

# jMDA: A Simple Approach to Model Driven Architecture

- White Paper -

*by Roger Wegner*



# **jMDA: A Simple Approach to Model Driven Architecture**

- White Paper -

*by Roger Wegner*

## **Abstract:**

jMDA is a simple yet very powerful approach to model driven architecture (MDA) that has a strong focus on leveraging only few but standardised Java technologies. It largely relies on tools and APIs delivered with the Java Development Kit 5 (and above), namely the annotation processing tool and the associated Mirror API. As necessary these basic technologies can be complemented by others to fit particular demands.

Inherently MDA is a complex area and there are many MDA approaches that even add complexity by introducing proprietary frameworks, platforms, tools, languages, libraries, process models, workflow engines and many more. As a result many people generally regard MDA as oversized and / or esoteric and they don't see MDA to be applicable successfully in time and within budget in real world's software development projects. In addition to that many MDA approaches are regarded as too difficult to understand, to customize or to extend if e. g. the generated artefacts do not meet actual needs.

In contrast to many well known MDA approaches jMDA is a KISS (keep it simple and stupid) approach with a minimum of dependencies to other technologies, a maximum of flexibility and extensibility and that is both easy to learn and easy to use.

Undoubtedly the generation of high quality source code and other artefacts is a very important aspect of MDA. Although prefixed with a "j" jMDA is not at all limited to the generation of Java source code. jMDA supports common MDA proceedings like transformations of platform independent models and can easily be targeted to generation of various artefacts for arbitrary technical platforms.

This paper summarises some intentions and techniques that are followed by the jMDA approach and outlines how jMDA may contribute to higher efficiency and quality of the overall software development process.

## Modelling Language

The jMDA modelling language is the static type system of the Java 5 (and above) programming language. What does that mean?

It means that the application domain can be modelled using both standard and custom Java types together with Java annotations to express metadata if necessary. This needs some additional explanation. Think of the following simple application domain: A company has employees. This application domain can be expressed by the following Java source code:

```
class Employee
{
    String name;
}

class Company
{
    String name;
    Set<Employee> employees;
}
```

It is hard to think of an even more simple source code representation of the application domain. Although this code compiles and may be executed it is not meant to be used in a real software system<sup>1</sup>. In jMDA there is a strict distinction of Java code representing a domain model and code that is generated to be executed as part of a software system.

## Modelling and the jMDA Process

The development of domain models is crucial for any MDA approach and it has just been said that the jMDA modelling language is the static type system of the Java 5 and above programming language. Hence an obvious way to develop jMDA models would be to open a text editor or Java IDE and write Java 5 source code.

Fortunately this is not the only way to go. In early phases of real world's software development projects software engineers make workshops and interviews with domain experts to learn about their application domain. Often these domain experts do not have any software development experience and so there would usually be great irritation if you presented them a bunch of Java source code files as the result of your interviews and workshops and ask them what they think about the domain model you developed for them.

Experience shows that domain experts become familiar with a model even of their own application domain more easily if it is written down using a graphical notation. With jMDA you can for example use the widely adopted unified modelling language (UML) and one of the well established UML tools to develop domain models. Illustration 1 outlines a typical jMDA process: The starting point is the elaboration of a domain model.

---

<sup>1</sup> But without going into details it can be said that if a domain model is compilable and executable this makes sure that the model has a certain formal correctness.

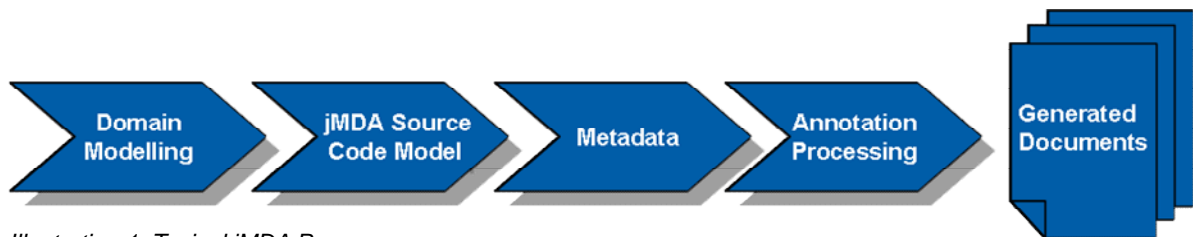


Illustration 1: Typical jMDA Process

If you wish to evolve your domain model iteratively and if you like to represent your model using UML a respective tool that supports round trip engineering will integrate best into your modelling / development process. Illustration 2 shows a UML representation of the Company / Employees domain model.

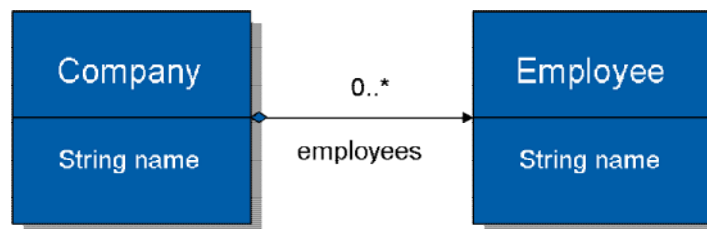


Illustration 2: UML Class Diagram Representation of Company / Employees Domain Model

In a jMDA process the result of a domain modelling iteration is a Java source code model. If the domain model was developed using an up to date UML tool it is usually possible to export the relevant UML classes to Java source code like that shown above.

However simple UML models as well as simple Java source code files often do not provide enough information for ambitious artefact generation as it is intended by jMDA. There are several approaches to address this shortcoming – jMDA simply makes use of Java annotations to provide metadata for the jMDA code model. It's possible to use standard JDK annotations as well as custom annotations that were defined for special purposes. The definition of custom annotations is pretty simple and at the same time provides enormous possibilities for jMDA.

Based on the domain model that was charged with useful metadata information the annotation processing may start and generate the desired documents.

### Processing the Domain Model

In order to generate anything from a domain model you need to have programmatic access to the details of the model. jMDA models can be processed comfortably using the annotation processing tool (apt) shipped with JDK 5 and above. The following outlines the general apt process:

- a) Create the Java source code files that represent your domain model and annotate your model with metadata annotations as necessary<sup>2</sup>.
- b) Define an annotation processor factory. An annotation processor factory is a Java class with the main purpose to return an annotation processor (see below). The annotation processor factory is invoked automatically by apt.
- c) Define an annotation processor. An annotation processor is a Java class that also is invoked automatically by apt and that has fine grained access to the details of the domain model.

Now what does “fine grained access to the details of the domain model” mean? Annotation processors may use the Mirror API to find out about the structural aspects of Java types that are defined in Java source code files. The Mirror API documentation puts it like this: “It [the Mirror API] provides representations of the entities declared in a program, such as classes, methods, and fields. Constructs below the method level, such as individual statements and expressions, are not represented.”<sup>3</sup>

In fact there are plenty of ways to process a domain model based on Java source code with an apt style processor. The following outlines just one possibility: To get a starting point all declarations that are annotated with a particular annotation may be queried. The resulting list can be iterated and all type declarations can easily be found out. If a type declaration is a class declaration (there also are interface, annotation and enumeration declarations) one might be interested in the fields or methods that are declared in that particular class.

### Model Driven Generation

The information about the domain model provided by apt and the Mirror API is extremely useful to generate various artefacts like source code (not only Java source code), configuration files, SQL schema files, unit tests and so on. The jMDA generation process is started within apt annotation processors and it can be tailored deliberately to produce the desired output.

The apt program shipped with JDK 5 is an executable command line program similar to javac. However processing a domain model using a command line program is somewhat uncomfortable especially when it comes to managing different apt start configurations or even debugging the execution of a processor. To overcome these shortcomings jMDA provides a wrapper for apt with a graphical user interface called APT Launcher – download: [www.jmda.de](http://www.jmda.de) – APT Launcher allows comfortable configuration of apt, management of such configurations with configuration files and, if started in debug mode, debugging of annotation processing.

Illustration 3 shows the graphical user interface of APT Launcher. In the upper part the set of files that store the jMDA code model is specified. For the example just the location (direc-

- 
- 2 To get apt running there has to be at least one (maybe empty) annotation in your domain model.
  - 3 The Mirror API provides much more detailed and flexible access to the information that is contained in source code models than the Java Reflection API. Not only that it provides access even to the non public declarations of Java types, it also makes e. g. javadoc comments accessible and does not require the Java types of the source code model to be loaded at runtime.

tory) of the Company and Employee source code model files is listed in the directories area. For more complex models where source code model files are distributed over several sub directories the respective root directories may be specified in the root directories area. If there are particular files that belong to the model they can be specified in the files area. For each area there also is the possibility to exclude root directories, directories or particular files.

If the annotation processor needs a custom execution classpath it can be set in the custom classpath area and the annotation processor factory path can be specified in the respective area below. At the bottom a dedicated annotation processor factory can be declared. In the example the annotation processor factory developed for this example is specified.

Pressing the launch button starts the annotation processor with the currently loaded configuration. The file menu offers options to save the launch configuration to a file and to load saved configurations. If APT Launcher was started in debug mode the execution will stop at the breakpoints that you defined. Needless to say that having a debugger at hand is invaluable for most software development projects and so it is for developing jMDA generators.

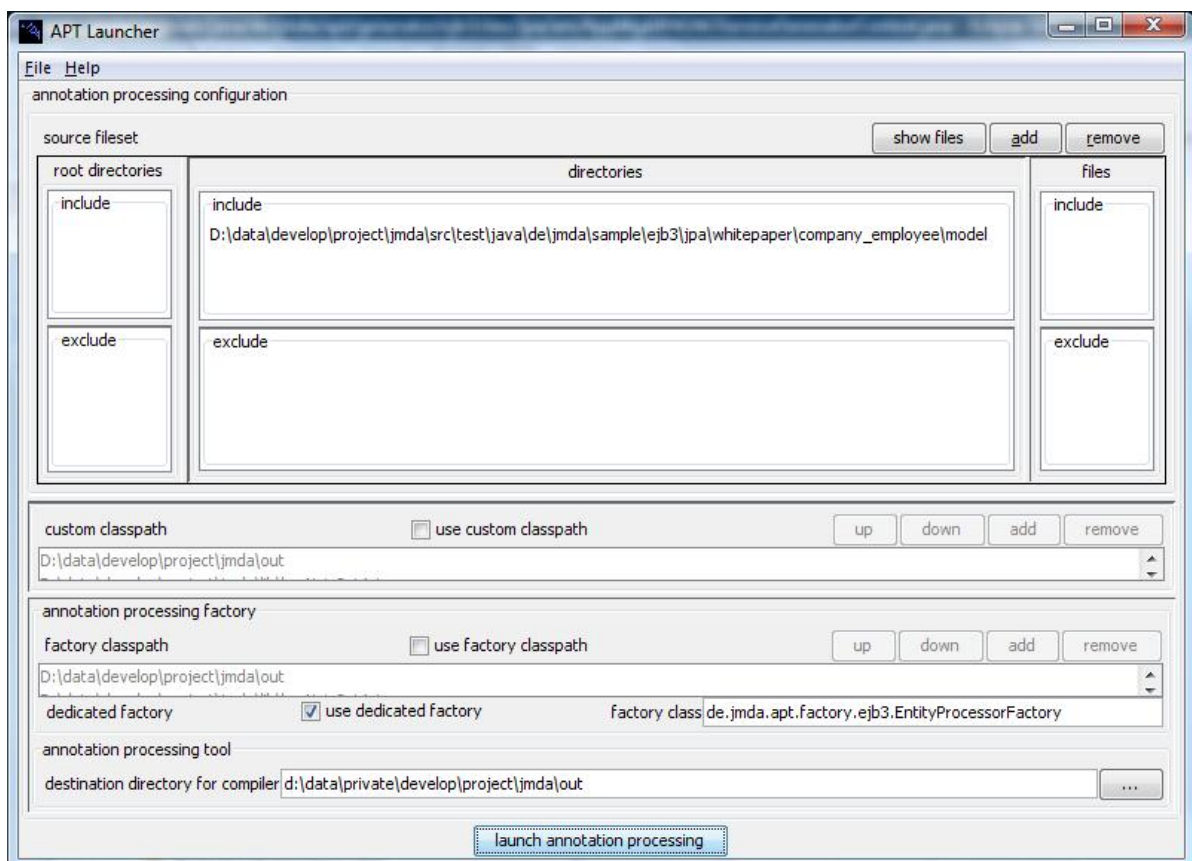


Illustration 3: GUI of APT Launcher

## jMDA in Action

So far some central concepts and intentions of jMDA were summarized. This chapter shows jMDA in action. The example presented here will add Java Persistence (JPA) to the Company and Employee classes that were introduced earlier.

As in other MDA approaches the starting point for jMDA is the definition of a domain model (here: a company has employees). Following the “keep it simple and stupid” paradigm of jMDA the domain model at first only contains the minimum amount of information that is necessary to express the relevant aspects as shown in chapter “Modelling Language”. To enable a jMDA annotation processor to provide persistence some metadata has to be added. Here the JPA annotation `javax.persistence.Entity` is used to define which classes should become persistent.

As Illustration 2 shows the sample domain model also has a relation. To demonstrate just a few capabilities of the jMDA approach the annotation processor that was implemented for this example has some built in functionality for convenient handling of persistent relations: if not explicitly specified otherwise it automatically handles collection type attributes of the model (such as type `java.util.Set<E>`) as a persistent unidirectional “one to many” relation. For demonstration purposes the default behaviour can be accepted here and so the usual `javax.persistence.OneToOne` JPA annotation is omitted for the relation in the sample model. In the following there will be shown how the sample processor adds the annotation to the generated code that still is required by JPA.

So the source code for the sample domain model is as follows:

```
@Entity
class Employee
{
    String name;
}

@Entity
class Company
{
    String name;
    Set<Employee> employees;
}
```

This simple Java source code is the complete input for the sample annotation processor. To generate full-blown classes with JPA persistence the processor at least has to accomplish the following tasks:

- a) The domain model should be left untouched by the generation process. This is a general recommendation that is followed in the example and thus the processor has to create new Java source code files for the persistent classes it generates.
- b) In JPA persistent classes have to possess an id representing a database table's primary key. The model does not expose one so the processor will create a default id field together with appropriate accessors.
- c) JPA requires persistent classes to have a default no-arg constructor.

To avoid an overload of the example the requirements for the annotation processor were intentionally reduced to a minimum (except for the mentioned built in functionality for con-

venient handling of persistent relations). Later there will be a short discussion of what else might easily be generated even on a model as simple as the sample model. In the following the few simple steps that have to be made to get the sample annotation processor running were outlined. After that the code generation process will be discussed.

As mentioned before first of all an annotation processor factory has to be defined. The main task for such a factory is to tell apt which annotation processor to instantiate and start when a particular annotation in the source code is discovered. Therefore the factory implements the Mirror API interface method `AnnotationProcessorFactory.getProcessorFor()`. In the example the implementation of this method takes just one line in which an annotation processor is created and returned:

```
return new EntityProcessor(  
    annotationProcessorEnvironment, annotationTypeDeclarations);
```

The first parameter deserves some special attention as the annotation processor environment provides the linchpin for access to the Mirror API which is used extensively by typical jMDA code generators.

An annotation processor has to implement the interface method `AnnotationProcessor.process()`. The first thing that happens here in the sample processor is that the querying features of the annotation processor environment are used to retrieve all jMDA source code model declarations that are annotated with the Entity annotation:

```
annotationProcessorEnvironment.getDeclarationsAnnotatedWith(Entity.class)
```

Not surprisingly two declaration objects will be returned for the example model, one representing class `Company` and the other representing class `Employee`.

The second thing happening in `EntityProcessor.process()` is that for each queried declaration object an `EntityGenerator` object is created and started. Together with the respective declaration object the processor hands over the annotation processing environment to the entity generator objects and in fact this is everything the sample annotation processor does.

At this point the example fulfils all requirements for annotation processing with the JDK apt tool by following the few conventions just described above. The main benefit for the generation process is that the custom class `EntityGenerator` obtains full access to the Mirror API which will turn out to be extremely useful for the generation task at hand. In the following there will be a short discussion about some details of the code generation for the sample domain.

### Good Practise – Decomposition of Generators

It has already been described how an `EntityGenerator` object is created and started for each class declaration of the domain model and the main tasks for the generator have also been summarised briefly. Now it will be outlined how `EntityGenerator` solves these tasks.

The first task is to generate new files for the domain object. Therefore a file name and maybe more interesting a package and a respective file location have to be specified. jMDA

offers utilities that can help with this (and many other issues). For example the `jMDA TargetPackageNameBuilder` evaluates metadata annotations that may be associated with the model's type or package declarations to determine the correct location and respective package. `TargetPackageNameBuilder` already is pretty capable but if this solution is still not sufficient a custom strategy can be used. This is only one example where the enormous flexibility of the jMDA approach shows up: jMDA offers a growing set of functionality for many tasks related to code generation that can be used as is, that can be extended or that can even be replaced completely.

The second task is to generate a .java source code file that contains a class with all the attributes defined in the domain model plus the primary key and the code for the accessors, the mutators and the default constructor. In the example the popular apache velocity templating engine is used to organise the generation process. The basic idea that a templating engine follows is to define a template text file that contains the static parts of an artefact type to be generated and markers for those parts that are built up dynamically during the generation process:

```

package ${context.getPackageName()};

${context.getClassJavaDoc()}
${context.getClassAnnotations()}
${context.getClassModifiers()}class ${context.getClassSimpleName()}
    ${context.getExtendsClause()}
    ${context.getImplementsClause()}
{
    // field declarations -----
    ${context.getFieldDeclarations()}

    // constructor -----
    public ${context.getClassSimpleName()}()
    {
    }

    // field accessors -----
    ${context.getFieldAccessors()}
}

```

In the template file above the dynamic parts are **highlighted**. When starting the velocity engine a context object has to be provided and this context object has to implement the methods that are called in the dynamic parts of the template. In the above template almost everything is dynamic and one might argue that introducing a templating engine to produce such little static output is like breaking a fly on a wheel. However the reason for this supposed disproportion is the intended simplicity of the example. And there is one thing that can be made very clear by this example: the more dynamic parts exist in a template the more reusable it becomes because it is easy to provide new context objects that inherit the features of existing objects and override or extend this functionality as necessary.

How does the implementation of such a context object look like? In jMDA the generators typically pass on the parameters that were injected by the jMDA annotation processors, namely the annotation processor environment and the declaration that is currently processed. Equipped with these objects a typical implementation of the `getFieldAccessors()` method could look like this:

```

/**
 * @return all complete field declarations
 */
public String getFieldDeclarations()
{
    // if necessary start with the @Id field declaration,
    // start with empty string buffer otherwise
    StringBuffer result = new StringBuffer(getFieldDeclarationId());

    for (FieldDeclaration fieldDeclaration : classDeclaration.getFields())
    {
        result.append(getFieldDeclaration(fieldDeclaration));
    }

    return result.toString();
}

```

Using the Mirror API this implementation simply iterates over all field declarations of the given class declaration from the domain model and delegates to the `getFieldDeclaration()` method. Additionally there is a delegation to the method `getFieldDeclarationId()` that is responsible for the generation of the primary key field in this example. The implementation of `getFieldDeclaration()` follows the same decomposition pattern:

```

protected String getFieldDeclaration(FieldDeclaration fieldDeclaration)
{
    StringBuffer result = new StringBuffer();
    ...
    result.append(comment + "\n\t")
        .append(getFieldAnnotations(fieldDeclaration) + "\t")
        .append(getFieldModifiers(fieldDeclaration))
        .append(fieldType + " ")
        .append(fieldDeclaration.getSimpleName())
        .append(getFieldInitialisation(fieldDeclaration) + ";\n\n");

    return result.toString();
}

```

It is obvious how these public or rather protected methods can easily be overridden if they don't fulfil actual needs and creating elaborate hierarchies for classes that provide functionality for the generation process is one of the main techniques that help JMDA achieve its enormous flexibility and extensibility.

Now finally here is an overview of the source code model that serves as input for the sample annotation processor and the code that the underlying generator emits. First here is the source code model from above once again:

```

@Entity
class Employee
{
    String name;
}

@Entity
class Company
{
    String name;
    Set<Employee> employees;
}

```

And here is the code that is generated from the above input:

```

@Entity
public class Employee implements Serializable {
    // field declarations -----
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    // constructor -----
    public Employee() {
    }

    // field accessors -----
    /**
     * getter for {@link #id}
     */
    public Long getId() {
        return id;
    }

    /**
     * setter for {@link #id}
     */
    protected void setId(Long id) {
        this.id = id;
    }

    /**
     * getter for {@link #name}
     */
    public String getName() {
        return name;
    }

    /**
     * setter for {@link #name}
     */
    public void setName(String name) {
        this.name = name;
    }
}

@Entity
public class Company implements Serializable {
    // field declarations -----
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    @OneToMany
    private Set<Employee> employees = new HashSet<Employee>();

    // constructor -----
    public Company() {
    }

    // field accessors -----

```

```
/**
 * getter for {@link #id}
 */
public Long getId() {
    return id;
}

/**
 * setter for {@link #id}
 */
protected void setId(Long id) {
    this.id = id;
}

/**
 * getter for {@link #name}
 */
public String getName() {
    return name;
}

/**
 * setter for {@link #name}
 */
public void setName(String name) {
    this.name = name;
}

/**
 * getter for {@link #employees}
 */
public Set<Employee> getEmployees() {
    return employees;
}

/**
 * setter for {@link #employees}
 */
public void setEmployees(Set<Employee> employees) {
    this.employees = employees;
}
}
```

The generated code contains significant enhancements compared to the very simple input source code model. These enhancements are necessary for JPA entities and the generated code fulfils all the requirements that were defined earlier. However it should be clear that this example gives only a small impression of the possibilities available with jMDA.

## Summary and Conclusion

The introduction to this paper claims that jMDA is a simple yet powerful approach to model driven architecture. The paper hopefully shows that both demands can be accomplished.

jMDA integrates smoothly into common software development processes. It supports modelling with UML and other modelling approaches, provides a convenient way to enrich domain models with expressive metadata, offers a comfortable tool to

govern the code generation process, supports debugging, uses and provides APIs that proved to be extremely useful for source code generation and so on. The outlined example shows how easy it is to create, customise and extend code generation.

Though this paper only presents a very simple example it should have made obvious how jMDA can be used to create significant parts of sophisticated software systems. For example jMDA was used successfully to create a complete persistence layer including persistent entity classes, DAOs and other service classes for a distributed software system. Using jMDA it is easily possible to automatically produce configuration files, unit tests, SQL schema files, constraint verification code and many more.

The simplicity of the jMDA approach shows up when you start a project and experience how easy it is to get the first generated results. You can find a simple example together with the APT Launcher tool and some brief instructions on how to get started at [www.jmda.de](http://www.jmda.de). The basic approach relies on only a minimum of commonly used technologies. This constitutes the certainty of control over the development process and takes away the myth of MDA being esoteric and non applicable. I hope that you can share this opinion after you tried out jMDA and I'd be very thankful if you would share your experiences and suggestions.

#### About the Author



Roger Wegner is a Senior System Consultant at MT AG, Ratingen, Germany.

He has long lasting experience from many software developing projects mainly in the Java Enterprise area.

You can contact him by e-mail at [roger.wegner@mt-ag.com](mailto:roger.wegner@mt-ag.com)  
Your comments are very welcome.

© Copyright 2008, MT AG Ratingen

Balcke-Dürr-Allee 9  
40882 Ratingen  
Tel. +49 (0) 21 02 309 61-0  
Fax +49 (0) 21 02 309 61-10  
[info@mt-ag.com](mailto:info@mt-ag.com)  
[www.mt-ag.com](http://www.mt-ag.com)