Trends on Adaptive Object Models Research

Filipe Figueiredo Correia[1,2] and Hugo Sereno Ferreira[1,3]

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

[2] 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/

[3] MAP-I Doctoral Programme in Computer Science,
hugo.ferreira@di.uminho.pt
http://www.map.edu.pt/i

Abstract. An Adaptive Object Model (AOM) is a meta-modeling dynamic technique, where a runtime model is used in order to allow for fast prototyping and model experimentation. It uses several levels of abstraction but differs from generative approaches and reflection in it's degree of dynamism and application domain.
We present a set of common AOM-related design patterns, along with several open issues. We also present the current version of Oghma, an AOM-based system that aims to become a framework for information systems development. Our intent was to compare Oghma with other systems of this sort. We believed some of Oghma's solutions belong to the current state of the art, but also that some benefit could be taken from other researcher's experiences with AOMs.
We have verified our beliefs to some extent, and briefly documented some of Oghma's solutions that we have not yet found applied to other AOMs. However, Oghma is still not close of being a comprehensive solution.

Key words: Adaptive object models, AOM, Model driven engineering, Design patterns, Meta-modeling, UML virtual machine, Oghma

1 Introduction

Creating abstractions has been a recurrent solution in the process of building software systems, allowing developers to direct more attention to software design instead of the idiosyncrasies of the platform being used. Model Driven Engineering (MDE) takes abstractions further, focusing on abstracting particular business domains, rather than only technology related issues [1]. Using this approach, domain models may play an important role on the process of requirements engineering, but their usefulness is also extendable to other software engineering activities.

A lot of current MDE efforts concentrate on model transformations, namely, the generation of implementations, and other artefacts, that effectively support developer's work. Generative approaches cover some typical pitfalls that appear when using MDE, they allow for increased reuse and fewer bugs, easier to understand systems, up to date documentation, a shorter time-to-market and they help making systems that are easier to change [2]. As such, models have proven to be very useful also at software design time, and not only during requirements engineering activities. Their usefulness, however, is also extendable to further software stages, as we will see.

Software requirements change increasingly faster, as organizations have to frequently adapt their business processes to different realities, or they acquire new knowledge that lead to different ways of understanding their business and, therefore, what they expect from systems used to support it. Software systems are called upon being adaptive to these new perceived realities, something that traditional systems are not good at, but models, meta-models, and meta-data in general, may be used in this regard. As been said in [3], in the context of models, *MetaData is just saying that if something is going to vary in a predictable way, store the description of the variation in a database so that it is easy to change. In other words, if something is going to change a lot, make it easy to change.*

Generative approaches to modeling are usually confined to static usage, while runtime model-based adaptivity brings an additional advantage, namely, it greatly reduces the time taken to put a new, or modified, model into execution. It thus allows for rapid prototyping, supporting model experimentation and innovation [2], [4]. Another difference is that runtime models make model semantics explicitly available at runtime. Code generation also makes model semantics available at runtime, but in an implicit way, encoded into the generated code.

The Adaptive Object Model (AOM) architecture allows for runtime adaptivity. It consists on using a meta-model as a first-class model; classes, attributes, relations and behaviour are represented and stored as data. At runtime, this information is interpreted, instructing the system which behaviour to take. Changing the model data results on the system following a different domain model and a different behaviour [2], [4], [5].

This paper presents previous research results on AOMs, relating them to the development of Oghma; a system based on an AOM that is currently being developed at ParadigmaXis. We expected to find solutions better than those we've achieved so far, from which Oghma would benefit, though we also believed some of our solutions would constitute contributions to the state of the art.

We will start by showing the role of abstraction in AOMs (section 2). Section 3 describes what is at stake when designing AOMs, along with related design patterns. It also presents the concrete example of the Oghma system, highlighting some topics we believe to be of particular relevance. Section 4 concentrates on future work using two different perspectives, namely, open issues on AOM systems in general, and issues that will soon be addressed on the context of the Oghma system. Finally, in section 5 some concluding remarks are made.

2 Abstraction

The concept of abstraction, in the sense of object-oriented design, plays an important role in AOMs. There are several levels of abstraction in use in an AOM, which frequently makes them difficult to understand [5]. We will see how AOMs fit among other techniques that also take advantage of multiple abstraction levels, better explaining the differences and similarities between them.

2.1 Level of Abstraction

Object-oriented (OO) languages provide two levels of abstraction, namely, class level, and instance level, which correspond respectively to compile-time and runtime activities. OO systems are bound to these levels, although more conceptual levels can be considered and implemented using these two levels alone [4].

The Meta Object Facility (MOF) is a standard from de Object Management Group (OMG) [6], based on the Unified Modeling Language (UML) and supporting model driven engineering. It provides four modeling levels (M3, M2, M1 and M0), which define an architecture for MOF-based systems, each level describing the next lower one. M3 models constitute meta-meta-models, M2 is used to define meta-models, M1 handles class-level elements and, finally, M0 corresponds to concrete instances [2], [7].

All four levels are useful when taking a meta-modeling approach, as more than the two levels supported by the OO paradigm (M1 and M0) are needed to account for the additional abstraction levels that meta-modeling requires. Meta-modeling is a fundamental concept when building AOMs, and MOF is one way of supporting it [4].

2.2 Reflective and Meta-modeling Techniques

The before mentioned generative and adaptive approaches are, respectively, static and dynamic approaches to meta-modeling. Reflection, like the use of AOMs, is also an adaptive technique, it is a process by which software can alter it's own execution using meta-information about it's structure. Comparing reflection with AOMs, both techniques have the concern of introducing flexibility by allowing dynamic behavior, but reflection has a wider scope, acting at the language level and using meta-information in an ad hoc way, while AOMs act at the business domain level and use meta-information in a well structured fashion [4], [5], [7].

3 The Design of AOMs

Figure 1 shows the basic design of an AOM, as described in [4], [5] and [8].

Two different levels are presented, a *meta* level and an *operational* level. While the former is used to define new types of objects, their respective attributes, relations and behaviors, the later is used to represent concrete objects, attributes and relations, and to enforce the defined behaviors.

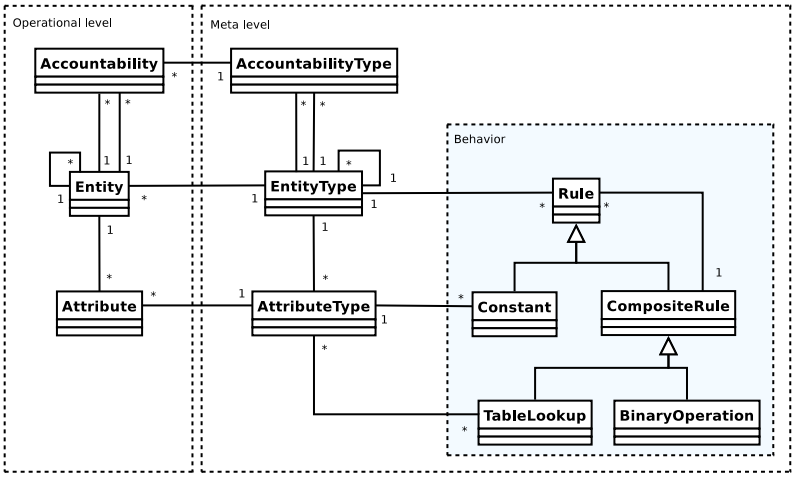

Fig. 1. Basic design of an Adaptive Object Model.

3.1 Patterns

When building AOMs, there are some typical issues that arise, as well as typical solutions for those very issues, which have been documented as *design patterns*.

A design pattern is a good solution for a recurring design problem. It's not meant to be a concrete solution, rather, it's meant to be a generic one that can be instantiated for a given type of problem, considering different contexts where it may arise. Solutions are presented in terms of interactions between elements of object-oriented design, such as classes, relations and objects [9].

There are many patterns useful in the context of an AOM, but the following ones are considered to be the most important when defining the essence of what an AOM is [4], [5].

Type Object. As described in [10], a *TypeObject decouple instances from their classes so that those classes can be implemented as instances of a class. Type Object allows new "classes" to be created dynamically at runtime, lets a system provide its own typechecking rules, and can lead to simpler, smaller systems.*

Property. The *Property* pattern gives a different solution to class attributes. Instead of being directly created as several class variables, attributes are kept in a collection, and stored as a single class variable. This makes it possible for different instances, of the same class, to have different attributes [11].

Type Square. The combined application of the *TypeObject* and *Property* patterns result in the *TypeSquare* pattern [11]. It's name comes from the resulting layout when represented in class diagram, as show in figure 1, with the classes *Entity*, *EntityType*, *Attribute* and *AttributeType*.

Accountability. Is used to represent different relations between parties, as described in [12], using an *AccountabilityType* to distinguish between different kinds of relation.

Composite. This pattern consists of a way of representing part-hole hierarquies. Is can be seen into practice in figure 1 with the *Rule* and *CompositeRule* classes [9].

Strategy. *Strategies* are a way to encapsulate behaviour, so that it is independent of the client that uses it. *Rules* are *Strategies*, as they define behaviour that can be attached to a given *EntityType* [9].

Rule Object. This pattern results from the application of the *Composite* and *Strategy* patterns, for the representation of business rules by combining simpler elementar constraints [13].

Interpreter. An AOM consists of a runtime interpretation of a model. The Interpreter pattern is used to extract meaning from a previously defined user representation of the model [9].

Builder. A model used to feed an AOM-based system is interpreted from it's user representation and a runtime representation of it is created. The *Builder* patter is used in order to separate a model's interpretation from it's runtime representation construction [9].

A lot of other patterns are used when building AOMs, though. Related issues like persistence, user-interfaces (UIs) and models maintenance can take great advantage of existing knowledge described as design patterns.

The patterns in [14], presented next, focus specifically on UI rendering issues when dealing with AOMs. In traditional systems, data presented in UIs is usually obtained from business domain objects, which are thus mapped to UI elements in some way. In AOMs business objects exist under an additional level of indirection, which has to be considered. In fact, it can be taken into our advantage, as the existing meta-information, used to achieve adaptivity, can be used for the same purpose regarding user interfaces. User interfaces can this way be adaptive to the domain model in use.

Property Renderer. Describes the handling of user-interface rendering for different types of properties.

Entity View. Explains how to deal with the rendering of *EntityTypes*, and how *PropertyRenderers* can be coordinated for that purpose.

Dynamic View. Approaches the rendering of a set of entities considering layout issues and the possibility of coordinating *EntityViews* and *PropertyRenderers* in that regard.

This growing group of patterns together describe a set of good practices for AOMs or, in other words, they constitute a *pattern language* for AOMs [15].

The following six categories include the patterns mentioned before, and where used while defining the pattern language presented in [15].

Core. This set of patterns constitute the basis for an AOM-supported system. The patterns included in this category are *Type Square*, *Type Object*, *Properties*, *Accountability*, *Value Object*, *Null Object* and *Smart Variables*.

Creational. These patterns are the ones used for creating runtime instances of AOMs: *Builder*, *AOM Builder*, *Dynamic Factory*, *Bootstrapping*, *Dependency Injection* and *Editor / Visual Language*.

Behavioral. Behavioral patterns are used for adding and removing behaviour of AOMs in a dynamic way. They are *Dynamic Hooks*, *Strategy*, *Rule Object*, *Rule Engine*, *Type Cube* and *Interpreter*

GUI. User-interface rendering patterns have already been mentioned: *Property Renderer*, *Entity View*, *Dynamic View*. Related to UI there's to add the *GUI Workflow* pattern.

Process. Includes the patterns used in the process of creating AOMs. An AOM has usually much of a framework in it. The following patterns are good practices when building a framework as well as when building an AOM: *Domain Specific Abstraction*, *Simple System*, *Three Examples*, *White Box Framework*, *Black Box Framework*, *Component Library*, *Hot Spots*, *Pluggable Objects*, *Fine-Grained Objects*, *Visual Builder* and *Language Tools*.

Instrumental. Patterns that help on the instrumentation of AOMs, namely, *Context Object*, *Versioning*, *History* and *Caching*.

3.2 The Oghma System

Oghma is a system based on an AOM. It is being developed at ParadigmaXis with the purpose of creating a framework for the development of information systems, although it hasn't yet been subjected to enough real-world cases in order to reach that status. It's development started without knowledge about existing research in AOMs and, as such, not all the design solutions employed match solutions described in literature on this topic, although a lot of them do. We find Oghma will benefit from some of these solutions, but we also believe some of the solutions in our system will constitute contributions to the current state of the art.

A detailed description of the system is outside the scope of this paper, but will certainly be further described in a future one. We will, however, highlight some of it's characteristics, which we haven't yet seen discussed to this extent in other AOM-related literature.

Modeling language. UML, as an executable language, presents some difficulties. Supporting it in a way that any UML model may be used by an AOM is a difficult task, as the UML specification is not formal, in order to be executed in a concise, standard way, and the several model types are not always seamlessly connected. As such, being a complete UML virtual machine is not one of the purposes of Oghma, although it uses a subset of UML, that allows for enough expressiveness.

Previous work exists on UML-based AOMs [2], but few details have been given about the extent of the supported UML specification. Oghma supports common AOM meta-model elements such as Classes, Attributes and Relations, along with relations' Cardinalities, but it also supports some UML-specific concepts, like Interfaces, Associative Classes and Navigability. These structures have shown to greatly simplify executing models that had been previously created using UML, while not over complexifying our models by trying to cover all the details in the UML specification.

Persistence. Persistence has before been pointed out as a typical issue when building AOMs. The most simple form of persistence may be achieved by using an object-oriented database, although using a relational database is also possible, in spite of the impedance mismatch between the relational and object-oriented worlds [2], [5].

The way to achieve persistence, when comparing with other Object-Relational Mapping (ORM) approaches [16], may be simplified in AOMs. In Oghma's case, a relational database is used. A runtime relational meta-model was conceived, along with rules to map between it and the existing runtime object-oriented meta-model. In this way we are combining model transformation techniques, common in generative approaches, with the dynamic technique which is an AOM.

Client-server. Oghma has a client-server architecture, and both kinds of nodes (clients and servers) take advantage of the possibilities offered by the underlying AOM. The way clients and server communicate with each other can be made independent of the fact an AOM is being used or not, but we have found that, the existing meta-model, allows the schema of messages, exchanged between clients and the server, to also be made adaptive.

The server accepts requests for both meta-level elements and operational-level elements. Allied with the fact that REST/XML (over HTTP)[17] was used as a communication architectural principle, we have obtained a server interface that is simple to use and constitutes a general purpose API, available for establishing interoperability with other systems. Using REST/XML over HTTP has also some additional advantages, namely, it simplifies debugging, provides caching mechanisms, and makes available standard ways of dealing with authentication and communication security.

Queries. Queries in the context of object-oriented environments have been addressed before in different perspectives [18], [19].

In Oghma, the way data is persisted is completely hidden from the server interface. In order not to break that abstraction, a querying model was designed, that allows queries to be defined in an object-oriented way. Instances of this object-oriented querying model can be transformed into an analogous relational-oriented querying model, in a similar way the relational meta-model is transformed to the object-oriented meta-model, and vice-versa. The relational-oriented querying model is directly translatable to SQL code, which is used to actually execute the intended query.

Because data is exchanged between the client and server in a RESTfull way, queries fit into this communication architecture encoded into URLs, and query results are returned as a set of resources.

Addressing. Something that directly derives from the adopted RESTfull communication architecture, is the fact that (meta-level and operational-level) model elements are seen and made available as resources. As such, by using the meta-model information, an adaptive and URL-based resource addressing scheme was defined. Considering an hypothetical model, the next example would obtain the model schema for "laptop" element types:

```
http://oghma.paradigmaxis.pt/computer/laptop/@schema
```

These examples would return existing information for a specific laptop, and a list of all of it's parts, respectively[4]:

```
http://oghma.paradigmaxis.pt/computer/laptop/4A3615F1-5A91-22E4-0B1D-1416F93D4412
```

```
http://oghma.paradigmaxis.pt/computer/laptop/4A3615F1-5A91-22E4-0B1D-1416F93D4412/parts
```

As mentioned before, queries also take part of the addressing scheme. The following example consists on a query that returns all the instances' of laptop computers bought before 2005:

```
http://oghma.paradigmaxis.pt/computer/laptop[yearbought lt 2005]
```

Business rules. Business rules in the context of AOMs are frequently made as pluggable components, based on the Strategy design pattern (see section 3.1), and these components' implementation vary according to the domain in use [11], [5], [4]. Oghma's business rules don't follow this approach, as they are added to the model in a declarative way, making them simpler to define and more reusable, although less powerful than using Strategies.
The runtime model enforces these business rules, and it does so both on the client and server sides. On the server side, business rule enforcement is done to ensure semantic integrity according to the model, while on the client side it is done to validate user input, giving quick feedback to the user and avoiding roundtrips to the server as much as possible. Validations exclusively on the client side are not sufficient, as the server is used concurrently by multiple clients, but also because it may be used by third party client software as well, which may not fully validate their input data.

User-Interface. Adaptivity is a pervasive concept when it comes to AOMs, and it reaches user interface issues too. Some solutions have recently been documented [14] that take an adaptive approach to UI issues in AOMs (see section 3.1), but some additional advancements can be found in Oghma's approach. Namely, PropertyRenderers are used to present not only value-properties (attributes), but also instance-properties (relations). When adapting the interface for a specific context, PropertyRenderers are chosen based on several meta-model characteristics. For value-properties, the kind of the AttributeType, as well as related business rules are used to determine which renderer should be applied. For instance-properties, the cardinality and navigability are used for the same purpose; there are specialized renderers for one-to-many, many-to-many and one-to-one relations. Has mentioned before, user-interface feedback is also based on business rules.
System navigation is also taken into account. Types may be declared as Entry Points, and modeled as belonging to specific Subsystems. Doing so makes such Types directly accessible from the system's menu, under the established subsystem structure.

[4] There is another important detail, that although not a direct consequence of the addressing scheme, shows in these examples: all instances are identified by Global Unique Identifiers (GUIDs). This makes decentralization easier; clients can create new objects, with their respective identifiers assigned, without having to request them to the server.

4 Future Work

Advantages of using AOMs stand out when using a domain model that changes frequently, but these advantages come at a cost. A lot of issues remain to be solved in their design and development, making them a fertile research area.

We will first address common open issues in AOMs, followed by some areas we will specifically like to explore in Oghma, and that we believe may also be of interest in the context of other AOM-based systems.

4.1 General Open Issues in AOMs

AOMs generally require a higher initial development effort, as they are more complex than traditional systems. This complexity makes them also harder to understand, specially by those who haven't had previous contact with this kind of architecture, and thus, they may be harder to maintain. Although AOMs are *adaptive* to model changes, they are not easily *adaptable* to new functional requirements. It is important to consider the degree of adaptivity that is in fact needed when starting the development of an AOM, as the introduced flexibility will tend to increase the system's complexity [5], [20], [21].

Model maintenance may also be an issue. Using Visual Editor tools or Domain Specific Languages (DSLs) may support model creation and later modifications, but these tools have usually to be developed from scratch. When developing a language, as when developing a DSL, other needs also arise, such as debuggers, version control and documentation [5]. However, we do believe the use of standard languages and tools may ease these issues.

Also related to modeling, running systems are limited to the expressiveness of the modeling language used, specially concerning behavior modeling, and it is an open issue to find the right level of abstraction the model should have [2].

Model evolution should also be taken into account. As models evolve, instances created according to the previous model version have to be migrated, which may require intervention of external tools or, alternatively, will require the system to handle different model versions simultaneously [2].

Being part of the production system, an AOM affects it's execution. AOMs are usually slower than traditional systems, and even other (generative) metamodelling techniques [2], [5], [20]. The approach followed in [2] is an interesting combination of AOMs and generative techniques, at a cost of an additional initial development effort: an AOM is used for system prototyping and model experimentation, but code is generated for the production system.

Our own experience with Oghma has show us some of AOM's advantages and disadvantages, but it is not trivial to assess the entire impact of using an AOM over a traditional approach. Yet, we haven't found a concrete analysis of how software quality metrics are affected by the use of this architecture. We believe such study would be of great interest, and it would also ease the comparison between different AOM implementations, including the Oghma system.

4.2 Future Work in Oghma

In the context of Oghma, the following AOM-related concerns would be of most interest to explore in the future.

Meta-model transformations. The transformation process between the object-oriented and relational-oriented models, as described in section 3.2, is done in a monolithic way. This process could be improved by taking a more modular approach, using a set of rules that together define the transformation, which would allow to prove the bijectiveness, or injectiveness, of transformations. It will also be useful to take this same approach towards the transformations between object-oriented models and their xml-oriented representation, which is also currently made in a monolitic manner.

Model evolution. Migration of object instances, between different model versions, is an issue we believe may be solved, to a certain degree, by taking benefit from the respective meta-models.

Elements from the source model will have to be mapped to the respective elements from the target model, but this mapping can be a complex process, considering the number of differences that may exist between two given models. Current knowledge on refactorings [22] and database evolutionary transformations [23] can be used in this regard. By simplifying complex model migrations into a more restricted set of standard object-oriented *refactorings*, we obtain an additional level of abstraction, from which a model-migrations-oriented language can be developed. The developer would be able to use this language to conceive a migration process between two given models. The next step would be to assist the developer in more easily creating these migration processes by automatically identifying model refactorings from the source and target models. In the best case scenario, the described assistant would be able to identify the entire process with minimal intervention from the developer, although that may not be possible for models very different from each other.

Ontology. An ontology is a form of knowledge representation[5]. It is a data model, and it can be used to describe classes, attributes, relations and events. This makes ontologies very similar to the subset of UML that Oghma uses, and thus, makes them a possible candidate to replace the current Oghma meta-model.

We believe ontologies may provide a richer and more formal meta-model than the one currently used by Oghma. It would also serve the purpose of standardizing the XML dialects used for model representation as well as for communication between the server and its clients. Well established formats such as the Ontology Web Language (OWL) can be used for this purpose, instead of the XML dialects we have developed [24], [25].

[5] We refer to the concept of ontologies in context of computer science, as it has a different meaning in the context of philosophy.

5 Conclusion

By using Model Driven Engineering (MDE) developers can more effectively focus on business domain modeling, as well as the modeling of more technology-related concerns. Generative approaches have shown to be very helpful in this context, but they are not appropriate for fast prototyping and model experimentation, while Adaptive Object Models (AOMs) present a way to achieve these objectives.

A characteristic of AOMs is that they use several levels of abstraction, as other techniques also do. However, AOMs are dynamic in nature (while generative approaches to meta-modeling are not) and act at the business domain level (while reflection act at the language level). AOMs take into account an operational level and a meta level. The first works at the instance level, while the second works at the class level.

Design patterns are a valuable resource when conceiving AOMs. Several patterns are used in the basic underlying architecture of an AOM, and many more can be used to solve related issues, such as persistence and user-interface adaptivity, among others. The set of patterns, that describe the ways an AOM may be designed, form a Pattern Language for AOMs.

The Oghma system is based on an AOM. It aims to be a framework for information systems development, and uses approaches that we believe to be of notice, namely, on the areas of the modeling language used, persistence, client-server interaction, queries and business rules enforcement.

AOMs still have a lot of unsolved related issues, although some of them have been being addressed on AOM's literature. Oghma also has a long way to go before we may see it as a comprehensive solution, and not all of the topics we will like to explore have already been covered before, in the context of AOMs. We believe we will find such topics to be fruitful research paths.

References

1. Schmidt, D., Schmidt, D.: Guest editor's introduction: Model-driven engineering. Computer **39** (2006) 25–31
2. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a UML virtual machine. In: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, Tampa Bay, FL, USA, ACM (2001) 327–341
3. Yoder, J.: Adaptive object models and metadata definition (2008) http://www.adaptiveobjectmodel.com/Define_Adaptive_Object_Models.html Accessed January 5, 2008.
4. Revault, N., Yoder, J.W.: Adaptive object-models and metamodeling techniques. In: ECOOP '01: Proceedings of the Workshops on Object-Oriented Technology, London, UK, Springer-Verlag (2002) 57–71
5. Yoder, J.W., Johnson, R.E.: The adaptive object-model architectural style. In: WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture, Kluwer, B.V (2002) 3–27
6. OMG: OMG's metaobject facility (MOF) home page (2008) http://www.omg.org/mof/ Accessed January 5, 2008.

7. Costa, F.M., Provensi, L.L., Vaz, F.F.: Using runtime models to unify and structure the handling of meta-information in reflective middleware. Volume 4364., Springer Berlin / Heidelberg (2006) 232–241
8. Welicki, L., Lovelle, J.C., Aguilar, L.J.: Meta-specification and cataloging of software patterns with domain specific languages and adaptive object models. In: EuroPLoP, Irsee, Germany (2006)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Professional (1995)
10. Johnson, R., Woolf, B.: The type object pattern (1997)
11. Yoder, J.W., Balaguer, F., Johnson, R.: Architecture and design of adaptive object-models. ACM SIG-PLAN Notices **36**(12) (2001) 50–60
12. Fowler, M.: Analysis patterns: reusable objects models. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (1997)
13. Arsanjani, A.: Rule object: A pattern language for adaptive and scalable business rule construction. (2000)
14. 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)
15. 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
16. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (2002)
17. Fielding, R.T. In: Representational State Transfer (REST). University of California, Irvine (2000)
18. ODMG: Object data management group home page (2008) http://www.odmg.org/ Accessed January 5, 2008.
19. Microsoft: The linq project (2008) http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx Accessed January 5, 2008.
20. Dantas, A., Yoder, J., Borba, P., Johnson, R.: Using aspects to make adaptive object-models adaptable. In: RAM-SE'04-ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, Oslo, Norway (2004) 9–19
21. Crous, T., Danzfuss, T., Liebenberg, A., Moolman, A.: Adaptive object modelling using the .NET framework (2005)
22. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)
23. Ambler, S.W., Sadalage, P.J.: Refactoring Databases: Evolutionary Database Design. Addison-Wesley Professional (2006)
24. W3C: Owl web ontology language overview (2004) http://www.w3.org/TR/owl-features/ Accessed January 5, 2008.
25. Knublauch, H.: Ramblings on agile methodologies and ontology-driven software development, Galway, Ireland (2005)