Teneo: Integrating EMF & EclipseLink

Model-Driven Development with Persistence

Shaun Smith
ORACLE

Martin Taal
elver

Stephan Eberle
geensys

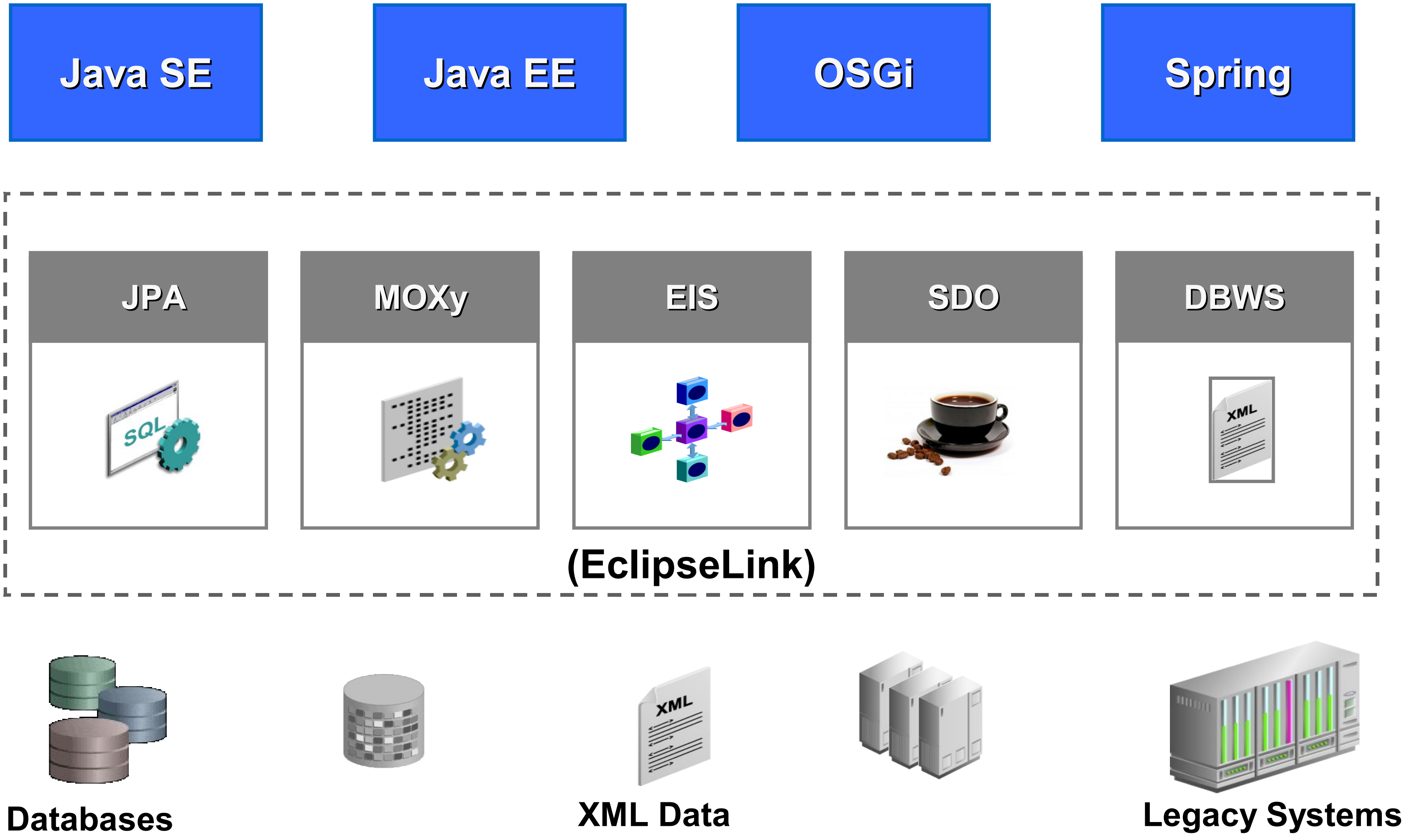
Teneo: Integrating EMF and EclipseLink JPA

- Teneo is...
  - Solution for: relational persistence for (EMF) model-driven-software development
    - Mapping of Ecore model to Relational DB
    - Override mapping behavior
    - Runtime behavior
  - Runtime framework that provides integration between EMF and EclipseLink JPA (Java Persistence API)

Why EMF and JPA?

- Teneo's EclipseLink/EMF runtime integration evolved from a collaboration between Oracle and Bosch to deal with persisting very large models.
    - e.g., engineering models of Bosch Electronic Control Unit software:
        - Definition of up to 4,000 components, 20,000 calibration parameters, etc.
        - ≈ 120 MB of specification data
- XML persistence not adequate
- Solution:
    - Put models into relational database
    - Use Java standard Java Persistence API for persisting/retrieving

Eclipse Persistence Services Project–*“EclipseLink”*



Java SE

Java EE

OSGi

Spring

JPA

MOXy

EIS

SDO

DBWS

(EclipseLink)

Databases

XML Data

Legacy Systems

Mapping with Annotations

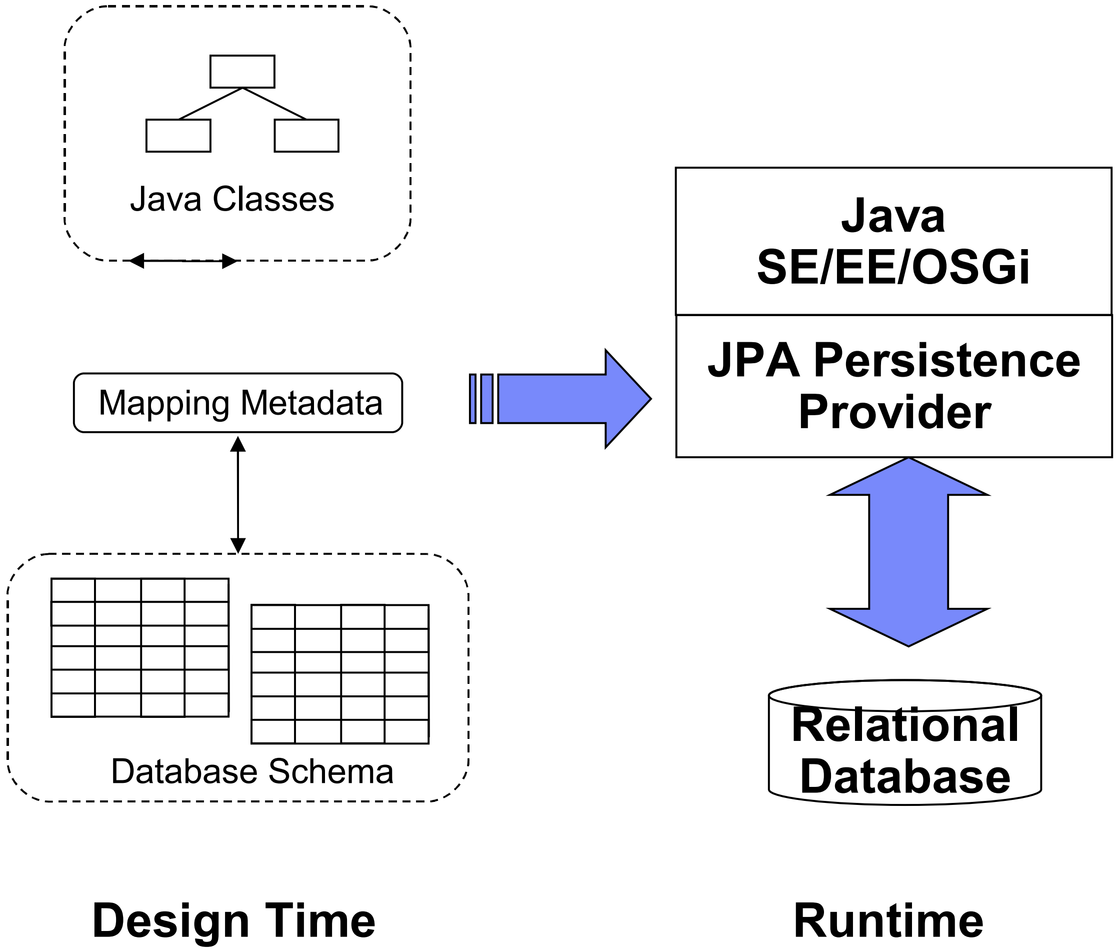
```
@Entity public class Customer {

    @Id
    private String name;
    @OneToOne
    private Account account;

    public String getName() { return name; }
    public void setName(String name) {
        this.name = name;
    }
    public Account getAccount() { return account; }
    public void setAccount(Account account) {
        this.account = account;
    }
}`
```

Mappings in XML

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/
  persistence/orm"
…
    <entity class=“Customer">
      <attributes>
        <id name=“name“/>
        <one-to-one name=“account“/>
      </attributes>
    </entity>
…
</entity-mappings>
```

JPA Design vs. Runtime

- Artifacts include:
  - Java Classes
  - Mapping Metadata
  - Database schema



EclipseLink JPA Summary

- JPA 1.0 compliant implementation
- Delivering the JPA 2.0 Reference Implementation (JSR 317)
- Java EE, Java SE, Web, Spring, and OSGi
- Any JDBC/SQL compliant database
- Schema generation
- Key infrastructure:
  - Caching, Locking, Query Framework, Mapping, …
  - JDBC connection pooling
  - Diagnostics: Logging, Profiling
  - Customization callbacks
- Highly Extensible
- … plus many valuable advanced features

EclipseLink JPA in OSGi

- EclipseLink is available as a set of generic OSGi bundles.
  - EclipseLink can run on any OSGi framework--currently being actively used on both Equinox and Felix.
  - Can also be used to build RCP and RAP applications

EMF Overview

- Eclipse Modeling Framework
- Modeling and Code Generation framework
- Some Specifics:
  - XML/XMI persistence
  - Notifications
  - Bi-directional/Structured Lists
  - Several abstractions (dynamic efeature, estore)
  - Complete XSD support (choice, list/union, substitution groups)
  - Runtime model

The EMF Persistence Challenge

- JPA is the Java standard for Relational Persistence
- JPA is designed to work with POJOs and standard java.util.* collection classes but 'default' EMF generated classes do not conform to the JPA requirements for an Entity (a persistent Java object).

EMF Collection Challenges—not java.util.*

- Collection instance variables of generated EMF models are not typed as one of JPA supported: List, Set, Map, Collection. They are EList or EMap, e.g.:

```
protected EList<Writer> writers;
```

- EclipseLink supports 'custom collection classes' but had to relax JPA mapping validation to allow for subclasses of List, etc.

EMF Collection Challenges—ownership, class?

- EMF collection classes are more complex than java.util collections
  - EMF collections know their 'owner', item type, and other information which must be provided in the constructor--EMF collections do not provide a no-arg constructor
  - The concrete implementation class for a collection varies greatly. Which class should EclipseLink instantiate when building an Entity?
  - Solution: EclipseLink uses the EMF model to instantiate the correct collection class for an instance variable collection, e.g.:

```
EListFactory.eINSTANCE.createEList(owner, attrName)
```

EMF Collection Challenges—lazy loading

- EclipseLink lazy loading uses a proxy (ValueHolder) that holds enough information to query the related object(s).
    - EMF collections know their 'owner', item type, and other information which must be provided in the constructor.
    - Solution: Teneo extends EclipseLink with ValueHolders that know their owner.
- EclipseLink provides lazy ('indirect') collections that implement java.util. classes but EMF collections are typed EList or EMap.
    - Teneo extends EclipseLink with indirect collections that implement EList and Emap
    - Teneo's indirect collections are a type of ValueHolder that know their owner, item type, etc.

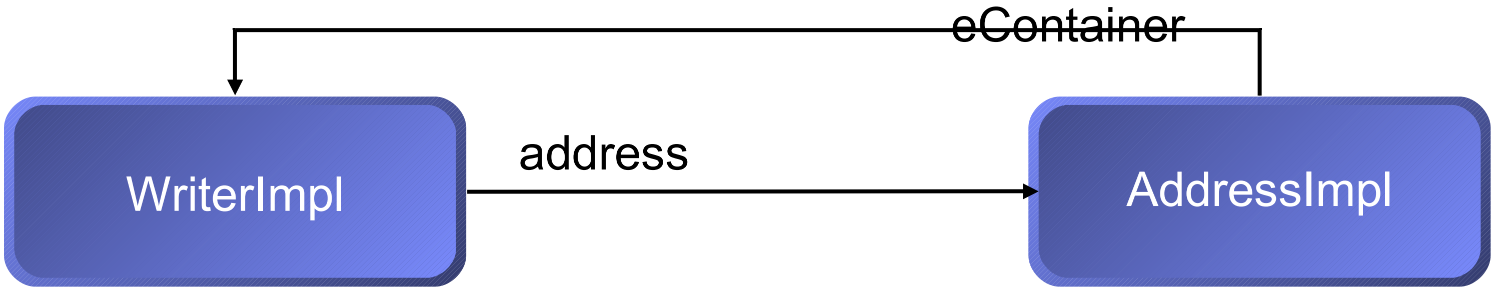
Relationship Management

- JPA Persistence Providers construct objects from relational data
- EclipseLink has to either work around or disable relationship management during construction
- EMF initializes custom collection classes in getter

```
public EMap<String, Book> getBooks() {
   if (books == null) {
    books = new EcoreEMap<String,Book>(
      LibraryPackage.Literals.STRING_TO_BOOK_MAP_ENTRY,
      StringToBookMapEntryImpl.class,
      this,
      LibraryPackage.LIBRARY__BOOKS);
   }
   return books;
 }
```

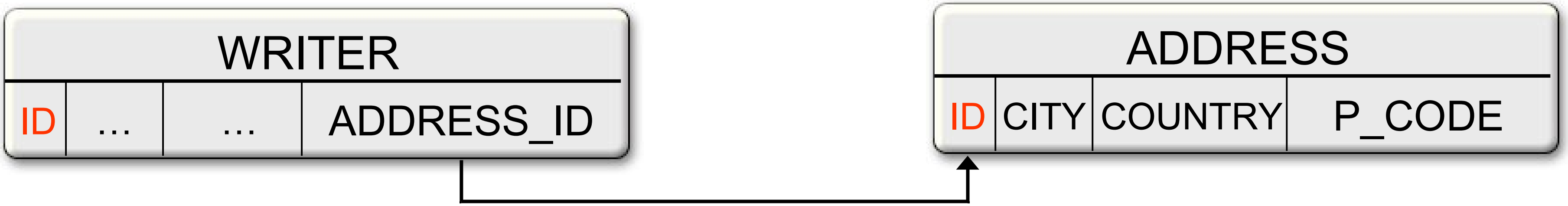
- Solution:
  - EclipseLink sets fields directly through reflection
  - EclipseLink gets using getter to leverage lazy loading

EMF JPA Idioms: eContainer



```
<entity name="Writer"
     class="WriterImpl">
   <attributes>
    …
     <one-to-one name="address"
       target-entity="AddressImpl" >
```

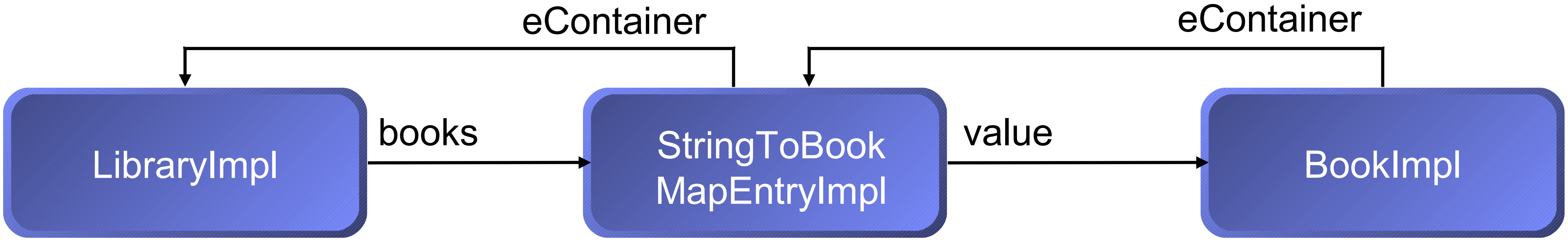
```
<entity name="Address"
    class="AddressImpl">
  <attributes>
    …
    <one-to-one name="eContainer"
       target-entity="WriterImpl"
       mapped-by="address">
    </one-to-one>
    …
```



WRITER			
ID	…	…	ADDRESS_ID

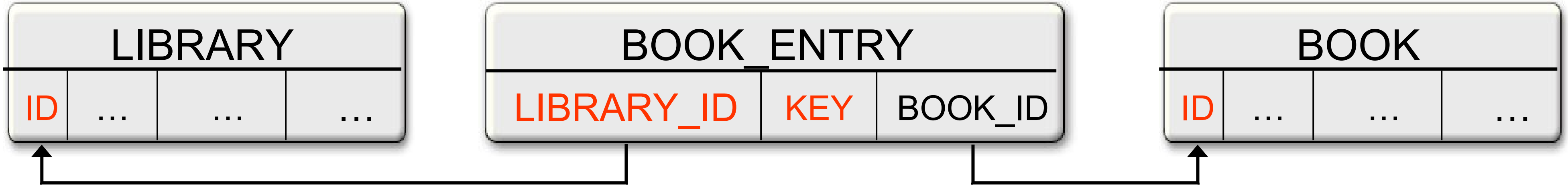
ADDRESS			
ID	CITY	COUNTRY	P_CODE

EMF JPA Idioms: MapEntry



```
<entity name="Library"
    class="LibraryImpl">
  <attributes>
    …
    <one-to-many name="books"
      target-entity="StringToBookMapEntryImpl"
      mapped-by="eContainer">
```

```
<entity name="Book"
    class="BookImpl">
  <attributes>
    …
    <one-to-one name="eContainer"
      target-entity=
         "StringToBookMapEntryImpl"
      mapped-by="value">
    </one-to-one>
```



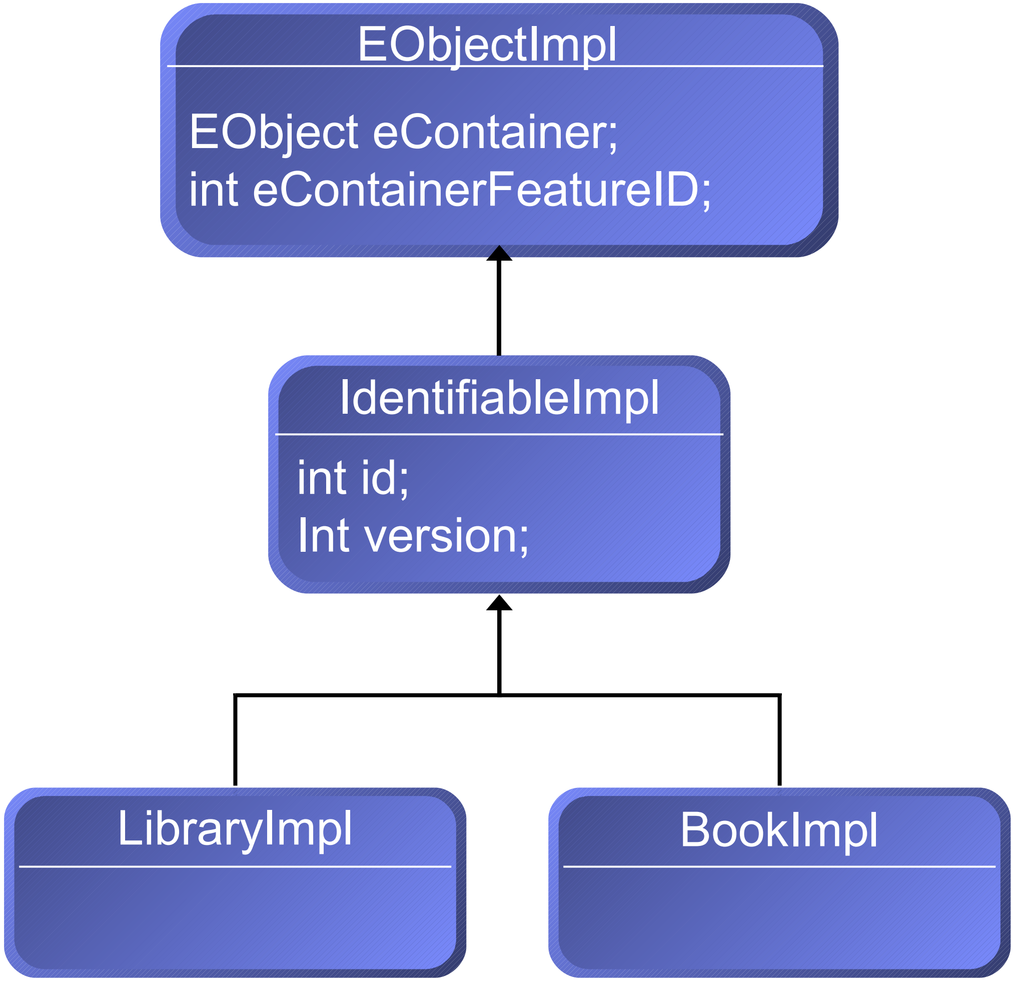
LIBRARY			
ID	…	…	…

BOOK_ENTRY		
LIBRARY_ID	KEY	BOOK_ID

BOOK			
ID	…	…	…

EMF JPA Idioms: Common Fields

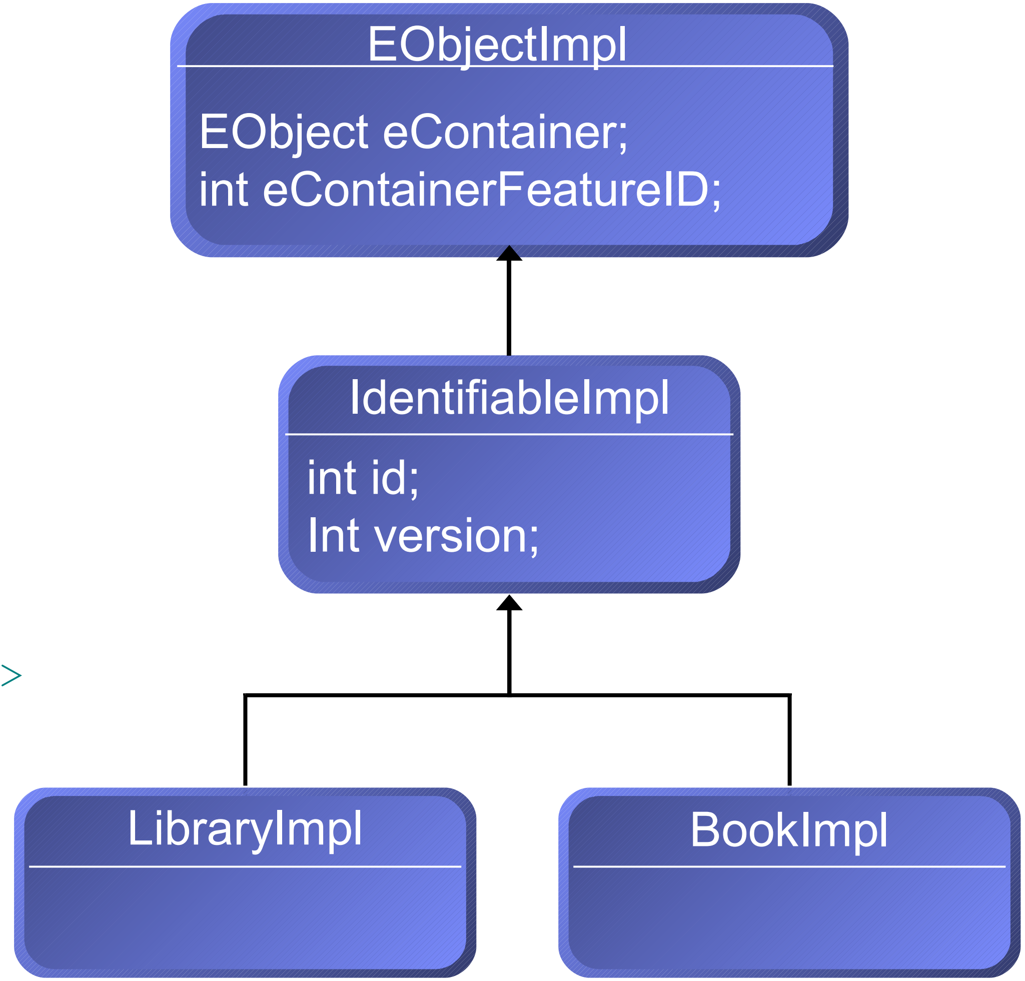
- All EObjects have an eContainerFeatureID.
- Entities must have an id field and should have a version field for optimistic locking.
- Best practice: define a common abstract root class in your model with common fields.



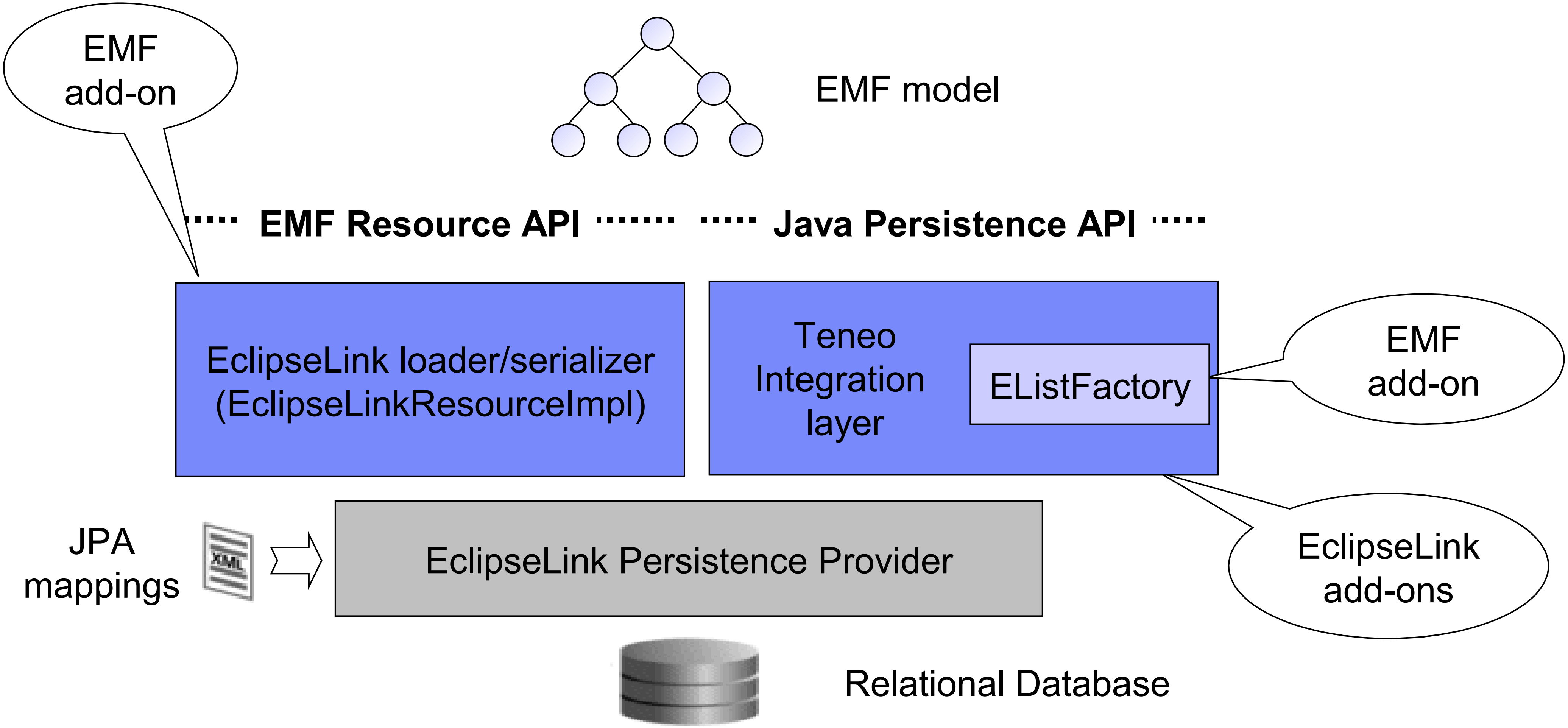
EMF JPA Idioms: Common Fields (cont.)

- Map this abstract class as a MappedSuperclass

```
<mapped-superclass
  class="IdentifiableImpl"
  access="FIELD">
  <attributes>
    <id name="id">
        <column name="ID"/>
        <generated-value/>
    </id>
    <basic name="eContainerFeatureID"/>
    <version name="version"/>
  </attributes>
</mapped-superclass>
```



Teneo EclipseLink Runtime



Development Approaches

- Meet in the middle
  - Map existing database schema to a generated EMF model
- Top down model-centric generating:
  - Model classes
  - Database schema
  - JPA mapping metadata

Meet-in-the-middle mapping

- Teneo supports the meet-in-the-middle approach of mapping EMF classes to an existing relational schema to enable the construction of model driven applications on top of existing or legacy databases.
- Teneo can be combined with the Dali JPA Tools for meet-in-the-middle development with Dali providing intelligent mapping assistance and validation against the target relational schema.

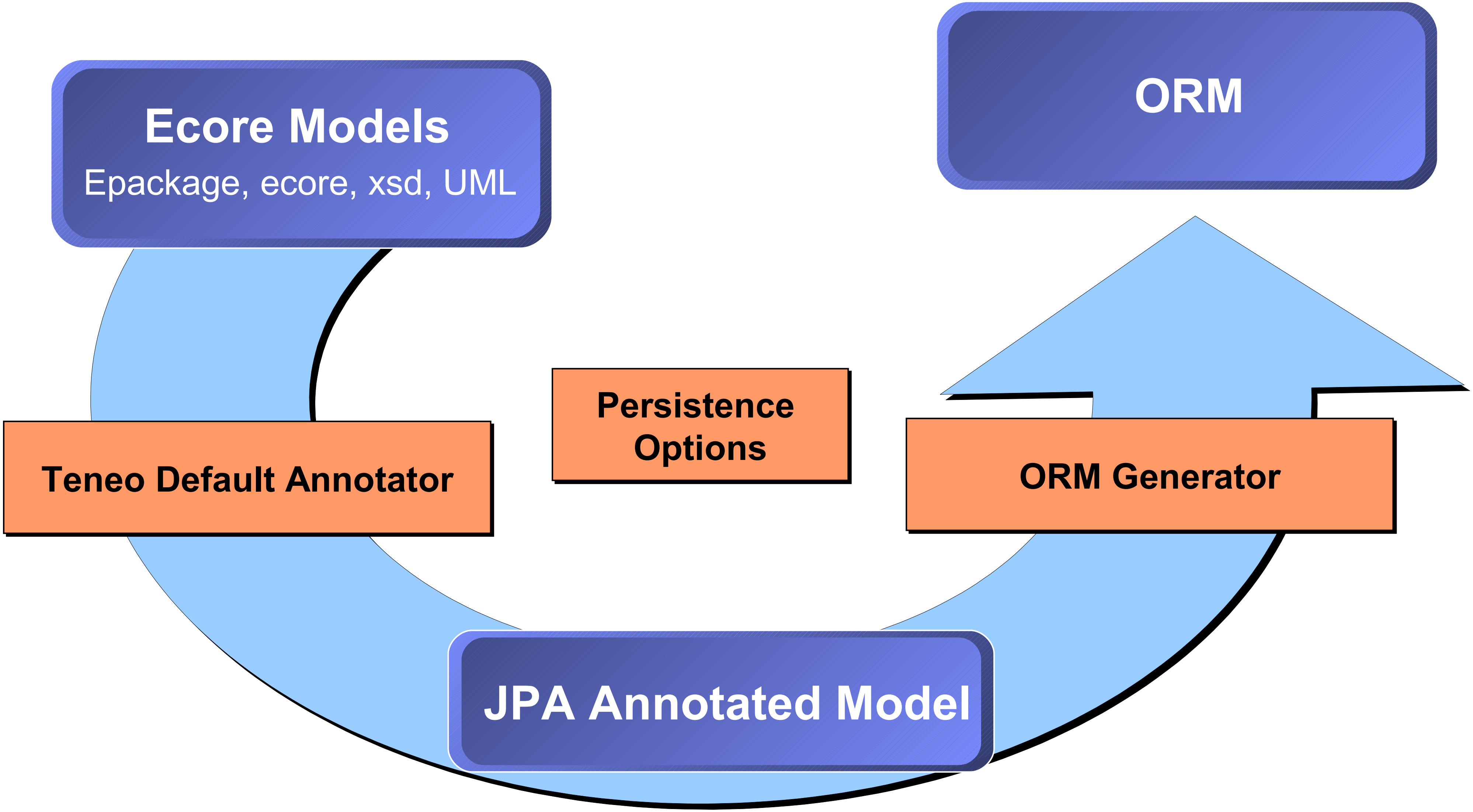
Top-down/Model-driven development

- Start is an ecore/xsd/uml model
- Generates the mapping directly
- Annotate the model:
  - In the model itself
  - In a separate xml file
- Use specific persistence options

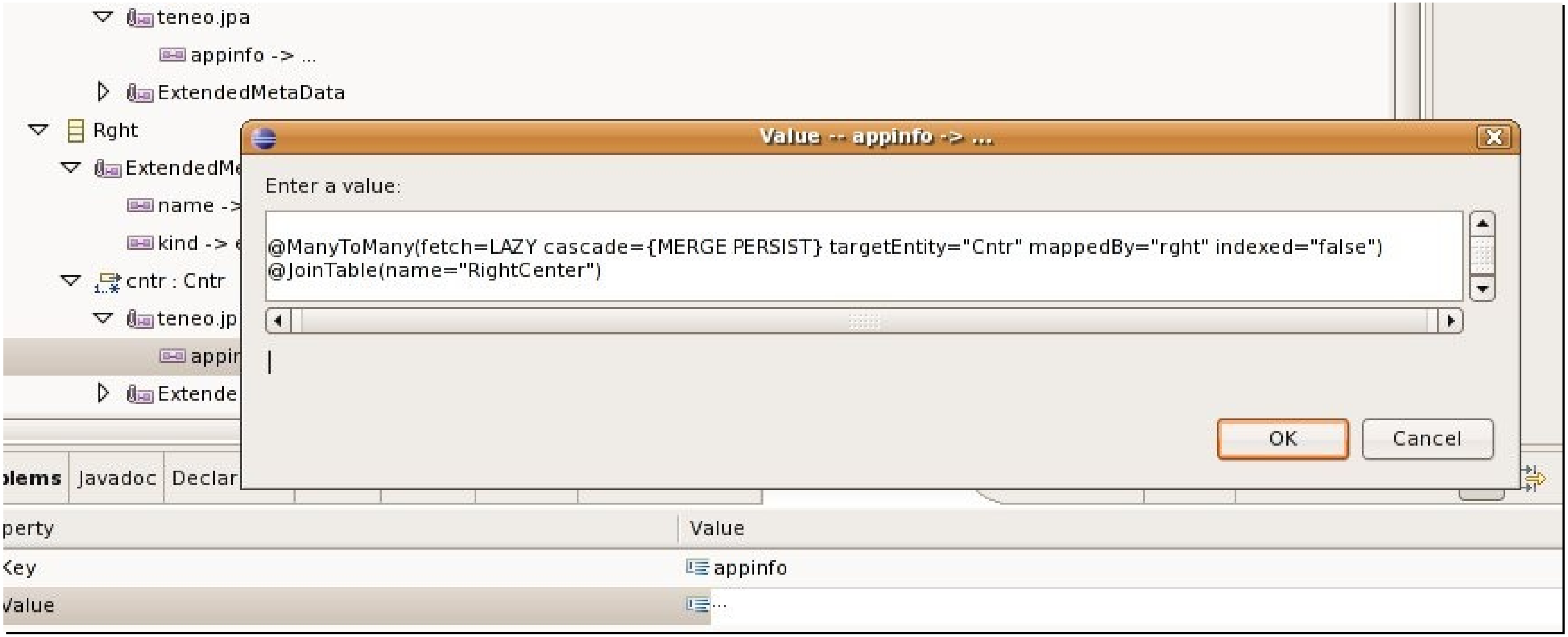
Generate OR-Mapping

- Decides on join tables versus foreign key
- Chooses side for join-column
- Set cascades
- Handles primitive type mapping
- Handles bi-directional relations
- Takes care of mapping list, set, map
- Ensures unique naming
    - Readable/Logical
    - Unique, prevent name clashes
    - Handle name length constraints

Generate directly



Adding Annotations: In the Model



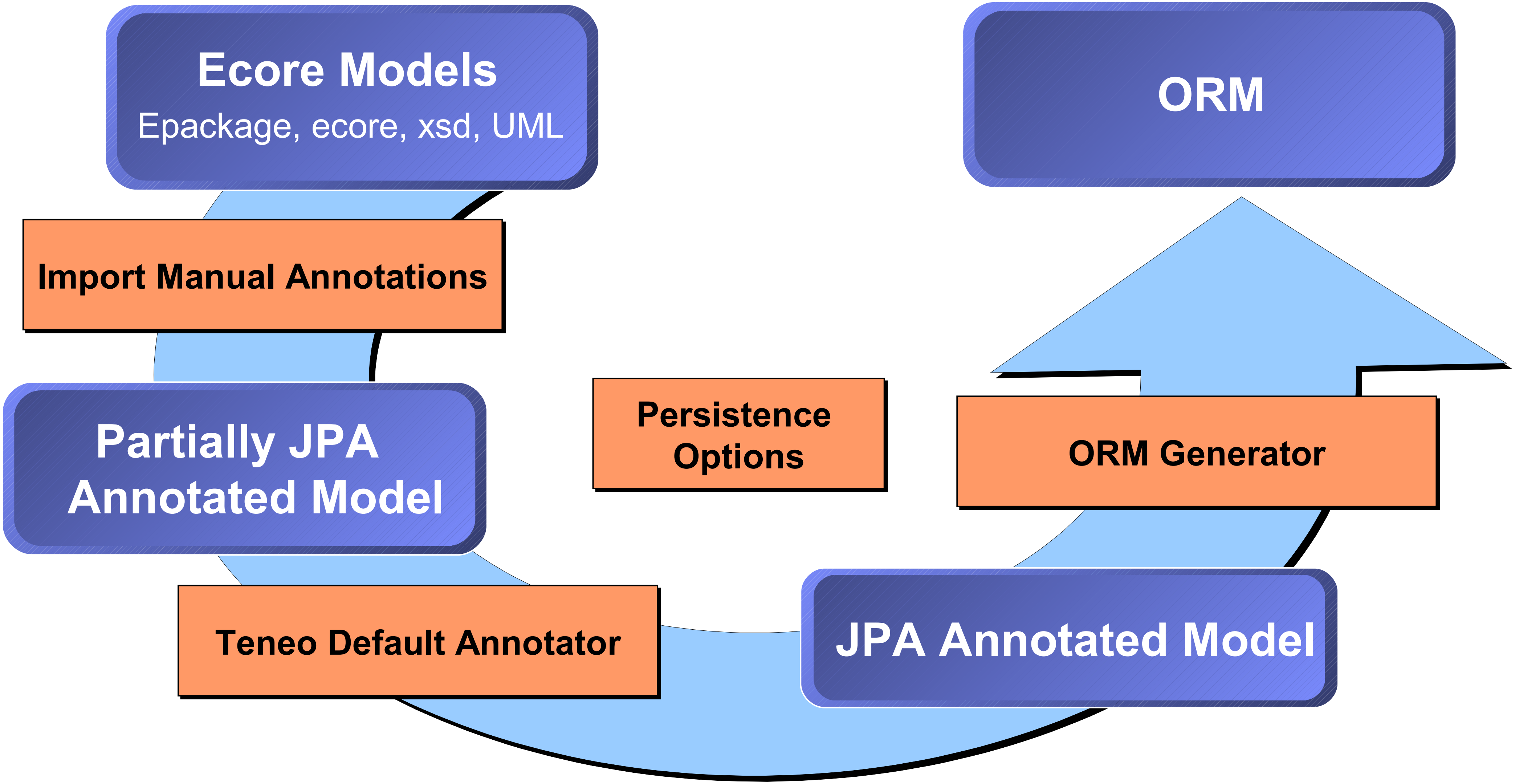
```
@ManyToMany(fetch=LAZY cascade={MERGE PERSIST} targetEntity="Cntr" mappedBy="rght" indexed="false")
@JoinTable(name="RightCenter")
```

perty	Value
Key	appinfo
Value	...

Annotate in XML

```
<eclass name="Person">
        <property name="address">
                <many-to-one fetch="EAGER" target-entity="Address" optional="true">
                        <cascade>MERGE</cascade>
                        <cascade>PERSIST</cascade>
                        <cascade>REFRESH</cascade>
                </many-to-one>
        </property>
</eclass>
<edatatype name="WeightType">
        <column name="weight" nullable="true" precision="5" scale="2"/>
</edatatype>
```

Generate from annotated models



Ecore Models
Epackage, ecore, xsd, UML

Import Manual Annotations

Partially JPA Annotated Model

Teneo Default Annotator

Persistence Options

JPA Annotated Model

ORM Generator

ORM

Summary

- Relational persistence is the natural choice for very large EMF models.
- JPA is the Java standard for relational persistence.
- Therefore, JPA should be used for EMF relational persistence.
- EMF has a number of 'unique' features that conflict with the expectations and requirements of JPA
- EclipseLink is highly extensible so it is has been extended in Teneo with special support for EMF.
- Teneo supports both model centric and meet in the middle JPA/EMF development.
- All the core features are working but we still have work to do…