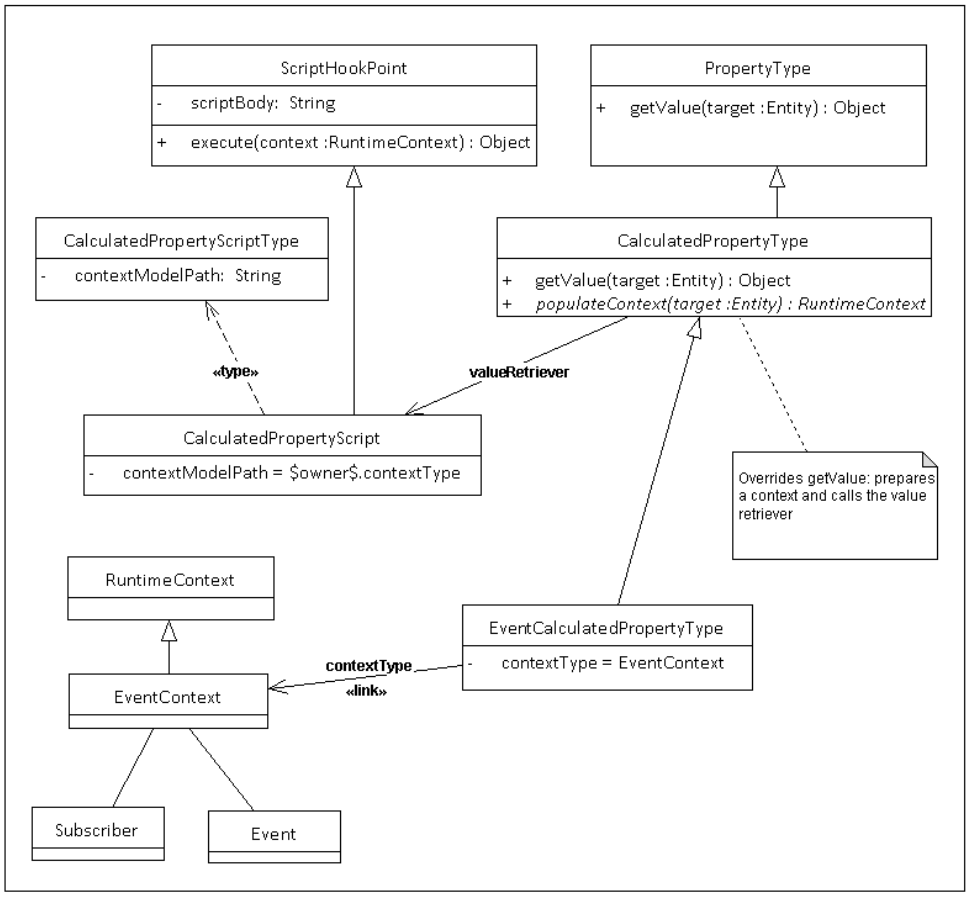

ScriptHookPoint
- scriptBody: String
+ execute(context :RuntimeContext) : Object
PropertyType
+ getValue(target :Entity) : Object
CalculatedPropertyScriptType
- contextModelPath: String
CalculatedPropertyType
+ getValue(target :Entity) : Object
+ populateContext(target :Entity) : RuntimeContext
«type»
valueRetriever
CalculatedPropertyScript
- contextModelPath = $owner$.contextType
Overrides getValue: prepares a context and calls the value retriever
RuntimeContext
EventCalculatedPropertyType
- contextType = EventContext
contextType
«link»
EventContext
Subscriber
Event

Figure 12. Class diagram of Calculated Property Hook Point

3.8 Resulting Context

- The scripting code uses primitive values and not model elements.
- Script code is shorter and focused on business logic.
- When a change in an entity type breaks the contract of a script, the errors are reported to the user.
- Developing the supporting infrastructure to support script hooks and script contracts is costly.
- The solution doesn't cover more complex scripts, e.g., when the script has to use system services.
- Evaluating the model-paths at runtime has performance penalties
- The user may be tempted to only "fix" the model in scripting code instead of making changes to the AOM model. For example, there can be some new attributes or behavior added through the scripts that would have been more appropriate to add to the core architecture of the AOM.
- The same fix might be applied to several scripts, resulting in redundant scripting code. For example you might have duplicated JavaScript for different pages doing similar validation. This problem can be addressed in some scripting languages that allow you to refactor and reuse the scripts across the pages.

3.9 Related Patterns

Break and Correct can be used during the validation process. *Dynamic Hooks* can be used as a means to implement *Evolution Resilient Scripts*. *Dynamic Hooks* is a pattern outlined by the authors and yet to be written. *Dynamic Hooks* allows you to have hooks in your code as places where your scripts can invoke new dynamic behavior.

3.10 Known Uses

Pontis Ltd. (www.pontis.com) is a provider of Online Marketing solutions for Communication Service Providers. Pontis' ModelTalk [HLPS09] AOM GUI environment includes several script hook points allowing users to add imperative JavaScript code to `EntityType` definitions. The user may choose between writing a script that directly access the AOM model or a *Evolution Resilient Script* that accesses the model via contracts.

4. SUMMARY

This paper has discussed two patterns that help minimize the impacts of evolving Adaptive Object-Model definitions. The first pattern, *Break and Correct*, describes one way to guarantee consistent changes to an Adaptive Object-Model in an environment where the AOM must run continually (24x7). The second pattern, *Evolution Resilient Scripts*, enables the refinement of AOM type definitions without having to change scripting code.

There are many other patterns that deal with changing systems and many specific to AOMs are being catalogued and described [AOM].

5. ACKNOWLEDGEMENTS

We would like to thank our shepherd Vander Alves for his valuable comments on these patterns during the SugarLoaf PLoP 2010 Shepherding process and our writer's workshop group members, Eduardo Fernandez and Fabio Kon, for their valuable comments during SugarLoaf PLoP. We also thank Pontis Ltd. (www.pontis.com) for granting us access to their development process.

REFERENCES

[AOM] Adaptive Object-Models. http://www.adaptiveobjectmodel.com.

[FY98] Foote, B. and Yoder, J. W. (1998) "Metadata and Active Object-Models", Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998.

[GoF95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) "Design Patterns: Elements of Reusable Object Oriented Software", Addison-Wesley. 1995.

[HLN10] Hen-Tov, A, Lorenz, D. H., Nikolaev, L., Schachter, L., Wirfs-Brock, R., and Yoder, J. W. (2010) "Dynamic Model Evolution". In Proceedings of the 17th Conference on Pattern Languages of Programs (Reno/Tahoe, Nevada, October 17 - 21, 2010). SPLASH/OOPSLA 2010. ACM, New York, NY. To appear.

[HLPS09] Hen-Tov, A, Lorenz, D. H., Pinhasi, A. and Schachter, L. (2009) "ModelTalk: When Everything Is a Domain-Specific Language", IEEE Software, vol. 26, no. 4, pp. 39-46, July/Aug. 2009.

[JW98] Johnson, R. and Wolf, R. (1998) "Type Object. Pattern Languages of Program Design 3", Addison-Wesley, 1998.

[RFBO01] Riehle, D., Fraleigh, S., Bucka-Lassen, D. and Omorogbe, N. (2001) "The Architecture of a UML Virtual Machine:, Proceedings of the 2001 Conference on Object-Oriented Program Systems, Languages and Applications (OOPSLA 2001), October 2001.

[RTJ05] Riehle, D., Tilman, M., and Johnson, R. (2005) "Dynamic Object Model", In Pattern Languages of Program Design 5. Edited by Dragos Manolescu, Markus Völter, James Noble. Reading, MA: Addison-Wesley, 2005.

[RY01] Revault, N. and Yoder, J. W. (2001) "Adaptive Object-Models and Metamodeling Techniques Workshop Results", Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001). Budapest, Hungary, 2001.

[VAL] Validator reference implementation.
http://static.springsource.org/spring/docs/3.0.x/reference/validation.html.

[WYWJ07] Welicki, L., Yoder, J.W., Wirfs-Brock, R. and Johnson, R. (2007) "Towards a Pattern Language for Adaptive Object-Models", Companion of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2007), Montreal, Canada, 2007.

[WYW07] Welicki, L., Yoder, J. W. and Wirfs-Brock, R. (2007) "Rendering Patterns for Adaptive Object-Models", 14th Pattern Language of Programs Conference (PLoP 2007), Monticello, Illinois, USA, 2007.

[WYW08] Welicki, L., Yoder, J. W. and Wirfs-Brock, R. (2008) "The Dynamic Factory Pattern", 15th Pattern Language of Programs Conference (PLoP 2008), Nashville, Tennessee, USA, 2008.

[WYW09] Welicki, L., Yoder, J. W. and Wirfs-Brock, R. (2009) "Adaptive Object-Model Builder" 16th Pattern Language of Programs Conference (PLoP 2009), Chicago, Illinois, USA, 2009.

[YBJ01] Yoder, J. W., Balaguer, F. and Johnson, R. (2001) "Architecture and Design of Adaptive Object-Models", Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001), Tampa, Florida, USA, 2001.

[YJ02] Yoder, J. W. and Johnson, R. (2002) "The Adaptive Object-Model Architectural Style", IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (WICSA 2002), Montréal, Québec, Canada, 2002.

Appendix to:
Adaptive Object-Model Evolution Patterns

IMPORTANT NOTICE: THIS SECTION IS A SUMMARY EXTRACTED FROM (YODER AND JOHNSON 2002, YODER ET AL. 2001) AND HAS BEEN INCLUDED TO HELP READERS UNFAMILIAR WITH THE AOM ARCHITECTURAL STYLE TO GET A MORE COMPLETE VIEW. WE STRONGLY RECOMMEND THE READER READS THE ORIGINAL PAPERS FOUND AT WWW.ADAPTIVEOBJECTMODEL.COM.

A. Summary of the Architectural Style of Adaptive Object-Models

A.1. Design

The design of Adaptive Object-Models (AOMs) differs from most object-oriented designs. Normally, object-oriented designs have classes that model the different types of business entities and associate attributes and methods with them. The classes model the business, so a change in the business causes a change to the code, which leads to a new version of the application. An AOM does not model these business entities as classes. Rather, they are modeled by descriptions (metadata) that are interpreted at runtime. Thus, whenever a business change is needed, these descriptions are changed, and can be immediately reflected in a running application.

AOM architectures are usually made up of several smaller patterns. TYPE OBJECT (Johnson and Wolf 1998) provides a way to dynamically define new business entities for the system. TYPE OBJECT is used to separate an ENTITY from an ENTITYTYPE. Entities have attributes, which are implemented using the PROPERTY pattern (Foote and Yoder 1998).

A.2. Type Object

In most AOMs, TYPE OBJECT is used twice: once before using the PROPERTY pattern, and once after it. TYPE OBJECT divides the system into entities and entity types. Entities have attributes that can be defined using properties. Each PROPERTY has a type, called PROPERTYTYPE, and each ENTITYTYPE can then specify the types of the properties for its entities (Fig 13).

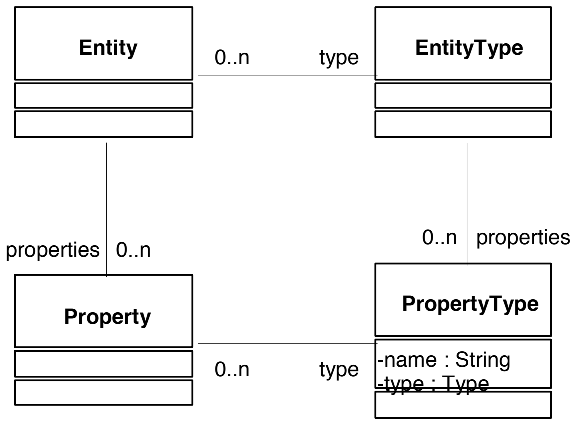


Fig.13. Type Square

A.3. Results

Type Square often keeps track of the name of the property and whether the value of the property is a number, a date, a string, etc. Sometimes objects differ only in having different properties. Fig. 13 represents the resulting architecture after applying these two patterns, which we call Type Square (Yoder et al. 2001).

As is common in Entity-Relationship (ER) modeling, an AOM usually separates attributes from relationships. In these cases the Type Object pattern is applied again to define the legal relationships between types of entities.

A.4. Strategy

The Strategy pattern (Gamma et al. 1995) can be used to define the behavior of entity types. These strategies can evolve, if needed into a rule-based language that gets interpreted at runtime. Finally, there is usually an interface for non-programmers, which allows them to define the new types of objects, attributes and behaviors needed for the specified domain. Therefore, we can say that the core patterns that may help to describe the AOM architectural style are: Type Object, Property, Entity-Relationship, Accountability, Strategy, Rule Object. AOMs are usually built from applying one or more of these patterns in conjunction with other design patterns such as Composite, Interpreter, and Builder (Gamma et al. 1995) (Fig 14).

A.5. Composite

Composite is used for building dynamic tree structure types or rules. For example, if the entities need to be composed in a dynamic tree like structure, the Composite pattern is applied. Builders and interpreters are commonly used for building the structures from the meta-model or interpreting the results.

A.6. AOM Core Architecture

But, these are just patterns; they are not a framework for building AOMs. Every AOM is a framework of a sort but there is currently no generic framework for building them. A generic framework for building the type objects, properties, and their respective relationships could probably be built, but these are fairly easy to define and the hard work is generally associated with rules described by the business language. These are usually very domain-specific and varied from application to application.

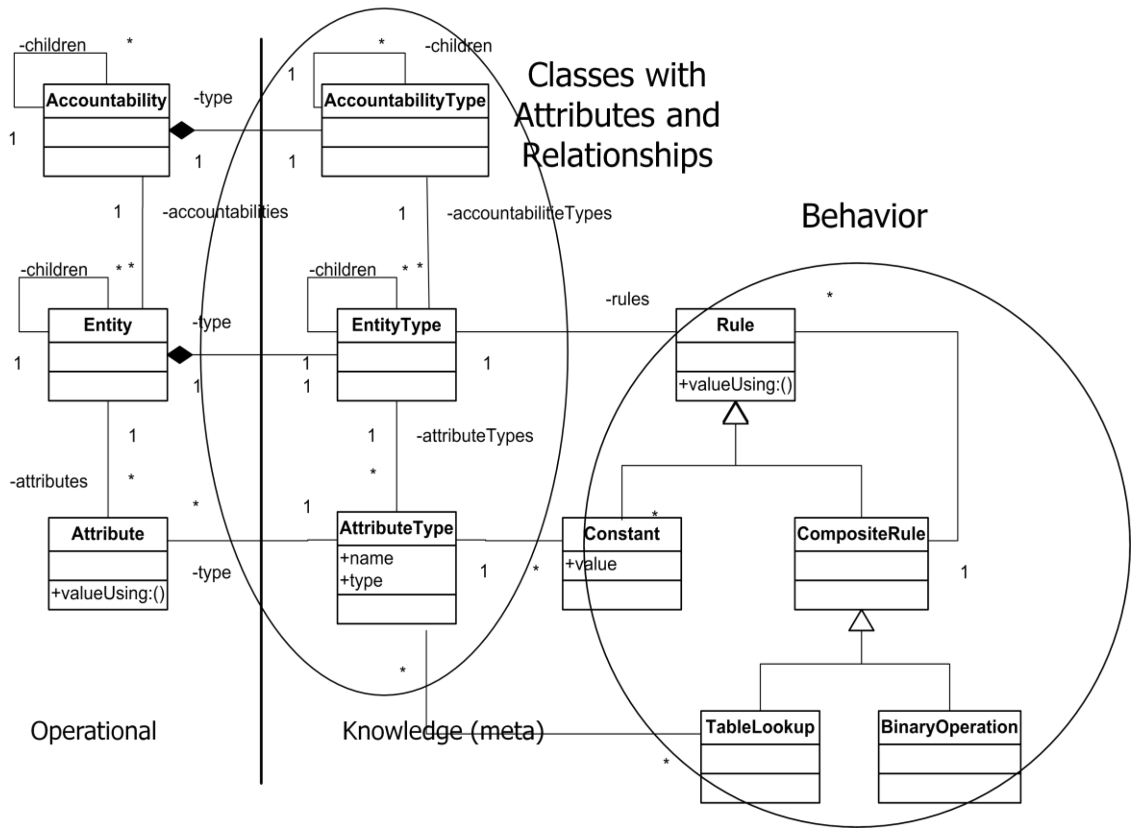


Fig.14. Core AOM Architecture.