Roma Meta Framework v. 2.0

Ideas, thoughts and trends

After 4 years of development & use,

1 year of Romulus

and a lot of suggestions by the users

Luca Garulli
CTO @AssetData.it
Romulus partner

Facts...

- **Roma Meta Framework** is really a full stack framework and aims to cover all the application stacks and sectors
- The winning card is taking the advantage of many other tools and frameworks by integrating all together using **Aspect** concepts
- It's used by many **developers** in all the world (and growing)
- The "**POJO** for all" remove the need to learn other languages and technologies
- Users report a huge gain of **time-to-market** for their applications: 10x in comparison to classic J2EE stack and 3x in comparison to the RoR-like technologies
- It's not a one-man project any more, but there is an active **community** of committers (12 nowadays) and many users

...But we learned that: (I)

- The **learning curve**:
  - can be high for experienced developers (since they tend to make things always complicated as learned by old technology)
  - and low for beginners: really strange don't you?
- The "**POJO** for all" is not always the best solution, specially for the presentation where scripting and templating are more productive. Of course you'll loose portability across frameworks but sometimes it's preferable
- The framework is growth in the 4 years of its life with a lot of **refactoring**, sometimes with patches in order to get compatibility with the past

...But we learned that: (II)

- POJO is ok, but too often I need to create wrapper classes to express complex concepts.
- Documentation is good to start but we would need a lot of **examples, advanced tutorials** and **videos**
- The **concurrency** is very strong and the RAD frameworks born every day giving new cool ideas. A real good community is open to the changes and ready to improve the product
- The **IDE Eclipse plugin** not helps very much since it has few features and most of them are in the modules management that is used at first and very rarely in the course of development

...Course of Action (I)

So we need to empower the Roma meta framework with:

- Consolidating and improving the winner features
- Implement the new cool ideas by other competitors
- Supporting more powerful expressiveness to reduce the number of POJOs created
- And... Break with the compatibility with 1.x!
  - We could provide a migration tool to help the porting of old applications
  - Last 1.x version is the release 1.2 branched under svn in the directory **/branches/1.2.0**

Winner features: POJO for all

- The full POJO architecture allows to use Roma with the knowledge of the Java language only plus the Roma platform, of course. Most of other competitors use other languages to being more productive (Groovy, Ruby, Python, etc.)
- The Java-only feature (no scripting) is loved by the big companies because:
  - Supports very well the **refactoring**
  - It's very easy to **debug**
  - A lot of good **IDEs** stable and for free (Eclipse, NetBeans)
  - Allow to use existent **investments** in software such as Application Servers (exceptions for RoR on top of JRuby)
  - **Save the skill** of Java developers
  - Test the **GUI** with JUnit

Component Aspect

- Some users asked us to support other **IoC** engine (such as JBoss) or nothing at all just instancing all the components by hand. Is Roma feasible without an IoC engine at all?
- Proposal (*to study yet*):
  - Create a generic **Component Aspect** interface to work with components
  - Create the Spring implementation **SpringComponentAspect** and package it as a new module called **roma-component-spring**. The user can choose to use Spring or not just as other modules
  - ObjectContext.getInstance().getComponent() will demand to the new Component Aspect
- Where can help **OSGi**?

Refactoring of roma-core

- Now the **roma-core** module contains a lot of generic classes used in the view:
  - **Error Message** popup classes
  - The entire **CRUD** package
  - etc.
- We need to create the **roma-view** module with all these classes in order to get the roma-core.jar smaller than now
- All the View Aspect implementation will depends by the **roma-view** module

Removing of unused modules

- We have to clean the central repository removing the not more supported modules:
  - **persistence-jpox**, since **persistence-datanucleus** replace it at all
  - **workflow-pojo**, since it's pretty old and was been replaced by Tevere workflow
- Some modules will be merged:
  - **web** module will be part of **project-web** since it contains parts needed by all web applications
  - **persistence-jdo** makes not sense anymore since libraries are coupled with the persistence-* dependent. This module will be merged in **persistence-datanucleus**

Modules (plug-ins)

- New target called "**info**" to display information about the module
- Each module version can contain a **text message** to display to the user, useful to give information to the user such as:
  - *"WARNING: this version is latest and the module will be not more supported. Use XXX instead."* or
  - *"WARNING: the module was moved to the new repository http://yyy.com/roma-modules"*
- Explore in deep if and where **OSGi** can help

Fast access to the aspect impls

- New **Roma** class to being used by user's POJOs.
- Each method knows how to retrieve the right Aspect implementation
- Persistence Aspect:
  - `Roma.persistence().createObject( pojo );`
  - `Roma.context().persistence().createObject( pojo );`
  - `Roma.context().component("PersistenceAspect").createObject( pojo );`
  - `Roma.component("PersistenceAspect").createObject( pojo );`
- Methods can cache the reference to the Aspect implementation as for the ViewAspect
  - `Roma.view().show( pojo ); // view aspect cached`
- Other examples:
  - `Roma.flow().forward( new Employee() );`

Multiple Aspect impl. (I)

- We need to support multiple aspect implementations at the same time
- Think to export a POJO as Web Service and another one via HTTP REST or rendering some POJO using Echo2 and others using Janiculum
- We need to reach a component by using:
  - The classic mode to reach a generic component by “id”:
    - ObjectContext.getInstance().**getComponent**(<id>)
  - And the new API to get the *default* implementation for a given Aspect in this way:
    - ObjectContext.getInstance().**getAspect**(<aspect>)
    - This API will search inside the AspectManager component (see before)

Multiple Aspect impl. (II)

- Aspects will be not any more **auto-registrant** but they have to be enlisted inside the property “default” of the **AspectManager** bean.
- **Wizards** have to update this entry to became the default one
- Example:

```
<bean id="AspectManager">
  <property name="default">
    <map>
      <entry key="view"><ref="Echo2ViewAspect"></entry>
      <entry key="session"><ref="Echo2SessionAspect"></entry>
      <entry key="i18n"><ref="I18NResourceBundleImpl"></entry>
    </map>
  </property>
</bean>
```

Virtual POJOs (I)

- Since the “POJO for all” is not always the best option (specially for presentation tier) the best solution to be coherent is leaving intact the current architecture but introducing the **Virtual POJO** concept
- A Virtual POJO for Roma engine is like a real POJO at all, but it has no a Java implementation
- The idea is to allow to define a POJO just using the well known **XML Annotation** but enriching it of new features:
  - Use the **core** aspect in the **<field>** tag to express the strictly language properties such as type
  - The **<action>** tag can contain the implementation using the Roma **scripting** aspect and selecting the language
  - The scripting can access to the “**this**“ object as **“me”** and other context variables

Virtual POJOs (II)

Example of Message.xml:

```
<class>
  <field name="from">
    <aspects>
      <core type="string"/>
    </aspects>
  <field name="to">
  <field name="subject">
  <field name="content">

  <action name="send">
    <aspects>
      <scripting language="java"><!CDATA[[
        MessageHelper.sendMessage( me.from,
                                   me.to,
                                   me.subject,
                                   me.content);]]>

      </scripting>
    </aspects>
  </action>
</class>
```

Use the **"core"** aspect to express the strictly language properties

Use the **"scripting"** aspect to write the action implementation

Virtual POJOs (III)

In this case we create a virtual POJO to handle a form with 2 tabs. No Java class will be created.

Example of HomePage.xml:

```
<class>
  <field name="tabs">
    <aspects>
      <core type="list"/>
      <view render="tabbed"/>
    </aspects>
    <content>
      <object class="Cockpit"/>
      <object id="BackOfficeBean"/>
    </content>
  </field>
</class>
```

Virtual POJOs (IV)

Extends a POJO rendering without creating a Java class.

Example of CustomerEditable.xml:

```
<class>
  <aspects>
    <core extends="Customer"/>
  </aspects>
  <field name="notes">
    <aspects>
      <view enabled="true"/>
    </aspects>
  </field>
</class>
```

View Aspect: order of fields

- Roma 1.x (today):
  - When layout mode = **EXPAND** all the fields are rendered in the parent object.
- Roma 2.0 (tomorrow):
  - As today but the fields are seen as owned by parent instances and can be ordered among the others

```
<class>
  <fields>
    <field name="entity.id" />
    <field name="entity.surname" />
    <field name="entity.name" />
    <field name="entity.city" />
    <field name="entity">
      <aspects>
        <view layout="expand" />
      </aspects>
    </field>
  </fields>
</class>
```

This avoid the generation of wrapper getter and setter in the container class

View Aspect: XML for collections

Use the embedded XML annotation also for collections. This avoid the use of wrapper classes created only to hide/show fields.

```
<class>
  <fields>
    <field name="items">
      <class>
        <field name="date">
          <aspects>
            <view visible="false">
          </aspects>
        <field>
        <field name="ordered">
          <aspects>
            <view render="select"/>
          </aspects>
        <field>
      </class>
    </field>
  </fields>
</class>
```

ServiceAspect: REST support

- We need to add a REST support to ROMA using a Servlet as gateway
- The proposal is:
  - Create something similar to the framework “Play” but supporting the Screen concepts
  - Map URL to **static methods**
    - `/app/<service-name>/<method>`
  - Parameters can be passed by position or by key:
    - `(<parameter-key>=<parameter-value>)|<parameter-value>]*`

Example:

```
@ServiceClass(serviceName=”controller”)
public class MyController{
  // mapped as: /app/controller/index
  public static String index();
  // mapped as: /app/controller/show?value=3&position=bottom or /app/controller/show/3/bottom
  public static String show(int value, String position);
}
```

JDO enhancement

- Some vertical modules such as **ADMIN** and **USERS** are composed of persistence classes
- Enhancing is not a problem but is preferable to enhance in the final project because:
  - Modules can be compiled using other ORM or previous version of the same ORM
  - Developer could need to tune and change the persistence mapping in the final project without modifying the modules
- The proposal is:
  - Copy the **package.jdo** files in the final project
  - Find a solution to enhance classes in the jar

Comments & Questions?