Encodo Systems AG – Garnmarkt 1 – 8400 Winterthur
Telefon +41 52 511 80 80 – www.encodo.com

Metadata in Software Development

White Paper

Abstract

This paper sketches a brief background of metadata, lists the advantages and drawbacks to existing approaches and provides some examples on where metadata can be useful. It describes Encodo's approach to metadata with a brief overview of the basic elements and ideas on how to avoid the limitations of existing solutions.

Authors

Marco von Ballmoos (MvB)
Remo von Balmoos (RvB)

Table of Contents

Version History

Version	Date	Author	Comments
1.0	03.07.2007	marco	First Version

Referenced Documents

Nr/Ref	Document	Version	Date
[1]			

Open Issues

Nr./Ref.	Document	Version	Date

Terms and Abbreviations

Term / Abbreviation	Definition / Explanation
Attributes (.NET) Annotations (Java)	Very similar syntax for attaching meta-data directly to code constructs (classes, fields or methods).
CrUD	Create/Update/Delete
D.R.Y.	Don’t Repeat Yourself
GUI	Graphical User Interface
K.I.S.S.	Keep It Simple, Stupid
MDA™	Model Driven Architecture[1]
MDD™	Model Driven Development[2]
MDE	Model Driven Engineering
OMG	Object Management Group
ORM	Object-Relational Mapper

[1] Trademark OMG
[2] Trademark OMG
All trademarks referenced are property of their respective holders and are used for. identification purposes only.

1 What is Metadata?

Metadata is, by definition, data about an application's data. It describes the properties and capabilities of and connections between different types of information in a particular application domain. Examples of application domains are bookkeeping, document management, a lending library or inventory management.

Listed below are a few commonly used terms for modeling methodologies:

- **MDA** – *Model Driven Architecture* – This approach focuses on complete separation of design from architecture, including the use of UML in order to avoid committing to a single programming language. The abstract UML model is transformed to the target architecture and is generally not further available to the application.
- **MDD** – *Model Driven Development* – This approach also focuses on the "systematic use of models as primary engineering artifacts throughout the engineering lifecycle" [Wikipedia]. As with MDA, the model is a first-class entity from which code can be generated, but is not usually represented in code itself.
- **MDE** – *Model Driven Engineering* – Essentially a synonym for MDD, but is not trademarked by the OMG (Object Management Group).

The approach recommended in the following paper can be generally described as MDD, but diverges from traditional approaches in several key ways.

2 Why use Metadata?

All software defines and uses metadata, but usually defines it implicitly, encoding it in the constructs of the programming language (e.g. classes, field, methods, references, etc.) in which it is written. A non-trivial application—one that renders a GUI or responds to a web-service request—needs much, much more information than that. More often than not, that extra information is encoded in application-specific code that is usually not generalized (though it may follow a pattern).

Software increases in complexity and size with time. In order to maintain any sort of control over larger software projects, it is important to strike a balance between the following two precepts:

- **K.I.S.S.** – *Keep It Simple Stupid* – Don't make things more complicated than they need to be for the foreseeable future
- **D.R.Y.** – *Don't Repeat Yourself* – Any single piece of information should exist in exactly one place in an application.

Using a standardized metadata framework helps an application apply the D.R.Y. principle, but seems to violate K.I.S.S. However, applying K.I.S.S. to the metadata framework itself results in a simple API that makes any application using it also much simpler than it would have been with an ad-hoc solution. Using the same simple pattern and library throughout several applications decreases the overall complexity and drastically improves maintainability.

2.1 A Real-World Example

Imagine an application that generates a list of entities, say `Books`. Regardless of whether it generates a report, a web page or a graphical user interface (GUI), it needs to identify columns of data with labels. Traditionally, each application solves the problem of retrieval and internationalization of labels on its own. Programmers without much time or experience—and without framework support—will most likely hard-code the text for the labels, making maintenance more time-consuming and error-prone.

A metadata-based application, on the other hand, starts off with a model, in which the labels for bits of data are already defined. This application need concern itself only with transforming the information in the metadata to the output format and not with retrieval and internationalization of the metadata itself. On top of that, the application is free to use other available metadata which maps to the output format, like color, borders or font-size.

Apps using metadata have the following advantages:

- They address only their assigned tasks, rather than building ad-hoc frameworks for common tasks.
- They are easier to understand and more maintainable because their non–task-specific code uses standard APIs.
- They profit from functionality that would be otherwise impossible to provide within time- or budget-constraints

3 Who Uses Explicit Metadata?

Those few systems that do use explicit metadata do so in order to provide object-relational mapping (ORM) and CrUD (Create/Update/Delete) access to a database. To name just a few:

- Cayenne (Java)
- Hibernate (Java)
- LINQ (.NET)
- Django (Python)

These solutions use various mechanisms to specify the metadata (uniqueness, primary key, relationships, etc.) needed to communicate with a database: Cayenne uses XML files, Hibernate allows either XML files or Java annotations, LINQ uses the class definition along with optional attributes and Django uses inner classes.

Django goes further still by providing metadata for web UIs as well as data access. It uses this metadata to generate the entire administrative back-end—including sortable, filterable lists and CrUD UI for all objects—automatically. The .NET framework has grids that automatically provide CrUD from a LINQ dataset, but these cannot be customized by tweaking central metadata, as in Django[3]. Other frameworks have been similarly inspired—Rails (Ruby) and Grails (Groovy) come to mind—but are still in very early stages and don't seem to use much central metadata. For example, the BeanForm component in Tapestry generates web forms from objects using Java reflection and Hibernate properties (if present). However, it can only be further customized (e.g. styles and classes or layout) with metadata hard-coded in the HTML document.

3.1 Limitations of Existing Implementations

All of the approaches above use the class hierarchy as the model and the reflection/introspection services of the language itself as the metadata API. Metadata not directly expressible in the language is attached using attributes (LINQ), annotations (Hibernate) or inner classes (Django).

[3] The BLINQ library provides more flexibility for .NET, but does not seem to be an officially supported part of the .NET library. See http://forums.asp.net/1076.aspx for more information.

This raises a few issues:

- Instead of being defined centrally, the application’s model is spread throughout its source files.
- The metadata model is tied to a particular implementation (since it is tied to the classes themselves).
- Decorating source code with annotations or attributes is not a scalable way of defining metadata, quickly resulting in an unreadable mess.
- The metadata model is not easily available to the application at run-time (reflection APIs are very low-level and limited for this purpose).

These factors limit the re-usability of metadata from application to application. While a classic GUI application would likely use a set of customized classes to represent its data (in order to benefit from business logic defined therein), a report generator accessing the same data and using the same model would be far better off using generic data objects. If the metadata is defined with classes, the reporting tool must include business logic which it neither needs nor uses.

With the class as model, all possible extensions to the metadata (web-, GUI-, reporting-specific, etc.) are defined in one place. This drastically increases the interdependence between completely different applications. Programmers of GUI applications should not be confronted with web-specific notation throughout their code.

4 A New Approach

With metadata so central to the application, it makes no sense to cede control of it to a single external component (like an ORM). Instead, it must be independently and centrally defined and under application control.

In order to be both truly useful, metadata must satisfy the following conditions:

- It must be independent of any other components in an application.
- It must be available as first-class objects in the application.
- It must not be monolithic (e.g. GUI applications not required to load web-specific metadata).
- There must be a single API for all metadata consumers, be they application components or external components like GUI controls or an ORM.
- Integrating application components must be simple and straightforward, limited by the complexity or API of the component rather than that of the metadata.
- It must not be tied to any single instance representation (e.g. given an entity named `Book`, nothing prevents one application from using class `Book1` to represent an instance while another application uses `Book2`).
- It must be an *enhancement* to the existing programming library and should not, through its use, force replacement of any part of that programming library (e.g. an application should not be required to use only customized web or GUI controls in order to benefit from metadata).

4.1 Integration with Existing Components

None of the requirements listed above places restrictions on the software environment. Rather, it emphasizes only that none of the components gets to control the metadata. For example, it is trivial to generate the data classes needed by Hibernate or LINQ from the metadata. Similarly, nothing prevents a project from generating the in-memory metadata from more traditional modeling approaches like UML.

The application benefits from the centralized metadata, but can continue to any external components without restriction.

4.2 Aspects of Meta-Data

The first step is to define the boundaries of an application domain with entities, properties, operations and relationships (described in section 5 – "Elements of metadata"). However, as mentioned above, an non-trivial application needs specialized metadata in order to work with one or more external components, such as:

- Reading from a database (relational, flat file, XML, etc.)
- Responding to a web-service request
- Generating a user interface (web or classic GUI)

To address these different needs—and to stay centralized but decoupled—metadata is split into aspects, each of which encapsulates one of the tasks listed above. For example, an application might want to store the following information in its metadata:

- Which kind of GUI control to use to edit the value for a meta-property
- How to map a property to a database field (is it nullable, is it persistent, etc.)

Aspects that are very similar, like "gui" and "web", can put shared metadata into another aspect (e.g. "view"). This way, common metadata, like "color" and "font-size" are in the "view" aspect, while browser-specific metadata is defined in the "web" aspect. A desktop application includes the "gui" and "view" aspects, while a web application includes "web" and "view". The reporting tool mentioned above includes only "view" so that it has access to the required display metadata.

5 Elements of metadata

The sections below briefly sketch the basic parts of metadata and are not meant to be comprehensive.

Since metadata is descriptive, all of its elements have the following basic properties:

- title (singular and plural)
- description

Complicating things slightly, each textual property—like the "singular title" above—has multiple values, one for each language supported by the application. In addition to these basic, shared properties (of which there can be many, many more), specific types of elements add other features.

A lending library application is used in the examples below.

5.1 Entities

Entities describe particular types of objects in the application domain, like books, authors, publishers or customers. Less obvious entities are languages, media types, genres or lending transactions (when a customer loans a book for a period of time). Entities include the following lists:

- properties
- operations
- relationships

Each of these is described in more detail in the following sections.

5.2 Properties

A property is a single feature of an entity, like a book's title, an author's first or last name or the due date for a loaned book. In addition to the basic features above (title, description), a meta-property has the following features:

- **Type** – Integer, String, etc.
- **IsPrimary** – The property is in the primary key of the entity
- **Validations** – required greater than a value, etc.

There are, of course, dozens of other features that an application might associate with a property, but things like "color", "font-size" or "visible" are only interesting to particular application domains. Therefore, these features belong in domain-specific aspects (as described above in section 4.2 – "Aspects of Meta-Data").

5.3 Operations

An operation is an action that can be executed against an entity (or, more precisely, an instance of an entity). In addition to the basic features above (title, description), a meta-operation has the following features:

- **Signature** – Number and types of parameters
- **Implementation** – The program logic to execute when called

Including the signature in the metadata is useful for validation, but the implementation is best left in the application itself (including it in the metadata would violate K.I.S.S.).

5.4 Relationships

An application domain consists of more than just free-floating entities: it is the relationships between those entities that truly describe what is possible within a metadata model. A book has a list of authors as well as a publisher, whereas publishers and customers have lists of books. A relationship has the following features:

- **Source** – Entity and properties to map *from*
- **Target** – Entity and properties to map *to*
- **Cardinality** – How many elements can be in the target
- **Filter** – A condition restricting the entities in the target

As with properties, these are the basic features that all relationships have to describe them fully; domain-specific aspects may add more.

6 No Going Back

Using metadata explicitly is the kind of approach that comes after years of experience working with other methodologies and technologies. It arises from a need to avoid re-inventing the wheel with each new application. We started Encodo after working for years with modeling tools that offered some—though not all—of the advantages mentioned in this paper.

Most available components and libraries, however, don't work with metadata as we'd grown accustomed to. We tried using them as-is but, soon enough, realized we could adhere to neither K.I.S.S. nor D.R.Y. principles. Having had the advantage of using metadata in previous applications, our standards were raised to a level where we were no longer willing to work without it.