Patterns for Data and Metadata Evolution in Adaptive Object-Models

Hugo Sereno Ferreira[1,2], Filipe Figueiredo Correia[1,3], León Welicki[4]

[1] ParadigmaXis — Arquitectura e Engenharia de Software, S.A.,
Avenida da Boavista, 1043, 4100-129 Porto, Portugal
{hugo.ferreira,filipe.correia}@paradigmaxis.pt
http://www.paradigmaxis.pt/

[2] MAP-I Doctoral Programme in Computer Science
University of Minho, Aveiro and Porto, Portugal
hugo.ferreira@di.uminho.pt
http://www.map.edu.pt/i

[3] FEUP — Faculdade de Engenharia da Universidade do Porto,
Rua Dr. Roberto Frias, s/n 4200-465, Porto, Portugal
filipe.correia@fe.up.pt
http://www.fe.up.pt/

[4] ONO (Cableuropa S.A.)
lwelicki@acm.org

Abstract. An Adaptive Object-Model (AOM) is a dynamic meta-modeling technique where the object model of the system is explicitly defined as data to be interpreted at run-time. It fits the model-driven approach to software engineering. The object model comprehends the 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 for this architectural style. This paper approaches data and metadata evolution issues in the context of AOMs, by describing three additional patterns — History, Versioning and Migration. They establish ways to track, version, and evolve information, at the several abstraction levels that information may exist in an AOM.

Key words: Adaptive object models, AOM, Model driven engineering, MDE, Design patterns, Meta-modeling, Versioning, History, Migration.

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, usually converge into a common architecture style typically known as Adaptive Object-Model (AOM) [1].

The Adaptive Object-Model architecture fulfills particular needs of the several Model-Driven Development methodologies [2], 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.

One of the key aspects of Adaptive Object-Models is their ability to allow changes to the model. 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 different, yet complementary, issues:

Tracking. How to keep track of the changes introduced in the system as it evolves?
Time Travel. How to enable the system to access its past and present states?
Integrity. How to maintain integrity of the run-time models, especially concerning its structure?

This paper presents three domain specific design patterns that have risen from the experience on implementing Adaptive Object-Models, and researching how other systems, particularly Object-Oriented Databases and Version Control Systems, deal with these problems [3], [4], [5]. These patterns aim to contribute to the on-going effort on defining a pattern language for AOMs [6], [7].

Both the VERSIONING and HISTORY patterns have already been identified, though not described, in [6]. Besides describing these two patterns, we also propose a third — MIGRATION — by further decoupling the concerns of model evolution. It is our belief that the synergies between these three patterns provide a more complete understanding of the overall problem of data and metadata evolution.

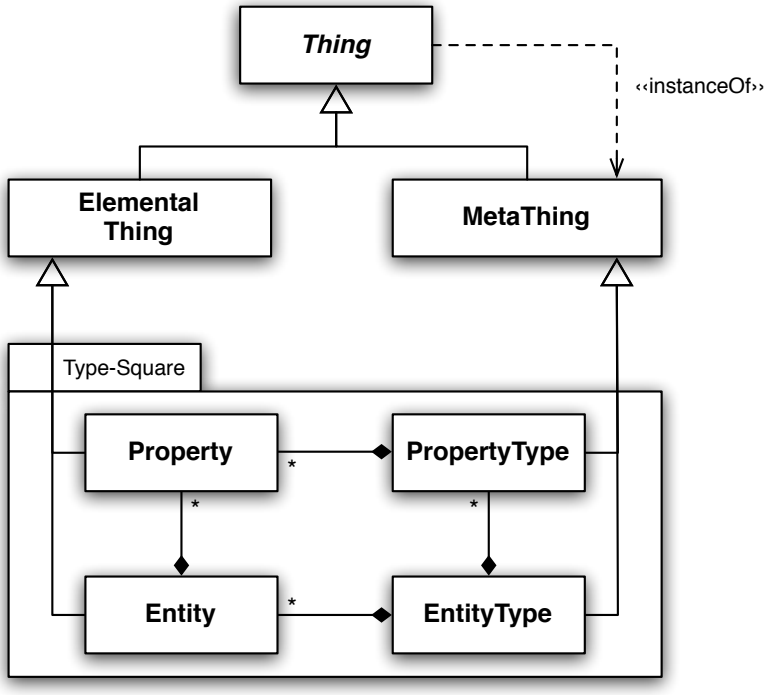

Fig. 1. Extension to the Type-Square pattern. A *Thing* is specialized into an *ElementalThing*, which represents data (i.e. *Entities* and *Properties*) and a *MetaThing*, which represents metadata (i.e. *EntityTypes* and *PropertyTypes*).

1.1 Levels of Abstraction

On the traditional literature on AOMs, two different levels are usually at stake: (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 lowest information level associated with the values of the domain [1]. Classic AOMs can thus be seen as having two explicit and one implicit level. Making the parallel with the nomenclature used by the OMG and their MOF initiative [8], instances of the operational and knowledge levels are equivalent to M_0 and M_1 levels respectively, where M_0 are instances (objects) of M_1 defined classes. M_2 is roughly equivalent to the models presented when defining an AOM. M_2 is implicitly defined using the very same artifacts (classes, methods, properties, etc.) provided by the programming language used to implement the AOM. MOF is a closed meta-modeling architecture, though: while it also defines an M_3-model, this last one is conformant to itself.

While the traditional AOM architecture stop at the M_1 level, nothing restrains system developers to define higher-level models. In this work, generalizing the model-level such that each pattern could be applied regardless of it, pushed us into defining a simple extension to the TYPE-SQUARE pattern [1].

Figure 1 illustrates the concept of a *Thing*. It's here defined as being specialized into either an *ElementalThing* or a *MetaThing*. Any object of type *Thing* is actually an instantiation of a *MetaThing*, thus allowing an unbounded definition of *MetaLevels*. 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 of an object (*EntityTypes*, *PropertyTypes*...). This diagram and all examples discussed in this paper follow the UML 2.1 and OCL 2.1 specification [].

1.2 Data and Metadata Patterns

This paper documents the following three patterns:

History. Addresses the problem of maintaining a history of operations that was taken upon the defined objects.

Versioning. Deals with preserving the several states each object has achieved during the evolution of the system.

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

All patterns further presented are closely related. In fact, MIGRATION is completely dependent on the HISTORY and VERSIONING patterns, and orchestrates the coordination between them, 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. Any developer either working or designing this type of systems, who recognizes the presented forces and problem statements as being part of her systems' functional requirements, will benefit from knowing these patterns.

Nonetheless, developers can still apply them 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.

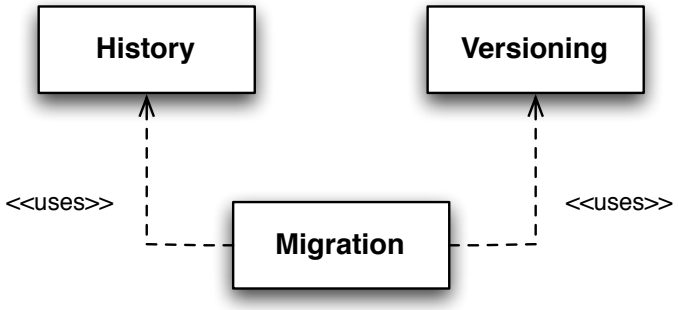

Fig. 2. Dependencies between data and metadata evolution patterns. MIGRATION is dependent upon the concepts of HISTORY and VERSIONING.

2 The HISTORY Pattern

Addresses the problem of maintaining a history of operations that was taken upon the defined 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.

2.2 Example

We are developing an information system based upon the AOM architecture, using a variant of the TYPE-SQUARE pattern [9], [1] that conforms to the previously mentioned design (i.e. every object and instance at any level is considered a *Thing*).

Imagine an insurance company in which its users keep changing information in the the system at a fast pace and there is a high need of keeping track of what, how and probably when and by whom it has been changed. For this example system, this meta-information is as important as the information itself.

Keeping track of the operations history can be useful for auditing purposes, but can also be used for statistical analysis (e.g. number of created instances per user), or controlling user behavior (i.e. without recurring to explicit user access control).

Because we are using an AOM based system, there are several levels at which information can change (data, model, meta-model...). As an example, suppose we have an *EntityType* named *Person* and a particular *Entity* named *John*. The kind of actions we may want to take can be simple CRUD operations (e.g. deleting the *Entity*, or changing its name), or model operations (e.g. adding the attribute *Number of Children*).

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 functions from the log into the objects, and into the user interface, a complex (if at all possible) activity. Further, as the underlying AOM interpreter evolves, the type of operations that should be recorded will also evolve, leaving us with an entangled web of messaging to parse, for which the details we are not interested in. We may also want to delegate some activities to the machine, like finding systematic modifications to the information or performing statistical analysis, as previously mentioned.

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

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

Extendability. We may want to store additional information (e.g. *before* and *after*, *user*, *time*...) along with the actions.

Operation's Semantics. Operations should have enough semantics to allow for machine activities based on them.

Simplicity. The logged operations should be easy to be written, as well as to read.

Maintenability. Correct factoring of responsibilities makes maintenance and evolution easier.

Modifiability. The kind of operations allowed by the system can evolve.

Performance. The design should allow for minimal performance impact regarding system throughput and access times.

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

Consistency. Operations should comply to semantic constraints (consider *pre* and *post* conditions) to validate their concrete execution within a specific context.

Space Consumption. Quantity of additional stored information should be carefully minimized.

2.5 Solution

Encapsulate all the Operations in a hierarchy of commands that operates over Things. A sequence of invoked commands constitutes the History.

Create commands with the responsibility of defining and encapsulating the types of modification (**Operations**) that can be made at any model-level. These **Operations** may be elemental — **Concrete Operations** — or be grouped in a sequence — **Macros** – through the use of the COMPOSITE pattern [10].

Every action taken by the application must occur by instantiating and executing these **Operations**. Creating a HISTORY object is as simple as storing the sequence of the invoked **Operations**. These should fulfill the system's needs in terms of data and model evolution. Each **Operation** will maintain enough semantics in order to be mappable to the **Things** it operates over.

By using the HISTORY pattern, developers can factor the responsibility of creating and storing modifications to the information in a semantically rich way, such that it is easier to evolve the underlying interpreter and to allow machine actions to be based upon this history.

2.6 Example Resolved

Consider five employees from the automobile insurances department, working as a team. During the given week they create and alter information on the system, according to demands external to the department, from clients and according to instructions from the rest of the company.

Namely, they add new policies for the same or different clients, answer to the events of new accident occurrences, and adjust the conditions for new policies.

Fig. 3. Class diagram of the History pattern. A history results from a set of operations done over Things.

On this particular week, the same client record happened to be edited by three different users. Yet, there was an accident occurrence incorrectly registered for that client, and it is now important to understand how it took place, so that it will not happen again in the future. The history of commands that was registered throughout the week now allows to find out what exactly happened, in what the system is concerned. The employees find out the accident occurrence was registered by one particular employee on Tuesday, and they conclude it happened because the client was initially wrongly chosen, has it was selected by searching by his name, when what should have been done was to search him by the respective 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 commands, and gets the idea it was in fact a particularly busy week.

2.7 Resulting Context

After applying this pattern, the following items should be regarded as benefits gained:

- Because every modification is abstracted into an evolution primitive, the history is made simply by storing the sequence of these commands.
- As a side-effect, modifications allowed by the system would be correctly mapped into a hierarchy of commands which would promote further **reutilization** and **easier maintenance**.
- If enough information is stored with each evolution primitive, the system can allow, at a later time, playback of the executed operations.
- Centralizing all allowable changes to the system, **simplifies** the control of semantic/constraints checking, specially regarding **security** issues.

However, the balance of forces result in 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.
- The **performance** may be affected because of the quantity of instantiated objects and the necessary pointer dereferencing/set joins associated with particular implementations.
- The **complexity** of implementation is considerably higher.
- The **resource management** of operation instances becomes more complex, specially when dealing with garbage-collection concerns.

2.8 Implementation Notes

Semantic Consistency. The semantic consistency of data and meta-data can be kept by enforcing the constraints defined at an upper abstraction level (ie, constraints applicable to M0-level data are defined at the M1 abstraction level, those applicable to M1-level data are defined at the M2 abstraction level, and so on). As described before, this enforcement of constraints can be done along with each operation, as *pre* and *post* conditions, but this may not be enough.

Keeping operations as general as possible will leverage their reusability and maintainability, but leads to operations of a lower granularity. The use of CRUD-like operations is a good example of this, as they focus on very straight forward 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. As an example, 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, connected through that same one to one relation. Suppose now we would like to replace b by a new instance b', as the other end of the said relation. If we consider only CRUD-like operations, two different operations would be needed: the deletion of the relation between a and b, and the creation of a new one, connecting a and b'. By the end of these two operations information would be in a consistent state, but that would not be the case when first operation completes, and the second hasn't yet started, as a relation, which is mandatory, would not exist.

As described before, the COMPOSITE pattern can be used to group **Operations** in sequence, creating **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, consistency can be checked at the end of macro in which they are enclosed.

2.9 Related Patterns

Operations are structured using the COMMAND pattern [10]. The hierarchy of operations are also related to the COMPOSITE pattern [10]. Registration of information is done similarly to the AUDITLOG pattern [11], but in a semantically richer way.

2.10 Known Uses

The HISTORY Pattern can be found in any system that possesses an undo mechanism. Some image editors, for instance, make available to the user the complete sequence of operations, from the latest one performed to the first. This allows to undo operations, and so to reach previous versions of the document. Object-Oriented Database Management Systems also use variants of this pattern.

3 The VERSIONING Pattern

Deals with preserving the several states each object has achieved during the evolution of the system.

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 [9], [1] that conforms to the previously mentioned design (i.e. every object and instance at any level is considered a *Thing*).

Imagine an 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 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 500AC and 200DC.

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 since. Yet, through acquired experience, users have now found the need to also 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 model-level (i.e. data and meta-data).

Decoupling. We don't want to pollute the system with versioning logic wherever it is needed.

Identity. Although several existences of an object co-exist simultaneously, one should be able to reference the object regardless of its latest version.

State. Although the identity of an object is maintained along the evolution of a system, one should be able to reference one of it's particular versions.

Semantics. Each version should group a semantically valid set of states, preserving data and meta-data integrity.

Globality. A version should be global to the whole system.

Time Independence. Past information should be kept in a way that allows recovery of as much of past versions as possible.

Space Consumption. The data-set at hand should be kept at manageable sizes.

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

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.

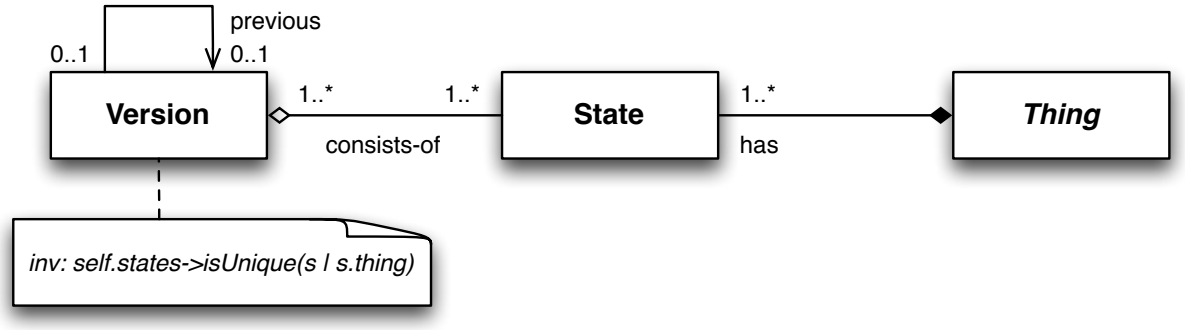

Fig. 4. Class diagram of the Versioning pattern. Each version consists on a collection of *States* from *Things*. Each version may only refer a single *State* of any particular *Thing* (i.e. there cannot be multiple states of the same thing in the same version).

Applying this pattern usually starts by decoupling **Things** from their **States**, as usually these two concepts appear as one in most systems [12]. While **Things** represent the identity of an object, **States** represent its content, which will evolve over the use of the system's information. A **Version** is merely a way of capturing the global state of the system, by referencing all the **States** at some point of the system's lifetime. Each **Version** also maintains references to the ones that gave origin to it, and to the ones that it gave origin to. Usually each **Version** will only be based on one single previous **Version**, and will give origin to one single other **Version** too, 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 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**, any **Version** need to be able to reference **States** from both levels. In fact, this is an essential issue for the MIGRATION pattern.

3.6 Example Resolved

Consider the *Survey of the Coliseum* previously described. Over the last year new information were acquired about the Coliseum, 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 has evolved over time. Conceptually, several **Versions** have been created, each representing a consistent point on the evolution of the information available. It is now possible to use the system to consult information as it was on any of these points, and even to recover previously deleted data.

Eventually, the model may also need to evolve. For example, a new attribute may be added to accommodate the name of the leader of each archeological expedition. The attribute itself is also a *Thing*, which means a new **Version** will be created that references model-level *States* and *Things*.

3.7 Resulting Context

After applying these pattern, the following items should be regarded as benefits gained:

- We are now able to use the same mechanism — **reusability** — regardless of the model-level.
- By **decoupling** the state from the object, we are able to preserve the object's **identity**.
- The concept of version is now at system level — **globality** — instead of object-level.
- **Concurrency** can be coped more easily through the use of versioning.

However, the balance of forces result in the following liabilities:

- The quantity — **space consumption** — of stored information can be daunting, if enough care isn't taken into choosing persistency strategies.
- The branching of versions, and further merge, will require complex mechanisms to deal with concurrency.
- **Performance** may be affected by the higher quantity of stored information.
- The versioning mechanism itself introduces additional **complexity**.

3.8 Implementation Notes

Coping with state explosion. While the model described above conceptually illustrates this design pattern, the reader may have already noted it can lead to an unnecessary growth in the data 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. Because this issue can determine the feasibility of a system, we present some notes overviewing this particular solution.

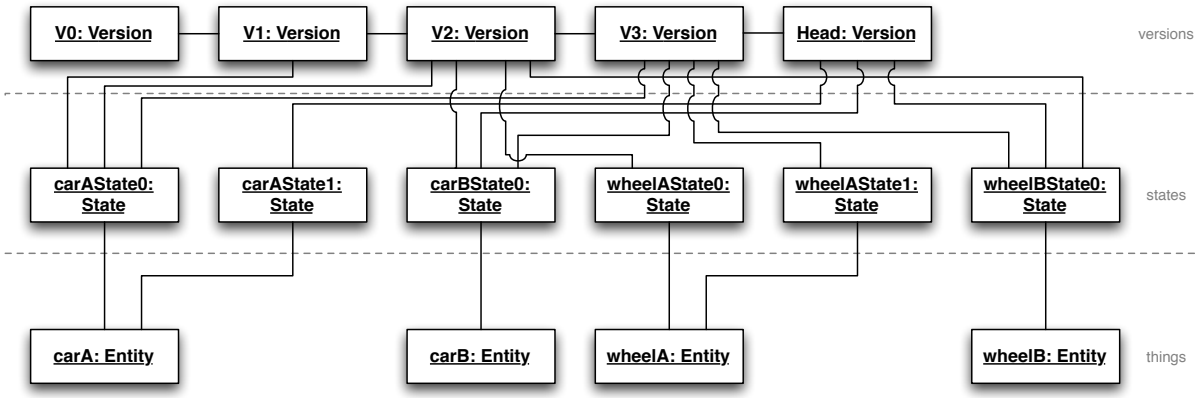

Fig. 5. Object diagram for an example instantiation of the VERSION pattern.

Let us define S_n^t as the n^{th} state of a particular thing (t), V_m as the m^{th} version defined, and V_H as the latest version. The evolution of any particular system is here defined as the set of versions V_m, where $0 \leq m \leq V_H$. Each V_m is a set of S_n^t, where each t is unique. The semantics of V_m is thus the set of the most actual, valid states of every existing thing in the m^{th} version.

Consider thus the following 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 (depicted as an object diagram in **Figure 6**) is the following:

$$
\begin{aligned}
V_H &= \{S_1^{carA}, S_0^{wheelB}, S_0^{carB}\} \\
V_3 &= \{S_0^{carA}, S_1^{wheelA}, S_0^{wheelB}, S_0^{carB}\} \\
V_2 &= \{S_0^{carA}, S_0^{wheelA}, S_0^{wheelB}, S_0^{carB}\} \\
V_1 &= \{S_0^{carA}\} \\
V_0 &= \emptyset
\end{aligned}
$$

Any thing that doesn't change its state in any subsequent version, would have its state replicated across those versions. Hence, we can devise a scheme where only changes to states are stored, from where we can infer the complete set of states for any version. The example above would then become:

$$
\begin{aligned}
\Delta_H &= \{S_1^{carA}, S_-^{wheelA}\} \\
\Delta_3 &= \{S_1^{wheelA}\} \\
\Delta_2 &= \{S_0^{wheelA}, S_0^{wheelB}, S_0^{carB}\} \\
\Delta_1 &= \{S_0^{carA}\}
\end{aligned}
$$

Where any state S_n^t may only occur in a single Δ. For every $m \geq n$, a function that converts Δ_m to V_m can be constructed by applying the following rules (a) any state S_n^t that belongs to Δ_n, also belongs to V_m, (b) if S_n^t belong to V_m then S_{n-1}^t doesn't belong to V_m, and (c) if S_-^t (null state) belongs to Δ_n then no S^t belongs to V_m. In english, these rules could 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.** This strategy results in a much smaller set of object relations as seen in **Figure 7**.

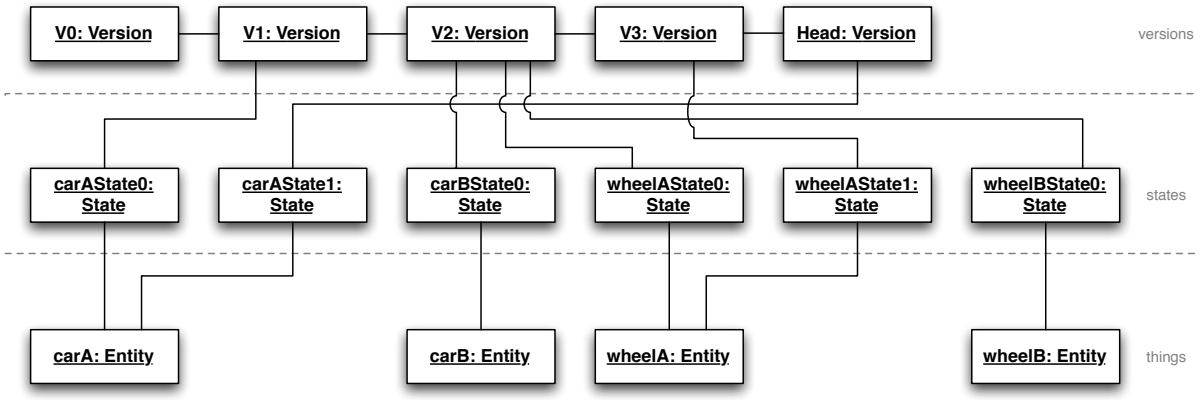

Fig. 6. Object diagram for the *delta* strategy instantiation of the VERSION pattern.

3.9 Related Patterns

The Temporal Property Pattern [13] addresses the issue of answering requests for a previous value of a given property. However, it does not explicitly account for the information of what was the global state of the system at a given point, like the Versioning Pattern.

The Effectivity Pattern [14] consists on marking an object with a time period, in which it is effective.

The Memento Pattern [10] consists on decoupling the identity of an object from its state, but do not address the concern of *Globality*.

The Temporal Object Pattern [15] addresses the issue of answering requests for a previous value of a given object. It differs from the Versioning Pattern in that it does not explicitly account for the information of what was the global state of the system at a given point (see *Globality* force).

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

Also check the Identity pattern [12].

3.10 Known Uses

This pattern is common on wikis and source Control Management systems. The work presented in [16] also details the implementation of several versioning techniques.

This pattern is also common in *Object-oriented Database Management Systems (OODBMSs)* and *Data Warehouses* [5], [4].

4 The 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 there is the need to evolve the model and data definition while maintaining consistency.

4.2 Example

We are developing an information system based upon the AOM architecture, using a variant of the Type-Square pattern [9] that conforms to the previously mentioned design (i.e. every object and instance at any level is considered a *Thing*).

Suppose we have an information system for an insurance company based on the mentioned architecture. 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 profound impact on the system. Valid objects may actually depend on the information we are changing,

thus leading to a system which is inconsistent. These issues need to be addresses upon each evolution step, to guarantee that the integrity of the system holds, and that it remains valid according to the specifications.

Another typical concern relates to maintaining legacy application programming interfaces (APIs). If our system must interoperate with third-party components, once the model definition evolves, the API can become invalid. In this case, the need to provide a layer of data transformation, which is able to maintain legacy communications over previous versions of models, adds to 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 and Versioning, the Migration pattern is subject to the same forces. Some additional forces specific to this pattern are presented below:

Simplicity. 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.
Integrability. 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 States, such that it becomes possible to achieve a new Version by applying the sequence of Operations in the history.

As has been described in the History and Versioning 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 [10], but instead of operating over **Things**, operations should occur over (or, generate new) **States**. Considering a one to one relation between the **History** and **Version** classes (see History and Versioning patterns), the later can fulfill both roles.

Like described in the History pattern, 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 [10].

The concept of an **Operation** spawning other **Operations** may allow the manipulation of metadata to generate operations at lower-levels (i.e. data) whose purpose is to maintain the consistency of the system. For example, a *Move Attribute to Superclass* operation at the model level would generate several operations at the data level, since data would also need to be moved.

The Migration act as an interpreter or patch engine which, by consulting the information stored at any given set of **Versions**, both by looking at their **States** and the respective sequence of **Operations**, would allow one to "migrate" from any source to target **Version** of the whole system.

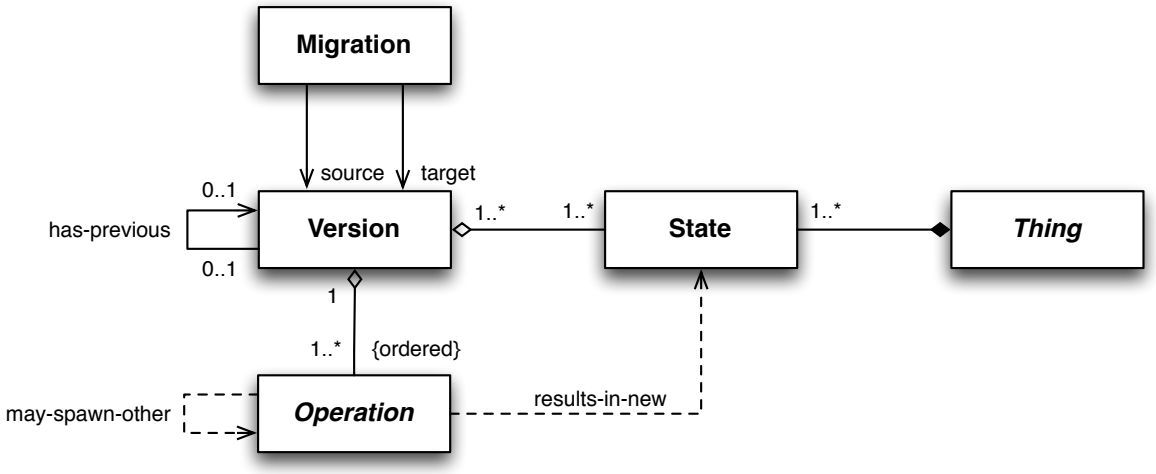

Fig. 7. Class diagram of the Migration pattern. A Migration between any two versions consists on applying, in the correct order, all the histories of operations that were executed between those versions. A single version may only refer a single state of any particular thing.

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.

For example, let us imagine that an insurance company is going to evolve its model because of new laws. Some fields are now needed to be created in existing entities (e.g. *number of dependent children*). Some fields, previously belonging to a particular sub-class, will now be moved into the super-class (e.g. *number of days oversea per year*).

Consider the system is at version V_1, and is going to be evolved to version V_2. Two M_1 operations (i.e. they occur at the model-level) are issued, *Create Attribute* and *Move Attribute to Superclass*. While the former doesn't need to spawn any M_0 (i.e. data-level) **Operation**, the later can be composed 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 history of commands, interweaving different level operations, are all stored upon the history of 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

Since this pattern is a combination between the HISTORY and VERSIONING patterns, the resulting context includes the combination of them both. Nonetheless, the following items should be regarded as benefits gained specifically by the use of this pattern:

– We are now able to 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. If they are correct, it can help to move towards a *correct by construction* evolution (though this would need proof based on the formal semantics of each concrete operation).

However, the balance of forces result in the following liabilities:

– If the system does not provide enough operations to perform complex tasks, it can be hard to express the intended semantics of the evolution.
– The migration mechanism, along with all the additional information that it requires, brings extra **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 [17]. The concept of refactoring applied to models (e.g. UML) have already been pointed out as a way to cope with system evolution. In the same way these works have shown that complex system evolution could be accomplished by composition of these primitives, we suggest this notion can be extrapolated into the **Operations**, such that they would represent a set of refactorings specifically designed for Adaptive Object-Models.

4.9 Related Patterns

All related patterns to History and Versioning also apply here.

4.10 Known Uses

This pattern is common for addressing **Schema Versioning** in *Object-oriented Database Management Systems (OODBMSs)* and *Data Warehouses* [5], [4].

The *Ruby on Rails (RoR)* framework uses the Migration concept [18], [19]. There are, however, some differences since RoR is based on the Active Record Pattern [12] and, as such, expresses operations within relational models, instead of object-oriented models as is here considered.

References

1. Yoder, J.W., Balaguer, F., Johnson, R.: Architecture and design of adaptive object-models. ACM SIG-PLAN Notices **36** (December 2001) 50–60
2. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap, IEEE Computer Society (2007) 37–54
3. Rashid, A., Leidenfrost, N.: Supporting flexible object database evolution with aspects. Generative Programming And Component Engineering (Jan 2004)
4. Wei, H., Elmasri, R.: Schema versioning and database conversion techniques for bi-temporal databases. Annals of Mathematics and Artificial Intelligence (Jan 2000)
5. Bebel, B., Eder, J., Koncilia, C., Morzy, T., Wrembel, R.: Creation and management of versions in multiversion data warehouse. portal.acm.org
6. Welicki, L., Yoder, J.W., Wirfs-Brock, R., Johnson, R.E.: Towards a pattern language for adaptive object models, Montreal, Quebec, Canada, ACM (2007) 787–788

7. Welicki, L., Yoder, J.W., Wirfs-Brock, R.: A pattern language for adaptive object models: Part I – rendering patterns. In: PLoP 2007, Monticello, Illinois (2007)
8. OMG: OMG's metaobject facility (MOF) home page. http://www.omg.org/mof/, Accessed on the 1st of May, 2008.
9. Johnson, R., Woolf, B.: Type object. Addison-Wesley Software Pattern Series (Jan 1997)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Professional (1995)
11. Fowler, M.: Analysis patterns: Audit log. http://www.martinfowler.com/ap2/auditLog.html, Accessed on the 1st of May, 2008.
12. Fowler, M., Rice, D.: Patterns of enterprise application architecture. (Jan 2003) 533
13. Fowler, M.: Analysis patterns: Temporal property. http://www.martinfowler.com/ap2/temporalProperty.html, Accessed on the 1st of May, 2008.
14. Fowler, M.: Analysis patterns: Effectivity. http://www.martinfowler.com/ap2/effectivity.html, Accessed on the 1st of May, 2008.
15. Fowler, M.: Analysis patterns: Temporal object. http://www.martinfowler.com/ap2/temporalObject.html, Accessed on the 1st of May, 2008.
16. Arnoldi, M., Beck, K., Bieri, M., Lange, M.: Time travel: A pattern language for values that change. (Jan 2005)
17. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
18. Ruby on Rails Community: Understanding migrations in ruby on rails http://wiki.rubyonrails.org/rails/pages/understandingmigrations, Accessed on the 14th of May, 2008.
19. Ruby on Rails Community: Using migrations in ruby on rails http://wiki.rubyonrails.org/rails/pages/UsingMigrations, Accessed on the 14th of May, 2008.