Adaptive Object-Model Builder

León Welicki
Microsoft
lwelicki@microsoft.com

Joseph W. Yoder
The Refactory, Inc.
joe@refactory.com

Rebecca Wirfs-Brock
Wirfs-Brock Associates
rebecca@wirfs-brock.com

Abstract

An Adaptive Object-Model system represents user-defined classes, attributes, relationships, and behavior as metadata. This paper presents the Adaptive Object-Model Builder pattern that is used to construct AOM entities. An AOM Builder reads an externally stored build description to construct a build process. This process is then executed to construct a properly initialized AOM entity. Since an AOM Builder is driven by metadata descriptions of entities and their build processes, a single generic AOM Builder implementation can construct different entity types.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 **[Design Tools and Techniques**]: Object-oriented design methods; D.2.11 **[Software Architectures]**: Patterns

General Terms

Design

Keywords

Factory Objects, Adaptive Object-Models, Creational Patterns

1. Introduction

An Adaptive Object-Model is a system that represents user-defined classes, attributes, relationships, and behavior in an object-oriented domain model as metadata [YBJ01; YJ02]. In an AOM system, domain entities are constructed from externally stored definitions (metadata) that are interpreted at run-time.

Users, who may not be programmers, can change externally stored metadata whenever they want to change the definitions of domain entities. Whenever externally stored definitions are modified, the system can immediately reflect those changes without recompiling the application. This is similar to a UML Virtual Machine implementation described by Riehle et. al [RFBO01]. As a consequence, the object model in an AOM system is dynamically adaptable.

This is in contrast to how domain models are typically built in traditional object-oriented programming languages. In normal OO design, the programmer defines domain entities and their behavior using programming-language classes. Whenever a change is required to a domain entity, one or more class definitions may need to be modified and the application recompiled.

The pattern presented in this paper describes the creation of instances of AOM entities using an AOM BUILDER. AOM BUILDER is one **Creational** pattern that is part of a pattern language for AOM systems [WYWJ07]. Figure 1 shows the context of this pattern with other creational patterns.

Adaptive Object-Model architectures are usually made up of several smaller patterns. In the existing literature they are documented by the patterns TYPE OBJECT, ATTRIBUTES, PROPERTY LIST, TYPE SQUARE, ACCOUNTABILITY (ENTITY-RELATIONSHIP), STRATEGY, RULE OBJECTS, COMPOSITE, BUILDER, and INTERPRETER.

More information about the AOM architectural style can be found in Appendix A. An overview of a larger pattern language for AOM systems is presented in Appendix B. For a more comprehensive treatment and bibliography on AOM systems and patterns, see *www.adaptiveobjectmodel.com*.

The AOM BUILDER pattern presented in Section 2 uses a pattern format which includes the context, problem, forces, solution, dynamics, implementation, resulting context, and related patterns sub-sections.

2. AOM Builder Pattern

Typically, at object construction time an entity's attributes are initialized to well-defined values and links are made to associated objects, which themselves are properly formed. This can be a complicated process in any system. But creating entity objects based on metadata definitions, as is the case for AOM systems, is slightly more involved. External definitions must be read and interpreted in order to construct a TYPEOBJECT. When constructing a TYPEOBJECT, its PROPERTIES, TYPE-SQUARE, STRATEGIES and ENTITY-RELATIONSHIP must also be created with valid values.

2.1 Context

You are creating an application using an Adaptive Object-Model. Your model relies on a variant of TYPE SQUARE so you are using a combination of TYPE OBJECT and PROPERTIES patterns.

You want to create instances of entities of a concrete type based on metadata. Since the creation process is complex, the BUILDER pattern can be used (which could be combined with the INTERPRETER pattern). However, a maintenance problem may arise if you hand code in the BUILDER steps to create an instance of entity which might vary according to its type or some arbitrary rules (specifically when these vary or evolve).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. Preliminary versions of these papers were presented in a writers' workshop at the 16th Conference on Pattern Languages of Programs (PLoP). PLoP'09, August 28-30, Chicago, IL, USA. Copyright 2009 is held by the author(s). ACM 978-1-60558-873-5

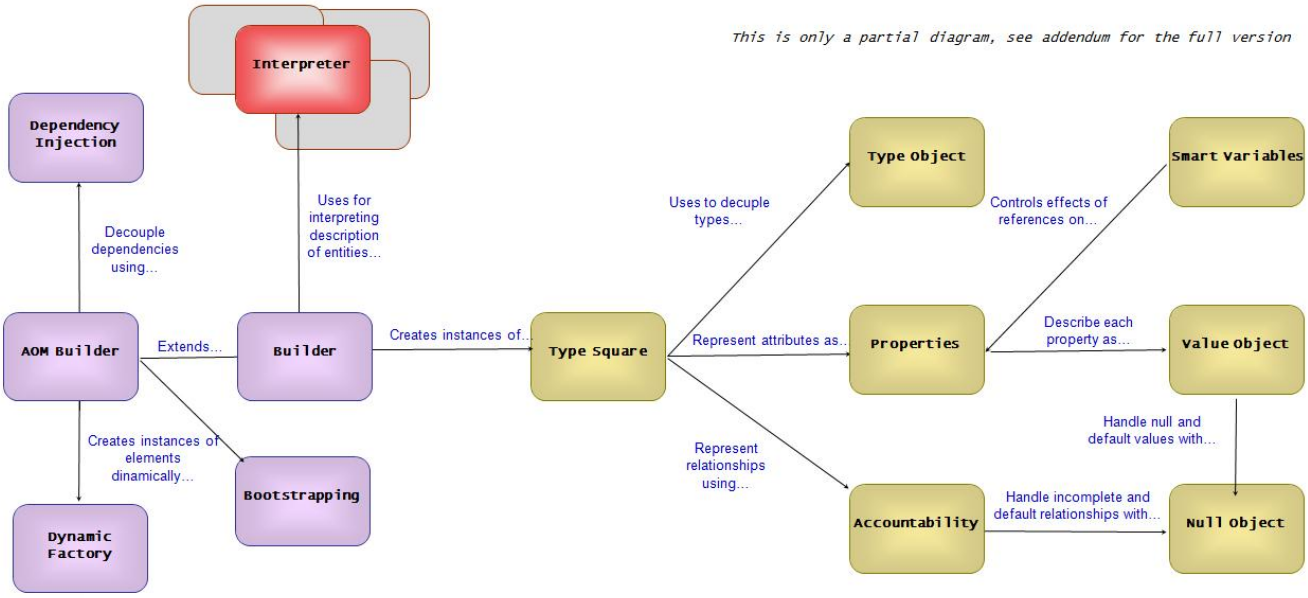

Figure 1 - AOM Pattern Language for Creational Patterns

2.2 Problem

How can you encapsulate the process of building instances of persisted entities allowing the process to change dynamically according to the composition rules of the entities types?

2.3 Forces

- The rules for creating an entity may vary according to its type or according to rules that apply to its data.
- You want to encapsulate the construction of entities.
- You want to reuse the different steps involved in creating an instance of an entity to create other entities.
- You want to be able to adapt to changes in the entity definition or to add new arbitrary steps in the creation process (like logging, security, etc.)
- You don't want to bloat your construction code with lots of conditional statements to handle different entity types.
- You don't want to have an explosion of Builders, one for each entity type, or cope with all the conformation rules of the concrete entities by writing builder code that must be rewritten and compiled whenever entity definitions change.

2.4 Solution

Abstract the building process into a well defined interface, break it into small steps, configure the steps using metadata based on the type of the entity to be built, and execute build steps in order.

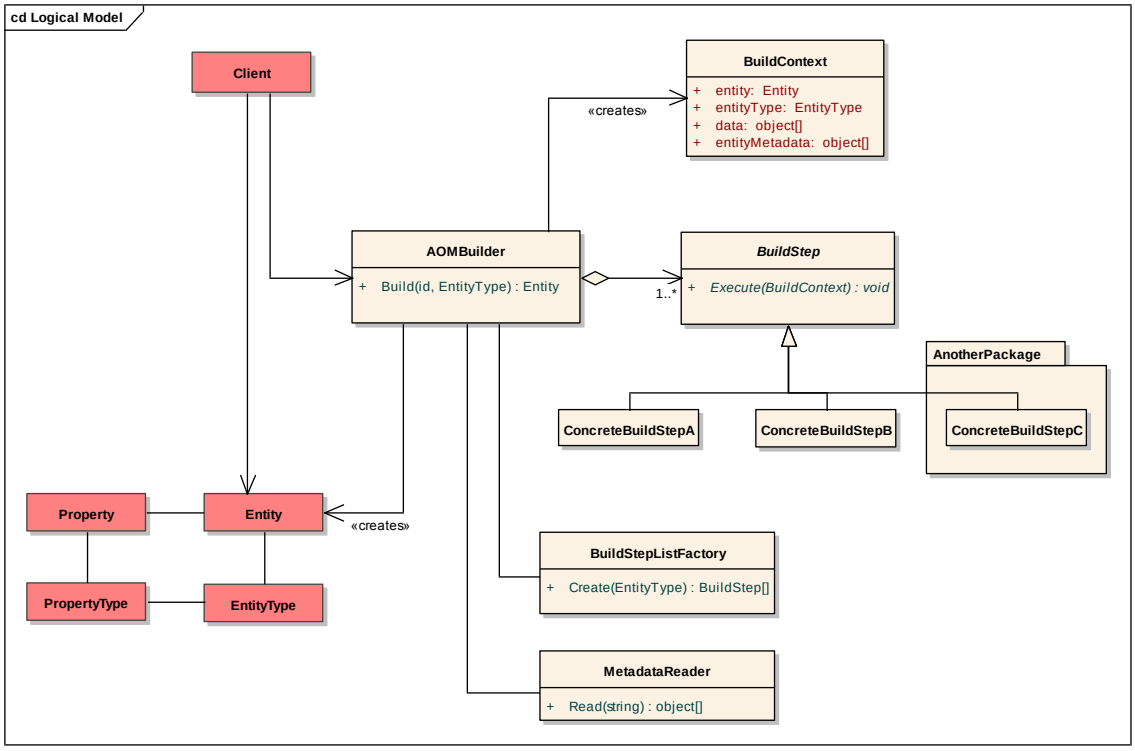

Figure 2 - AOM Builder Structure. The classes in red (the client and the type square instance) are not part of the solution

A complex entity build process can be divided into atomic steps that are executed in order. Build steps can share data, if necessary, using a context object [KSS05]. Specification of the steps can be done dynamically using externally stored metadata. The configuration of the steps should be based on type, since each type of entity may need different build steps. This also allows you to define a default build procedure which can be arbitrarily extended.

There are two main "sources" of metadata used by the implementation of the AOM BUILDER pattern: the definition of the build steps for each type and the metadata which defines the entities. The first is used to drive the overall process, the second to load the AOM entity with information.

The entry point to the building process is provided by the `AOMBuilder`. This object defines a generic interface for creating instances of several types of entities [YBJ01]. The `AOMBuilder` first initializes the process, asking for the building pipeline from the `BuildStepListFactory` (which loads the necessary build steps based on the given TYPE OBJECT). The `AOMBuilder` then creates the `BuildContext` and fills it with the metadata of the required entity (loading it from the metadata repository through the `MetadataReader` object).

Each `BuildStep` is a specialized part of the entity building process. The building process can be extended by defining new `ConcreteBuild` steps. `BuildStep` implementations can be in different packages or assemblies (as is the case for `ConcreteBuildC` shown in figure 2). A `BuildStep` can be loaded dynamically using REFLECTION [POSA1] or any other late binding technique.

The classes in red in figure 2 (`Client`, `Entity`, `EntityType`, `Property`, `PropertyType`) are not part of the solution itself: the `Client` uses the `AOMBuilder` and the `Entity`. The `Entity`, `EntityType`, `Property`, and `Property Type` represent a canonical implementation of TYPE SQUARE [YBJ01], the product of the building process.

For complex cases, the metadata that indicates the build steps for each type may contain additional rule definitions and constraints. While this will increase the complexity of the build process execution, it allows for an even more flexible build process.

The idea behind the AOM BUILDER pattern is the same as for the BUILDER [GoF95] pattern (dealing with the creation of complex objects in several steps. But the AOM BUILDER is targeted to a clearly different execution context and has different design goals. The BUILDER relies on composition and inheritance for dealing with flexibility and extensibility; the AOM BUILDER is based on composition, dependency injection, smart properties, and polymorphism driven by externally defined metadata.

2.5 Dynamics

Figure 3 shows how the participants interact to produce an AOM entity. The Client asks the `AOMBuilder` for an entity. The `AOMBuilder` is responsible for coordinating the build process. The `AOMBuilder` first asks the `MetadataReader` to read the requested entity's metadata from the metadata repository. It then creates the `BuildContext` and an ordered set of `BuildStep` objects using the `BuildStepListFactory`. The order of the `BuildStep` objects is defined by the metadata. Each `BuildStep` is executed in order. In our example there are just two build steps, the `ConcreteBuildStepA` and the `ConcreteBuildStepB`.

The reader may notice that the participants of the TYPE SQUARE pattern (`Entity`, `Entity Type`, `Property`, and `Property Type`) are not shown in figure 3. This wasn't shown so as to simplify the sequence diagram. The interaction with these entities is as follows: the `AOMBuilder` creates the empty `Entity` instance (based on the `Entity Type`) and also loads it into the `BuildContext`. Thereafter, only concrete `BuildSteps` interact with the `Entity` or any of its `Properties`, either to properly define their values or to perform any other arbitrary action such as logging, audit, security, or tamper checking.

2.6 Implementation

The complexity of implementing this pattern lies in the implementation of the concrete build steps, following the *Dependency Inversion Principle* as presented in [Martin02]. The main build control logic is the same and is contained in the implementtation of the `AOMBuilder Build()` method (see code 1).

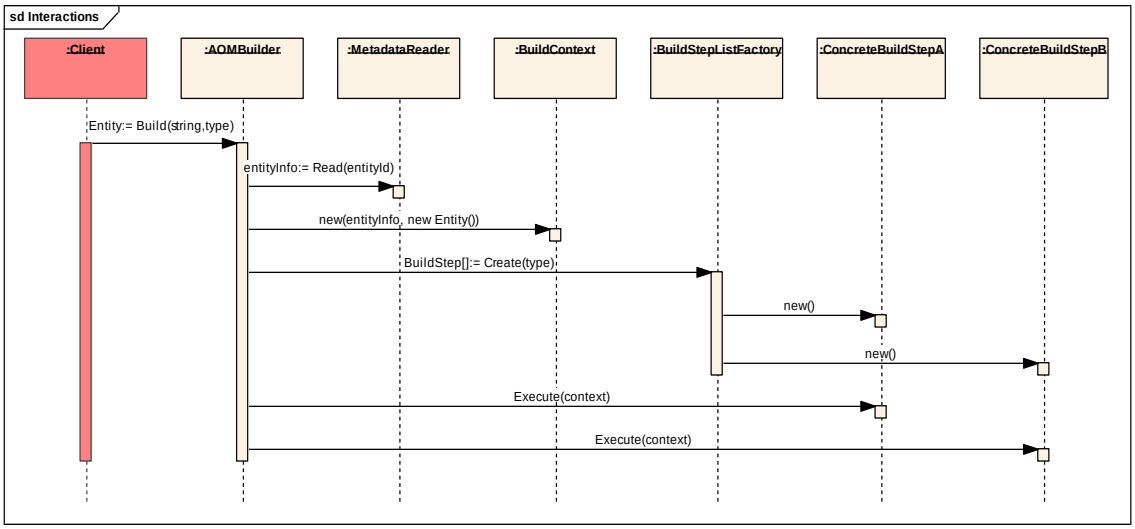

Figure 3 - AOM Builder Dynamics. The TypeSquare members have been left out to make the diagram simpler (they interact with the concrete build steps)

```
public class AomBuilder  {
   public Entity Build(string id, EntityType type)
  {
   // load entity metadata
  IEntityMetadataReader reader = new
      EntityMetadataReader();
  XmlDocument entityMetadata = reader.Load(id,
      type);
   // create building context
   BuildContext context = new
      BuildContext(entityMetadata,
              type.CreateInstance());
  // obtain building pipeline and execute it
   IList<IBuildStep> buildSteps =
      BuildStepListFactory.Create(entity.Type.ID);
  foreach (IBuildStep buildStep in buildSteps)   {
    buildStep.Execute(context);
  }
  // return result
  return context.Entity;
 }
}
```

Code 1 - Main body of the AOM Builder participant.

Variations in building behavior are controlled by the concrete implementations of the build steps which implement the `IBuildInterface`, as shown in code 2. Their order is specified by a configuration in the build metadata repository. Code 3 shows a configuration file with four build steps. Any common information that needs to be shared between the build steps, including the Entity, is passed using a context object, following the Context Object [KSS05] pattern. For each step a class name and assembly are specified. In this example, the last step registers audit information for statistical purposes and doesn't affect the entity.

```
public interface IBuildStep {
   void Execute(BuildContext context);
 }
```

Code 2 - Interface definition for build steps.

The sample configuration shown in code 3 contains several steps for dealing with various phases of construction: creating the entity, loading its properties, loading the relationships, and saving audit information for statistical purposes. Build steps can be complex and may need to be broken in several pieces. This is the often case for the `PropertiesBuildStep` (the step that loads the values into the properties), since each property may need to be handled differently. Each step can also manage its own metadata and be as complex as it needs to be (code 4 shows a sample of the configuration file for property loaders used by the `PropertiesBuildStep`). This metadata is used by the PropertyLoader build step

```
<buildSteps>
 <buildStep type="AOM.Builder.BuildSteps.
           EntityInfoBuildStep,AOM.Core"/>
 <buildStep type="AOM.Builder.BuildSteps.
           PropertiesBuildStep,AOM.Core"/>
 <buildStep type="AOM.Builder.BuildSteps.
        RelationshipsBuildStep,AOM.Core"/>
 <buildStep type="AOM.Builder.BuildSteps.
                AuditBuildStep,AOM.Core"/>
</buildSteps>
<propertyLoaders>
```

Code 3 - Build step metadata specification.

```
  <loaderFor
     type="AOM.Core.StringProperty"
     factory="AOM.Core.StringPropertyTypeLoader,
                                    AOM.Core"/>
  <loaderFor
     type="AOM.Core.NumberProperty"
     factory="AOM.Core.NumberPropertyTypeLoader,
                                    AOM.Core"/>
  <loaderFor
     type="AOM.Core.DateProperty"
     factory="AOM.Core.DatePropertyTypeLoader,
                                    AOM.Core"/>
  <loaderFor
     type="AOM.Core.FileProperty"
     factory="AOM.Core.FilePropertyTypeLoader,
                                    AOM.Core"/>
  <loaderFor
     type="AOM.Core.UrlProperty"
     factory="AOM.Core.UrlPropertyTypeLoader,
                                    AOM.Core"/>
  <loaderFor
     type="AOM.Core.EntityProperty"
     factory="AOM.Core.EntityPropertyTypeLoader,
                                    AOM.Core"/>
</propertyLoaders>
```

Code 4 - Metadata configuration for property loaders.

2.7 Resulting Context

- ✓ The complex process of creating instances of AOM entities is encapsulated into a single, well-known object.
- ✓ Responsibility for creating instances of properties, rules, etc. is factored into fine-grained building step objects.
- ✓ Creation code is separated from the consumer code.
- ✓ The pipeline of the building process is specified using metadata. It can be modified without needing to recompile the application.
- ✓ The build steps can be modified or extended dynamically.
- ✓ The build process of any AOM entity can be modified dynamically at run-time.
- ✓ Additional concerns can be easily added to the build process (e.g. by adding a build step for logging, another for security, etc.).
- ✗ Since the build process is specified using metadata there is no possible compile-time verification.
- ✗ More complexity. Although less flexible, the alternative of defining several factories (based on entity and property types) which contain hand-coded rules for creating instances of AOM entities can be simpler to understand.
- ✗ There is more indirection involved in reading and interpreting external metadata to build entities. This can lead to lower performance.

2.8 Related Patterns

AOM BUILDER is an evolution of the BUILDER [GoF95] pattern.

AOM BUILDER uses PIPES AND FILTERS [POSA1] to orchestrate the building steps.

Information shared between build steps can be accomplished using the CONTEXT [KSS05] pattern.

Build steps instances can be created using a PRODUCT TRADER. In this case the rules for selecting one step or another are not hard-coded in external definitions of metadata but determined at run-time using Specification objects [BR98].

The AOM BUILDER can be seen as a REGISTRY [Fowler02] for instances of entities in an AOM based application.

AOM BUILDER performance can be dramatically enhanced using CACHING [POSA3].

This pattern is similar to a COMPLETE CONSTRUCTOR [Beck08] as it attempts to create full constructed objects.

2.9 Known Uses

The entity loader in [WCJ06] uses the AOM Builder pattern to create instances of entities in the system. An entity is composed of several parts (tags, metadata, relationships, pattern definition, and implementation). The AOM builder is configured with a set of steps to build each one of these parts and then assemble a complete entity. These steps also include an audit step that saves data about the entity being loaded (e.g. last loaded date, user that is loading the entity, and hit count).

An AOM framework for medical systems built for the Illinois Department of Public Health uses Builder pattern to create instances of Observations and its related objects.

An AOM-based content management system developed and used at a telecom company where one of the authors worked uses this pattern to create instances of entities. The AOM Builder pattern implementation coordinates the work that needs to be done in order to create a new or load an existing entity instance.

3. Appendix A - A Brief Summary of the Architectural Style of AOMs

Notice: *This section is a summary extracted from [YJ02] and [YBJ01] and has been included with informative purposes to help readers that are not familiar with the AOM architectural style. To get a more complete view we recommend the reader see the original papers at www.adaptiveobjectmodel.com.*

The design of Adaptive Object-Models differs from most object-oriented designs. Normally, object-oriented design would have classes for describing the different types of business entities and associates attributes and methods with them. The classes model the business, so a change in the business causes a change to the code, which leads to a new version of the application. An Adaptive Object-Model does not model these business entities as classes. Rather, they are modeled by descriptions (metadata) that are interpreted at run-time. Thus, whenever a business change is needed, these descriptions are changed which are then immediately reflected in the running application.

Adaptive Object-Model architectures are usually made up of several smaller patterns. TYPE OBJECT [JW98] provides a way to dynamically define new business entities for the system. TYPE OBJECT is used to separate an Entity from an EntityType. Entities have Attributes, which are implemented with the Property pattern [FY98]. The TypeObject pattern is used a second time in order to define the legal types of Attributes, called AttributeTypes.

This core set of patterns working together is very common to most AOM architectures as described by Dynamic Object Models [RTJ05]. Therefore if the user is selling products, the AOM will describe different types of Entities to represent their different types of products. Non-AOM systems would model these with different product classes.

As is common in Entity-Relationship modeling, an Adaptive Object-Model usually separates attributes from relationships. In usual OO design, entity-relationships are commonly implemented through an attribute as a pointer or direct reference to the related objects. Also, methods are used to implement any rules about the relationship. However in AOMs these relationships are reified thus enabling a way to describe new types of relationships and rules governing the relationships via metadata. The STRATEGY pattern [GoF95] is used to define the behavior of `EntityTypes`. These strategies can evolve into a rule-based language that gets interpreted at runtime. Finally, there is usually an interface for non-programmers to define the new types of objects, attributes and behaviors needed for the specified domain. This also includes ways to define subtypes and relationships between objects.

Therefore, we can say that the core patterns that may help to describe the AOM architectural style are:

- TYPE OBJECT
- PROPERTY
- ENTITY-RELATIONSHIP / ACCOUNTABILITY
- STRATEGY / RULE OBJECT
- INTERPRETER (of Metadata)

Adaptive Object-Models are usually built from applying one or more of the above patterns in conjunction with other design patterns such as COMPOSITE, INTERPRETER, and BUILDER [GoF95]. COMPOSITE is used for building dynamic tree structure types or rules. For example, if the entities need to be composed in a dynamic tree like structure, the COMPOSITE pattern is applied. BUILDERS and INTERPRETERS are commonly used for building the structures from the meta-model or interpreting the results.

But, these are just patterns; they are not a framework for building Adaptive Object-Models. Every Adaptive Object-Model is a framework of a sort, but there is currently no generic framework for building them. A generic framework for building the TypeObjects, Properties, and their respective relationships could probably be built, but these are fairly easy to define and the hard work is generally associated with rules described by the business. This is something that is usually very domain-specific and varies quite a bit.

3.1 The Type Square

In most Adaptive Object Models, TYPE OBJECT is used twice, once before using the PROPERTY pattern, and once after it. TYPE OBJECT divides the system into `Entities` and `EntityTypes`. `Entities` have attributes that can be defined using `Properties`. Each property has a type, called `PropertyType`, and each `EntityType` can then specify the types of the properties for its entities. Figure 4 represents the resulting architecture after applying these two patterns, which we call TYPE SQUARE [YBJ01].

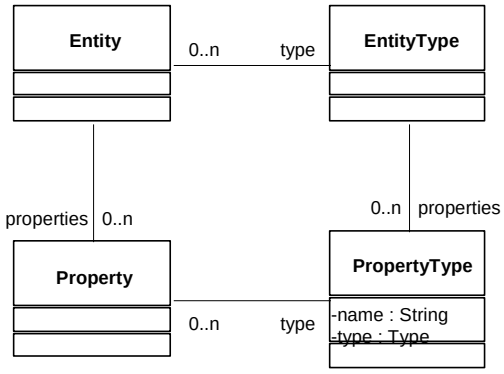

Figure 4 - The Type Square

It often keeps track of the name of the property, and also whether the value of the property is a number, a date, a string, etc. The result is an object model similar to the following: Sometimes objects differ only in having different properties. For example, a system that just reads and writes a database can use a Record with a set of Properties to represent a single record, and can use `RecordType` and `PropertyType` to represent a table.

4. Appendix B – An Overview of AOM-Related Patterns

Our primary goal is to document in a uniform and standardized way all the existing patterns that can be used to create adaptive object models. A secondary goal is to make the pattern language more complete. This will ease the task of creating this kind of architectures to designers, architects and developers.

We started with a brainstorming session where a big set of patterns (more than 40) was listed and categorized. We also classified the patterns in three groups according to their publishing status: published, not published, ongoing)

The pattern language map will help to establish a roadmap to document (or recast) all the patterns involved in creating applications using this architectural style.

4.1 Categories

We have grouped our patterns in the following categories:

- **Core**: includes the core patterns that are present in the basic implementation of AOMs. These are the basic patterns and they are the ones that govern this architectural style.
- **Process**: includes the patterns that deal with the process of creating AOMs. They establish guidelines for evolving frameworks and boundaries to avoid going up to the meta-levels far beyond than necessary.
- **Presentation**: includes the patterns that deal with how to present AOMs to end-users in applications.
- **Creational**: includes the patterns that help to create instances of AOMs
- **Behavioral**: includes the patterns for dynamically adding, removing or modifying behavior to the AOMs
- **Miscellaneous**: includes patterns that help to instrument the usage, control, and instrumentation of AOMs. They also help to provide guidelines for non-functional requirements such as performance and auditability.

4.2 Status

The status refers to the publishing state of the patterns. In our pattern mining session, we found more than forty patterns. Some of them were published, some of them where included in unpublished work and some of them where ideas.

- **Published**: patterns that have been published in previous works. These patterns have been through the community process (shepherding and writers workshops).
- **Unpublished**: patterns that we aware of their existence but haven't been publicly published yet.
- **Ongoing**: patterns that are being written at the moment of creating our patterns list.

4.3 Conclusions and Future Directions

Creating AOMs is not a trivial task. The architects and developers involved in creating AOM-based applications need to use and combine many patterns. Some patterns have been written about in published conference proceedings but the topic is still incomplete. Very often, developers don't even use the patterns and arrive at this kind of architecture intuitively. What we are trying to achieve with our research and further publications is to provide a comprehensive set of patterns for creating AOMs, thus making it easier for developers who are creating applications using this kind of architecture. The set of related AOM patterns and their relationship to other published patterns, as shown in Figure 5, is a clear step towards that objective. It establishes a visual roadmap for documenting the patterns involved in the AOM architectural style.

Besides these patterns, less widely known patterns are often used in AOM systems. Descriptions of these other patterns are scattered among a number of different papers patterns with different templates and styles. Additionally, not all the pattern examples use the same example. Some patterns haven't been updated to reflect current implementation trends or programming language environments or development platforms. We ultimately see the pattern described in this paper as part of a more complete pattern language for building Adaptive Object-Models.

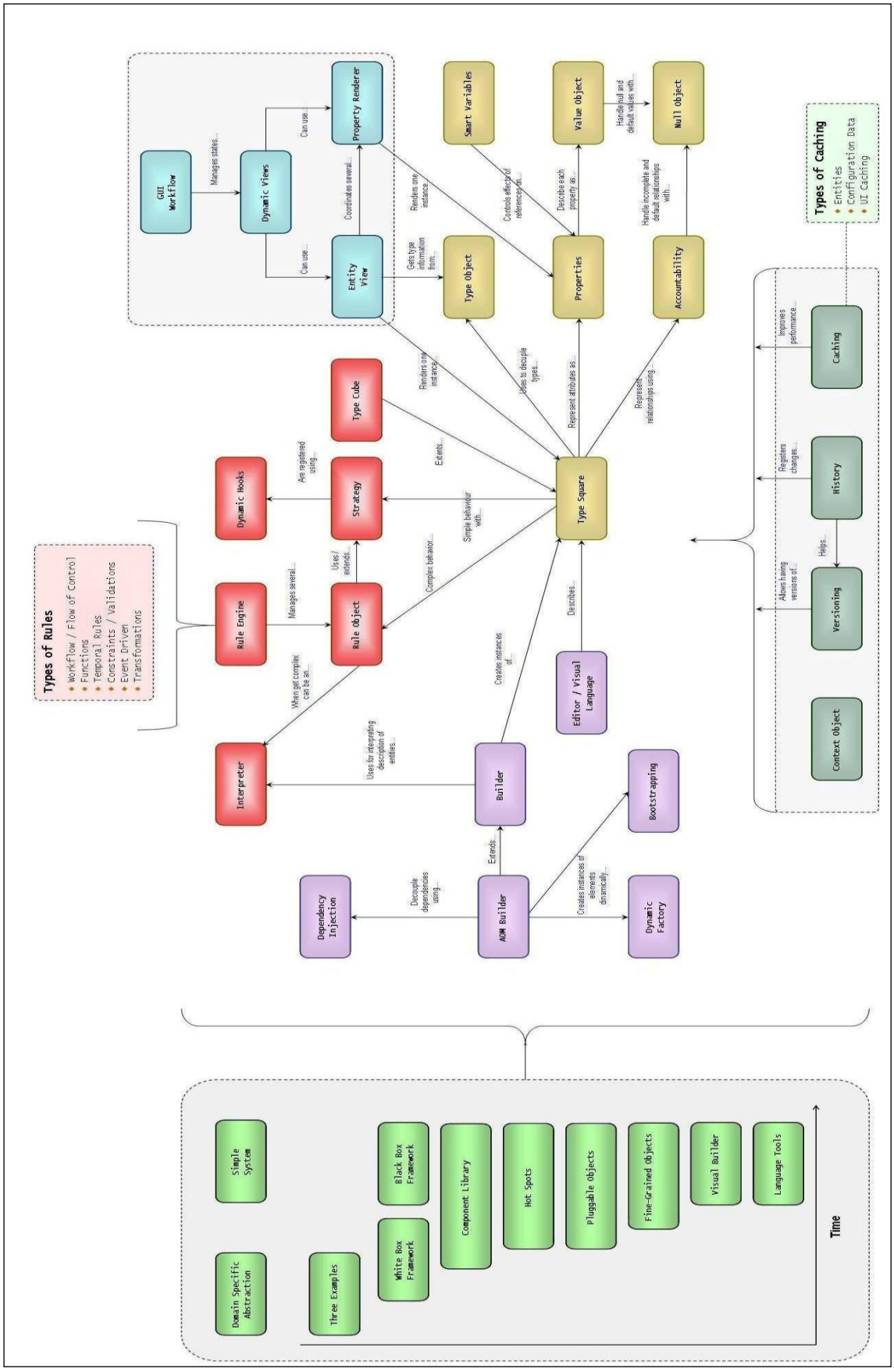

Figure 5 - AOM System patterns and their relationships to other patterns.

5. Acknowledgements

We would like to thank our shepherd Alejandra Garrido for help and advice on improving the contents of this paper. We would also like to gratefully thank to the participants of the PLoP 2009 Architecture Writers Workshop (Brian Foote, Alexander M. Ernst, Eduardo Guerra, Maurice Rabb, and James Siddle), and to Agile 2009 for supporting PLoP 2009 in Chicago, Illinois.

6. References

[AOM] AdaptiveObject-Models. http://www.adaptiveobjectmodel.com

[BR98] Bäumer, D; D. Riehle. *Product Trader*. Pattern Languages of Program Design 3. Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998.

[Beck08] Beck, K. *Implementation Pattern*. Pearson Education Inc. 2008

[Fowler97] Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley. 1997

[Fowler02] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesly. 2002.

[FY98] Foote B, J. Yoder. *Metadata and Active Object Models*. Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998.

[GoF95] Gamma, E.; R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 1995.

[JW98] Johnson, R., R. Wolf. *Type Object*. Pattern Languages of Program Design 3. Addison-Wesley, 1998.

[KSS05] Krishna, A., D.C Schmidt, M Stal. *Context Object: A Design Pattern for Efficient Middleware Request Processing*. 13th Pattern Language of Programs Conference (PLoP 2005), Monticello, Illinois, USA, 2005.

[Martin02] Martin, R. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[POSA1] Buschman, F. et al. *Pattern Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons. 1996

[POSA3] Kircher, M.; P. Jain. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley & Sons. 2004.

[RFBO01] Riehle, D., Fraleigh S., Bucka-Lassen D., Omorogbe N. *The Architecture of a UML Virtual Machine*. Proceedings of the 2001 Conference on Object-Oriented Program Systems, Languages and Applications (OOPSLA '01), October 2001

[RTJ05] Riehle D., M. Tilman, and R. Johnson. "Dynamic Object Model." In *Pattern Languages of Program Design 5*. Edited by Dragos Manolescu, Markus Völter, and James Noble. Reading, MA: Addison-Wesley, 2005.

[RY01] Revault, N, J. Yoder. *Adaptive Object-Models and Metamodeling Techniques Workshop Results*. Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001). Budapest, Hungary. 2001.

[WCJ06] Welicki, L.; J.M Cueva, L. Joyanes. Patterns Meta-Specification and Cataloging: Towards Knowledge Management in Software Engineering Proceedings of the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany, July 2006.

[WYWJ07] Welicki, L.; J. Yoder; R. Wirfs-Brock; R. Johnson. Towards a Pattern Language for Adaptive Object-Models. Companion of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2007), Montreal, Canada, 2007.

[WYW07] Welicki, L, J. Yoder, R. Wirfs-Brock: Rendering Patterns for Adaptive Object Models. 14th Pattern Language of Programs Conference (PLoP 2007), Monticello, Illinois, USA, 2007

[YBJ01] Yoder, J.; F. Balaguer; R. Johnson. *Architecture and Design of Adaptive Object-Models*. Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001), Tampa, Florida, USA, 2001.

[YJ02] Yoder, J.; R. Johnson. *The Adaptive Object-Model Architectural Style*. IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (WICSA 2002), Montréal, Québec, Canada, 2002

[YR00] Yoder, J.; R. Razavi. *Metadata and Adaptive Object-Models*. ECOOP Workshops (ECOOP 2000), Cannes, France, 2000.