Design for an Adaptive Object-Model Framework

An Overview

Hugo Sereno Ferreira[1,2], Filipe Figueiredo Correia[2], and Ademar Aguiar[1,2]

[1] INESC Porto

[2] Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
{hugo.sereno, filipe.correia, ademar.aguiar}@fe.up.pt

Abstract. The Adaptive Object-Model (AOM) architectural pattern has been significantly documented in literature, but there is not yet enough documentation explaining how to design and build a full AOM-based system. A AOM framework would need to address an additional number of issues that go well beyond individual software patterns. In this paper, we propose a design for a AOM framework that addresses several issues of building AOM-based systems, namely: integrity, run-time co-evolution, persistency, user-interface generation, communication and concurrency. We borrow concepts from distributed version-control systems. We show how applications based on a concrete realization of this framework, called Oghma, helps to avoid a traditional two-level domain classification, reduces accidental complexity, and directly exposes confined model evolution to the end-user.

Key words: Adaptive Object-Models; Frameworks; Software Design

1 Introduction

The industrialization of software development has been increasingly faced with the growth of software complexity. A considerable effort in development is also repeatedly applied to the same tasks, despite all the effort in research of reuse techniques and good practices, thus suggesting that reuse may need to be performed even more and at higher-levels. Hiding these inherent complexities of technological concerns by creating abstractions as been a recurrent reaction, at the cost of widening the existing gap between specification and implementation artifacts [9]. To make these abstractions useful not only for modeling, documentation, analytical and reasoning purposes [5, 12], models have to be made executable, by systematic transformation [15] or interpretation [14] of problem-level abstractions (i.e. specifications) into implementations (i.e. algorithms).

1.1 System Variability and Evolution

The difficulty of acquiring, inferring, capturing and formalizing software requirements is a recurrent problem in software engineering. This is because not only those processes are dependent upon the stakeholders' perspectives, but also because requirements often change faster than implementations. Since evolving software requires a considerable effort, as the implementation progresses, a strong resistance against changing the requirements is developed. What seems to be dismissed is that not only some business domains rely on constant adaptation of their own processes, but also that new knowledge is incrementally acquired as development unfolds, leading to new insights and expectations from software. If it is known *a priori* that the systems being developed are incomplete by design, won't there be benefits in designing for incompleteness? [11] Even so, current practices often focus on quick functional change, disregarding conceptual design, leading to a Big Ball of Mud, and eventually facing total reconstruction with a significant impact in economy [8]. Successful software needs to increase its resilience to change [13].

1.2 Framework for Adaptive Object-Models

Approaches to the use of models have traditionally been generative, automatically refining models into code artifacts during development. However, such techniques are based upon two premises: (a) that changes are always introduced by developers, within the development environment, and (b) that a full compile cycle (e.g. shutting down the system) is affordable. When these premises fail to hold, generative approaches may reveal insufficient, thus leading to the use of runtime domain models [14]. The systematic search for higher levels of abstractions — both to improve analysis and increase reuse — associated with the pervasive adoption of object-oriented models, converged to a common architectural style called the Adaptive Object-Model (AOM) [19, 18], which is founded on a growing collection of AOM-related patterns [7, 17, 16]. As patterns, they usually occur not as reusable components, but as perceived abstractions within the design of each particular system. In this paper, whenever referring to a pattern we use a SmallCase typographical style.

Frameworks are both reusable designs and implementations, that orchestrate the collaboration between core entities of a system. While they establish part of the system's behavior, they are deliberately open to specialization by providing hooks and specialization points. The framework dictates the architecture of the underlying system, defining its overall structure, key responsibilities of each component, and the main thread of control. It captures design decisions common to its application domain, thus emphasizing design reuse over code reuse. Patterns differ from frameworks because (a) are more abstract, (b) have smaller architectural elements, and (c) are less specialized [10].

Section 2 will provide a general overview of a framework which supports the development of AOM-based systems, and then proceed to detail each concern independently. Section 3 focus on its use within industry. Section 4, will draw some conclusions and present remaining issues to be addressed in the future.

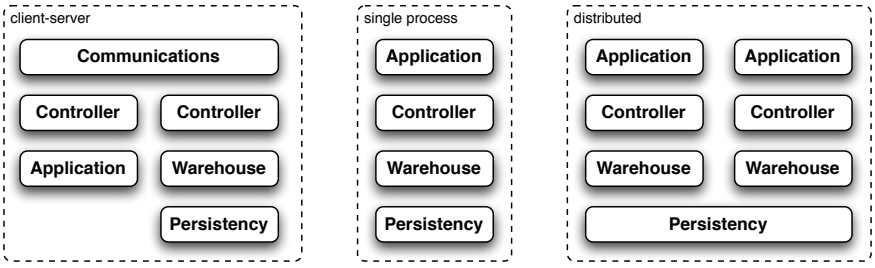

Fig. 1. Three possible component configurations of the framework: (a) client-server, where several processes are controlled by a centralized server, (b) single-process, only allowing a single running application, and (c) distributed, which takes advantage of the data-replication mechanisms from the underlying persistency engines.

2 Oghma: Architecture and Design

Oghma, which components are depicted in Fig. 1, is a framework to develop AOM-based systems, that balances adaptability and reuse. It supports the creation of models resembling MOF [1] and UML [2], and aims at covering the entire cycle of system creation and evolution. It also allows the introduction of changes to the system during runtime, thus providing a particular kind of confined end-user development.

Furthermore, the framework leverages the infrastructure used to support system evolution to provide additional features, such as auditing over the system's usage, and *time-traveling* to an arbitrary point along its evolution (i.e. to set the system in a past state).

Oghma includes a set of interchangeable components designed to have an high degree of flexibility — it supports several types of persistency engines, including relational, object-oriented, key-value and document-oriented, and architecture styles, such as single-process, client-server, and distributed.

2.1 Core — Structural

Fig. 2 depicts the design of the structural core of Oghma, resembling the TypeSquare [19] pattern: (a) `ObjectType`, which is refined into `Entities` (that represent classes) and `Interfaces`, (b) `Instance`, which complies to a given `Entity`, (c) `PropertyType`, which is refined into `RelationNodes` and `Attribute-Types`, and (d) `Property` which complies to its `PropertyType`. Each `Relation-Node`, besides specifying cardinality, navigability and role, must be connected to another node, thus establishing a `RelationType`. In order for a `RelationType` to have properties (similar to the *Associative Class* in UML) it can relate to an `Entity`. `Entities` can also inherit from other `Entities` and/or multiple `Interfaces`. Model-defined `Entities` and `Instances` can be made *Types* and *Objects* of the underlying programming language through the use of Plugins.

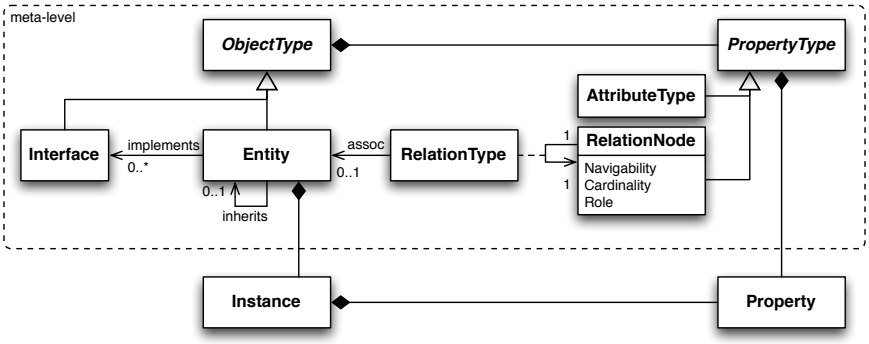

Fig. 2. Core design of the structural meta-model.

2.2 Core — Behavioral

Fig. 3 depicts `Expression` as the central concept of the behavioral core, extending the Rule Object pattern [17]. Expressions are stated in a *Domain Specific Language* and may be evaluated using an Interpreter [10] or a *Virtual Machine*. They're widely used to define: (a) `ObjectType` invariants, (b) derivation rules in `PropertyTypes` and `Views`, (c) body of `Methods`, (d) guard-conditions of `Operations`, etc. As such, they play an important role in assuring semantic integrity during model evolution (see Section 2.7). Structural rules, such as the cardinality and uniqueness of a `PropertyType` are translated to `ObjectType` invariants. `Methods`, which are used-defined `Batches` of `Operations`, may be invoked manually, or triggered by `Events`, thus allowing the specification of *State Machines*.

2.3 Controller

As seen in Fig. 1, the Controller serves as a entry layer for the GUI and Communications components, and it's key responsibility is to orchestrate the several other components in the framework by establishing a thread of control. It bootstraps the system by loading the meta-model, and the necessary versions of the domain-model from the Warehouse. It manages data requests by interacting with the Warehouse. It also provides several *hooks* to the framework through Chains of Responsibility and Plugins (e.g. interoperability with third-party systems by allowing subscribers to intercept requests).

2.4 Warehousing and Persistency

Because of the evolutive nature of the model, mapping to a classic relational database through the use of ORMs complexifies co-evolution. Warehousing (Fig. 1) hides the details of persistency from the Controller, exposing and consuming data and meta-data (i.e. `Things`), and managing versioning (i.e. through

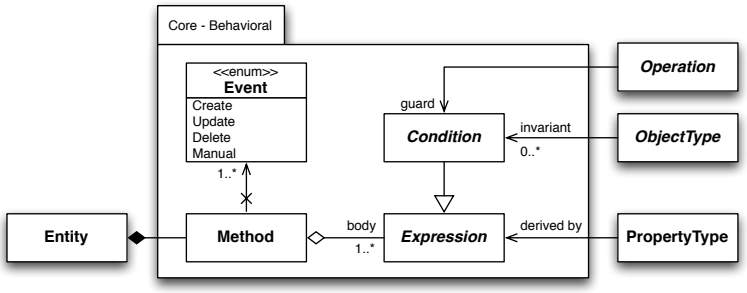

Fig. 3. Core design of the behavioral meta-model.

`Versions` and `States`). Its behavior can be extended and modified through inheritance and composition, as by the DECORATOR [10] pattern. Transient memory-only, direct data-base access, lazy and journaling strategies (e.g. using CACHES [10]) are just a few examples of existing (and sometimes simultaneous) configurations. Also, `Things` are always regarded as opaque, key-valued objects.

2.5 Communications

This design allows to assemble several types of communication stacks. If a client-server HTTP-based stack is chosen, Oghma currently provides a RESTful API for communication between the `Server Controller` and `Client Controller` through a pair of `HTTP Bridge` and `Server Dispatcher` acting as PROXIES [10]. Every `Thing` is addressable by its unique identifier as a resource. The contents of `States` and `Changesets` are serialized in XML. Simple queries can be expressed directly in the URL; those more complex require POST methods.

By specialization of the communication layer, other types of technology can be used for bridging the controllers (e.g .NET Remoting). For example, in the case of a single-user stack, the `Client Controller` would interact directly with the `Server Controller`. A different approach from the client-server architecture is to use distributed key-value databases (e.g. CouchDB), to handle both persistency and communication. Here, every application would assume direct access to the Controller, delegating the responsibility of disseminating contents to the underlying data warehouse.

2.6 Integrity

The **Structural Integrity** of the run-time model is asserted through rules stated in the meta-model. For example, `Instances` should conform to their specified `Entity` (e.g. they should only hold `Properties` which `PropertyType` belong to its `Entity`). Nonetheless, evolving the model may corrupt structural integrity, such as when moving a mandatory `PropertyType` to its superclass (e.g. if it doesn't have a default value, it can render some `Instances` non-compliant).

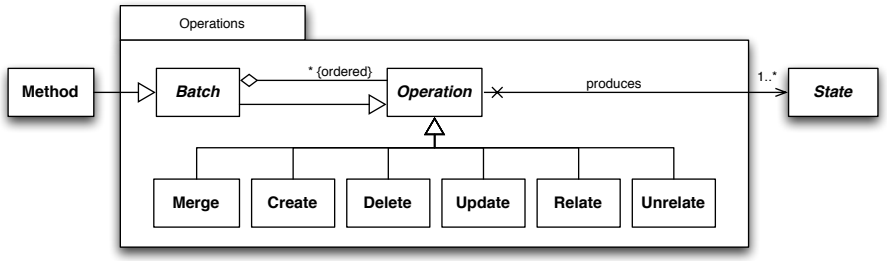

Fig. 4. Data and meta-data are manipulated through `Operations`, similarly to the COMMAND [10] pattern, which produces new `States` of `Things`.

Some model evolutions can be solved by foreseeing integrity violations and applying prior steps to avoid them (e.g. one could first introduce a default value before moving the `PropertyType` to its superclass).

Some steps of a particular evolution may also violate model integrity, although the end result would be valid. For example if a `PropertyType` is mandatory, one cannot delete its `Properties` without deleting itself and vice-versa. This problem is solved by the use of `Changesets`, and only enforcing integrity at the end.

Semantic Integrity, on the other hand, is much harder to ensure since it's not encoded as rules in the meta-model. One cannot just look to the results of an arbitrary evolution and infer the steps which have lead to it. Consider the scenario where an `AttributeType age` is renamed to `date-of-birth`, recalculated according to the current date, and moved to it's superclass `Person`. Would we rely on the direct comparison of the initial and final models, a possible solution would be to delete the attribute `age` in `Employee` and create the attribute `date-of-birth` in `Person`. However, the original meaning of the intended evolution (e.g. that we wanted to store birth-dates instead of ages) would be missed. To solve this problem, Oghma makes use of MIGRATIONS [7], providing and storing sequences of model-level operations that cascade into instance-level changes.

2.7 Evolution

Allowing collaborative co-evolution of model and data by the end-user introduces a new set of concerns not usually found in classic systems. They are (a) how to preserve model and data integrity, (b) how to reproduce previously introduced changes, (c) how to access the state of the system at any arbitrary point in the past, and (d) how to allow concurrent changes. These concerns can be summarized into traceability, reproducibility, auditability, disagreement and safety, and are commonly found on version-control systems.

Typically, *evolution* is understood as the introduction of changes to the model. Yet, the presented design doesn't establish a difference between changing data or meta-data; both are regarded as *evolutions* of `Things`, expressed as

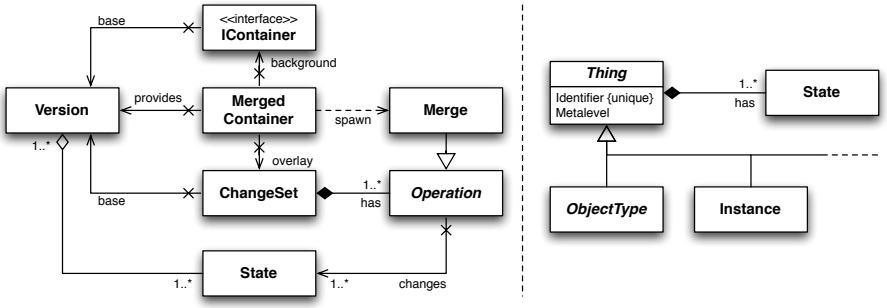

Fig. 5. (a) Merging mechanism used to validate and apply operations stored in a `Changeset`. (b) Data and meta-data are both viewed as `Things` and `States`.

`Operations` over `States`, and performed by the same underlying mechanisms as depicted in Fig. 4. To provide enough expressivity such that semantic integrity can be preserved during co-evolution, model-level `Batches` operate simultaneously over data and meta-data.

Sequences of `Operations` are encapsulated as `ChangeSets`, following the History of Operations pattern [7], along with meta-information such as user, date and time, base version, textual observations, and data hashes. Whenever the framework validates or commits a `ChangeSet`, the Controller uses the merge mechanism depicted in Fig. 5 (similarly to the System Memento pattern [7]), which dynamically overlays the modifications onto the base version by orderly applying each `Operation`, allowing for behavioral rules to be evaluated, and finally resulting in a new version.

If all changes to the data and model are preserved, one can easily recover past information. This not only solves the aforementioned issues, but it also brings to information systems the same notions of versioning that changed the scenario of collaboration in *wikis* and software development.

2.8 User-Interface

Oghma provides a run-time adaptive UI, by inspection and interpretation of the model and using a set of pre-defined heuristics and patterns. While detailing every heuristic applied is outside the scope of this paper, an overview is provided.

A set of grouped entry-points are presented to the user through the GUI. Groups correspond to `Packages` and entry-points to selected `ObjectTypes`. When choosing an entry-point, a list of the associated instances is presented, showing several details in distinct columns, inferred from special annotations made in the model; along with generic search mechanisms. `ObjectTypes` have two default views: *edition* and *visualization*. During edition, Oghma uses heuristics to render each `Property` by inspecting the cardinality, navigability and role of both nodes of a `RelationType`. The result is a different input panel, according to the

property in question: text-fields, text-areas, combo-boxes, tree-views, lists, embedded forms, etc.. Visualization is defined using `Views` — virtual `Instances` where every `Property` is derived — and are subsequently transformed through templates before being presented to the user. Mechanisms as clipboard (using object identifiers), undo and redo (using `Operations`) are orthogonally supported. Custom panels, either for special types (e.g. dates) or model-chunks (e.g. user administration), can also be loaded as PLUGINS.

The user workflow resembles those when using version-control systems. User changes are not immediately applied; instead, `Operations` are stored into the user `Changeset`, and sent to the `Server Controller`, allowing it to assert integrity and provide feedback on behavioral rules. When a user chooses to, it can *commit* its work to the server, by reviewing the list of `Operations` and additionally submitting a descriptive text about his work.

Awareness is also addressed through several feedback techniques: graphics that show the history of changes either in a particular `ObjectType`, by user or globally; alerts to the user for simultaneous pendent changes in the same objects, and presenting reconciliation screens when conflicts are detected; etc.

3 Oghma in the Industry

Oghma has been implemented in C# using the Microsoft .NET Framework v3.5, although the design here presented doesn't depend on a specific technology.

3.1 Use Cases

Oghma was already used to create production-level applications: (a) *Locvs*, an Information System for Management of Architectonic and Archaeological Heritage, and (b) *Zephyr*, a tool for document records management [3, 4]. Of particular interest is *Locvs*, whose domain model currently consists of more than 300 classes, and has gone through more than 1000 model versions, 12k instance-level commits, and 200k `Things`, throughout approximately 2 years of usage and evolution. It is deployed in dozens of machines using a client-server architecture. Performance tests have lead it to currently use SQLite as a storage and Full-Text Search engine. A custom-made DSL for specifying behavior was also implemented. This application will be a valuable asset to further research the role of AOMs in the development of large-scale information systems.

3.2 Lessons Learned

The development and usage of Oghma targeting adaptive applications allows us to elicit some lessons. First, the skills needed to deal with this type of architecture aren't trivial to find, and developers are not necessarily at ease to work at these levels of abstraction. From a framework standpoint, there's also a thin balance between a framework that makes the creation of new systems a quick and easy process, and one that is flexible enough to cover a wide scope of systems.

Because it's very tempting to make the framework address all use-cases using an adaptive and model-driven approach, there is a risk of the final models becoming as elaborate and complex as a full-blown programming language. In this sense, hooks are a key issue, as they are not always easy to foresee, but they establish the border line between what should be regarded as part of the framework and what is particular behavior of a specific instantiation.

Nonetheless, the conduction of small model experiences seem to show it's easy to quickly build a functional prototype which can be shown to the customer, thus providing very early feedback before refining it into a production-level application. Not only the costumer involvement in this process is also increased due to the end-user development capabilities offered by the framework, but it also reduces the burden of up-front design by allowing an incremental approach to formalization of the underlying business model.

4 Conclusions

Adaptive Object-Models and application frameworks are both solutions for a common problem: to increase software reuse. They try to minimize the effort of developing and evolving a software system. A AOM is a meta-architecture for domain variability; frameworks focus on providing code and design reuse. In this paper, they are combined and presented as a conceptual framework design which allows the creation of AOM-based systems, along with some details of a particular implementation being used within an industrial environment: *Oghma*.

Leveraging the concept of adaptability, end-users are empowered to introduce (confined) changes to the model at run-time. This choice raised several issues, such as traceability, reproducibility, auditability, disagreement and safety, which were addressed in the framework by borrowing concepts from distributed version-control systems. One of the side-effects of unifying data and meta-data evolution was that the classical two-level domain classification, where *types* are static entities, is diluted, thus reducing accidental complexity of applications [6]. For example, a user can edit an *enumeration*, or add a new *specialization* of a class, by directly editing the model, hence preserving the classification levels.

Yet, several open-issues remain to be addressed. While automatic run-time generated user-interfaces may not be on pair with custom-made ones regarding usability, they seem to be consistent and based on a strict set of metaphors, supporting a quick learning process by users. Nonetheless, which mechanisms should the framework provide to improve usability and customization of GUIs, while retaining the capability of automatically generating them?

Furthermore, no studies on the performance, robustness, usability, evolvability, maintainability, consistency, composability, scalability and several other software quality attributes regarding Adaptive Object-Models, in comparison to classical systems, were published yet, and even less regarding AOM frameworks.

In the design proposed in this paper, we've addressed data and meta-data evolution through an unified architecture. However, the whole framework depends on the definition of a well-known meta-model. Operations that support

model evolution are thus dependent on this definition. How often would the meta-model change, and how can we easily cope with its evolution? Should the abstraction be raised yet another level, or is a self-compliant meta-model enough to provide mechanisms for its own evolution?

Finally, this paper provides only an overview of the framework. There are several issues that should be taken into account when implementing it. Such details are outside of the scope of the work developed so far, but they are expected to be addressed in future work.

5 Acknowledgments

We would like to thank both *FCT* and *ParadigmaXis, S.A.* for sponsoring this research through the grant SFRH / BDE / 33298 / 2008.

References

1. MOF version 2.0. http://www.omg.org/spec/MOF/2.0/, Accessed on 2009/08/06.
2. UML version 2.2. http://www.omg.org/spec/UML/2.2/, Accessed on 2009/08/06.
3. Locvs. Technical report, ParadigmaXis, S.A. produced to CMP, 2009.
4. Zephyr. Technical report, ParadigmaXis, S.A. produced to CMP, 2009.
5. J. Arlow, W. Emmerich, and J. Quinn. Literate modelling—capturing business knowledge with the uml. *The Unified Modeling Language: UML'98*, 1999.
6. C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. 2008.
7. H. Ferreira, F. Correia, and L. Welicki. Patterns for data and metadata evolution in adaptive object-models. *Proceedings of the 15th Conference on PLoP*, 2008.
8. B. Foote and J. Yoder. Big ball of mud. *PLoP'00*, 2000.
9. R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. *International Conference on Software Engineering*, Jan 2007.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.
11. R. Garud, S. Jain, and P. Tuertscher. Incomplete by design and designing for incompleteness. *Organization Studies*, Jan 2008.
12. J. Krogstie, A. Opdahl, and G. Sindre. Advanced information systems engineering: 19th international conference. Jan 2007.
13. D. Riehle and E. Dubach. Why a bank needs dynamic object models. *OOPSLA Workshop on Metadata and Active Object Models*, 1998.
14. D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The architecture of a uml virtual machine. Jan 2001.
15. M. Voelter. A catalog of patterns for program generation. 2003.
16. L. Welicki, J. Yoder, and R. Wirfs-Brock. A pattern language for adaptive object models: Part i-rendering patterns. *hillside.net*.
17. L. Welicki, J. Yoder, R. Wirfs-Brock, and R. Johnson. Towards a pattern language for adaptive object models. *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on OO programming systems and applications companion*, Oct 2007.
18. J. Yoder, F. Balaguer, and R. Johnson. Adaptive object-models for implementing business rules. *Urbana*.
19. J. Yoder, F. Balaguer, and R. Johnson. Architecture and design of adaptive object-models. *ACM SIGPLAN Notices*, Jan 2001.