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 which is used to construct AOM entities. An AOM Builder first reads externally stored build description to construct a build process which is then executed to construct an AOM entity.

Introduction

An Adaptive Object-Model is a system that represents user-defined classes, attributes, relationships, and behavior as metadata [YBJ01; YJ02]. The system is a model based on instances rather than classes. Users change the metadata (object model) to reflect changes in the domain. These changes modify the system’s behavior. In other words, an AOM stores its Object Model in a database and interprets it. Consequently, the object model is adaptable; when the descriptive information is modified, the system immediately reflects those changes similar to a UML Virtual Machine described by Riehle et. al [RFBO01].

The design of Adaptive Object-Models differ 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. In a sense, what we normally would model as a class is now being modeled by metadata which is being interpreted by the AOM.

Adaptive Object-Model architectures are usually made up of several smaller patterns. TypeObject [JW98] provides a way to dynamically define new business entities for the system. TypeObject 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 can even include ways to define subtypes and relationships between objects.

More information about the AOM architectural style is included in Appendix A. For a comprehensive set of documents and bibliography on the subject, visit www.adaptiveobjectmodel.com.

**Towards an Adaptive Object-Model Pattern Language**

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.

Besides these patterns, less widely-known patterns are often used in AOM systems. In the AOM current literature 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 work through 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 patterns described in this paper as part of a more complete pattern language for building Adaptive Object-Models. Patterns in this pattern language are organized into these 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.

- **Creatational**: 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 auditable.
The pattern presented in this paper (AOM Builder) is included in the **Creational** category.

**Target Audience**

The pattern presented in this paper deals with the creation of instances of entities of AOMs. Therefore, any developer working with this kind of systems (mainly **Type Object**, **Properties** [JW98], and **Type-Square** [YBJ01] or **Dynamic Object Model** [RTJ05] based architectures) would benefit from using the patterns in this paper to cope with presentation issues.

Figure 1 below shows the creational patterns from our AOM pattern language [WYWJ07].

![AOM Pattern Language Diagram](image-url)

**Figure 1 - AOM Pattern Language**
AOM Builder

Context

You are creating an application using an Adaptive Object-Model. Your model relies on a variant of TYPE SQUARE and therefore 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 maintainability problem may arise since the steps to create an instance of entity may vary according to its type or to some arbitrary rules.

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?

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 components with lots of conditional statements according to the type of the entity.
- You don’t want to have an explosion of Builders for each entity type (or to cope with all the conformation rules of the concrete entities).

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. The complex build process is then divided into atomic steps that are executed in order which can share data using a context object [KSS05]. The configuration of the steps can be done dynamically based on external metadata. The configuration of the steps is done based on type, since each type of entity may need different build steps. This also allows you to have a default build procedure which can be arbitrarily extend.
There are two main “sources” of metadata in the implementation of this pattern: the definition of the build steps for each type and the metadata of the persisted entities. The first is used to drive the process, the second to load the AOM entity with information.

The entry point to the building process is provided by the AOMBuilder participant. This entity provides the general interface for creating instances of entities [YBJ01]. This entity can be considered a FACADE [Gof95] to the complex and dynamic building process. It initializes the process, asking for the building pipeline for the BuildStepListFactory (that loads the necessary build steps based on 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 in the build pipeline is specialized as a concrete part of the building process. The building process can be extended with new ConcreteBuild steps. The new BuildStep implementations can be in different packages or assemblies (like the case of ConcreteBuildC in figure 2). The build steps 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 show a canonical implementation of TYPE SQUARE [YBJ01] and are the product of the building process.
In advanced cases, the metadata that indicates the build steps for each type may also contain rules and constraints. This increases considerably the complexity in the implementation, but also allows for a more flexible build process.

The idea behind this pattern is the same as for the BUILDER [GoF95] pattern (dealing with the creation of complex objects in several steps), but it is targeted to a clearly different context (AOM based architectures) and is based on different principles. 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. We could say that the biggest differences are adding support for an extensible build process based on Open-Closed Principle [Martin02] to the solution and using metadata instead of a fixed interface for defining the building process.

Dynamics

Figure 3 shows how the participants interact to get an instance of an existing entity. The Client asks the AOMBuilder for an entity and then a set of messages between the participants are passed with the goal of dynamically loading the entity with its corresponding metadata.

The AOMBuilder is the entry point and the participant in charge of the orchestration of the other participants. It reads the requested entity’s metadata from the metadata repository (through the MetadataReader), creates the building steps pipeline (using the BuildStepListFactory) and creates and sets the building context (creates and loads an instance of BuildContext). After all this “setup” work is done, it runs through the building pipeline (a list of BuildStep instances) and executes them in order (in this case, there are just two build steps, ConcreteBuildStepA and ConcreteBuildStepB). The order of the BuildStep instances is determined in the metadata used for their creation (which is read by the BuildStepListFactory).
The reader may notice that the participants of the TYPE SQUARE pattern (Entity, Entity Type, Property, and Property Type) are not present in figure 3. This omission is deliberate 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 loads it into theBuildContext. Thereafter, only the concrete BuildSteps interact with the Entity or any of its Properties (for loading them or in order to perform any other arbitrary action such as logging, audit, security, tampering checking, etc.).

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 flow is always the same and is contained in the implementation of the AOMBuilder Build() method where the requested entity’s metadata is read, the building context is established, and the creation pipeline is created and executed (code snippet 1).

```csharp
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 = newBuildContext(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.

The variations on the behaviour are dictated by the concrete implementations of the build steps (the implementers of the IBuildInterface, shown in code 2) and how are they configured in the build metadata repository (code 3 shows a configuration file with four steps for creating an instance of an entity).

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

Code 2 – Interface definition for build steps. The information between the buildsteps is passed using a context object, following the Context Object [KSS05] pattern.
The sample configuration shown above 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 case of the PropertiesBuildStep (the step that loads the values into the properties), since each property may need to be handled in a different way. Each step can manage also 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).

Code 4 – Metadata configuration for property loaders. This metadata is used by the PropertyLoader build step.

**Resulting Context**

- The complex process of creating instances of persisted entities is encapsulated in a well known location.
- There is a single point for retrieving the entities.
- Responsibility for creating instances of properties, rules, etc. is factored in fine-grained building step objects.
- Creation code is separated from the consumer code.
✓ The pipeline of the building process is specified in metadata. It can be modified without needing to recompile the application.

✓ The build steps can be modified / extended dynamically.

✓ The build process of persisted entities can be modified dynamically at run-time.

✓ Additional concerns can be added to build process (e.g. by adding a build step for logging, another for security, etc.).

✗ Since the build process is specified in metadata there is no possible compile-time verification.

✗ More complexity. Having several factories with hard-coded rules for creating instances of entities of AOMs (based on each entity and property types) may be simpler but less flexible and extensible.

✗ There is more indirection which can lead to lower performance.

Related Patterns

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

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

The information needed to be shared between build steps is accomplished using the CONTEXT [KSS05] pattern.

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

PROPERTY RENDERERS have code for creating instances of Entities when using TYPE SQUARE.

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].
Appendix - 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 to get 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 TYPE OBJECT pattern is used a second time in order to define the legal types of Attributes, called AttributeTypes. As is common in Entity-Relationship modelling, an Adaptive Object-Model usually separates attributes from relationships.

The STRATEGY pattern [GoF95] is used to define the behaviour 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 behaviours needed for the specified domain.

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 language. This is something that is usually very domain-specific and varies quite a bit.

**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](image)

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.

AOM BUILDER performance can be dramatically enhanced using CACHING [POSA3].
Addendum – An Overview of AOM-Related Patterns
References


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


