

AndroMDA

INTRODUCTION

AndroMDA (pronounced "Andromeda") is an extensible generator framework that adheres to the Model Driven Architecture (MDA) paradigm. Models from UML tools will be transformed into deployable components for your favorite platform (J2EE, Spring, .NET). Unlike other MDA toolkits, AndroMDA comes with a host of ready-made cartridges that target today's development toolkits like Axis, jBPM, Struts, JSF, Spring and Hibernate.

This information was extracted from the "Getting Started Java" tutorial from (http://galaxy.andromda.org/index.php?option=com_content&task=category§ionid=11&id=42&Itemid=89).

1	INTRODUCTION	1
1.1	WHAT IS ANDROMDA?	1
2	APPLICATION ARCHITECTURE	2
2.1	ARCHITECTURE OF ANDROMDA GENERATED APPLICATIONS.....	3
2.2	DATA PROPAGATION BETWEEN LAYERS	4
2.3	SERVICES AND HIBERNATE SESSIONS.....	4
3	TIMETRACKER TOUR	5

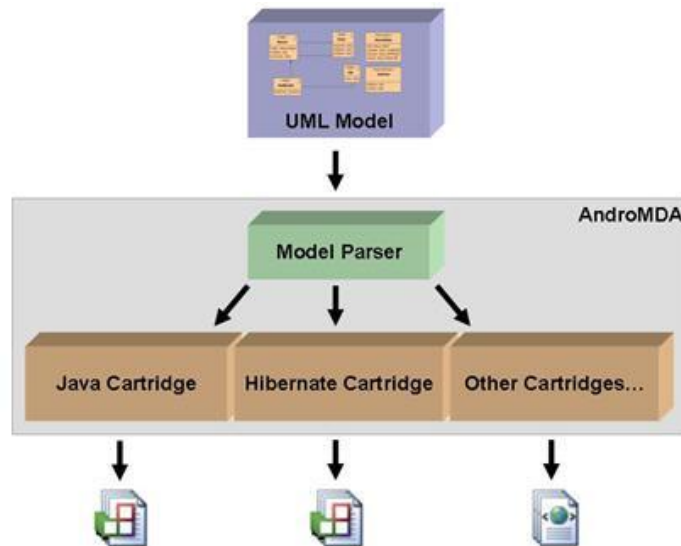
1 Introduction

1.1 What is AndroMDA?

AndroMDA (pronounced "Andromeda") is an extensible generator framework that adheres to the Model Driven Architecture (MDA) paradigm. It transforms UML models into deployable components for your favorite platform. While AndroMDA ships with cartridges that can generate code for several platforms and technologies, this tutorial will focus on generating a Java application.

The paragraph above might have sounded like a lot of acronyms and marketing speak, but using AndroMDA means one main thing: write less code. Not only that, AndroMDA also lets you create better applications and maintain order on large projects. AndroMDA enforces best practices and lets developers focus on high level problems instead of wasting time on repetitive plumbing code. Additionally, AndroMDA can generate highly customized enterprise quality code to meet your project's very special needs.

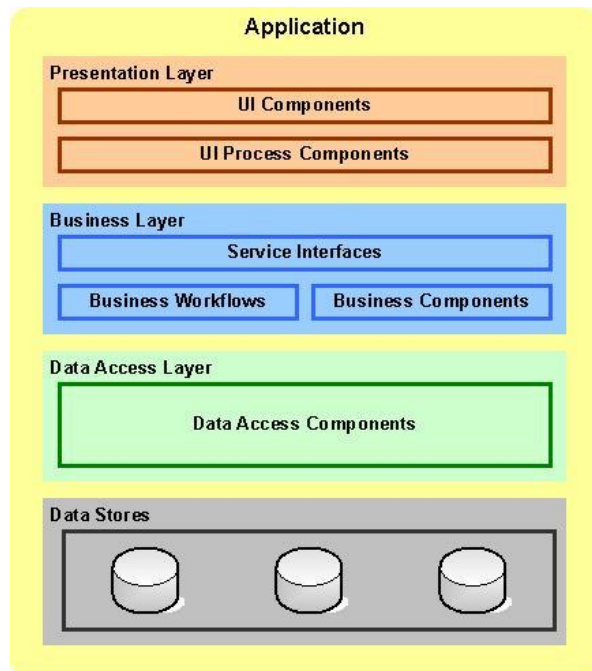
During development of large applications, most architects and developers already create class diagrams and data diagrams. These diagrams are usually made in tools like Visio, and the resulting artifacts are static pictures. When code changes, the diagrams must be updated. With AndroMDA, these diagrams become a living part of your application -- they are used to generate large portions of your application, and hence always reflect the current state of the system. When you need to modify your application, you change the model first, regenerate the code, and then add or update custom code as necessary. Thus, you get a production quality application out of assets that you had to create anyway.



AndroMDA provides several cartridges out-of-the-box. For example, the Hibernate and Spring cartridges generate robust service and data layers for your application. In addition, database schema can be exported to script files to allow the creation of your application's database. There is also an easy way to map your model to an existing schema if your database has already been defined. If you wish to generate custom artifacts from your model, you can write a custom cartridge to accomplish this.

2 Application Architecture

Modern enterprise applications are built using several components connected to one another, each providing a specific functionality. Components that perform similar functions are generally grouped into layers. These layers are further organized as a stack where components in a higher layer use the services of components in a lower layer. A component in a given layer will generally use the functionality of other components in its own layer or the layers below it. The diagram below shows a popular layer structure for an enterprise application.



Presentation Layer: The presentation layer contains components needed to interact with the user of the application. Examples of such components are web pages, rich-client forms, user interaction process components etc.

Business Layer: The business layer encapsulates the core business functionality of the application. Simple business functions can be implemented using stateless components, whereas complex, long-running transactions can be implemented using stateful workflows. The

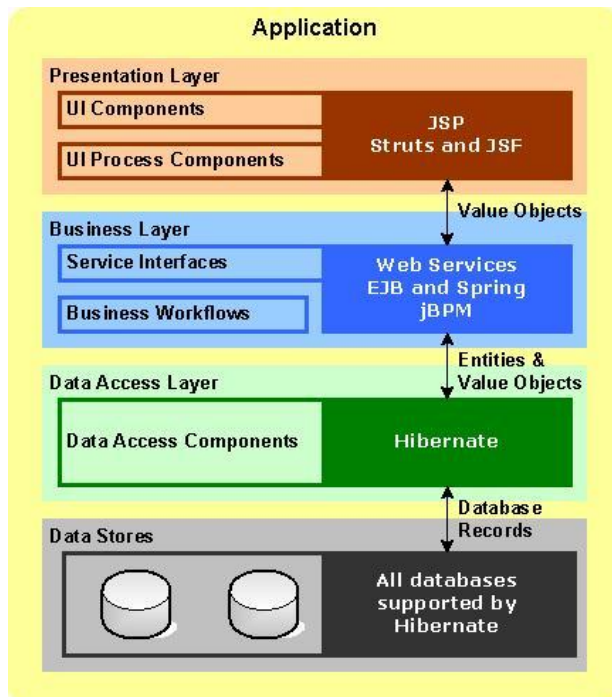
business components are generally front-ended by a service interface that acts as a facade to hide the complexity of the business logic. This is commonly known as Service-Oriented Architecture (SOA).

Data Access Layer: The data access layer provides a simple API for accessing and manipulating data. The components in this layer abstract the semantics of the underlying data access technology thus allowing the business layer to focus on business logic. Each component typically provides methods to perform Create, Read, Update, and Delete (CRUD) operations for a specific business entity.

Data Stores: Enterprise applications store their data in one or more data stores. Databases and file systems are two very common types of data stores.

2.1 Architecture of AndroMDA Generated Applications

Now that we understand the basic concepts behind modern enterprise applications, let's discuss how AndroMDA implements these concepts. AndroMDA takes as its input a business model specified in the Unified Modeling Language (UML) and generates significant portions of the layers needed to build a Java application. AndroMDA's ability to automatically translate high-level business specifications into production quality code results in significant time savings when implementing Java applications. The diagram below maps various application layers to Java technologies supported by AndroMDA.



Presentation Layer: AndroMDA currently offers two technology options to build web based presentation layers: Struts and JSF. It accepts UML activity diagrams as input to specify page flows and generates Web components that conform to the Struts or JSF frameworks.

Business Layer: The business layer generated by AndroMDA consists primarily of services that are configured using the Spring Framework. These services are implemented manually in AndroMDA-generated blank methods, where business logic can be defined. These generated services can optionally be front-ended with EJBs, in which case the services must be deployed in an EJB container (e.g., JBoss). Services can also be exposed as Web Services, providing a platform independent way for clients to access their functionality. AndroMDA can even generate business processes and workflows for the jBPM workflow engine (part of the JBoss product line).

Data Access Layer: AndroMDA leverages the popular object-relational mapping tool called Hibernate to generate the data access layer for applications. AndroMDA does this by generating data access objects (DAOs) for entities defined in the UML model. These data access objects use the Hibernate API to convert database records into objects and vice-versa. AndroMDA also supports EJB3/Seam for data access layer (pre-release).

Data Stores: Since AndroMDA generated applications use Hibernate to access the data, you can use any of the databases supported by Hibernate.

2.2 Data Propagation Between Layers

In addition to the concepts discussed previously, it is important to understand how data propagates between various layers of an application. Follow along the diagram above as we start from the bottom up.

As you know, relational databases store data as records in tables. The data access layer fetches these records from the database and transforms them into objects that represent entities in the business domain. Hence, these objects are called business entities.

Going one level up, the data access layer passes the entities to the business layer where business logic is performed.

The last thing to discuss is the propagation of data between the business layer and the presentation layer, for which there are two schools of thought. Some people recommend that the presentation layer should be given direct access to business entities. Others recommend just the opposite, i.e. business entities should be off limits to the presentation layer and that the business layer should package necessary information into so-called "value objects" and transfer these value objects to the presentation layer. Let's look at the pros and cons of these two approaches.

The first approach (entities only, no value objects) is simpler to implement. You do not have to create value objects or write any code to transfer information between entities and value objects. In fact, this approach will probably work well for simple, small applications where the presentation layer and the service layer run on the same machine. However, this approach does not scale well for larger and more complex applications. Here's why:

- Business logic is no longer contained in the business layer. It is tempting to freely manipulate entities in the presentation layer and thus spread the business logic in multiple places -- definitely a maintenance nightmare. In case there are multiple front-ends to a service, business logic must be duplicated in all these front-ends. In addition, there is no protection against the presentation layer corrupting the entities - intentionally or unintentionally!
- When the presentation layer is running on a different machine (as in the case of a rich client), it is very inefficient to serialize a whole network of entities and send it across the wire.
- Passing real entities to the client may pose a security risk. Do you want the client application to have access to the salary information inside the Employee object or your profit margins inside the Order object?

Value objects provide a solution for all these problems. Yes, they require you to write a little extra code; but in return, you get a bullet-proof business layer that communicates efficiently with the presentation layer. You can think of a value object as a controlled view into one or more entities relevant to your client application. Note that AndroMDA provides some basic support for translation between entities and value objects, as you will see in the tutorial.

2.3 Services and Hibernate Sessions

Another key concept of AndroMDA-generated applications is the strong association between service methods (i.e., operations exposed by a service) and Hibernate sessions. But before we introduce this concept, we must lay out some ground work. The Hibernate session is a runtime object that allows an application to create, read, update and delete entities in the data store. As long as the session is "open", these entities are attached to the session and you can navigate from one entity to another using relationships between them. If a related entity is not yet in

memory, Hibernate will automatically pull it in for you (this is called "lazy loading"). However as soon you close the Hibernate session, the entities that exist in memory are considered to be "detached"; i.e. Hibernate no longer knows about them. You are free to hold references to such entities, but Hibernate will no longer pull in associated entities if they don't exist in memory already. If you accidentally try to access such associated entities, you will get a Hibernate `LazyInitializationException`.

Now that we understand this background material, let us discuss the relationship between a service method and a Hibernate session. When a client application calls a service method, a new Hibernate session is opened automatically -- you do not have to write any code to do this. Similarly, when the service method exits, the associated Hibernate session is closed automatically. In other words, the lifespan of a Hibernate session is bounded by the beginning and ending of a service method call. Consequently, entities are attached to Hibernate session for the entire duration of the service call, but are detached from this session as soon as the service call ends. As a result, if your service method returns raw entities, the client must be extra careful not to access related entities that are not in memory already. You can avoid all this mess by following the recommendation in the earlier section; that is, transfer all relevant information into value objects while the session is still open, and pass these value objects back to your caller as a return value. In general, think of a service method as a logical transactional boundary - do everything you need to do within the method and then return the results as value objects.

Another implication of the strong association between a service method and a Hibernate session is that client applications should not try to bypass the service layer and interact directly with the lower layers. You may be able to brute force your way into one data access object, but sooner or later you will get into trouble!

3 TimeTracker Tour

Before we start building TimeTracker from scratch, let's take a test drive of the finished application. This will give you a good feel for what you are about to embark on and also give you some direction in case you get stuck.

Follow these steps to build and run TimeTracker:

1. Open a Command Prompt in the directory `C:\timetracker-completed`.
2. Execute the command `mvn install` to build the application. This step downloads all the libraries needed to build the TimeTracker application from remote Maven repositories. These libraries have been specified in the various `pom.xml` files under the TimeTracker source tree. Due to the sheer number and size of these libraries, this step will take a significant amount of time. However, be assured that your next build will be much faster because all the necessary dependencies will be available locally. Note that sometimes due to bad Internet connectivity or server load, Maven may not be able to download all dependencies in one go. If your build fails due to this reason, try to issue the command again until you get a successful build.
3. Now let's populate the timetracker schema with the tables required by the application. To do this run the following command in the Command Prompt opened earlier:
`mvn -f core/pom.xml andromdapp:schema -Dtasks=create`

4. Now that the tables have been created we need to populate the tables with some sample data. Follow the steps below to do this:
 - Open MySQL Query Browser. Login as timetracker.
 - Select File > OpenScript and open the following script file: C:\timetracker-completed\app\src\main\sql\static-data-insert.sql. Click the Execute button to on the top right to execute the script and populate static data in the database (users and tasks).
 - Now open the script file test-data-insert.sql in the same directory and execute it. This will insert test data in the database (timecards and time allocations).
 - Close MySQL Query Browser.
5. Let's start the JBoss server so we can deploy the TimeTracker application to it. To do this, open a second Command Prompt in the JBoss bin directory (C:\jboss-4.0.5\bin) and execute the command run. JBoss will take some time to start up. Wait for a message similar to this to make sure JBoss has started successfully:


```
19:50:01,285 INFO [Server] JBoss (MX MicroKernel) [4.0.5.GA (build: CVSTag=JBoss_4_0_4_GA date=200605151000)] Started in 39s:517ms.
```
6. Now we are ready to deploy the finished application to the JBoss server. Go to the previous Command Prompt (in the directory C:\timetracker-completed) and execute the command:


```
mvn -f app/pom.xml -Ddeploy
```

 You will notice that the TimeTracker EAR (Executable Archive) file is copied over to the JBoss deploy directory (C:\jboss-4.0.5\server\default\deploy) and the JBoss console displays several messages indicating that it is deploying the application. Wait for a message similar to this to make sure TimeTracker has started successfully:


```
19:59:53,767 INFO [EARDeployer] Started J2EE application: file:/D:/jboss-4.0.5/server/default/deploy/timetracker-1.0-SNAPSHOT.ear
```
7. Now we are ready to test the application. Open a browser and make it point to <http://localhost:8080/timetracker>. (If you have changed the JBoss HTTP port, then replace 8080 with the appropriate port number.) The login page should appear.
8. Enter nbhatia as the username and cooldude as the password. Click the *Login* button. The TimeTracker home page should appear with links to various sections.
9. Click on *Search Timecards*. The search screen should appear. Try changing the search criteria and click the *Search* button. The *Search Results* panel should show appropriate results. Note that the remaining TimeTracker screens are under construction.
10. When you are satisfied with the TimeTracker tour, you may stop the JBoss Server by typing Control-C in the JBoss Command Prompt.
11. We can now drop all the tables in the TimeTracker database in preparation of your turn to build the application from scratch. Go to the Command Prompt and excute the command:


```
mvn -f core/pom.xml andromdapp:schema -Dtasks=drop
```

Well, now that you have seen the TimeTracker application in action, are you ready to recreate it from scratch? I am hearing a resounding yes!