Patterns for Data and Metadata Evolution in Adaptive Object-Models

Hugo Sereno Ferreira
Faculty of Engineering
University of Porto
Rua Dr. Roberto Frias, s/n
hugo.sereno@fe.up.pt

Filipe Figueiredo Correia
Faculty of Engineering
University of Porto
Rua Dr. Roberto Frias, s/n
filipe.correia@fe.up.pt

Leon Welicki
ONO (Cableuropa S.A.)
lwelicki@acm.org

ABSTRACT

An Adaptive Object-Model (AOM) is an architectural pattern based upon a dynamic meta-modeling technique where the object model of the system is explicitly defined as data to be interpreted at run-time. The object model encompasses the full specification of domain objects, states, events, conditions, constraints and business rules. Several design patterns, that have before been documented, describe a set of good-practices within this domain. This paper approaches data and metadata evolution issues in the context of AOMs, by describing three additional patterns — HISTORY OF OPERATIONS, SYSTEM MEMENTO and MIGRATION. They establish ways to track, version, and evolve information, at the several abstraction levels that may exist in an AOM.

Keywords

Adaptive object models, Model driven engineering, Design patterns, Meta-modeling, System Memento, History of Operations, Migration.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Patterns

1. INTRODUCTION

Developers who are faced with the system requirement of a highly-variable domain model, by systematically searching for higher flexibility of object-oriented models, often converge into a common architecture style typically known as Adaptive Object-Model (AOM) [28].

The Adaptive Object-Model architecture fulfills particular needs of the several Model-Driven Development methodologies [13], and allows for on-the-fly adaptivity by the use of runtime models. It can be summarized as an architectural style that uses an object-based meta-model as a first-class artifact from where all domain information can be obtained, or derived from: structure (such as classes, attributes and relations), behavior (rules and workflow) and presentation (graphical user interfaces). At runtime this information is interpreted, instructing the system which behavior to take. Changing the model immediately results on the system following a different business domain model. For the purposes of this paper, whenever we refer to *system*, we mean both *data* and *metadata*.

One of the key aspects of Adaptive Object-Models is their ability to allow changes to the model even at run-time. Model evolution is thus a recurrent problem that developers adopting this architecture face, since it may introduce inconsistency in its structure. This problem can be split into three complementary issues:

Track. How to keep track of the operations performed for evolving the system?

Time Travel. How to access specific key states of the system at any particular point of its evolution?

Evolution. How to introduce changes into the system while preserving its integrity?

This paper presents three domain specific design patterns that have risen from the experience implementing Adaptive Object-Models, and researching how other systems, particularly Object-Oriented Databases and Version Control Systems, deal with these problems [22, 25, 4]. These patterns aim to contribute to the on-going effort on defining a pattern language for AOMs [27, 26].

Patterns under the name of HISTORY and VERSIONING have been foreseen as part of a Pattern Language for AOMs [27]. In this work, we now call these two patterns HISTORY OF OPERATIONS and SYSTEM MEMENTO, and add a third one — MIGRATION — which aims to further decouple the concerns of system evolution.

1.1 Levels of Abstraction

Traditional literature on AOMs usually describe two different levels: (a) the knowledge level, which defines the domain model, such as classes, attributes, relationships, and behavior, and (b) the operational level which consists in the run-time instances of the domain model [28]. However, there's also a third level: the model which describes the concept of an AOM. Making the parallel with the nomenclature used by the OMG and their MOF initiative [19], instances of the operational and knowledge levels are equivalent to M_0 and M_1 levels respectively, where M_0 are instances (entities) of M_1 defined elements (entity types). M_2 is roughly equivalent to the models used to define an AOM. M_2 may be defined either implicitly, through the target programming language during implementation, or explicitly, through the usage of a meta-metamodel (see **Figure 1**). It should be noted that MOF is a closed meta-modeling architecture, since M_3 is compliant to itself.

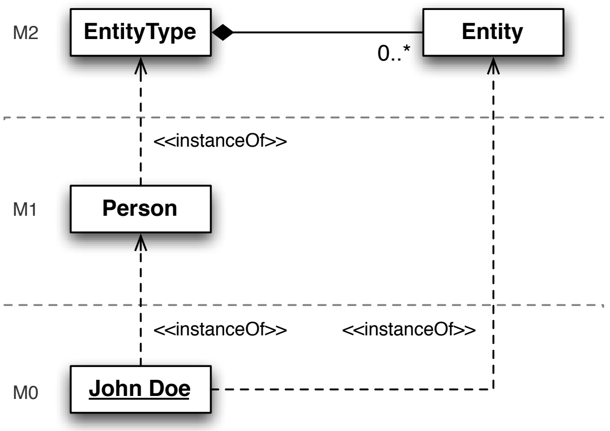

Figure 1: AOM as a meta-modeling technique.

While the traditional AOM architecture only considers the M_0 and M_1 levels, nothing keeps system developers from defining higher-level models. To decouple each of these patterns from a particular model-level, we define a simple extension to the TYPE-SQUARE pattern [28] (see **Figure 2**).

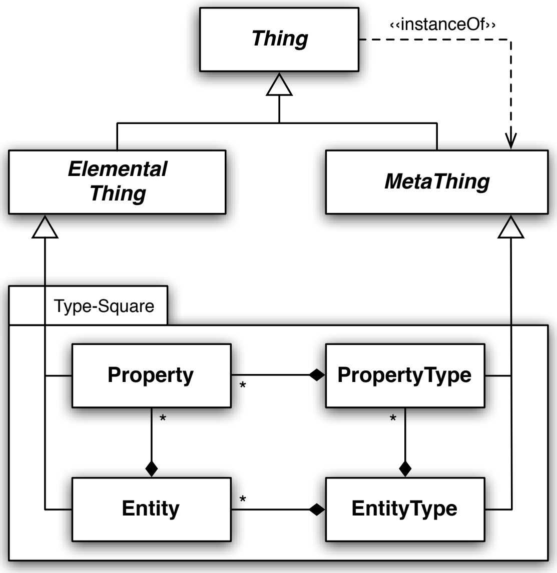

Figure 2: Extension to the TYPE-SQUARE pattern.

The concept of `Thing` is here defined as being specialized into either an `ElementalThing`, which represents data (i.e. `Entities` and `Properties`), or a `MetaThing`, which represents metadata (i.e. `EntityTypes` and `PropertyTypes`). Any object of type `Thing` is actually an instantiation of a `MetaThing`, thus allowing an unbounded definition of meta-levels. Eventually, the upper-bound may be delimited when a defined `MetaThing` is regarded as an instantiation of itself (or simply not defined). Because every class in our model and meta-model derives from `Thing`, this extension allows one to explicitly state the IDENTITY [12] of an object (`EntityTypes`, `PropertyTypes`...).

It's worth highlighting that, in the context of this paper, `Things` may be both `ElementalThings` and `MetaThings` so, when something is said to be applicable to a `Thing`, it may be used regardless of the model-level.

The notation used in this paper complies to the UML 2.1 and OCL 2.1 specifications [20, 18].

1.2 Data and Metadata Patterns

This paper documents the following three patterns:

History of Operations. Addresses the problem of maintaining a history of operations that were taken upon a set of objects.

System Memento. Deals with preserving the several states the system has achieved upon its evolution.

Migration. Addresses the concern of performing evolution upon the system while maintaing its structural integrity.

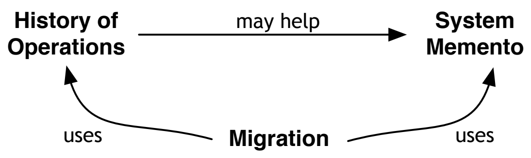

Figure 3: Data and metadata evolution patterns.

All patterns further presented are closely related (see **Figure 3**). MIGRATION depends upon the concepts of HISTORY OF OPERATIONS and SYSTEM MEMENTO (which in turn may be helped by HISTORY OF OPERATIONS). MIGRATION orchestrates the coordination between the other two patterns, so that enough semantics is gathered to fulfill its intended purpose.

1.3 Target Audience

The patterns presented in this paper deal with instrumentation issues, specifically, evolution concerns, that arise when working with AOMs [27]. Developers, either working or designing these type of systems, who recognizes the presented forces and problem statements as being part of their systems' functional requirements, will benefit from knowing these patterns.

Nonetheless, developers can still adapt them to uses outside the scope of AOMs, particularly in other meta-modeling based architectures, always taking into consideration AOM-specific issues in these solutions which probably should be re-evaluated outside this context.

2. HISTORY OF OPERATIONS PATTERN

Addresses the problem of maintaining a history of operations that were taken upon a set of objects.

2.1 Context

An application based on the Adaptive Object-Model as the main architectural style is being developed, and there is the need to track the system's usage by end-users, including changes to both the knowledge and operational levels.

2.2 Example

Consider an information system based upon the AOM architecture, using a variant of the TYPE-SQUARE pattern [28, 16] that conforms to the previously mentioned design (see **Figure 2**).

Imagine an insurance company who's users keep changing the system's information at a fast pace. There is the need to keep track of *what*, *how* and probably *when* and by *whom* it has been changed. For this example system, meta-information is as important as the information itself.

Keeping track of the operations' history can go beyond auditing purposes, like performing statistical analysis (e.g. number of created instances per user), controlling user behavior (i.e. without recurring to explicit user access control), automating activities (e.g. finding systematic modifications to the information) or recovering past states of the system.

In an AOM based application, there are different levels at which `Things` can change (data, model...). For example, consider an `EntityType` named *Person* and a particular `Entity` named *John*. The kind of actions we may perform can be as simple as CRUD-like operations (e.g. deleting the `Entity`, or changing its name), or model operations (e.g. adding the attribute *Number of Children*, or moving it to the superclass).

The history of operations must be made available in the application, since it will be used by end-users. However, simply storing messages in a file or database makes the mapping between them and the operations over things, a complex (if at all possible) activity. Furthermore, as the underlying AOM interpreter evolves, the type of operations that should be recorded may also evolve. This can result in an set of messages to parse, with obsolete syntactic details that may no longer be directly mappable.

2.3 Problem

Given a set of Things, how do we keep track of the history of operations that were performed upon them, without knowing the specific details of each operation?

2.4 Forces

Encapsulation. We don't want to pollute the system with logging structures wherever they are needed.

Extendability. We may want to add additional information to the history (e.g. *before*, *user*, *time*...).

Operations' Semantics. Operations should have enough semantics to allow automatization.

Simplicity. Occurred operations should be easy to store and retrieve.

Modifiability. The allowed operations can be expanded or evolved.

Performance. There should be minimal performance impact on the system.

Reusability. We want to use the same mechanism regardless of the model-level.

Consistency. Operations should comply to semantic constraints assuring system's integrity.

Reproducibility. Operations should be able to be re-executed and achieve the same result (e.g. deterministic).

Resource Consumption. Additional manipulated and stored information should be carefully minimized.

2.5 Solution

Encapsulate the allowed Operations in a set of commands that operate over Things. A sequence of invoked commands constitutes the History of Operations.

Create `Operations`, using the COMMAND pattern[14], with the responsibility of defining and encapsulating the types of modifications allowed (i.e. Evolution Primitives [22]). These may be elemental — `Concrete Operations` —, or grouped in a sequence — `Macros` – through the use of COMPOSITE pattern [14].

Every action taken by the application must occur by instantiating and executing a defined `Operation`. Creating an `History` object is as simple as storing the sequence of the invoked `Operations`. Each `Operation` will retain enough information in order to be mappable to the `Things` it operates over (see **Figure 4**). However, note that if operations are not versioned, they should be made either static or semantically equivalent upon evolution, otherwise the history may become unusable.

By using the HISTORY OF OPERATIONS pattern, developers can factor the responsibility of creating and storing modifications in a semantically rich way. This will allow an easier evolution of the underlying interpreter and other automatizations.

2.6 Example Resolved

Consider five employees from the automobile insurance department, working as a team. During a week, they create and alter information on the system, either from external demands (e.g. clients) or from the rest of the company. Namely, they subscribe clients to policies, answer to the events of new occurrences, and redefine conditions for upcoming policies.

On this particular week, the same client record happened to be edited by three different users. Yet, there was an incorrectly registered occurrence for that client, and it's important to understand why it happened in order to prevent future mistakes. The history of operations registered throughout the week allows users to find out exactly what happened: the occurrence was registered by one particular employee on Tuesday, because the client was wrongly chosen to begin with, since it was selected by searching his name, instead of his client-number.

By Friday, the department's director wants to know how the week went, before the weekly meeting with his staff. He uses the system's functionality that collects several statistics from that week's history of operations, and realizes it was in fact a particularly busy week.

2.7 Resulting Context

This pattern results in the following benefits:

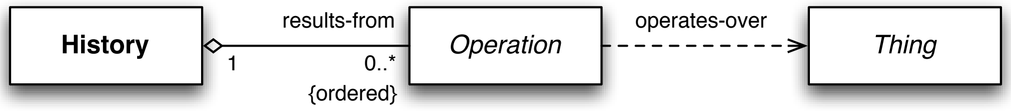

Figure 4: Class diagram of the History of Operations pattern. A `History` results from a set of `Operations` done over `Things`.

- Because every modification is abstracted into an evolution primitive (as an `Operation`), the history is made simply by storing the sequence of performed commands — **encapsulation** – which also **simplifies** the control of semantic/constraints checking, **auditing** and **security** issues.
- A side-effect of mapping the allowed operations to Commands is the further promotion of **reuse**, **easier maintenance** and **consistency**.
- If enough information is stored with each evolution primitive — **semantics** — it becomes possible to playback the executed operations.
- The use of the Composite pattern to create **Macros** of operations addresses the issue of atomicity, thus preserving **consistency**.

This patterns has the following liabilities:

- The quantity — **space consumption** — of additionally stored meta-information may be considerable, as it will always grow with time, despite the size of the current valid objects and meta-objects. However, the use of compression techniques and the external archiving of unnecessary history may lessen the impact of this liability.
- The **performance** may be affected because of the quantity of instantiated objects and the necessary pointer dereferencing/set joins associated with particular implementations.
- The implementation may be more **complex**.

2.8 Implementation Notes

Semantic Consistency. The semantic consistency of `Things` can be kept by enforcing constraints defined at an upper abstraction level (i.e. operational-level constraints are defined at the knowledge-level). One way to enforce these constraints is to use *pre* and *post* operation conditions.

Keeping operations as general as possible will leverage their reusability and maintainability, but leads to operations of low granularity. The use of CRUD-like operations is a good example, as they focus on very straightforward tasks, and cover a wide scope of use cases when combined.

However, some sequences of operations may be impossible to carry out while ensuring consistency at the end of each of them, although information would be in a consistent state upon completion of the entire sequence. Consider two classes, A and B, with a mandatory one to one relation between them, and two particular instances of these classes, a and b, thus connected through that same relation. Suppose we replace b by a new instance b', as the other end of the relation. If we consider only CRUD-like operations, three different operations would be needed: the deletion of the relation between a and b, the creation of a new relation connecting a and b' and the deletion of b. By the end of these operations, information would be in a consistent state, but that would not be the case just after each individual operation completes, since mandatory relations would not exist.

As described, through the use of the Composite pattern, `Operations` can be grouped in sequence — `Macros`. These macros are a means to the reuse of operations, but may also be used to establish consistency-checking frames. Instead of checking the consistency of information after each individual elemental operation, it may be checked only at the end of the macro in which they are enclosed. This notion is akin to the concept of **transactions** in database systems.

2.9 Related Patterns

Operations are structured using the Command pattern [14]. The hierarchy of operations are also related to the Composite pattern [14]. The storage of information may be done similarly to the AuditLog pattern [7], though with more semantics to increase traceability and automation.

The Identity pattern [12] is also used as described in **Section 1**.

2.10 Known Uses

This pattern is common in *Object-oriented Database Management Systems* and *Data Warehouses* [22, 25, 4]. The *Prevayler framework* [21] and the *COPE tool* [15] are also known to use this pattern, as well as the work presented in [1].

3. SYSTEM MEMENTO PATTERN

Deals with preserving the several states the system has achieved upon its evolution.

3.1 Context

An application based on the AOM architectural style is being developed, and there is the need to access the state of the system at any point (present or past) of its evolution.

3.2 Example

We are developing an information system based upon the AOM architecture, using a variant of the Type-Square pattern [16, 28] that conforms to the previously mentioned design (see **Figure 2**).

Imagine a heritage research center where its users keep collecting information as they perform their regular activities. Due to the nature of the research, uncertainty of the information is common, leading to several changes over time. While the pace of collected information is not high, any change in the system is critical since no previous information should be lost, and even if deleted at one point, should

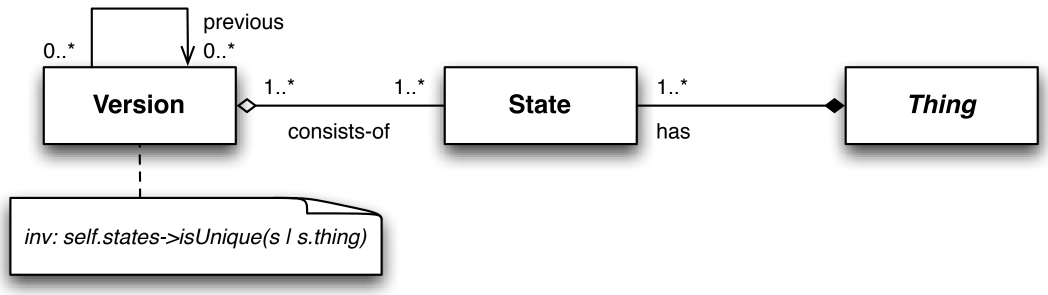

Figure 5: Class diagram of the System Memento **pattern. A** `Version` **is a collection of** `States`**, one per** `Thing` **(i.e. there cannot be multiple states of the same thing in the same version).**

be recoverable in the future, by the same or other user.

Because we are using an AOM based system, there are several levels at which we want to persist the state of the objects as they are evolved (data, model, meta-model...). For example, suppose we have an `EntityType` named *Archeological Survey* and a particular `Entity` called *Survey of the Coliseum.* At a certain point in time, the *Coliseum* could have been dated as 100AC, but recent research has casted doubt on that date, and it has since been oscillating between 200BC and 500AC.

One can also consider a case in which this system has been running for a considerable amount of time, and several thousand *Archeological Surveys* have been registered. Yet, through acquired experience, users have now found the need to additionally register the leader of each archeological expedition. As such, an evolution would need to take place at the model level, to accommodate this new property of the *Archeological Survey's* `EntityType`.

3.3 Problem

How can we access the state of a system, at any particular point of its evolution?

3.4 Forces

Reusability. Usage of the same versioning mechanism regardless of model-level (i.e. data and metadata).

Encapsulation. We don't want to pollute the system with versioning logic everywhere it's needed.

Identity. It should be possible to reference either an object or one of its states, independently of each other.

System-Level Semantics. Versions should represent the evolution of the system, and not of a particular object.

Time Independence. Though evolution usually occurs with the passage of time, the system shouldn't have to be aware of it (the concern is the sequence of changes).

Accessibility. It should be possible to access the system at any arbitrary point of its evolution.

Space Consumption. The data-set at hand should be kept to a manageable size.

Branching. Allowing information to be branched may require merge mechanisms.

Consistency. Any particular state of the system must comply to integrity constraints (e.g. an M_0 object must be compliant to its M_1 definition).

3.5 Solution

Separate the identity of a Thing from its properties such that, by aggregating a particular State of Things, one can capture the global state of the system at any particular point of its evolution.

Applying this pattern usually starts by decoupling `Things` from their `States` [12]. While `Things` represent the identity of an object, `States` represent its content, which will evolve over the use of the system's information (see **Figure 5**). A `Version` thus captures the global state of the system, by referencing all the valid `States` at some point of the system's lifetime. Each `Version` maintains references to the those that gave origin to it (previous), and to those that originated from it (subsequent). Usually, however, each `Version` is based on a single previous `Version`, and will give origin to a single other `Version`, thus resulting in a linear time-line. However, more than one previous and/or next `Versions` may be considered, specially in concurrent usage environments, for purposes of data reconciliation.

Each individual `Version` may accommodate both instance and model-level `Things`. This results in a particularly useful design, since a change at the model-level can often lead to changes at the instance-level. In order to aggregate a consistent group of `States`, every `Version` need to be able to reference `States` from both levels. In fact, this is an essential issue for the Migration pattern, since changes to the model usually require changes to the data.

3.6 Example Resolved

Consider the aforementioned *Survey of the Coliseum* (see **Example**). Over the last year new information about the Coliseum was acquired, through the study of newly found manuscripts. Users updated the information on the system, such that it would reflect their best knowledge at each phase of the research. Therefore, the description of this monument evolved over time. As such, several `Versions` may have been created, each representing a consistent point on the evolution of the available information. Thus, it becomes possible to access, and even recover, previous states of the system.

Eventually, the model may also need to evolve. As described in the example, a new `AttributeType` may be added to accommodate the name of the leader of each archeolog-

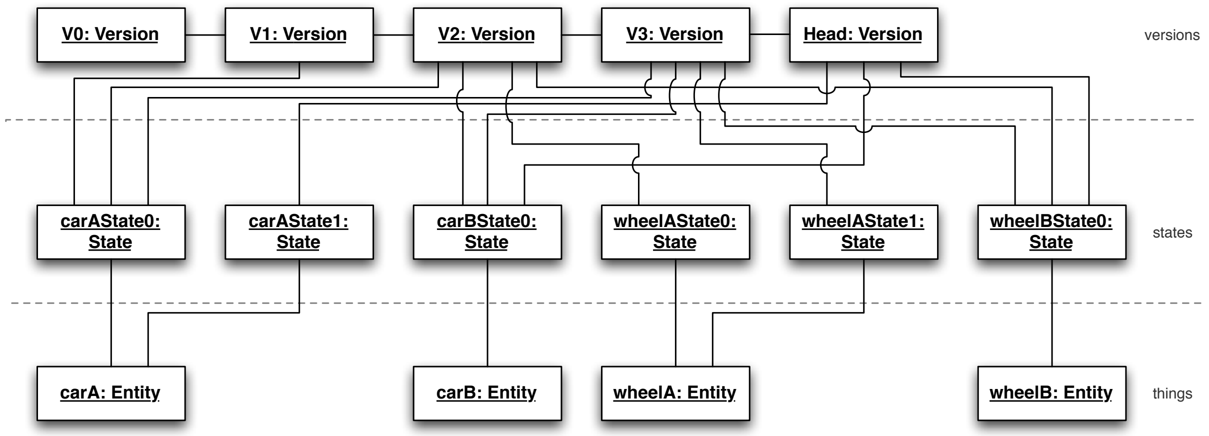

Figure 6: Object diagram for an example instantiation of the System Memento pattern.

ical expedition. Since an `AttributeType` is a `Thing`, a new `Version` will be created that references model-level `States` and `Things`.

3.7 Resulting Context

This pattern results in the following benefits:

- We are now able to use the same versioning mechanism regardless of the model-level — **reusability**.
- By **decoupling** the state from the object, we are able to isolate the object's **identity**.
- Because the concept of version is now at system level — **system-level semantics** — instead of object-level, we are now able to address **consistency**.
- If multiple evolution branches are used, **concurrency** may be coped with more easily.

This pattern has the following liabilities:

- The quantity of stored information may be larger than affordable — **space consumption**. The choice of appropriate persistency strategies may reduce this issue (see **Implementation Notes**).
- The branching of versions will require additional merging mechanisms.
- **Performance** may be affected by the overhead introduced while changing, storing and accessing information.
- It may increase the systems' **complexity** due to additional object dereferenciation.

3.8 Implementation Notes

Coping with state explosion. It should be noted that a literal implementation of this approach may lead to an unnecessary use of space as the system evolves. Versioning systems typically deal with this issue by partially inferring, instead of explicitly storing, the complete set of states that define a particular version (i.e. by just keeping the *deltas*). Because this issue can determine the feasibility of a system, we present some notes overviewing one possible solution.

Consider the following sets of operations (a) create *carA*, (b) create *wheelA*, *wheelB* and *carB*, (c) modify *wheelA*, and (d) modify *carA* and delete *wheelA*. The resulting set of versions can be observed as an object diagram in **Figure 6**.

Any `Thing` that doesn't change its `State` in any subsequent version, would have its `State` replicated across those versions. Using a strategy where only changes to states are stored, thus inferring (instead of storing) the complete set of states for any version, the stated example would become as observed in **Figure 7**.

In english, the inference rules can be summarized as: *if a state belongs to a delta, then it also belongs to the corresponding and subsequent versions, until a new state is defined or the null state is reached (i.e. when an object is deleted).*

3.9 Related Patterns

The patterns Temporal Property [10], Effectivity [8], Memento [14], Temporal Object [9], Snapshot [5], and several others [2, 3], are directly related to the problem of storing the changing values of an object. However, none of them explicitly addresses the concerns of *system-level semantics* (i.e. they focus on the change of a single object instead of the whole system) and *Meta-modeling* (i.e. the change of an object's specification).

The Migration Pattern, described in this work, uses System Memento to allow arbitrary evolution of the system between any two versions.

The Identity pattern [12] is also used as described in **Section 1**.

3.10 Known Uses

This pattern is common on Wikis and Version Control Systems. The work presented in [3] also details the implementation of several versioning techniques in object-oriented design. Several *Object-oriented Database Management Systems* and *Data Warehouses* [22, 25, 4], as well as the *Prevayler framework* [21] and the *AMOR system* [1], are known uses of this pattern.

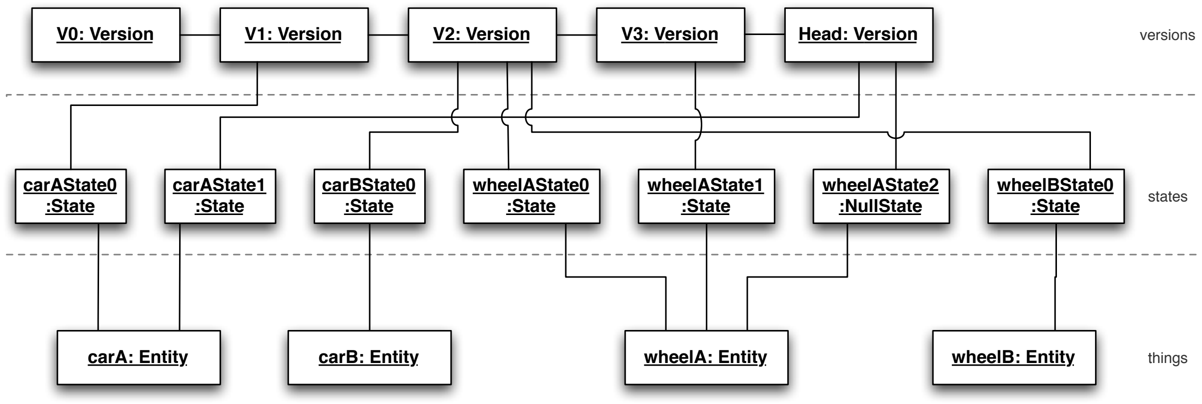

Figure 7: Object diagram for the *delta* strategy instantiation of the System Memento pattern.

4. MIGRATION PATTERN

Addresses the concern of performing evolution upon the system, while maintaing its structural integrity.

4.1 Context

An application based on the AOM architectural style is being developed, and it will be necessary to evolve model and data definition (assuring consistency) after system's deployment.

4.2 Example

We are developing an information system based upon the AOM architecture, using a variant of the Type-Square pattern [16] that conforms to the previously mentioned design (see **Figure 2**).

Consider an insurance company where several domain rules and structure, due to the nature of the business, keep changing to fulfill market needs. One example is the insurance payback for any particular kind of incident, which is based on a complex formula that takes into account several factors. Not only the formula changes as the system evolves, but also the factors taken into account change, thus needing new information to be either collected or inferred (e.g. the number of children of an individual while calculating his life insurance payment).

However, even simple evolutions of the structure or behavior, like removal of information, can have a significant impact in the system. Valid objects may depend on the information being changed, thus leading to inconsistency. These issues need to be addressed upon each evolution step, to guarantee that the integrity of the system holds to the specification.

Another typical concern is maintaining legacy interfaces. If the system must interoperate with third-party components, once the model definition evolves, the interface can become invalid. In this case, it may be necessary to provide a layer of data transformation, thus maintaining legacy interfaces over previous versions of the system. This approach may increase the complexity of the underlying architecture.

4.3 Problem

How do we support the evolution of a system while maintaining its integrity?

4.4 Forces

Due to the use of the History of Operations and System Memento, the Migration pattern is subject to the same forces. Some additional forces specific to this pattern are presented below:

Automation. We want to automate the evolution instead of relying on monolithic, custom made scripts.

Integrity. Applying a migration should result in a consistent state of the system.

Control. We want to restrict the kind of evolutions allowed upon the system.

Interoperability. We may need to maintain interoperability with third-party systems not aware of the model evolution.

4.5 Solution

Use the History of Operations to support the Versioning of Things. Achieving a target Version is the result of applying the sequence of Operations defined between two Versions.

As described in the History of Operations and System Memento patterns, first start by decoupling the `State` of a `Thing` from its identity (see the Identity pattern [12]). Also, every `Operation` over a `Thing` should be structured as a Command [14]. Instead of operating over `Things`, operations should occur over (or generate new) `States`. Considering there is a one-to-one relationship between the `History` and `Version` classes (see History of Operations and System Memento patterns), the later can fulfill both roles (see **Figure 8**).

An `Operation` can be specialized into either `Concrete Operations`, or `Macros` that establish a sequenced group of other `Operations`, through the use of the Composite pattern [14].

The ability of an `Operation` to spawn other `Operations`, allows changes on the knowledge-level to be reflected upon the operational-level, whose purpose is to maintain the consistency of the system. For example, a *Move Attribute to Superclass* at the knowledge-level may generate several `Operations` at the operational-level, since data may also need to be moved.

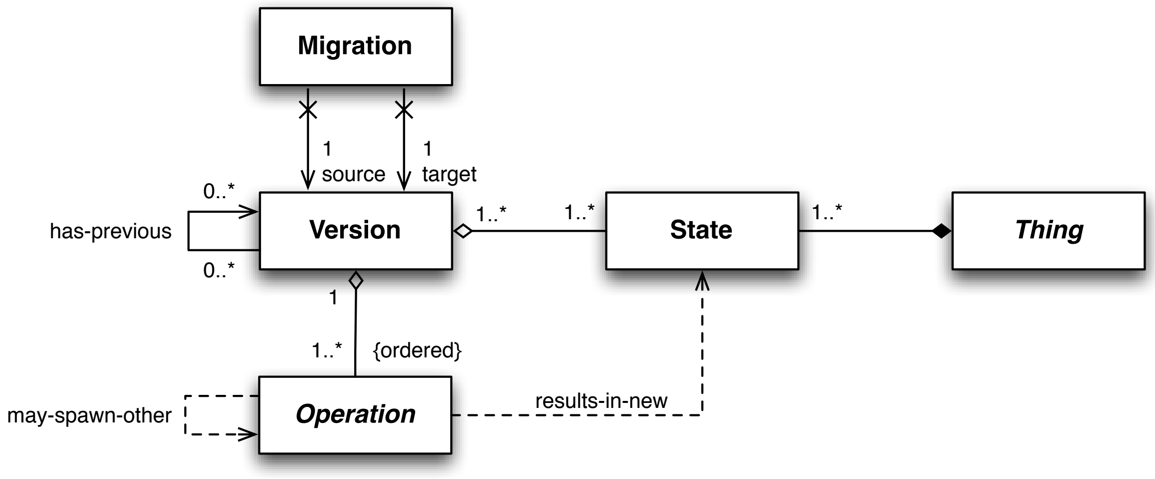

Figure 8: Class diagram of the MIGRATION pattern. A `Migration` between any two `Versions` consists on applying, in the correct order, all the histories of `Operations` that were executed between those versions. A `Version` may not refer more than one `State` from the same `Thing`.

The MIGRATION acts as an interpreter, or patch engine, which, given a `Version` and a set of `Operations`, achieves a target `Version`.

4.6 Example Resolved

One of the most complex examples this pattern support is the ability to evolve what is normally called the schema (in this case, the word *model* is more appropriate) and to immediately affect data at lower levels.

Let us consider the aforementioned example. The introduction of new laws will require the creation of new fields in existing entities (e.g. *number of dependent children*). Others, previously belonging to a particular sub-class, will now be moved into the super-class (e.g. *number of days overseas per year*).

Consider this evolution will occur from version V_1 to version V_2. Two M_1 (knowledge-level) operations are issued: (a) *Create Attribute* and (b) *Move Attribute to Superclass*. While the former doesn't need to spawn any M_0 `Operations`, the later should be defined as a `Macro`, mixing sequential M_0 and M_1 operations (e.g. *Create Attribute* at M_1, *Duplicate Data to Attribute* at M_0, *Delete Attribute* at M_1 and *Dispose Data* at M_0). Each `Operation` will act upon a specific given `State` of a set of `Things` to generate new `States`. This sequence of commands, interweaving different level **operations**, may be stored in the new `Version` (V_2).

In summary, *a migration between any two versions consist on applying, in the correct order, all the histories of operations that were executed between those versions.*

4.7 Resulting Context

The resulting context of applying this pattern is the combined resulting contexts of the HISTORY OF OPERATIONS and SYSTEM MEMENTO patterns. The following benefits are particular to this pattern:

- We are now able to **automatically** evolve between any two versions of the system, provided that we issue a semantically correct sequence of operations.
- Consistency of the system is dependent on the consistency of the operations. This functional decomposition may help achieving higher confidence in the model **integrity** after a migration procedure.

This pattern has the following liabilities:

- If the system does not provide enough operations to perform complex tasks, it can be difficult (or even impossible) to express the intended semantics of the evolution.
- The migration mechanism, along with all the additional information that it requires, adds **complexity** to the system.

4.8 Implementation Notes

Refactorings as Evolution Primitives. In object-oriented programming, behavior-preserving source-to-source transformations are known as refactorings [11]. The concept of refactoring applied to models [6, 17, 15] has already been pointed out as a way to cope with system evolution. This notion may be applied when designing `Operations`, such that they represent a set of refactorings specifically designed for evolving `Things`. Each refactoring should assure system integrity upon its completion.

4.9 Related Patterns

All related patterns to HISTORY OF OPERATIONS and SYSTEM MEMENTO apply.

4.10 Known Uses

Several *Object-oriented Database Management Systems* and *Data Warehouses* [22, 25, 4], as well as the *Prevayler framework* [21], the *COPE tool* [15], and the *AMOR system* [1], are known uses of this pattern.

The *Ruby on Rails (RoR)* framework uses a variation [23, 24] of MIGRATION, but expresses operations within relational models, since it's based upon the ACTIVE RECORD Pattern [12].

5. ACKNOWLEDGMENTS

We would like to thank our shepherd, Jorge Ortega Arjona, whose experience was invaluable; every participant of the *Design and Architecture Writers' Workshop*: Ademar Aguiar, Adriana Chis, Alexander Ernst, Atsuto Kubo, Joseph Yoder, Nuno Flores, Peter Sommerlad, Peter Swinburne, Ralph Johnson, Rebecca Wirfs-Brock and Srinivas Rao — it was a great honor for us to be part of this workshop; and to our family, friends and colleagues, particularly Hugo Silva, for their support during this research. We would also like to thank the Portuguese Foundation for Science and Technology and to ParadigmaXis, S.A., for sponsoring this research through the doctorate scholarship grant SFRH / BDE / 33298 / 2008.

6. REFERENCES

[1] K. Altmanninger, G. Kappel, and A. Kusel. Amor–towards adaptable model versioning. *info.fundp.ac.be.*

[2] F. Anderson. A collection of history patterns. *Collected papers from the PLoP'98 and EuroPLoP'98 Conference*, 1998.

[3] M. Arnoldi, K. Beck, M. Bieri, and M. Lange. Time travel: A pattern language for values that change. Jan 2005.

[4] B. Bebel, J. Eder, C. Koncilia, T. Morzy, and R. Wrembel. Creation and management of versions in multiversion data warehouse. *portal.acm.org.*

[5] A. Carlson, S. Estepp, and M. Fowler. Temporal patterns. *Pattern Languages of Program Design*, page 19, Aug 1998.

[6] A. Correa and C. Werner. Applying refactoring techniques to uml/ocl models. *UML*, Jan 2004.

[7] M. Fowler. Analysis patterns: Audit log. http://www.martinfowler.com/ap2/auditLog.html, Accessed on the 1st of May, 2008.

[8] M. Fowler. Analysis patterns: Effectivity. http://www.martinfowler.com/ap2/effectivity.html, Accessed on the 1st of May, 2008.

[9] M. Fowler. Analysis patterns: Temporal object. http://www.martinfowler.com/ap2/temporalObject.html, Accessed on the 1st of May, 2008.

[10] M. Fowler. Analysis patterns: Temporal property. http://www.martinfowler.com/ap2/temporalProperty.html, Accessed on the 1st of May, 2008.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[12] M. Fowler and D. Rice. Patterns of enterprise application architecture. page 533, Jan 2003.

[13] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. pages 37–54. IEEE Computer Society, 2007.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[15] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE: a language for the coupled evolution of metamodels and models. In *MCCM'08 Proceedings*, 2008.

[16] R. Johnson and B. Woolf. Type object. *Addison-Wesley Software Pattern Series*, Jan 1997.

[17] P. Muller, F. Fleurey, D. Vojtisek, Z. Drey, and D. Pollet. On executable meta-languages applied to model transformations. *Model Transformations In Practice Workshop*, Jan 2005.

[18] OMG. Ocl specification 2.0. http://www.omg.org/spec/OCL/2.0/, Accessed on the 12th of December, 2008.

[19] OMG. OMG's metaobject facility (MOF) home page. http://www.omg.org/mof/, Accessed on the 1st of May, 2008.

[20] OMG. Uml infrastructure 2.1.2. http://www.omg.org/spec/UML/2.1.2/, Accessed on the 12th of December, 2008.

[21] Open Source. Prevayler — the open source prevalence layer. http://www.prevayler.org, Accessed on the 12th of December, 2008.

[22] A. Rashid and N. Leidenfrost. Supporting flexible object database evolution with aspects. *Generative Programming And Component Engineering*, Jan 2004.

[23] Ruby on Rails Community. Understanding migrations in ruby on rails. http://wiki.rubyonrails.org/rails/pages/understandingmigrations, Accessed on the 14th of May, 2008.

[24] Ruby on Rails Community. Using migrations in ruby on rails. http://wiki.rubyonrails.org/rails/pages/UsingMigrations, Accessed on the 14th of May, 2008.

[25] H. Wei and R. Elmasri. Schema versioning and database conversion techniques for bi-temporal databases. *Annals of Mathematics and Artificial Intelligence*, Jan 2000.

[26] L. Welicki, J. W. Yoder, and R. Wirfs-Brock. A pattern language for adaptive object models: Part I – rendering patterns. In *PLoP 2007*, Monticello, Illinois, 2007.

[27] L. Welicki, J. W. Yoder, R. Wirfs-Brock, and R. E. Johnson. Towards a pattern language for adaptive object models. pages 787–788, Montreal, Quebec, Canada, 2007. ACM.

[28] J. W. Yoder, F. Balaguer, and R. Johnson. Architecture and design of adaptive object-models. *ACM SIG-PLAN Notices*, 36:50–60, Dec. 2001.