Towards a UML profile for model-driven object-relational mapping

Alexandre Torres, Renata Galante and Marcelo S. Pimenta
Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
{atorres,galante,mpimenta}@inf.ufrgs.br

Abstract—By using UML (Unified Modeling Language) and ER (Entity Relationship) notations to describe a system, the specification of persistent and transient details became separated. UML and ER models use distinct notations, that cannot be checked for consistence, and therefore are not suitable to be used in a model driven development approach. This paper proposes a synergistic approach to persistence modeling by extending UML with a profile that represents object-relational mapping concepts, and provides rules to check the coherence between persistence and object oriented concepts. Model driven transformations are proposed, and implemented by a tool, in order to turn models into object relational mapped systems based upon the JPA (Java Persistence API) standard.

Keywords—Persistence, UML, MDD, JPA, Database, Java.

I. INTRODUCTION

The UML is the “de facto” standard software modeling language. Despite its success on representing structural and dynamic features of object oriented systems [1], the UML does not provide resources to model database persistence [2]. When relational database persistence is the concern, the ER model [3] and derivatives are still employed as the main artifacts. Hence, both UML and ER models are used in a complementary way.

The Model Driven Development (MDD) proposes that models take on the main role on the system development process, replacing text based languages at parts or the whole process of software development [4]. In order to use the MDD approach, the information represented by models should be coherent, integrated, and computable, so that automatic transformations could turn models into executable system [5]. Using separated UML and ER models turned out to be a problem for transformation construction: there is neither integration between elements nor a common metamodel.

Furthermore, relational databases and object-oriented programming languages are based upon distinct paradigms, with technical, conceptual, and cultural incompatibilities. The set of those incompatibilities is commonly referred as the object-relational impedance mismatch problem [2].

Object-Relational Mapping (ORM) frameworks address the impedance problem on software implementation level [2], providing the developer with ways to declare how each technical incompatibility should be treated. ORM tools brings to the same level relational resources, such as data querying and object oriented resources, such as inheritance and polymorphism, enabling to explore the synergy between those constructs.

The Java Persistence API (JPA) [6] is the result of a community process involving major ORM tool providers, in order to standardize ORM frameworks for the Java platform, and seems to be the most important (if not the only) cross-vendor standard to ORM frameworks. The mappings of JPA can be used not only to tell the framework how to translate objects to database tuples, but also to generate the database structure of the system [6].

This paper proposes the MD-JPA (Model Driven Java Persistence API) profile, a UML extension that enables the representation of the object-relational mapping resources of the JPA standard. An open source tool was developed to provide model evaluation and its transformation to a JPA implementation within the MDD approach. The model verification is performed by a set of constraints that detects designs incompatible with ORM, reducing the round-trip effect of adapting models to development needs.

The MD-JPA profile is our first step in the SOUL (Synergistic Object-relational UmL) modeling approach, that also aims in the development of more general ORM modeling solutions. The main goal of SOUL is to allow the modeler to employ the synergy between relational and object oriented paradigms by the usage of a unified notation based on UML.

The main contribution of this paper is a UML profile that implements aforementioned extensions and can be applied to models created by standard UML design tools. The cognitive dimensional framework [7] is employed in order to evaluate the use of MD-JPA profile and reason about the SOUL approach adequateness for future works.

The remaining sections of the paper are organized as follows. Section 2 presents basic and related work on persistence modeling, ORM, and transformations. Section 3 specifies the MD-JPA profile modeling elements and section 4 discuss the constraints used to validate models. Section 5 presents a cognitive dimensional analysis of MD-JPA. Section 6 shows how the profile is used in the MDD approach by presenting transformations that take models built with the proposed profile, producing Java classes with
ORM annotations. The last section brings the final remarks and considerations about concerning works.

II. RELATED WORK

The agile database modeling [8] is a well known proposal for database modeling using UML extensions. It is mainly based upon the class diagrams for representing data models with a set of model, class, and property stereotypes, allowing the creation of models in 3 abstraction levels: conceptual, logical, and physical.

The conceptual and logical models focus on the representation of the ER concepts such as entities, relationships, and attributes, by annotation of classes, properties, and associations. The physical model represents database concepts such as tables, columns, views, and foreign keys. The result is an extension that successfully allows the data modeling with UML but does not tackle the integration of object oriented and relational concepts.

The Object Management Group (OMG) has an underway proposal for data modeling representation. Again the focus seems to be the representation of ER concepts with UML, aided by database reverse engineering, providing a wider range of concepts to allow XML representation [9].

Both proposals enable the use of UML for data modeling, therefore allowing the construction of transformations over a common meta model. However, to successfully tackle the object-relational impedance mismatch problem, the data and object modeling should be more integrated and coherent [2], hence facilitating the visualization of the relations between tables and classes that access those tables.

The ORM tackles the impedance mismatch problem at implementation level, allowing the developer to use the relational and object oriented techniques together, abstracting the database structure. The proposals of Ambler and OMG do not map classes that represent information in the system to classes that represent tables and explicitly represents the database structure.

UML extensions related to persistence, approaching relational database operations [10] and multidimensional modeling in data warehouses [11], are focused in the modeling details of data representation. The object oriented implementation of such data representation with ORM is not part of those works.

Similar works were proposed for modeling object oriented databases with UML, as discussed in [12]. Although those studies focused on transformations, and therefore are suited to be used with the MDD approach, their main concern was with the representation of the object oriented database structure with UML, instead of the integration between object oriented languages constructs and relational database structures. The present study is more concerned with the object oriented viewpoint of persistence.

III. THE MD-JPA PROFILE

This paper complements our previous works [13] and [14] by proposing the MD-JPA profile, an extension of UML to represent the most important concepts of ORM, using the JPA standard as the source for those concepts. The MD-JPA profile is mainly comprised of stereotypes, each of them extending a meta-class of UML, such as classes, properties or use cases. With those stereotypes, all annotation mappings of JPA can be represented with UML based models.

Table 1 summarizes the annotations of JPA specification and their equivalent stereotypes on MD-JPA. The third column describes which elements of UML are extended by the proposed stereotypes. Annotations such as Table, AttributeOverride and UniqueConstraint are represented as properties of those stereotypes.

TABLE I. ANNOTATIONS OF JPA AND EQUIVALENT MD-JPA STEREOTYPES.

<table>
<thead>
<tr>
<th>JPA</th>
<th>Stereotypes</th>
<th>UML Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Entity</td>
<td>Entity</td>
<td>Class</td>
</tr>
<tr>
<td>@Inheritance (Strategy)</td>
<td>SingleTable, Joined,</td>
<td>Generalization</td>
</tr>
<tr>
<td></td>
<td>TablePerClass</td>
<td></td>
</tr>
<tr>
<td>@ (One/Many) To (One/</td>
<td>AssociationMapping</td>
<td>Property</td>
</tr>
<tr>
<td>Many)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@Embeddable</td>
<td>Embeddable</td>
<td>Class</td>
</tr>
<tr>
<td>@Embedded</td>
<td>Embedded</td>
<td>Property</td>
</tr>
<tr>
<td>@Transient</td>
<td>Transient</td>
<td>Property</td>
</tr>
<tr>
<td>@Id</td>
<td>Id</td>
<td>Property</td>
</tr>
<tr>
<td>@IdClass</td>
<td>IdClass</td>
<td>Class</td>
</tr>
<tr>
<td>@EmbeddedId</td>
<td>EmbeddedId</td>
<td>Property</td>
</tr>
<tr>
<td>@Column</td>
<td>Column</td>
<td>Property</td>
</tr>
<tr>
<td>@Version</td>
<td>Version</td>
<td>Property</td>
</tr>
<tr>
<td>@Enumerated</td>
<td>Enumerated</td>
<td>Class</td>
</tr>
<tr>
<td>@MappedSuperclass</td>
<td>MappedSuperclass</td>
<td>Class</td>
</tr>
<tr>
<td>@GeneratedValue</td>
<td>GeneratedValue</td>
<td>Property</td>
</tr>
<tr>
<td>@Lob</td>
<td>Lob</td>
<td>Property</td>
</tr>
<tr>
<td>@Temporal</td>
<td>Date, Time, TimeStamp</td>
<td>Class, Property</td>
</tr>
<tr>
<td>@AttributeOverride(s)</td>
<td>AttributeOverrides</td>
<td>Class, Property</td>
</tr>
<tr>
<td>@OrderBy</td>
<td>OrderBy</td>
<td>Property</td>
</tr>
<tr>
<td>@DiscriminatorColumn</td>
<td>DiscriminatorColumn</td>
<td>Class</td>
</tr>
<tr>
<td>@SequenceGenerator</td>
<td>SequenceGenerator</td>
<td>Property/Class</td>
</tr>
<tr>
<td>@TableGenerator</td>
<td>TableGenerator</td>
<td>Property/Class</td>
</tr>
<tr>
<td>@AssociationOverride</td>
<td>AssociationOverrides</td>
<td>Generalization</td>
</tr>
</tbody>
</table>

Fig. 1 shows the main stereotype diagram of MD-JPA, containing the stereotypes that extends the Class and Property elements of UML. A stereotype may have properties and relationships with other UML elements (the dark arrow relationships) or classes that represents structured information and enumerations. A stereotype may specialize another stereotype, as represented by the empty arrow inheritance relationship.

The following subsections specify the most important mapping resources of our JPA proposal. The stereotypes of MD-JPA are exemplified by diagrams of an evaluation of scientific production system.
A. Entities and properties

The entity concept of JPA is represented by the Entity stereotype, suitable to UML classes. An entity denotes a class in which each instance is persistent and has a unique identifier. An entity may have a set of persistent properties using types according to the ORM mapping limitations.

The entity class is persisted, by default, in a table with the same name of the class. The table characteristics, like name, catalog, schema, and unique constraints to name a few, may be detailed in the table property. It is also possible to define any number of secondary tables definitions used to persist the entity data.

Each entity hierarchy must implement the same access method for persistent properties, namely field or method access [6]. Therefore, the access method on MD-JPA is defined in the entity stereotype (access property), and not by the position of annotations on properties or methods.

By default each property of a Persistent class is persistent, except for those marked with the Transient stereotype. On JPA it is expected that each property presents getter and setter methods. In order to simplify the modeling, each UML property is considered as a JavaBean property [15] with private instance variable by default. No getter or setter method specification is required, and the place of annotation is inferred by the access method of the entity.

The Column stereotype enables a more precise description of the property mapping to a database column. It allows the definition of the name, precision, scale, acceptance of null values, and table, when the entity is persisted on more than one table.

Other stereotypes can be applied on persistent properties, as shown in Fig. 1. The specializations of Temporal stereotype help mapping the Java Date class to a more specific date, time or time-stamp database type. The Lob stereotype marks a property as a large text or binary object. The Basic stereotype allows definition of a database fetch strategy and if the property allows null values in program level. The Embedded and AssociationMapping stereotypes will be explained in the further subsections.

B. Embeddable classes

Embeddable classes can represent part of the persistent state of entities. They do not have persistent identity, and each instance cannot belong to more than one entity instance at the same time [6]. Embeddable classes may be used to encapsulate common logic of a fine-grained component, reused both in transient and persistent classes. Embedded
values are persisted with multiple columns, one for each property of the embeddable class.

The **Embeddable** stereotype indicates a class that can be embedded as properties of entities. A property that references an embeddable class should use the **Embeddable** stereotype.

<table>
<thead>
<tr>
<th>Interval</th>
<th>embeddable DateInterval</th>
<th>CourseEvaluation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>start: Date [1]</td>
<td><code>@embeddable</code> Date start: Date[1]</td>
<td><code>@embeddable</code> idCourseEval</td>
<td><code>@embedded</code> interval: DateInterval[0..1] score: Integer [1]</td>
</tr>
<tr>
<td>end: Date [0..1]</td>
<td><code>@date</code> end: Date[0..1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>after(Interval): Boolean</td>
<td>before(Interval): Boolean</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 2.** Entities and embeddable classes example.

Fig. 2 is a diagram exhibiting a number of classes from an academic evaluation system, including entities and embeddable classes. The **DateInterval** class extends the transient **Interval** class by implementing an interval of time with day granularity. Each **Evaluation** instance stores a score for a predefined date interval, which is in turn an embedded object. The **CourseEvaluation** class is an entity that implements the abstract **Evaluation** class and inherits the interval property, which is mapped to two date columns in the course evaluation table: **start** and **end**.

C. **Primary Keys**

Every entity hierarchy must have a primary key, defined in the topmost entity of the hierarchy tree [6]. Simple primary keys are just represented as annotated attributes. Composite keys must have a separate embedded class, where each property is a column of the key. In such case, it is possible to specify one property that makes references to the embedded class or a set of properties that makes references to each of the embedded classes’ properties.

There are four important stereotypes used on the representation of primary keys: (i) **Id** - Used to identify a single key, or each property of a composite key, when combined with IdClass; (ii) **IdClass** - Identifies what class will be used to represent the primary composite key mapped by the Id stereotype; (iii) **EmbeddedId** - Defines a composite primary key as an embedded property; (iv) **GeneratedValue** - Defines a property as having automatic generated values. It is possible to define the generation strategy or to leave the decision to the ORM framework. It is also possible to describe a **TableGenerator** or a **SequenceGenerator**, both derived from the abstract **Generator** stereotype. On Fig. 2 the entity **Course** is identified by the **idCourse** property, while the **Graduation** and **Doctorate** specializations cannot define its keys.

D. **Relationships**

All relationships between entities are, by default, persistent. The mapped by property, mapping type, and direction are respectively expressed on the model by the association relation, cardinality and navigability, thus making unnecessary specific stereotypes to distinguish the one/many to one/many mappings. For the sake of simplicity, associations are mapped to Java collections, and Generics [16] is used to assure type checking. Aggregation and composition follow the same rules as associations, but the specialization relationship is shown in the inheritance section.

The **AssociationMapping** stereotype provides further definitions that allows to specify what kind of collection will be used (sets or lists), ordering expression, cascade type, fetch strategy, and join tables. It is applicable on the **Property** element instead of the association itself, because each UML class association has one property member in each side that can be naturally mapped as connecting properties on the implementation classes. The join columns property of **AssociationMapping** can contain one or more mappings detailing how the foreign key should be generated. If the foreign key is part of the primary key, the **pkJoinCols** should be used instead.

**Fig. 3.** Table related classes in MD-JPA.
Another resource of MD-JPA is the join table, defined as another property of AssociationMapping stereotype. It can define the structure of the table used to many to many or one to many unidirectional associations. Join tables, secondary tables, and the table definition are all meta-classes sharing the AbstractTable super meta-class. The OrderBy stereotype allows the definition of an order clause for the fetch of an ordered collection of objects. It can be only used in a many side of the relationship. Those elements are shown in Fig. 3.

Fig. 4 contains an example of relationships detailed by the AssociationMapping and OrderBy stereotypes. A Paper relates to Researcher in a many-to-many relationship, and relates to PaperVersion in a one-to-many relationship. Each PaperVersion can be commented by other researcher, and those comments are stored in Comment instances. Each comment can be a response to another comment, as expressed in the auto-relationship properties replyTo and responses. The responses property should be ordered by the cdate column, and be stored in a ordered List. The collection type is specified in the AssociationMapping stereotype, and the order by expression is specified by the OrderBy stereotype.

E. Inheritance

An entity can generalize or specialize another class [6]. When the super class is another entity, there are three possible strategies to object-relational mapping on JPA. For each, there is a stereotype that extends the Generalization element of UML, as shown in Fig. 5.

The SingleTable stereotype represents a mapping of the entire hierarchy of classes to one table only (also called “universal table”), with as many columns as the sum of all properties and associations of the classes, plus the discriminator. Fig. 2 exemplifies the single table inheritance of Graduation and Doctorate courses.

The TablePerClass stereotype represents a mapping where each concrete class has a separate table. In a similar way, the Joined stereotype also employs one table for each class, but the properties from the super class are stored on the super class table. The Joined stereotype can also be detailed with foreign key to primary key mappings.

The discriminator column is always used on SingleTable, but optional on Joined strategy, hence the abstract stereotype DiscriminableGeneralization can hold the discriminator value. The DiscriminatorColumn stereotype can be used to detail the column. If no discriminator stereotype is used, all mapping, including the discriminator naming, is done automatically by the framework.

F. Mapping transient classes

The MappedSuperclass stereotype allows the definition of ORM on transient classes, and this mapping can be used by its specializations. Otherwise, the ORM framework will ignore inherited properties of non entity super classes.
Fig. 6 depicts the main relationships between entities of our scientific evaluation system. The Categorized class represents the common characteristic of papers and scientific event editions of being related to knowledge areas of science. However, in the database generated for this model, paper and events are unrelated entities, the Categorized concept does not exist, and there are two tables to implement the many to many relation “Categorized to KnowledgeCategory”, one to the paper and another to the scientific event.

The evaluation is another example of transient super class with mappings. It defines the common elements of a evaluation (our date interval of Fig. 2 and a score) but let the specialization relate this evaluation to the correct entity, and implement its specific logic. The scientific event also implements an interface called WithEvaluations that has generic methods to access evaluations.

The AttributeOverrides stereotype can be applied to both classes and properties. It allows the redefinition of one or more column definitions done in a super or embedded class. Each AttributeOverride object makes a reference to a property name and to a new ColumnDefinition object. If a relationship needs to be overridden, the AssociationOverrides can be applied to the generalization element containing one or more properties that participates in relationships of one of the super classes.

In the Fig. 2 DateInterval is an embeddable class that extends the common class Interval to represent dates. The start and end properties will now be overridden to be stored in the startDate and endDate columns. Fig. 7 compares two ways to override the property: in the left, by applying the stereotype to the class itself without overriding the inherited properties; or in the right, where both properties are stereotyped with AttributeOverrides.

G. Parameterizable types and relationships

Parameterizable types in Java are used to define classes that have a parameter type that should be bound in instantiation time. For instance, the Set<E> class in the collection framework can be instantiated to Set<Course> in order to store a set or Course objects. The UML notation has similar mechanism to represent classes that take parameters, called templates.

Fig. 8 shows a diagram with the interface WithEvaluations and its template signature declaring the type parameter E as being a class that is an Evaluation. On Figures 2 and 6, Evaluation is an abstract mapped superclass implemented by EventEvaluation and CourseEvaluation. The property evaluations of WithEvaluations uses the template class Set binded to the parameter E.

Entities ScientificEvent and Course implement WithEvaluations by binding the variable E with one evaluation type. Both ScientificEvent and Course are related to EventEvaluation and CourseEvaluation classes on Figures 2 and 6, by a relationship property also called evaluations. This property overrides the evaluations property on the WithEvaluations interface.

The use of parameterizable types enable powerful polymorphic constructs, such as in the relationship between evaluable and evaluating instances. For instance, one can create a collection of WithEvaluation objects and access its evaluations without bothering about what kind of evaluation is used, and if its stored on table CourseEvaluation or EventEvaluation.

H. Remarks about the MD-JPA profile

The aim of the agile database modeling proposal [2] and [8] was to enable relational database modeling with UML, whereas the MD-JPA is focused in modeling classes with object relational mapping. Unlike the agile database modeling approach, the MD-JPA profile does not distinguish persistent diagrams, and does not segregate persistent from transient elements in different diagrams. MD-JPA stimulates the drawing of entities and transient classes together, encouraging the user to explore the synergy of its relations, such as mapped superclass inheritance, interface realization and embedded class utilization.

One of the key features of JPA is that it allows the writing of code with few details about software persistence, or it allows a very detailed mapping that can be later used to create the database objects. The MD-JPA preserves this same characteristic by offering a complete set of mappings, from which the user can choose to pick just the most relevant.

Moreover, the approach of MD-JPA takes advantage of the UML ability to display an element in more than one diagram. It allows the creation of diagrams with a higher level of abstraction, depicting elements without persistence information, and diagrams detailing the persistence mapping of those elements. All information displayed on diagrams is stored in a single model, verified for well-formedness by a set of rules implemented as constraints.

The present section described the stereotypes, attributes, and classes that constitute the MD-JPA profile. By applying those elements its possible to enrich class models with
stereotype, which generalizes variable pointing to the class element and its opposite Embedded [16], what in UML is expressed by the checking if the base class of the entity is not a nested class enforces that “the entity class must be a top-level class”, by private, as evaluated by the second section of the operator in the rule.

Furthermore, this set of invariants can be expanded by the user, with the aid of the OCL interpreter of eclipse [18].

The JPA specification states several requirements for entities, fields, keys, inheritance, and relationships; this specification was the base to create most of our OCL invariants. For instance, the invariant rule21_p3 (Fig. 9) enforces that “the entity class must be a top-level class”, by checking if the base class of the entity is not a nested class [16], what in UML is expressed by the owner relationship.

inv rule21_p3 :
(base_class.owner.oclIsKindOf.uml::Package) or (base_class.owner.oclIsKindOf.uml::Model) ) and not ( base_class.visibility=uml::VisibilityKind:protected or base_class.visibility=uml::VisibilityKind:private )

Figure 9. Example of constraint for model verification

The context of rule21_3 is the abstract Persistent stereotype, which generalizes Entity, MappedSuperClass and Embedded stereotypes. It provides a common base_class variable pointing to the class element and its opposite variable extension_Persistent was redefined by each sub stereotype.

A class whose owner is another class or anything other than a package of model cannot be considered as top level class. Besides this, a top level class cannot be protected or private, as evaluated by the second section of the and operator in the rule.

The profile has a set of another 30 invariant checks for model well-formedness, not detailed here due to space limitations. About half the rules were direct references to the specification of JPA, and the remaining rules are additional checks to enforce the good use of the profile and UML modeling for JPA. Some examples of additional verifications are: only one class can generalize any other class in Java; the AssociationMapping stereotype can only be applied to the association ends; if a property is overridden, it must exist in one of the superclasses.

V. COGNITIVE DIMENSIONAL ANALYSIS

The MD-JPA profile was analyzed in the context of the cognitive dimensions [7], with the purpose of evaluating its adequateness as a modeling artifact. In the cognitive dimensions analysis, the software modeling task is rated as an exploratory design activity. The four most important dimensions at this activity are activity are viscosity, hidden dependencies, premature commit and abstractions; although the visibility and role-expressiveness dimensions are also pointed out as important for exploratory design activities.

In order to explore the six cognitive dimensions of MD-JPA, it is important to clarify the distinction between UML models and diagrams. A UML model comprises all elements that specify a software unit, and is concerned about the standard representation of the structure of UML. A UML diagram is an element, and therefore part of a UML model, that represents a view over this model, displaying some elements of this model.

A. Viscosity

The Viscosity dimension evaluates how many user actions are needed to accomplish one goal. The UML class model supports editing most of its elements independently, and operations like move and copy are a common resource of UML tools. However, the relationship between the class diagram and the model presents some viscosity problems.

Each class diagram represents a view over a set of elements in the model. Usually, the user will make any change in a diagram, that is stored in the model. The viscosity problem may arise when the same element is displayed in more than one diagram, and a change may impact in the visual disposition of the graphical representation of classes, relationships and its features.

For example, supposing a model with two diagrams that depicts the class Comment, like the Figures 4 and 6. If Comment is renamed to ResearcherFeedback on diagram of Fig. 4, the model will change, and the diagram of Fig. 6 will also show the new class name. But the box used to represent the class will be too small for the new name, thus requiring to be resized. If the UML tool automatically resizes the box, it may overlap another class or relationship, thus needing to be checked by the user.

The viscosity problem in such cases are connected to how each UML tool solves the problem of diagram maintainability. If the tool is able to automatically re-dispose the elements in a acceptable way, after a change is made, and if it records what diagrams need a review by the user, it will surely reduce the viscosity problem.
B. Hidden dependencies

Hidden dependencies happen when changing a property of an element causes unexpected changes in other unrelated elements, or elements related in a one way fashion. On class diagrams, the most typical hidden dependency problem occur when a class which is referenced as type of another property is removed.

The MD-JPA profile introduces hidden dependencies on embeddable classes and inheritance relationships, when they override its mappings. When a property of an embeddable or mapped super class is removed, and this property is overridden by another entity, the override will became inconsistent, and the user will need to remove it.

Hidden dependencies are a problem when there is no information about what happens after a change is made. The constraints for model verification are able to detect inconsistencies, and inform the user of what elements must be corrected. OCL queries may be used to show what elements are related, before executing a change in the model.

C. Premature commitment

Premature commitment happens when there is a fixed order to perform some task, and the user may have to make a premature decision. The MD-JPA profile, as the UML itself, does not impose a rigorous order to the modeling task.

The user may start its model by defining some classes, properties, operations and relationships. After that, he may apply the Entity, Embeddable, and Mappedsuperclass stereotypes. In the end, he may add the identifiers and a more detailed mapping by using stereotypes like Column and AssociationMapping.

In a different approach, the user may create each class, detailing how each property is mapped for persistence and which ones compose the identifier. If an Embeddable class is needed, he can define it, and use it as a property type. After each class is created, he may add the relationships and the inheritance detailing.

D. Abstractions

Adding new abstractions, such as MD-JPA profile is in relation to the UML notation, can increase the learning curve to new users. The MD-JPA profile does not add new abstraction mechanisms to the UML, but it can be used with the built in UML abstractions, such as other profiles and the ability to import elements from other models.

E. Visibility

The visibility measures how much information can be displayed at the same time, evaluating the expression power and juxtaposition ability of the notation. UML tools are very flexible on diagram display, because they allow an element to be displayed on various class diagrams, and allow diagrams to be displayed side by side in order to compare distinct constructions.

However, some information about UML elements are not represented on UML diagrams that comprise some advanced properties, and profile properties. To display such information, the user may use secondary notation like comments, or use a secondary editor to inspect those properties.

The stereotypes of MD-JPA profile, listed on Table 1, can be represented in class diagrams, but its properties are not shown by standard UML tools. In order to compare properties not shown in diagrams, the user may use OCL queries that select the desired information.

F. Role-expressiveness

The role-expressiveness evaluates how easily a user can discover why some structure in the model was built in a specific way. In a low role-expressive language, it is difficult to distinguish one element from another.

The MD-JPA profile increases the role-expressiveness of class models by pointing out the role of each class in the ORM. By using meaningful names, based upon the widespread concepts of JPA, the profile also offers a notation with high closeness of mapping, what helps the understanding of diagrams by JPA users.

VI. MODEL DRIVEN APPLICATION OF MD-JPA

This section describes a tool implemented in order to validate the MD-JPA profile as a MDD artifact, showing the ways through which it is used as a development artifact. The tool is composed by two components: a set of transformations rules that takes MD-JPA models, generating models of Java annotated implementation; and a translator that generates human readable implementation from the Java models.

In order to check the transformation completeness, a model with the example implementation of JPA specification was created by reverse engineering, comparing the results obtained by the tool with the original code. Fig. 10 shows an overview of our transformation process.

The ATL language [19] was employed to build transformations between MD-JPA models and Java models. The ATL is a model to model transformation language that work with UML models and user defined meta models like
JAS, and allows both imperative and relational transformation rules. The Java models are represented following the Java Abstract Syntax (JAS) meta model [20]. The JAS provides a complete mapping of the Java language to XMI meta model, allowing the transformation of Java code into models, as shown in [21].

Classes, properties, methods, enumerations, and interfaces, all among UML base elements, are matched by the transformation rules. Each matched element produces one or more Java elements: a top level class will generate a class and a program unit; a property will generate a private instance variable and accessor methods.

Some elements cannot be produced solely by match rules and are addressed by imperative rules called inside some match. For instance, the modifiers generated from a single class, attribute or operation, depends on the visibility and the abstract attribute. In this case, there is not a clear source for the generated element, other than the class or feature, and it is generated only in certain circumstances – a private visibility or abstract being true. The imperative rules are an alternative to create one match rule for each combination of circumstances in the source element [19].

The translator component of the tool takes the resulting model as input, generating the implementation as indented human readable code. The input model, in this stage, follows the JAS meta model, which is based upon the eclipse JDT framework. The translator just needs to parse this model, and generates the code by using the JDT API, that is responsible for the text generating and formatting. The transformations rules, the translator, the set of OCL rules and the MD-JPA profile were bundled together in one open source tool, that works as an eclipse “plugin” [22]. The implementation generated by the plugin is transformed into the database structure by any ORM framework that follows the JPA standard.

In order to validate the transformations, we present a model created from the JPA example system, comparing the generated code with the implementation proposed at the JPA specification. Fig. 11 shows a detailed MD-JPA class diagram with each class, property, and applied stereotype containing the mapping information of the acme example.

The only differences between the two acme sources were the instance variable names, when the property name is distinct from the private variable, and the use of generics in all relationships. Both are due to decisions concerning the mapping of Java concepts of properties and collections in MD-JPA profile. However, the database generated by hand written and generated code was identical, both with OpenJPA [23] and Hibernate [24] JPA implementations, for all available database dialects on those tools.

VII. CONCLUSIONS

Despite the wide adoption of UML, there is not a clear, consensual and easy-to-use solution for the problem of persistence modeling. Separate ER and UML models are possibly the most common approach. However the MDD approach demands integrated models that can correctly represent the implementation of the system.

This paper has presented a UML profile which allows the modeling of the persistent elements of a system, following the JPA persistence standard. The focus of MD-JPA profile is the creation of models depicting jointly both persistent and object oriented elements. Those models can be verified, in order to detect a design incompatible with ORM, and then transformed in parts of the software implementation.

The cognitive dimensional analysis shows that the MD-JPA profile preserves most advantages of the UML notation, aggregating an “use on demand” set of new resources to persistence modeling. It allows the use of ORM advanced resources as needed, like the code only approach; and preserves the UML ability to create distinct diagrams with different levels of abstraction, hiding the unnecessary details in top level diagrams, without losing the mapping information stored in the low level diagrams.

A simple example illustrates how we can validate the MD-JPA profile by the use of transformations and also shows the utility of our approach: a model was built according to the complete actual example available in the Figure 11. The model used with the “acme complex example” of JPA specification.
JPA specification. This model was then transformed using our defined ATL transformations into an implementation that is equivalent to the example, thus showing that in fact our MD-JPA profile can represent adequately the annotation mappings of JPA.

Although the JPA standard scope does not comprise all ORM solutions, and is fashioned to be used with the Java language, it deals with the most important concepts of ORM, and presents mapping standards that can be applied to ORM of other object oriented platforms. In this sense, the MD-JPA profile may be used as a generic ORM modeling notation, despite the necessary changes in the transformation tool to target a different language model. Future studies might also pursue a canonical standard for ORM, what would enable a better understanding of how to build transformations between the different ORM solutions.

ACKNOWLEDGMENT

This work was partially financed by project CNPq/CT-INF 550891/2007-2 of CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

REFERENCES