Dynamic Model Evolution

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

In the Adaptive Object-Model (AOM) architectural style, user-defined domain entities and their relationships, attributes and behavior are described by externally stored metadata. AOM implementations reify the domain entities at runtime by interpreting the metadata. Users change the object model by editing the metadata to reflect changes in the domain. This paper describes a pattern for incorporating model changes into an AOM production system without taking the system off-line. Dynamic evolution of the object model is done during loading of a domain object. The Dynamic Model Evolution pattern is applicable when a system needs to keep running, and the alternative of updating the entire object model at the time a new version comes online is too time consuming to be a practical option.

Categories and Subject Descriptors: **D.1.5 [Programming Techniques]:** Object-oriented Programming; **D.2.2 [Design Tools and Techniques]:** Object-oriented design methods; **D.2.11 [Software Architectures]:** Patterns—*Adaptive Object Model*

General Terms: Design

Additional Key Words and Phrases: Factory Objects, Adaptive Object Model (AOM), Creational Patterns, Lazy Loading, ModelTalk

ACM Reference Format:

HEN-TOV, A., LORENZ, D. H., NIKOLAEV, L., SCHACHTER, L., WIRFS-BROCK, R., AND YODER, J. W. Dynamic model evolution. In *Proceedings of the 17th ACM Conference on Pattern Languages of Programs (PLoP 2010)* (SPLASH, Reno/Tahoe, Nevada, USA, October 16-18 2010).

1. INTRODUCTION

An Adaptive Object-Model (AOM) (Yoder and Johnson 2001) system represents user-defined domain entities, their relationships, attributes and behavior as metadata (Foote and Yoder 1998, Yoder et al. 2001, Yoder and Razavi 2000). An AOM system relies on an object model, which is constructed at run time by interpreting externally stored definitions (metadata). Users change the object model (or the metadata) to reflect changes in the domain.

Refining the definition of an AOM entity may require other dependent AOM entity definitions to change as well. To preserve model consistency under such evolution, the required series of changes to domain entities need to appear to be applied as a single modification. One way to support atomicity when making a change to a number of different domain entities is to take the system off-line, evolve the metadata and all the affected domain objects, and then perform a clean restart of the system. However, when a system must keep running, or when the system cannot be taken offline for the long period of time required for performing system-wide changes, then *dynamic model evolution* is a viable option: the incremental evolution of the metadata and the domain objects can be done when objects are loaded into memory.

1.1 Contribution

The pattern presented in this paper handles dynamic evolution of the domain model of an AOM. Aspects of this pattern can be applied to dynamically evolve metadata and object instances to co-exist with new versions of

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; D. H. Lorenz, Open University of Israel, Raanana 43107, Israel; email: lorenz@openu.ac.il; 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.

AOM infrastructure code. This pattern may also be relevant to those who need to evolve other types of meta-architectures.

1.2 Background

To understand the details of this pattern, the reader is expected to have some background knowledge of AOM systems. AOM architectures are usually implemented by applying several patterns to represent a domain model and its behavior. The TYPE OBJECT (Johnson and Wolf 1998) pattern is used to dynamically define new business entities for the system. Entities have attributes, which are implemented with the PROPERTY (Foote and Yoder 1998) pattern. The TYPE OBJECT pattern is used a second time to define the legal types of attributes, called ATTRIBUTETYPE. Thus, to represent an entity, TYPE SQUARE (Yoder et al. 2001) is used, i.e., a combination of TYPE OBJECT and PROPERTY patterns. Additionally, an AOM might use these patterns: ENTITYRELATIONSHIP (ACCOUNTABILITY), (Welicki et al. 2009), RULEOBJECT (Arsanjani 2000), HISTORY OF OPERATIONS (Ferreira et al. 2008), SYSTEM MEMENTO (Ferreira et al.2008), and MIGRATION (Ferreira et al. 2008). For an overview of the core patterns of this architectural style, see Appendix A.

2. CONTEXT

You are upgrading a 24x7 production system that is implemented in an AOM architecture style. Implementing new functionality requires modifications to definitions of existing domain entities. Your production system has considerable pre-existing metadata and domain objects, and you don't want to halt the running production system to perform an off-line upgrade.

3. PROBLEM

How can one dynamically evolve at runtime the metadata definitions and the entities of a running AOM?

4. FORCES

There are four main forces:

- *High Availability:* The AOM system must keep running 24x7. A long period of downtime for performing model upgrades is unacceptable.
- *Legacy Metadata:* There are large amounts of stored instances, excluding the possibility for off-line data migration within a short maintenance window.
- *Type Refactoring:* The changes include redefinitions of attribute types, subtypes and super-types. (Hence, an upgrade support that is unable to handle such redefinitions will be of limited utility).
- *Multiple Custom AOM Versions:* There are several customers using the product and each customer may modify its own AOM entity definitions independently. (Hence, the architecture needs to be able to handle different versions of AOM definitions in a way that lets the customer team resolve conflicts autonomously without extensive reworking of any local customizations already made to domain entity definitions).

5. SOLUTION

The overall process requires a *staging system* as a mediator between the development and the production systems (Fig. 1). The new version and the customized domain model from the production system are both installed in the staging system.

In the development system, the AOM core team prepares a new version of the core software product ("*develop next version*," Fig. 1). For each change in the domain model that is not backward compatible, the core team prepares an upgrader that will transform instances of that domain entity to the new model as part of the de-serialization of existing AOM objects ("*prepare upgraders*," Fig. 1). The new version along with the set of upgraders is then deployed into the staging system ("*install*," Fig. 1).

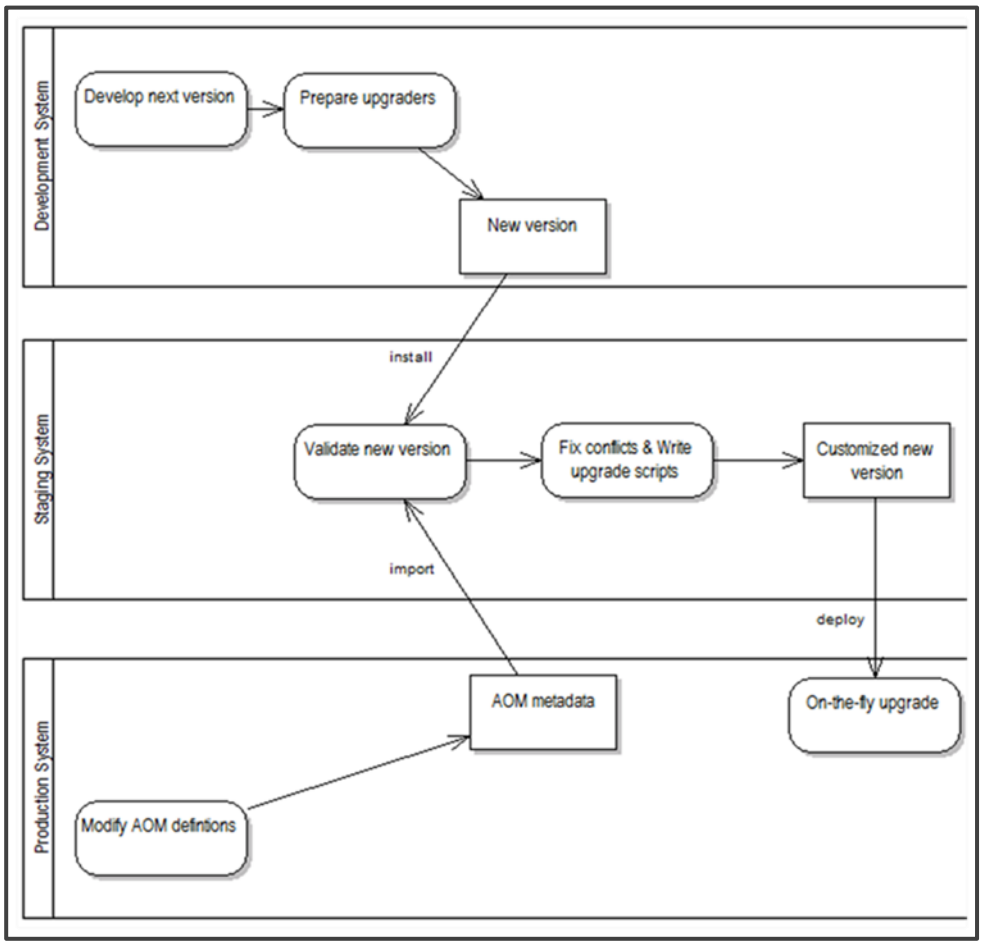

Fig. 1. Activity diagram for the AOM evolution process.

The customized domain model from the production system is also deployed onto the staging system ("*import*," Fig. 1). The combined model then undergoes validation to detect inconsistencies ("*validate new version*," Fig. 1). These inconsistencies are reported in a *problems view*, e.g., using the BREAK AND CORRECT pattern (Hen-tov et al. 2010), to be resolved by an *AOM engineer* (an end-user domain expert).

For example, a conflict will be detected when a new core property in a revised entity type happens to have the same name as that of a property already defined in a custom AOM type. To resolve this conflict, the AOM engineer will need to rename the dynamically defined property. After the property is renamed, the AOM engineer has to deal with backward compatibility. For example, in order to correctly handle de-serialization of old instances, a value needs to be associated with the renamed property. This is accomplished by either adding new upgraders or extending the behavior of existing ones ("*fix conflicts and write upgrade scripts*," Fig. 1).

The fix can be either written in a scripting language or specified declaratively using a library of generic configurable upgraders (e.g., *property rename* upgrader). Once all conflicts are resolved and the custom upgrade scripts are written, the customer can test the new version along with the entire set of upgraders. During production, the upgrade framework is invoked as a first step in the AOM BUILDER (Welicki et al. 2009) and performs just-in-time instance upgrade ("o*n-the-fly upgrade*," Fig. 1).

The upgrade framework can be further integrated with specific technologies that support dynamic evolution. In Java, for example, a new release of a core product system typically includes a binary distribution in the form of executable jars. To avoid restarting the application server, there are various Java techniques that do not require any downtime when installing the new binary version. We briefly list here two

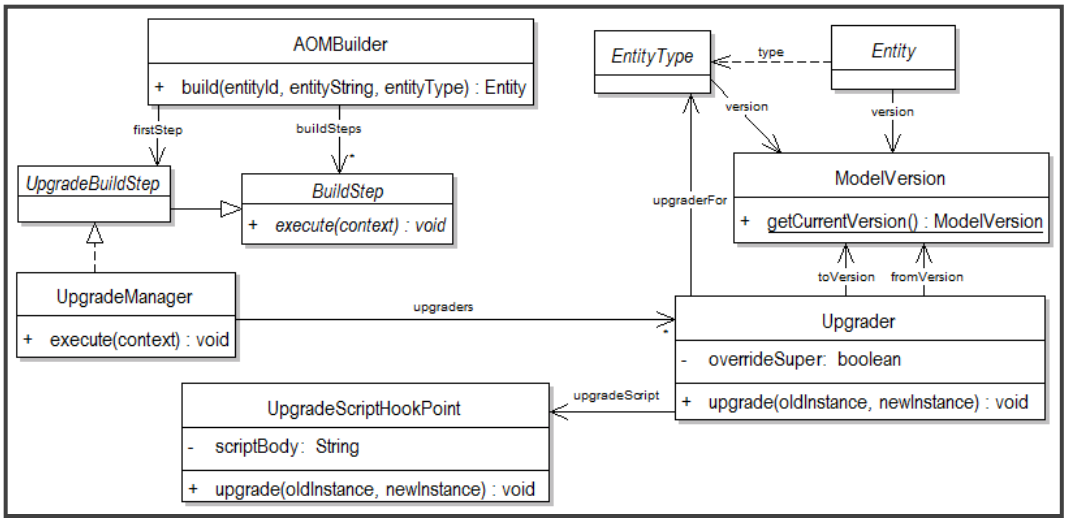

Fig. 2. AOM Builder with upgrade support (class diagram).

such technologies:

– *Dynamic class loading* is part of the Java Core Reflection API. However, this technique is not commonly used, because it requires supporting infrastructure for error handling and robustness, such as handling existing instances, error recovery, etc.

– *OSGi* is a set of specifications that define a dynamic module system for Java. It inherently supports versioning and evolution of Java modules (jars). There are several implementations of the OSGi specifications, such as Apache Felix (felix.apache.org) and Eclipse Equinox (www.eclipse.org/equinox).

6. DYNAMICS

When an entity is loaded into a running AOM system, a version migration commences. Fig. 2 depicts the main classes involved in the just-in-time evolution of an entity. Fig. 3 is the corresponding sequence diagram.

An `AOMBuilder` (Welicki et al. 2009) is responsible for de-serializing entities retrieved from a repository (e.g., a relational database). It invokes a collection of *`BuildSteps`* in order to gradually build the entity. The first step is the *`UpgradeBuildStep`* (an interface implemented by `UpgradeManager`). If the data associated with an *`EntityType`* has been defined in an earlier version of the AOM, an appropriate `Upgrader` is invoked. The build process may involve the migration of data from several prior versions. Therefore, a series of version upgrades may be applied sequentially.

An `Upgrader` is responsible for migrating data from one version to the next. For this to work, each *`Entity`* object needs a version property. A global variable holds the currently installed version. When upgrading the system, a collection of `ModelVersion` entities and the associated upgrade scripts are available to the `UpgradeManager`. The `UpgradeManager` determines which upgraders are necessary according to the *`EntityType`* and the `fromVersion` and `toVersion` that each upgrader handles. Each *`EntityType`* may have several upgraders, some of which were supplied with the new core product version and some that were added by the AOM engineer.

The `UpgradeManager` is also responsible for providing getters and setters (using the PROXY pattern) for manipulating raw data format of entities. This lets the script programmer concentrate on the transformation from the old version to the new one, rather than dealing directly with the raw data format of an *`Entity`*.

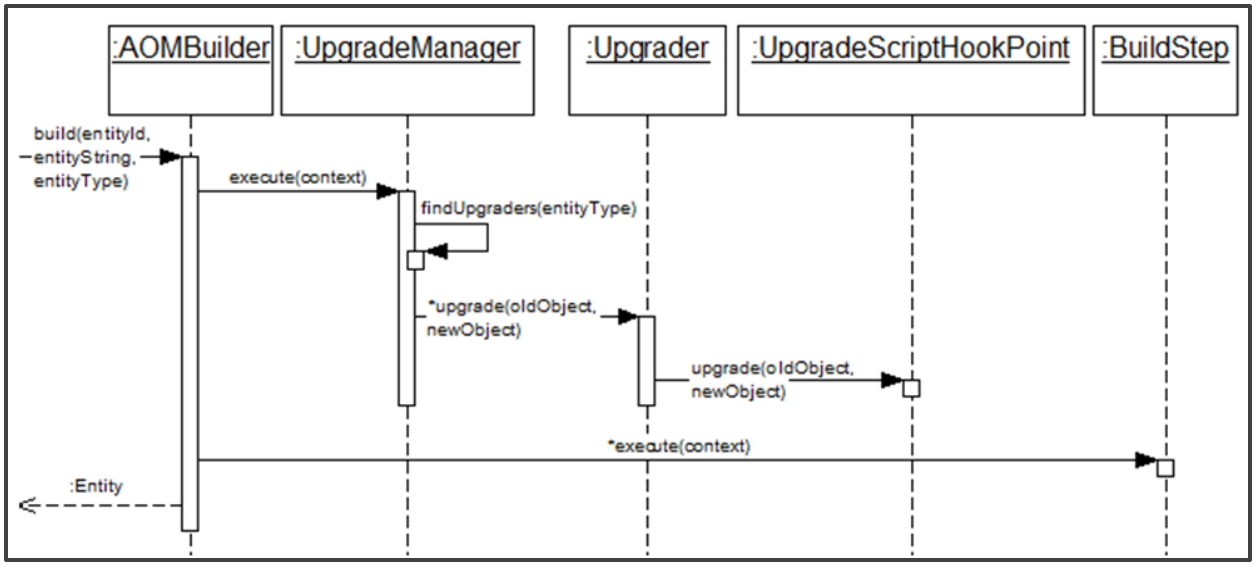

Fig.3. A Builder upgrade step (sequence diagram).

7. EXAMPLE

As an example, consider an AOM implementation of an online marketing system designed for running Telephony campaigns targeted at different subscriber segments (Hen-tov et al. 2009). Fig. 4 displays some of the classes in this AOM with their inheritance relationships, and several instantiated entities.

The core software product in this example comprises general-purpose AOM core classes (`Entity` and `EntityType`), and domain-specific AOM classes (`Benefit`, `BenefitType`, `Event`, and `EventType`). A `Benefit` describes specific campaigns. An `Event` describes the trigger for activating a `Benefit`. Each `BenefitType` is associated with an `EventType` to define which type of `Event` can trigger that `Benefit` type and grant the benefit. `BenefitType` and `EventType` are part of the AOM TYPE OBJECT pattern.

These domain specific classes can be specialized for a particular customer by creating new user defined metadata classes. In our example, the customer defined two new `Benefit` kinds: `FreeSms` and `FreeAstrologicalPrediction`; and three new `Event` kinds: `SmsEvent`, `TopupEvent`, and `VoiceCallEvent`. The `FreeSms` benefit is activated by an `SmsEvent` and grants the user with free units of SMS sends. The number of units is defined in the `Benefit` class. `FreeAstrologicalPrediction` is activated by a `TopUpEvent` and grants the user with free units of daily predictions. For simplicity, we omit details such as the conditioning for granting benefits, e.g., "send 200 text messages this week and get 100 more sends for free."

In addition, the customer took advantage of the AOM architecture and added a `CAP` property in `FreeAstrologicalPrediction`. This property was then used by the customer to set a special cap on the number of free units. Setting a cap limits the overall cost of the benefit to the service provider and thus helps in keeping within the marketing budget for the campaign.

At runtime, instantiations of specific benefits represent specific products. For example, `100FreeSms` sets the benefit to 100 units of free SMS's; `WeekFreeAstrologicalPrediction` sets the benefit to 7 days of free predictions with a maximum cap of 1904 days total for all users collectively to keep within a $10K budget.

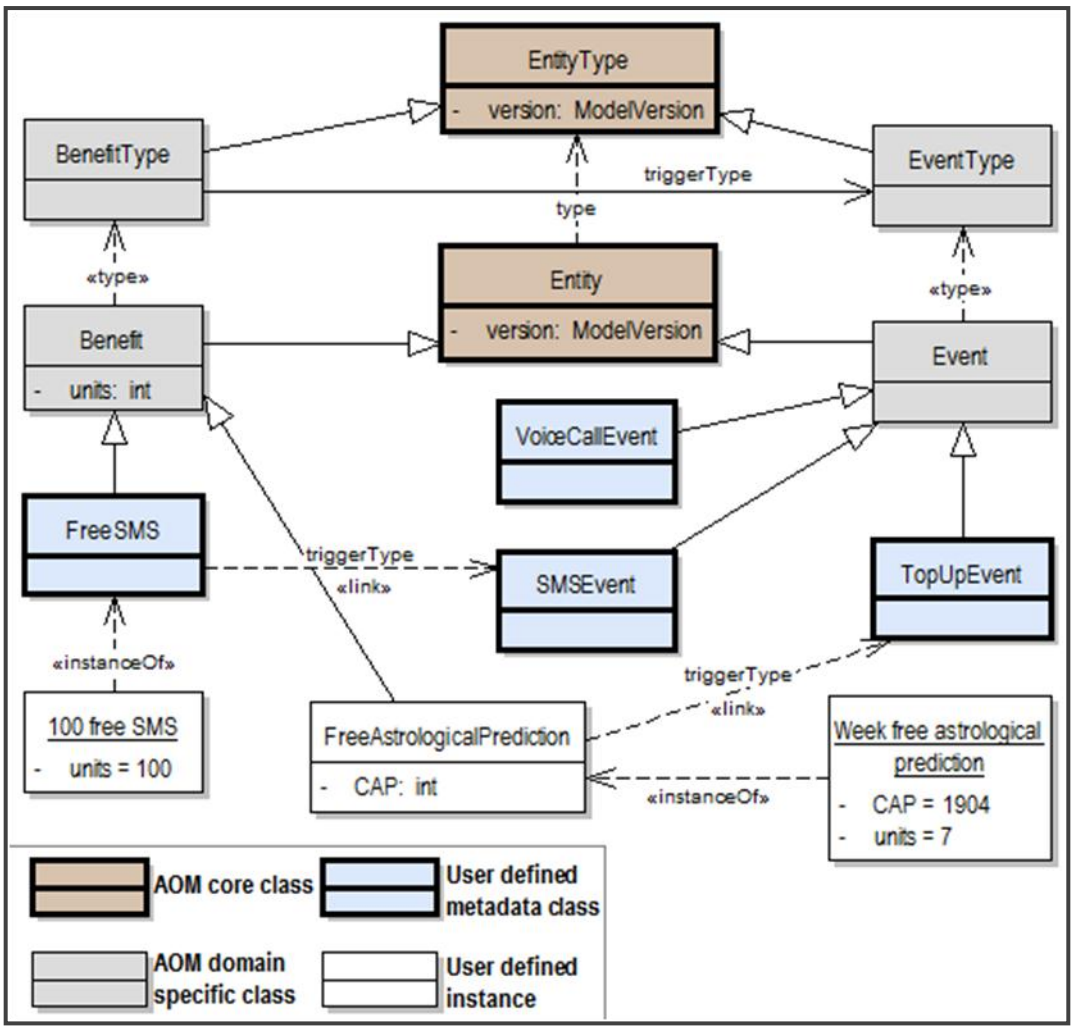

Fig.4. The original AOM model and instances.

8. UPGRADE SCENARIO

In our example scenario, the next version of the AOM system incorporates two improvements (Fig. 5). First, the cardinality of the association between `BenefitType` and `EventType` changed from 1:1 to 1:*. That is, in the new version several event types may trigger each benefit. Second, to improve budget control, a new monetary `CAP` was introduced in the core framework `Benefit` class, along with a `costPerUnit` property in `BenefitType`. When granting benefits to end users, the improved system checks that the `CAP` is not exceeded; otherwise it fails the benefit request. The new functionality is implemented in the `Benefit` class.

8.1 Core Upgraders

The cardinality change is a change to the type of benefits (`BenefitType.triggerType`), affecting `FreeSMS` and `FreeAstrologicalPrediction`. This is resolved by providing an upgrader that converts a benefit from the old structure to the new structure by adding the triggering event type from the old instance to the list of triggering event types in the new instance. The core upgrader is implemented in Java (Script 1):

Script 1. Core upgrade

```
if (oldInstance.getTriggerType() != null) {
      newInstance.getTriggerTypes().add(oldInstance.getTriggerType());
}
```

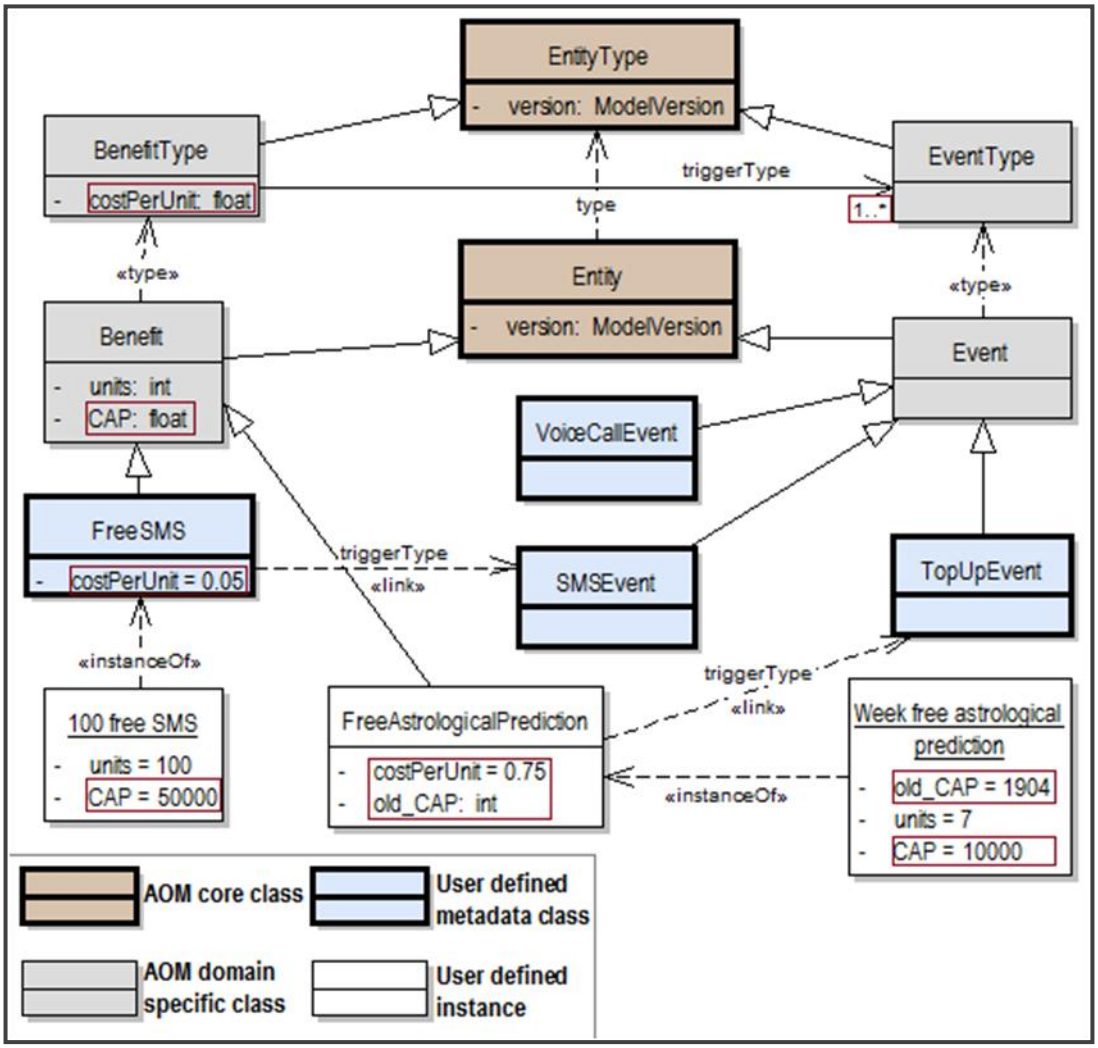

Fig.5. The revised AOM model and instances.

8.2 Conflicts and Custom Upgraders

Next we consider conflicts that may occur during upgrade and their resolution ("*Fix conflicts and write upgrade scripts," Fig 1).* Not all upgrade scenarios can be resolved automatically. In some cases, specific business logic needs to be applied in order to resolve the conflict. For example, the addition of the `CAP` property in `Benefit` clashes with the existing `CAP` property in `FreeAstrologicalPrediction`. The former was introduced by the new product version; the latter was introduced by the customer as part of the customization process. To resolve this conflict, the customer needs to rename the original `CAP` as `old_CAP` and provide an upgrade script that calculates the `CAP` value when loading existing model instances. The script essentially multiplies the `old_CAP` by the `costPerUnit` and the number of units (Script 2):

Script 2. Customized upgrade

```
newInstance.setOldCAP(oldInstance.getCAP());
newInstance.setCAP(oldInstance.getCAP()
      *newInstance.getType().getCostPerUnit()
       *oldInstance.getUnits());
```

When upgrading the `FreeSMS` class the customer needs to assign a default value for the `CAP` property and supply a value to the new `costPerUnit` property. These modifications cannot be resolved in the core product and the customer needs to manually supply values on a case-by-case basis (e.g., `costPerUnit = 0.5`).

Although this example presents an application domain change, the pattern applies just as well to core AOM architecture changes, such as changing the serialization format of AOM objects.

9. RESULTING CONTEXT

We list the pros and cons.

Pros:
- *High Availability:* changes to the AOM architecture can be made without having to stop a running system. Evolution of metadata and data can be done on the fly.
- *Amortized Cost:* the overhead of converting data to the new AOM architecture occurs incrementally.
- *Automatic Migration:* default migration can be provided for the changes to the core AOM metadata and architecture using upgraders. An AOM system may be upgraded from a much earlier version to the latest version without going through the intermediate versions
- *Reuse:* generic configurable upgraders can be provided by the upgrade framework (e.g., for rename property).
- *Usability:* the problem view assists the AOM engineer in resolving version conflicts.

Cons:
- *Higher Complexity:* the AOM BUILDER code is more complex, since hook methods and evolution code become part of the BUILDER. The addition of hook points introduces another level of indirection that makes the code more complex and harder to debug and maintain.
- *Performance Degradation:* the AOM BUILDER runs slower at first, because the upgrade to the metadata is invoked. Running version checks every time an entity is created may further degrade load performance. This overhead can be somewhat reduced by applying the versions check only when a system wide flag indicates that the system is in upgrade mode (and avoiding version checks when in normal operation mode).
- *Fragility:* cached AOM data may need to be flushed or invalidated to force data evolution of existing in-memory AOM instances.
- *Incompleteness*: semantic changes of the core model that require changes in customization are not covered by the upgrade process and need to be addressed separately (e.g., by inspecting the actual changes done).

10. ALTERNATIVE SOLUTION

The core solution presented in this pattern is based on a lazy modification of the metadata as the system evolves. The pattern is designed primarily for modifying the metadata in AOM systems that need to be running 24x7. When such execution constraints do not exist, there is a very common alternative (a different pattern) that can be used to modify the metadata safely: to suspend the AOM system and completely migrate all of the metadata before resuming the system.

This alternative requires a complete shutdown of the running AOM system. While the system is off-line, an upgrade script can completely evolve all of the metadata and data before bringing the AOM system back online. The benefit of the alternative solution is in avoiding the performance overhead at runtime since the system will not need to invoke checks and hooks in order to possibly evolve the metadata as it is loaded and instantiated.

There are several tradeoffs to consider for either solution, but the solution described in this pattern requires enhancements to the core of the AOM architecture, while the alternative solution does not affect the architecture of the AOM.

11. RELATED PATTERNS

- HISTORY OF OPERATIONS (Ferreira et al. 2008) can be used to transfer AOM metadata changes between the development, staging and production systems. This helps in managing the overall process.
- AOM BUILDER (Welicki et al. 2009) is an evolution of the BUILDER (Gamma et al. 1995) pattern.

– EVOLUTION RESISTANT SCRIPTS is used to invoke the UPGRADE SCRIPTS (Hen-tov et al. 2010).
– HOOKS (Fontura et al. 2001) are invoked in the AOM loader to enable customers to provide default values for AOM attributes when migrating to a new version.

12. KNOWN USES

The pattern presented in this paper is used intensively in online marketing systems developed at Pontis Ltd. (www.pontis.com), a provider of online marketing solutions for communication service providers. Pontis' Marketing Delivery Platform is implemented in ModelTalk (Hen-tov et al. 2009) and supports on-site customization and model evolution by non-programmers. Pontis' system is deployed in over 20 customer sites including Tier I Telco providers. A typical customer system handles tens of millions of transactions a day exhibiting Telco-Grade performance and robustness.

There is a mutation technique developed for migrating objects in Smalltalk-80 (Caudill and Wirfs-Brock, 1986) that is a known use of the core of this pattern. These authors made an enhancement to Smalltalk that changed how class redefinitions worked. The original Smalltak-80 system mutated all existing instances to use the new representation whenever a class was redefined. To eliminate the overhead of this bulk object mutation, the new approach used lazy mutation. Lazy mutation defers object mutation until it is actually used in a computation. Lazy mutation is accomplished by replacing the method dictionary of the old class definition with a dictionary which defines only the message `doesNotUnderstand:`. In addition, the superclass of the old class definition is set to `nil` and a reference to the new class definition is stored within the old definition. When a message is sent to such an instance of the old class, a response to the message will not be found and the `doesNotUnderstand:` method will be activated. This method contains the code to mutate the instance into an instance of the new class. The difference between this approach and the pattern described in this paper is that lazy mutation only applies a single evolution to an object, not a chain of replacements.

An AOM system developed for the Illinois Department of Public Health used the alternative solution mentioned above. There are also many other well-known uses of the alternative solution done by some of the authors.

13. SUMMARY

When evolving a production AOM, new functionality requires modifications to definitions of existing domain entities. An AOM production system can have considerable pre-existing metadata and domain objects that will need to be updated in order to adapt to the new version of the AOM architecture. Quite often it is the case that you will want to incorporate model changes into an AOM production system without taking the system off-line. The DYNAMIC MODEL EVOLUTION pattern describes a way to lazily update the metadata and the domain objects as a part of loading the domain objects. This allows for 24x7 systems to dynamically update the model to new features provided as the AOM evolves.

14. ACKNOWLEDGEMENTS

We thank our shepherd Filipe Correia for his valuable comments on this pattern during the PLoP 2010 Shepherding process and our writer's workshop group members, Christoph Hannebauer, Vivek Gondi, Joshua Kerievsky, Vincent Wolff Marting, Benjamin Schleinzer, and Hironori Washizaki, for their valuable comments during PLoP. We also thank Pontis Ltd. (www.pontis.com) for granting us access to their development process.

REFERENCES

Adaptive Object-Models. http://www.adaptiveobjectmodel.com.

ANDERSON, F. A collection of history patterns. In *Proceedings of the 5th Pattern Language of Programs Conference (PLoP'98)* (Allerton Park, Monticello, Illinois, USA, August 11-14 1998), ACM. Technical Report #WUCS-98-25, Dept. of Computer Science, Washington University, October 1998. http://hillside.net/plop/plop98/. In *Pattern Languages of Program Design 4*, N. Harrison, B. Foote, and H. Rohnert, Eds. Addison-Wesley, 2000, pp. 263–297.

ARSANJANI, A. Rule object: A pattern language for adaptive and scalable business rule construction. In *Proceedings of the 7th Conference on Pattern Languages of Programs (PLoP 2000)* (Allerton Park, Monticello, Illinois, USA, August 13-16 2000). Technical Report #WUCS-00-29, Dept. of Computer Science, Washington University, http://hillside.net/plop/plop2k/.

CAUDILL, P. J., AND WIRFS-BROCK A. A third generation Smalltalk-80 implementation. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)* (Portland, Oregon, USA, 1986), ACM SIGPLAN Notices 21(11) November 1986, pp. 119–130.

FERREIRA, H. S., CORREIA, F. F., AND WELICKI, L. Patterns for data and metadata evolution in adaptive object-models. In *Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP'08)* (Nashville, Tennessee, USA, October 18-20 2008), ACM. http://hillside.net/plop/2008/.

FONTURA, M., PREE, W., AND RUMP, B. The UML profile for framework architectures. Addison-Wesley Professional, 2002.

FOOTE, B., AND YODER, J. Metadata and active object models. In *Proceedings of the 5th Pattern Language of Programs Conference (PLoP'98)* (Allerton Park, Monticello, Illinois, USA, August 11-14 1998), ACM. Technical Report #WUCS-98-25, Dept. of Computer Science, Washington University, October 1998. http://hillside.net/plop/plop98/.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object Oriented Software.* Professional Computing, Addison-Wesley, Reading, MA, 1995.

HEN-TOV, A., NIKOLAEV, L., SCHACHTER, L., WIRFS-BROCK, R., AND YODER, J. W. Adaptive object-model evolution patterns, SugarLoafPLoP 2010.

HEN-TOV, A., LORENZ, D. H., PINHASI, A. AND SCHACHTER, L. ModelTalk: When everything is a domain-specific language, *IEEE Software*, vol. 26, no. 4, pp. 39-46, July/Aug. 2009.

JOHNSON, R., AND WOOLF, B. Type object. In *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, Eds., Software Patterns. Addison-Wesley, 1997, pp. 47–65.

WELICKI, L., YODER, J., AND WIRFS-BROCK, R. Adaptive object-model builder. In *Proceedings of the 16th Conference on Pattern Languages of Programs (PLoP'09)* (Chicago, Illinois, USA, August 28-30 2009), ACM. http://hillside.net/plop/2009/.

YODER, J. W., BALAGUER, F., AND JOHNSON, R. Architecture and design of adaptive object-models. *ACM SIGPLAN Notices 36, 12 (December 2001)*, 50–60. Special section on Intriguing Technology from OOPSLA.

YODER, J. W., AND JOHNSON, R. E. The adaptive object-model architectural style. In *IFIP 17th World Computer Congress - TC2 Stream / Proceedings of the 3rd IEEE/IFIP Conference on Software Architecture (WICSA3)* (Montréal, Québec, Canada, August 25-30 2002), J. Bosch, W. M. Gentleman, C. Hofmeister, and J. Kuusela, Eds., vol. 224 of *IFIP Conference Proceedings*, Kluwer, pp. 3–27.

YODER, J. W., AND RAZAVI, R. Metadata and adaptive object-models. In *Proceedings of the ECOOP 2000 Workshops, Panels, and Posters* (Sophia Antipolis and Cannes, France, June 12-16 2000), J. Malenfant, S. Moisan, and A. M. D. Moreira, Eds., vol. 1964 of *Lecture Notes in Computer Science*, Springer, pp. 104–112.

Appendix to:
Dynamic Model Evolution

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 6).

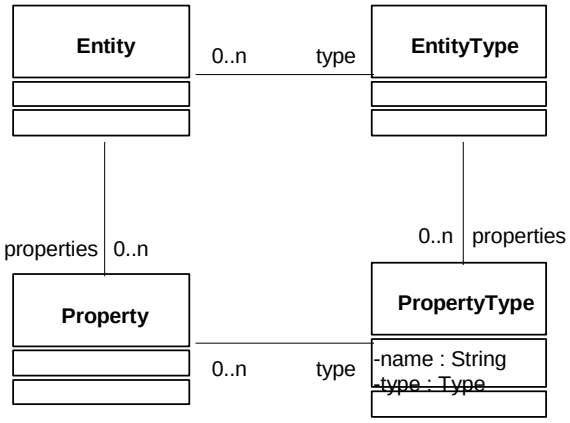

Fig.6. Type Square

PLoP'10, October 16-18, Reno, Nevada, USA. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7

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. 6 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, and 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 7).

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.

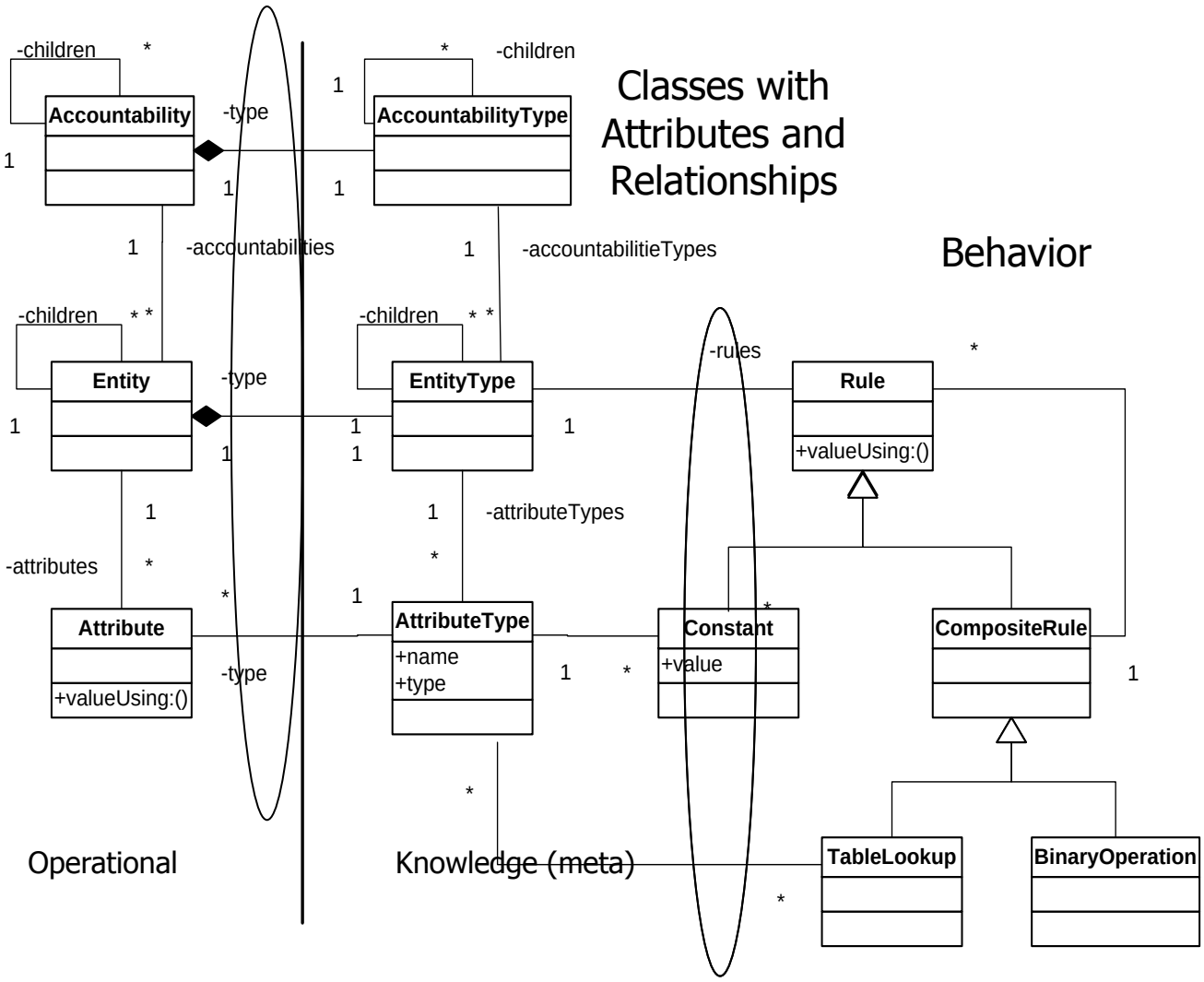

Fig.7. Core AOM Architecture.