The Good, the Bad and the Ugly

2 years with Java Persistence API

Björn Beskow

bjorn.beskow@callistaenterprise.se
www.callistaenterprise.se

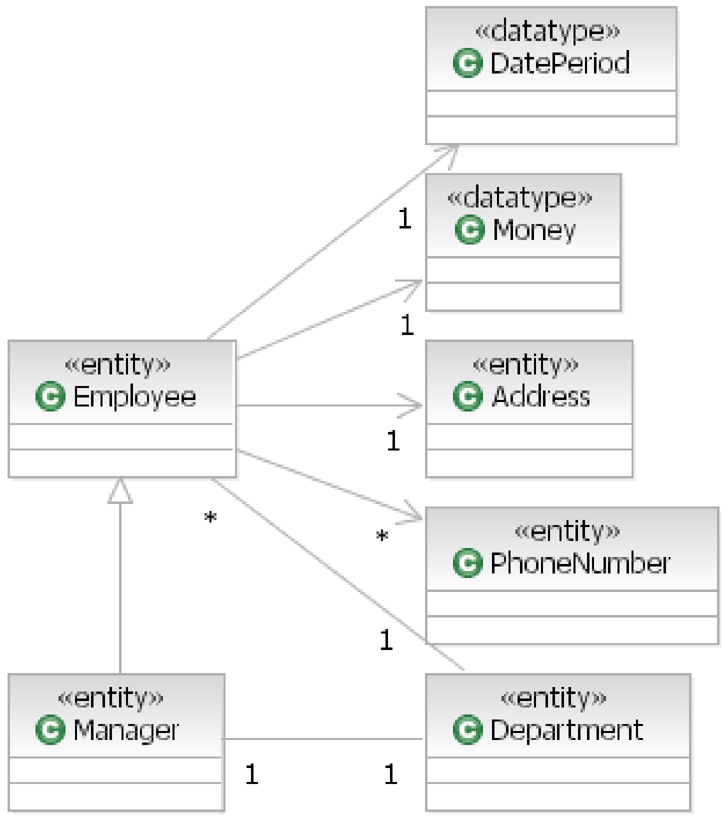
CALLISTA

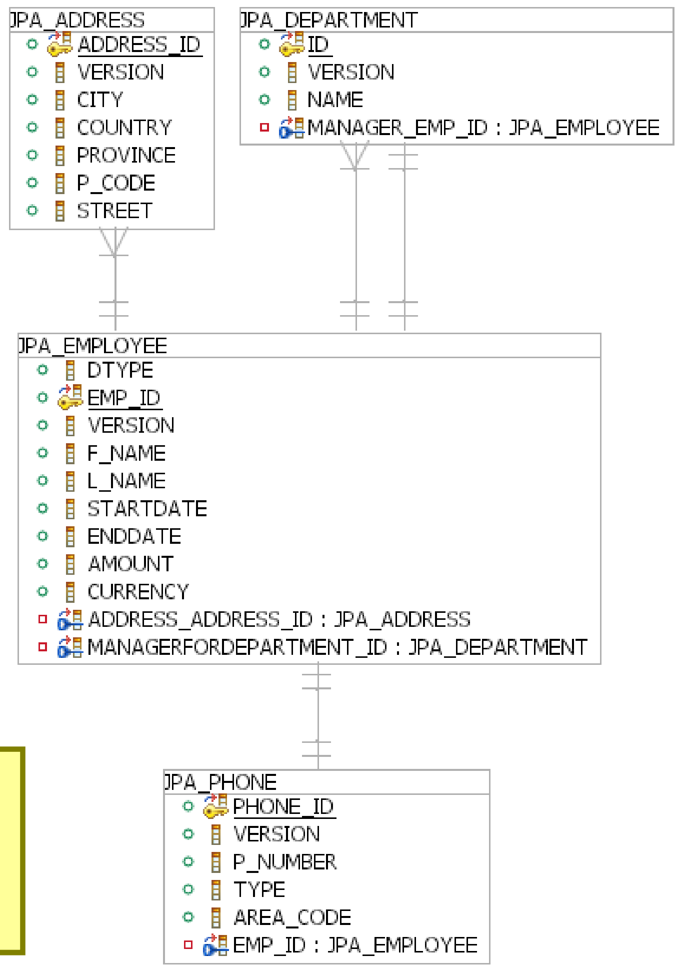
Agenda

- The Good
  - Wow! Transparency!
- The Bad
  - Not that transparent after all …
- The Ugly
  - JPA Deployment model and JavaEE integration …
- What's next?

CALLISTA

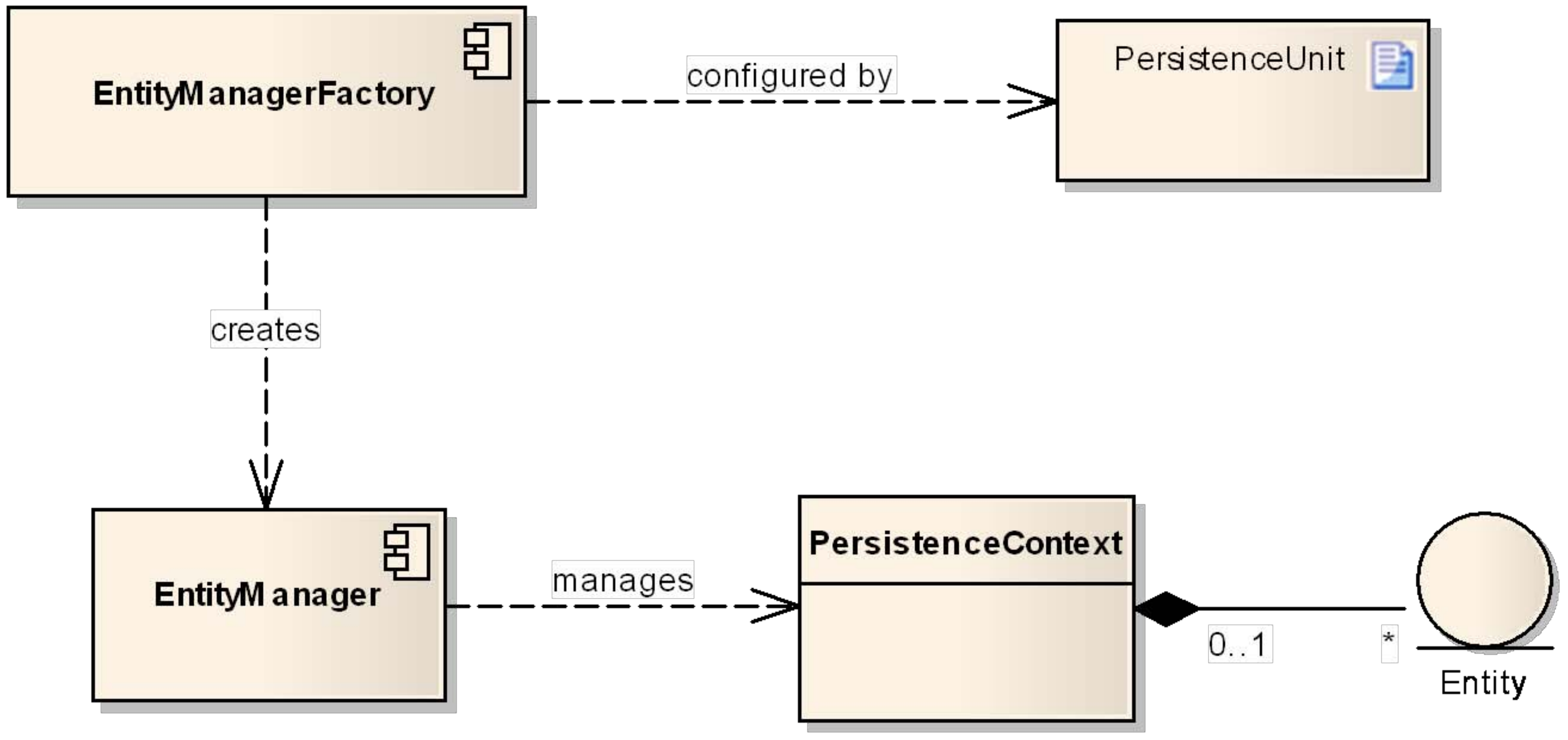
JPA 1.0 – Industry Standard Object Relational Mapping framework





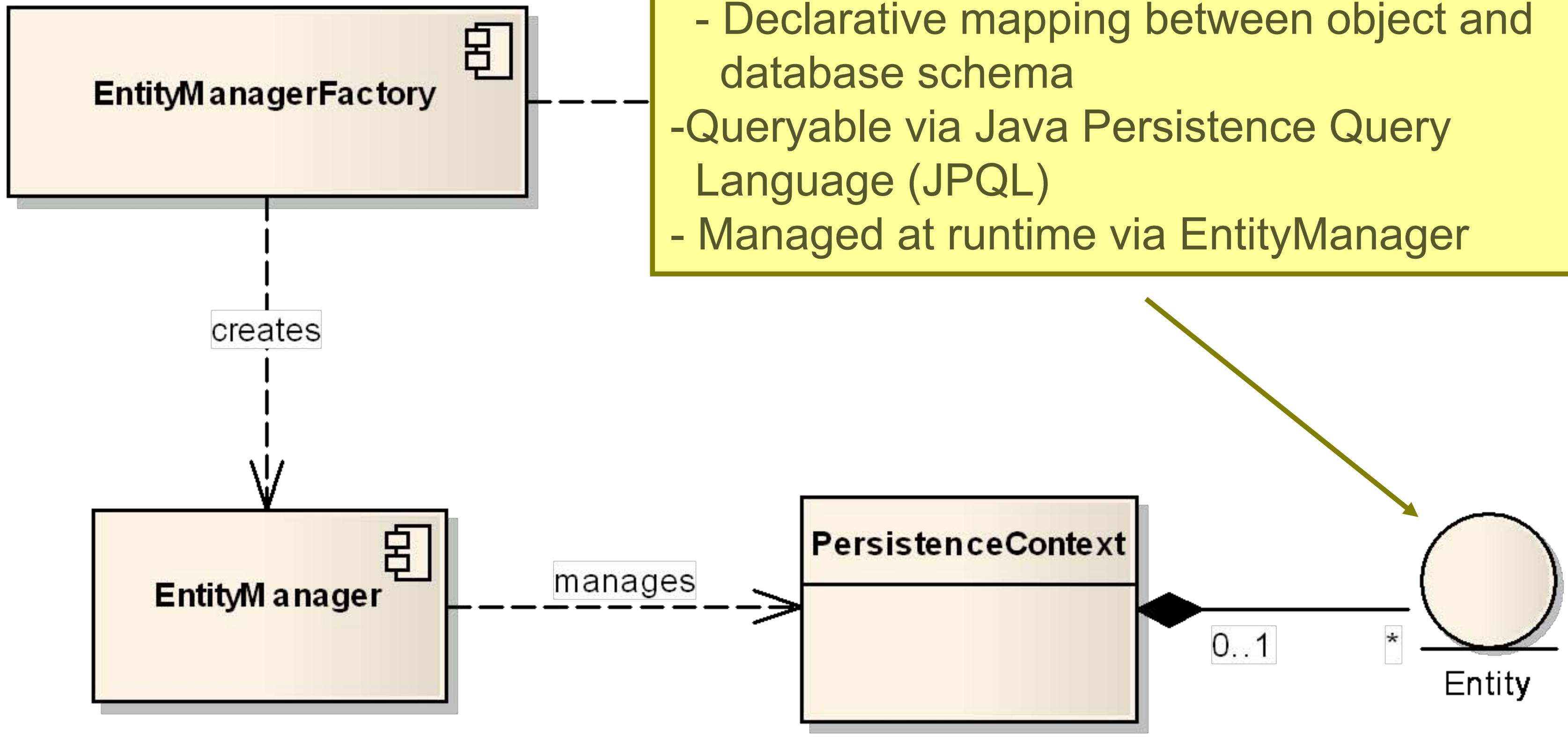
- **Declarative Mapping**
- **Framework API**
- **Query Language**
- **Transaction support**

JPA Key Concepts



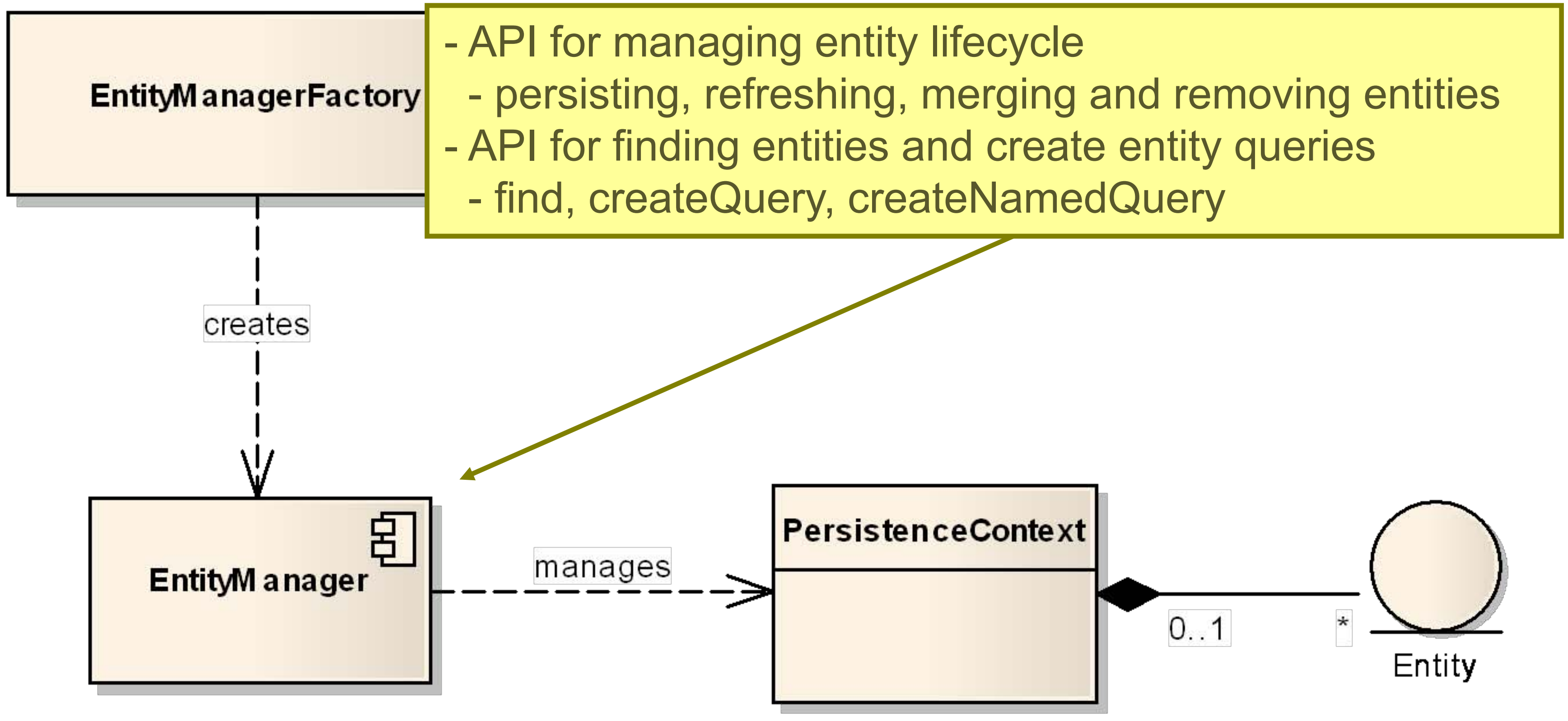
CALLISTA

Key Concepts: Entity



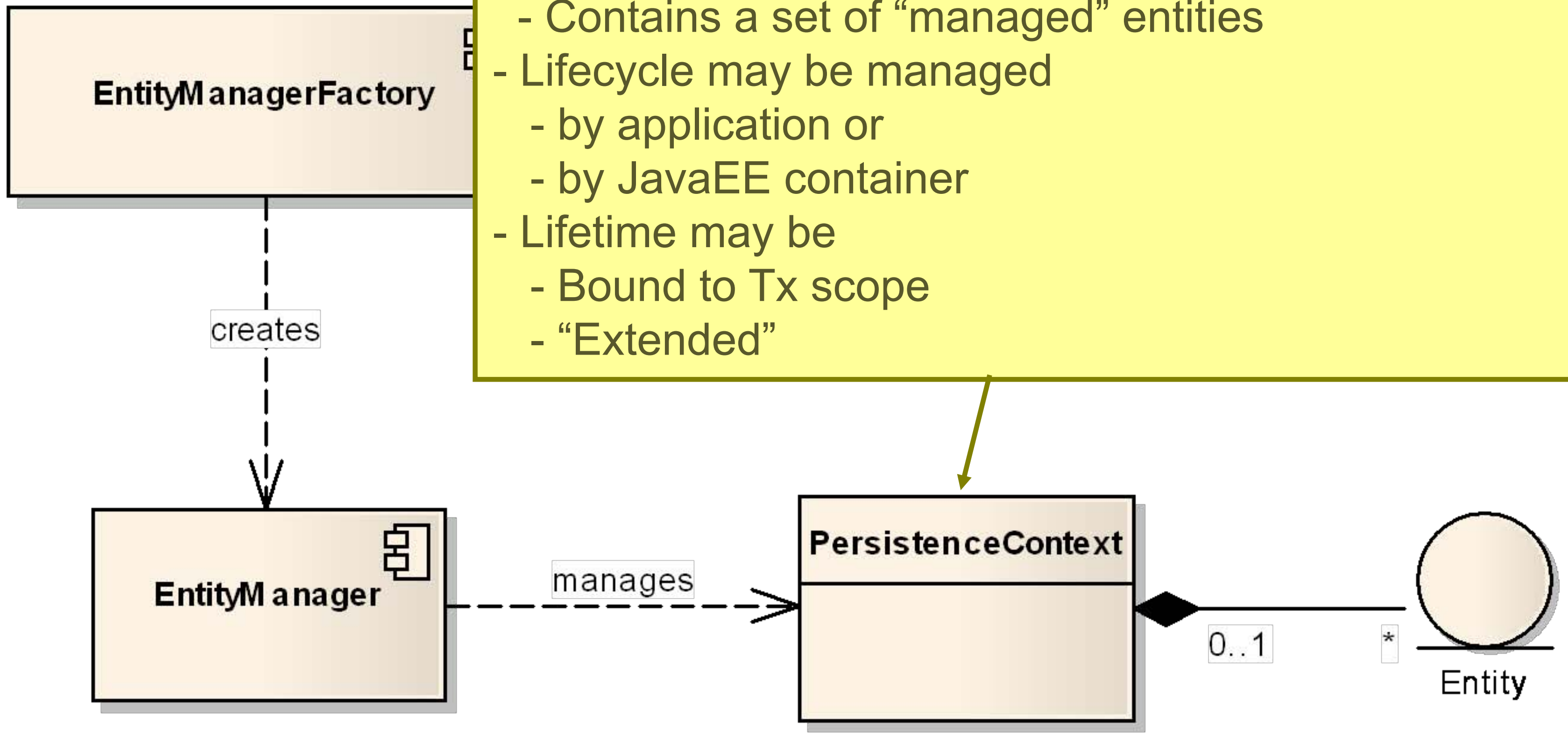
- Plain Old Java Object with persistent identity
  - Declarative mapping between object and database schema
- Queryable via Java Persistence Query Language (JPQL)
- Managed at runtime via EntityManager

Key Concepts: Entity Manager



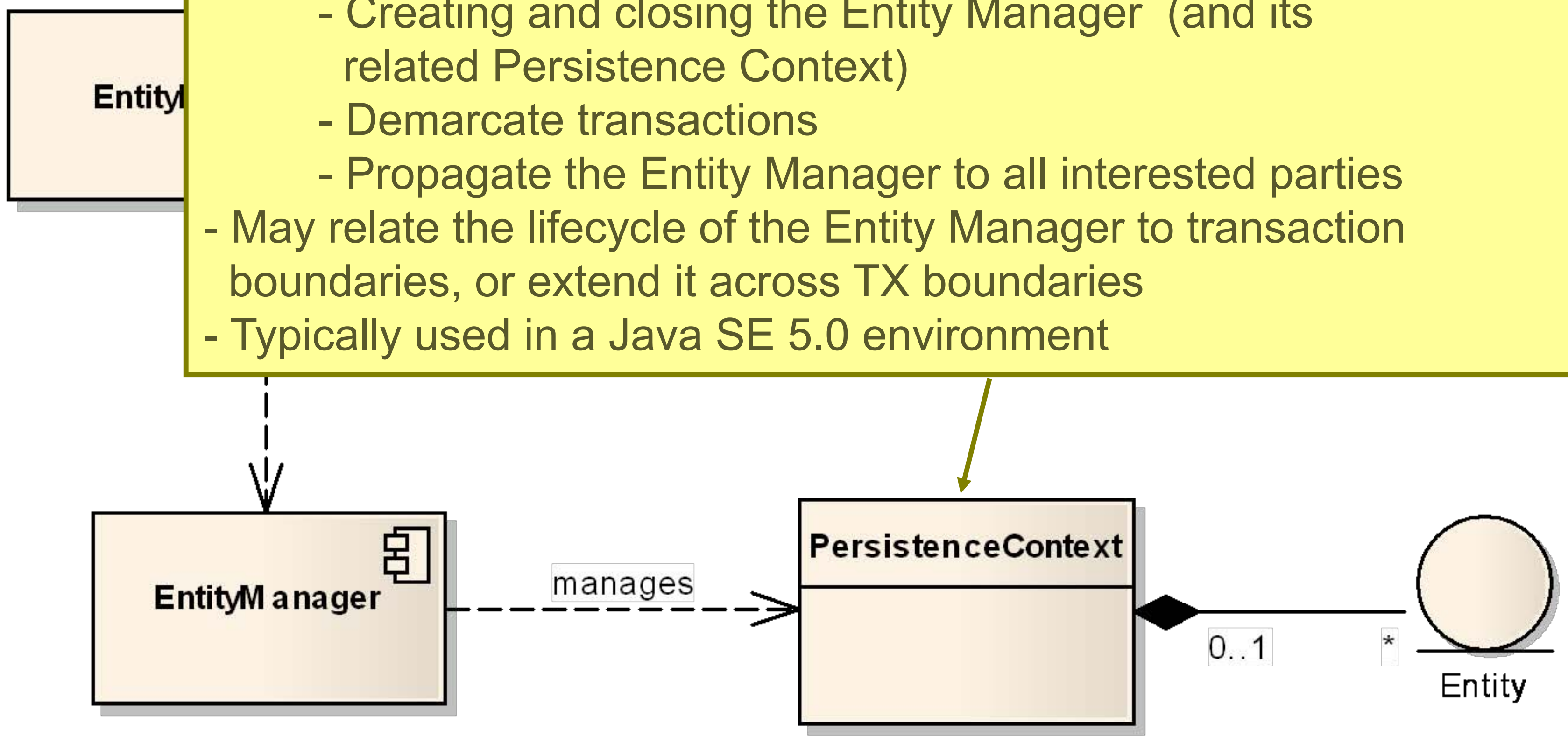
- API for managing entity lifecycle
  - persisting, refreshing, merging and removing entities
- API for finding entities and create entity queries
  - find, createQuery, createNamedQuery

Key Concept: Persistence Context



- Runtime context for an Entity Manager
  - Contains a set of "managed" entities
- Lifecycle may be managed
  - by application or
  - by JavaEE container
- Lifetime may be
  - Bound to Tx scope
  - "Extended"

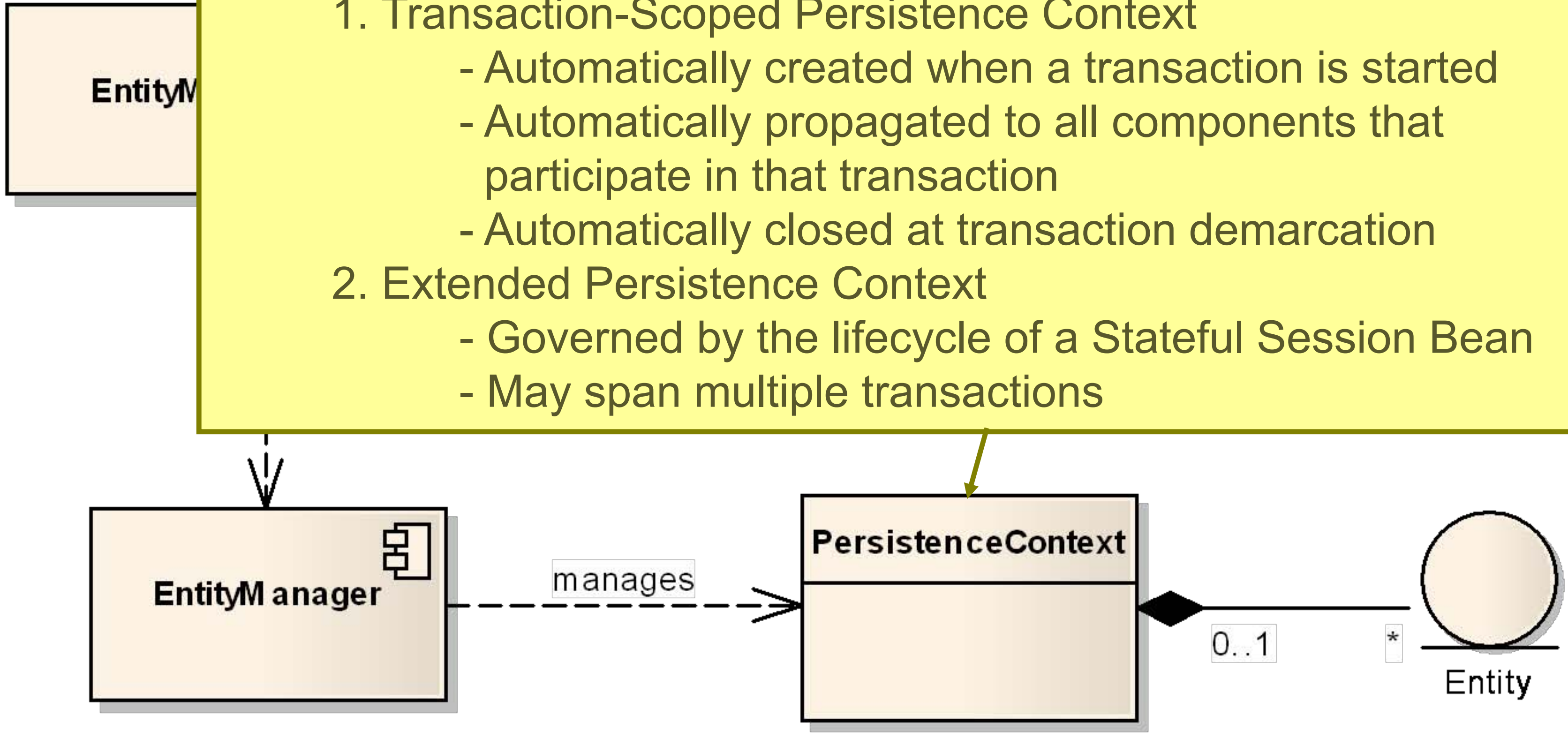
Application Managed Persistence Context



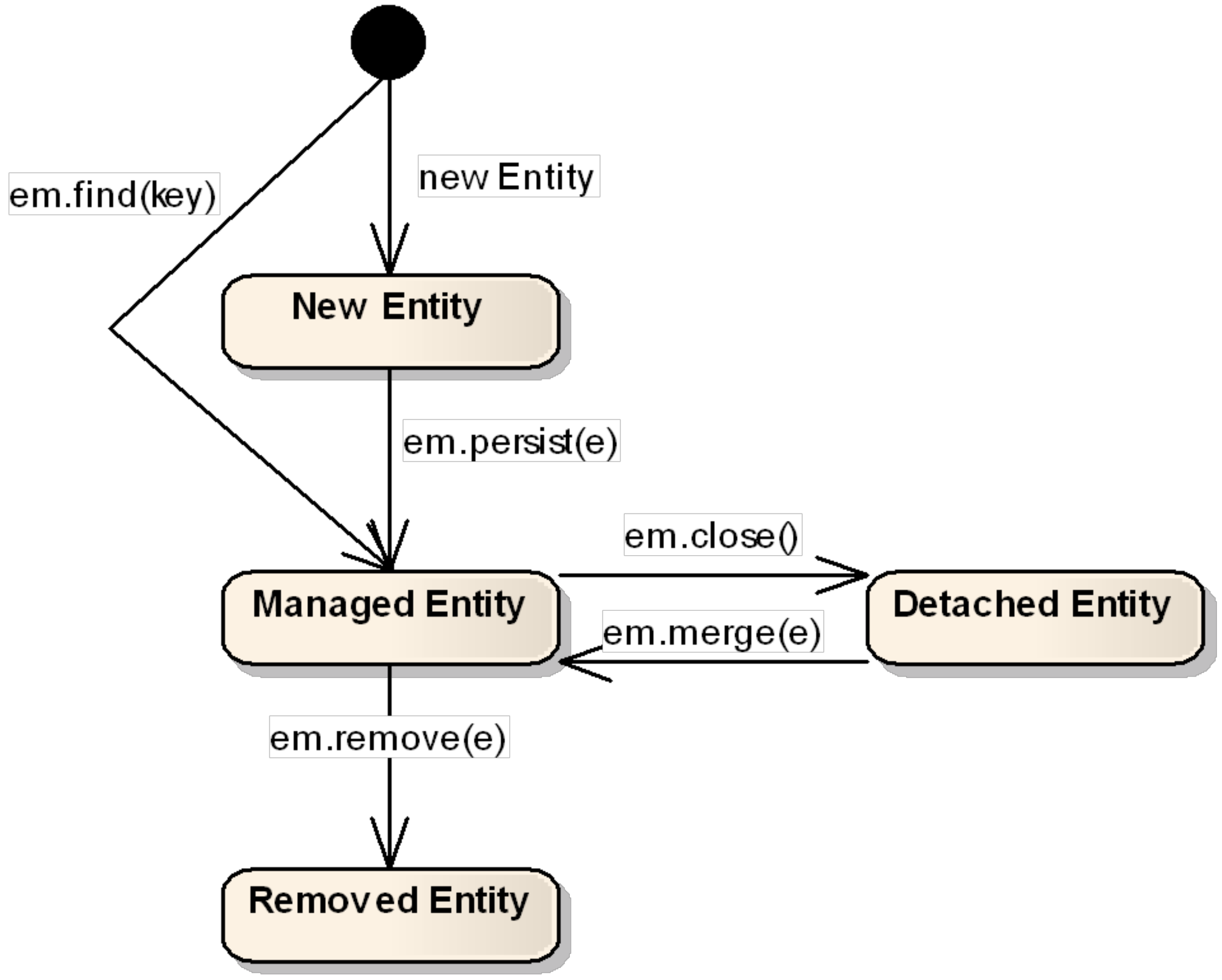
- The application manages the Entity Manager, i.e. is responsible for
    - Creating and closing the Entity Manager (and its related Persistence Context)
    - Demarcate transactions
    - Propagate the Entity Manager to all interested parties
- May relate the lifecycle of the Entity Manager to transaction boundaries, or extend it across TX boundaries
- Typically used in a Java SE 5.0 environment

Container-Managed Persistence Context

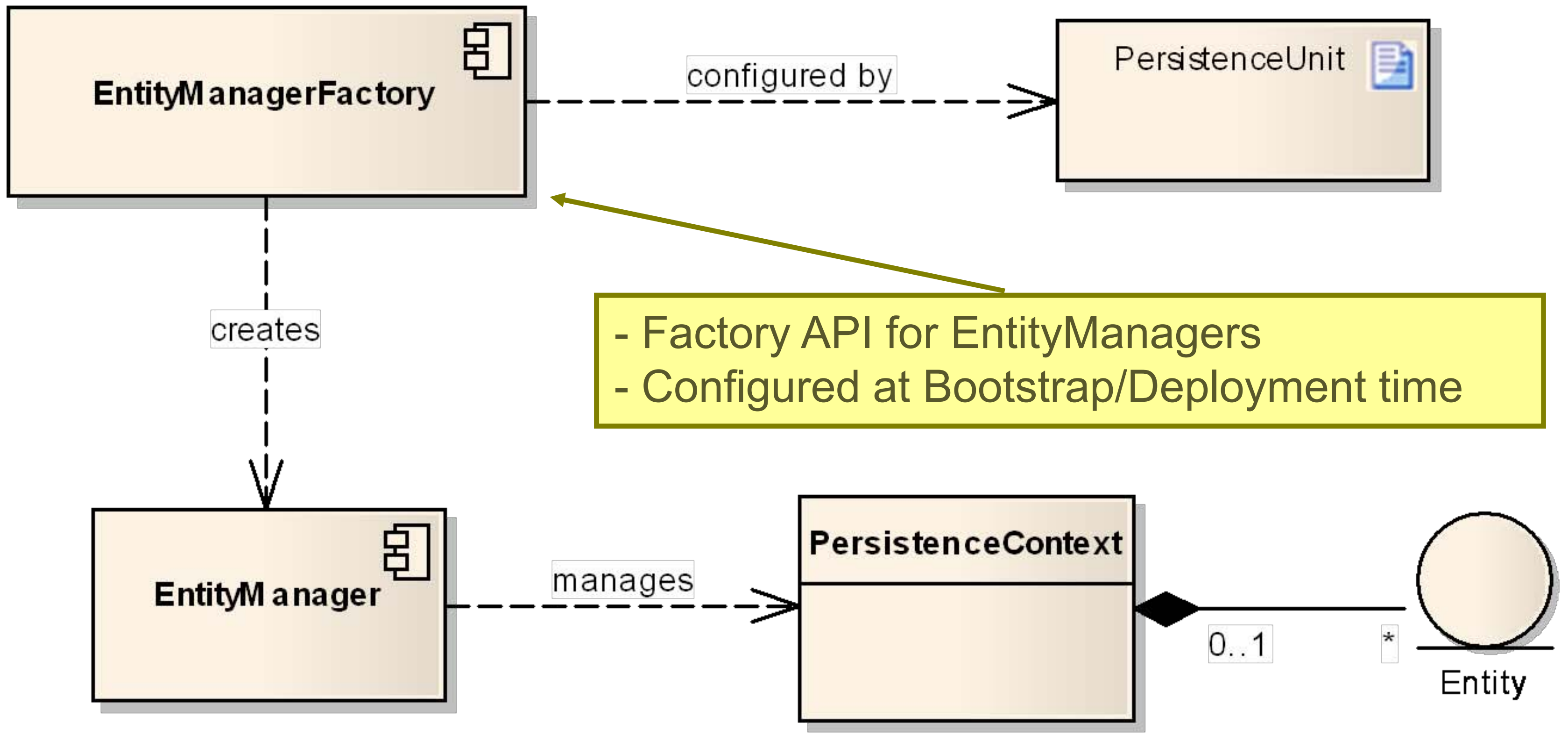
- A JavaEE container manages the Entity Manager and its related Persistence Context
- Comes in two flavors
  1. Transaction-Scoped Persistence Context
     - Automatically created when a transaction is started
     - Automatically propagated to all components that participate in that transaction
     - Automatically closed at transaction demarcation
  2. Extended Persistence Context
     - Governed by the lifecycle of a Stateful Session Bean
     - May span multiple transactions



State Model for JPA Entities



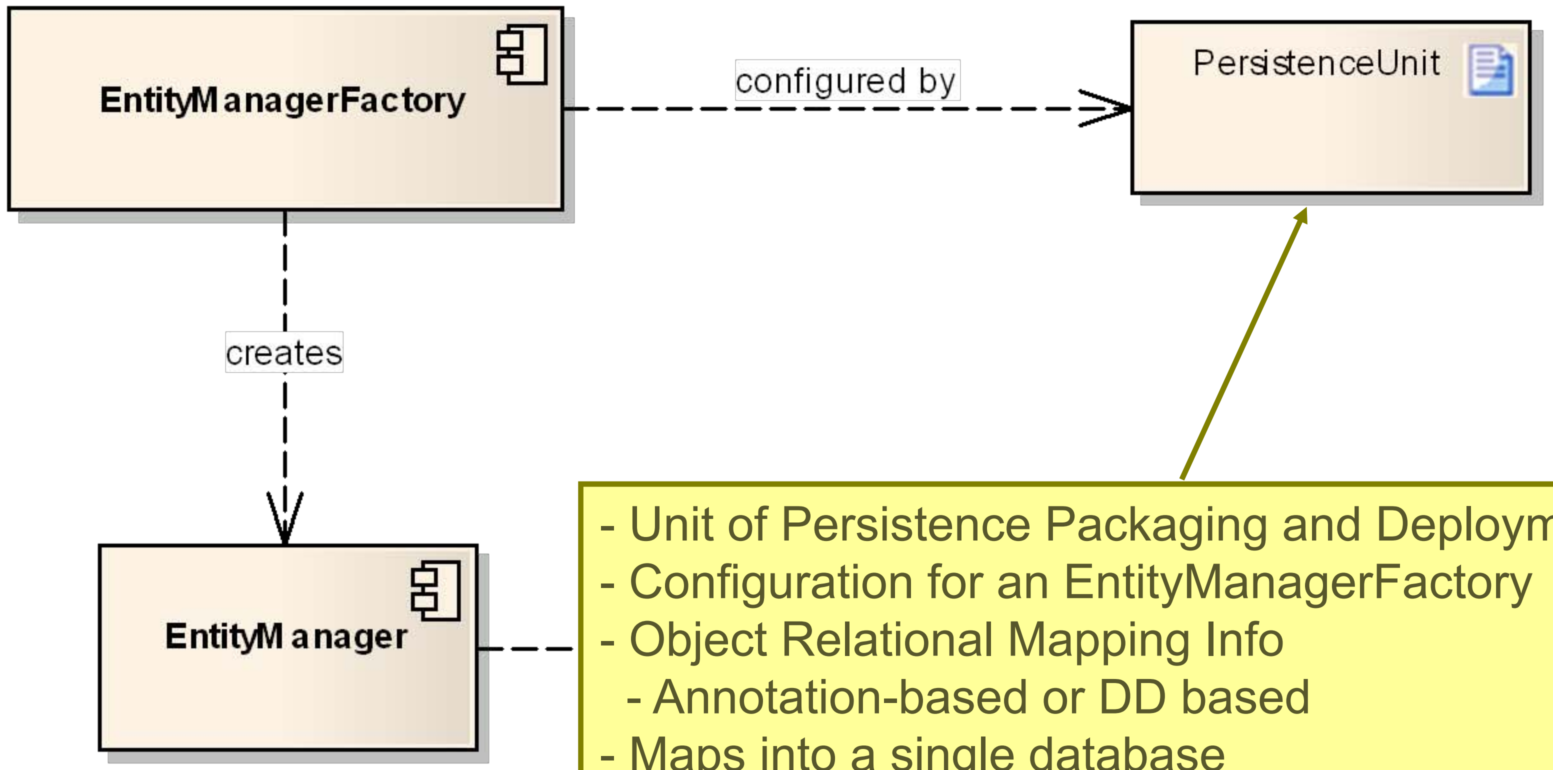
Key Concept: EntityManagerFactory



- Factory API for EntityManagers
- Configured at Bootstrap/Deployment time

CALLISTA

Key Concept: PersistenceUnit

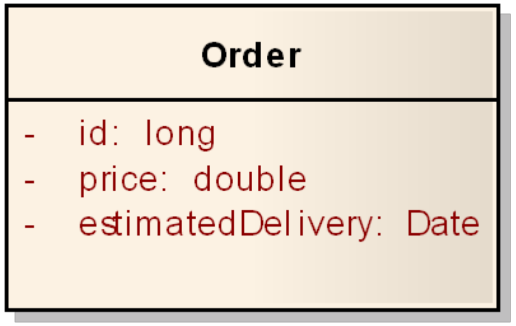


- Unit of Persistence Packaging and Deployment
- Configuration for an EntityManagerFactory
- Object Relational Mapping Info
  - Annotation-based or DD based
- Maps into a single database

CALLISTA

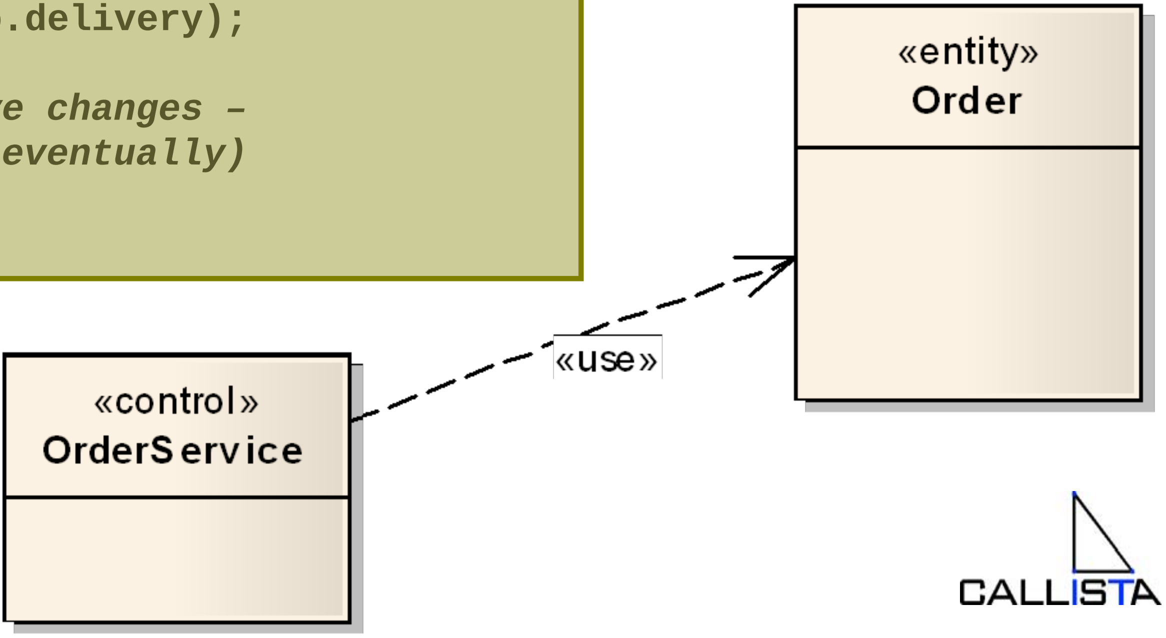
Example Entity

```
@Entity
public class Order implements Serializable {

  @Id @GeneratedValue
  private Long orderId;

  private double price;

  private Date estimatedDelivery;

  …
}
```



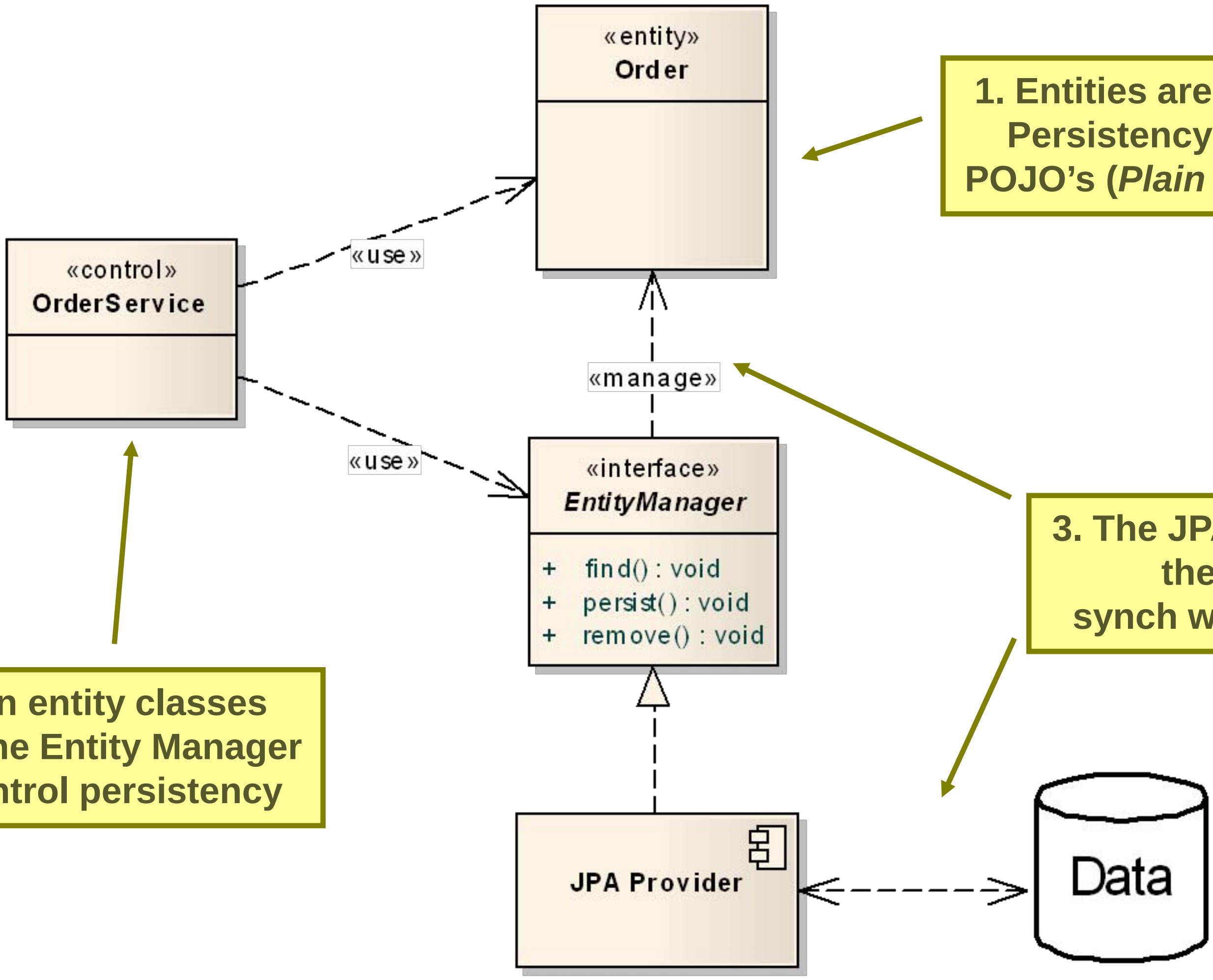
Example EntityManager usage

```
@Stateless
public class OrderServicesBean
  implements OrderServices {
  @PersistenceContext(unitName = "OrderPU")
  private EntityManager entityManager;
  public void processOrder(long orderId) {

    Order o = entityManager.find(orderId,
                                 Order.class);
    ...
    o.setPrice(calculatePrice(o.getCustomer());
    e.setEstimatedDelivery(erp.delivery);
    ...
    // No explicit call to save changes –
    // happens automatically (eventually)
  }
}
```



CALLISTA

The Good: POJO based, transparent persistence



1. Entities are unaware of their Persistency – they are pure POJO's (*Plain Old Java Objects*)

2. Non entity classes uses the Entity Manager to control persistency

3. The JPA provider keeps the entities in synch with the database

No more Embedded CRUD SQL …

```
Connection con = datasource.getConnection();
PreparedStatement stmt = null;
try {
    stmt = con.prepareStatement("UPDATE products SET price = ?");
    stmt.setInt(1, 200);
    stmt.executeUpdate();
} finally {
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException ex) {
            logger.warn("Could not close statement", ex);
        }
    }
    try {
        con.close();
    } catch (SQLException ex) {
        logger.warn("Could not close connection", ex);
    }
}
```

No more DAOs ...

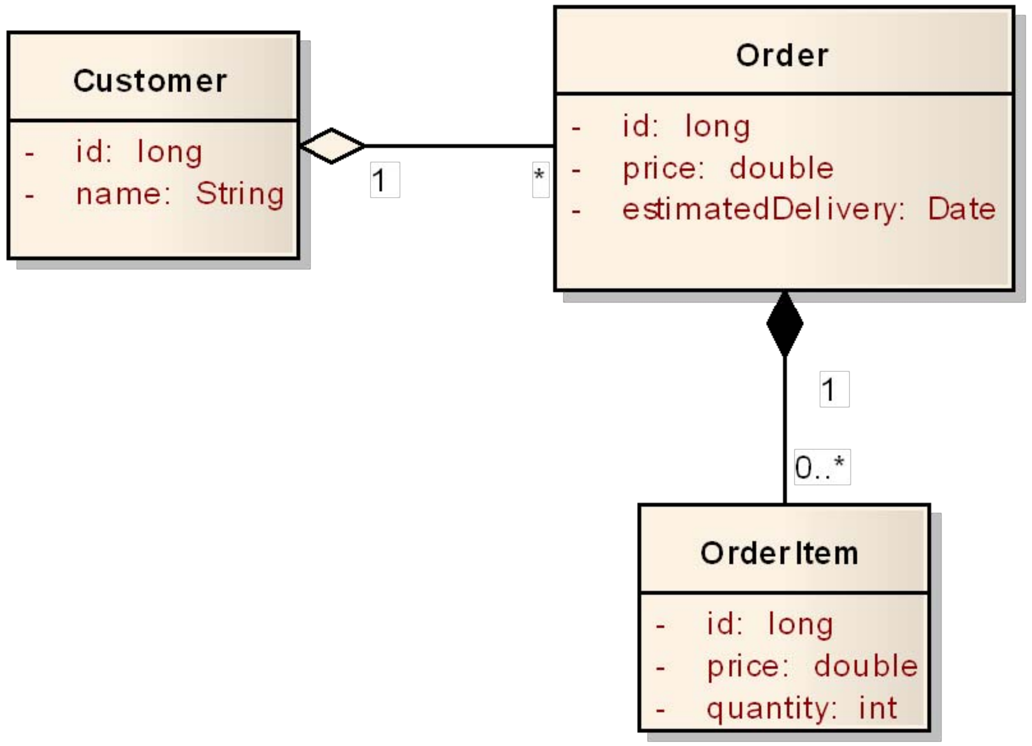
```
public interface CustomerDao {

   public void createCustomer(CustomerDTO customer)

   public CustomerDOT retrieveCustomer(String ssn)
                throws UnknownCustomerException;

   public CustomerDTO updateCustomer(CustomerDTO customer)
                throws UnknownCustomerException;

   public void deleteCustomer(String ssn)
                throws UnknownCustomerException;
}
```

No more DTOs ...

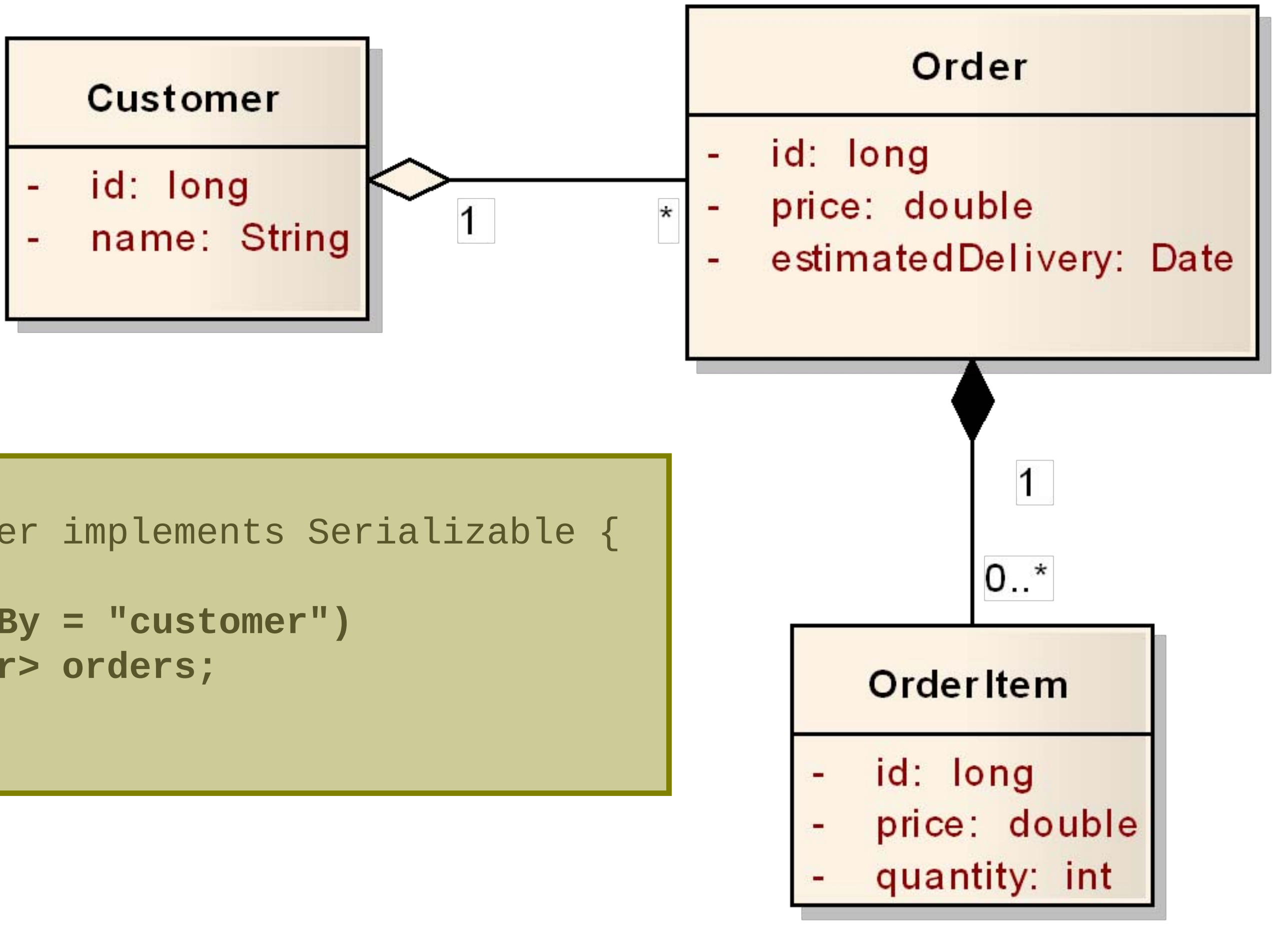
```
public class CustomerDTO {

  private String ssn;
  private String name;

  public String getSSN() {
      return ssn;
  }
  public void setSSN(String ssn) {
      this.ssn = ssn;
  }
  public String getName() {
      return name;
  }
  public void setName(String name) {
      this.name = name;
  }
}
```

Managing Relationships

- Expressed in Mapping Metadata as well
- Allows object graphs to be navigated – much more convenient than explicit Joins!



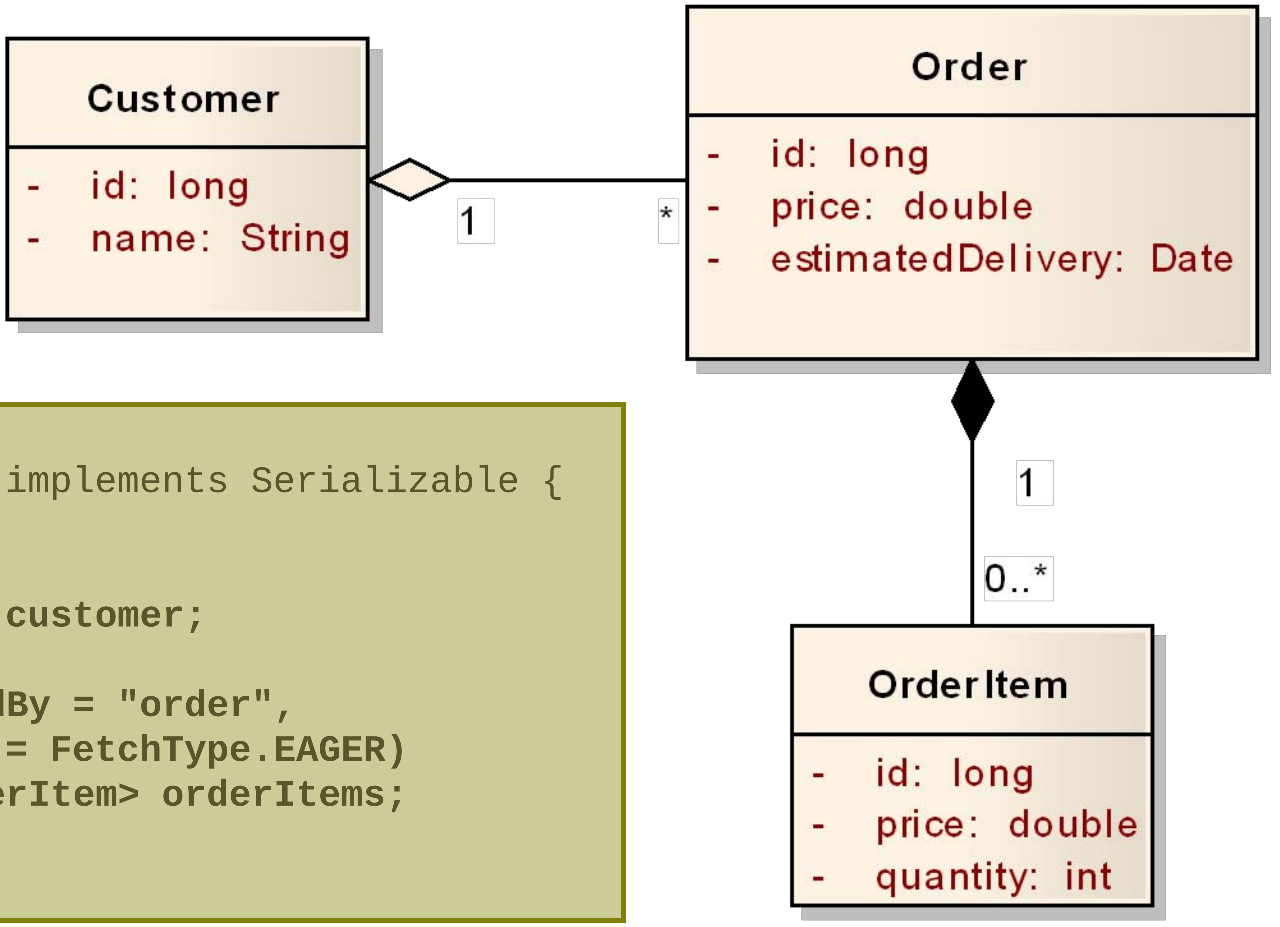
Managing Relationships: Mapping



```
@Entity
public class Customer implements Serializable {
  …
  @OneToMany(mappedBy = "customer")
  private List<Order> orders;
  …
}
```

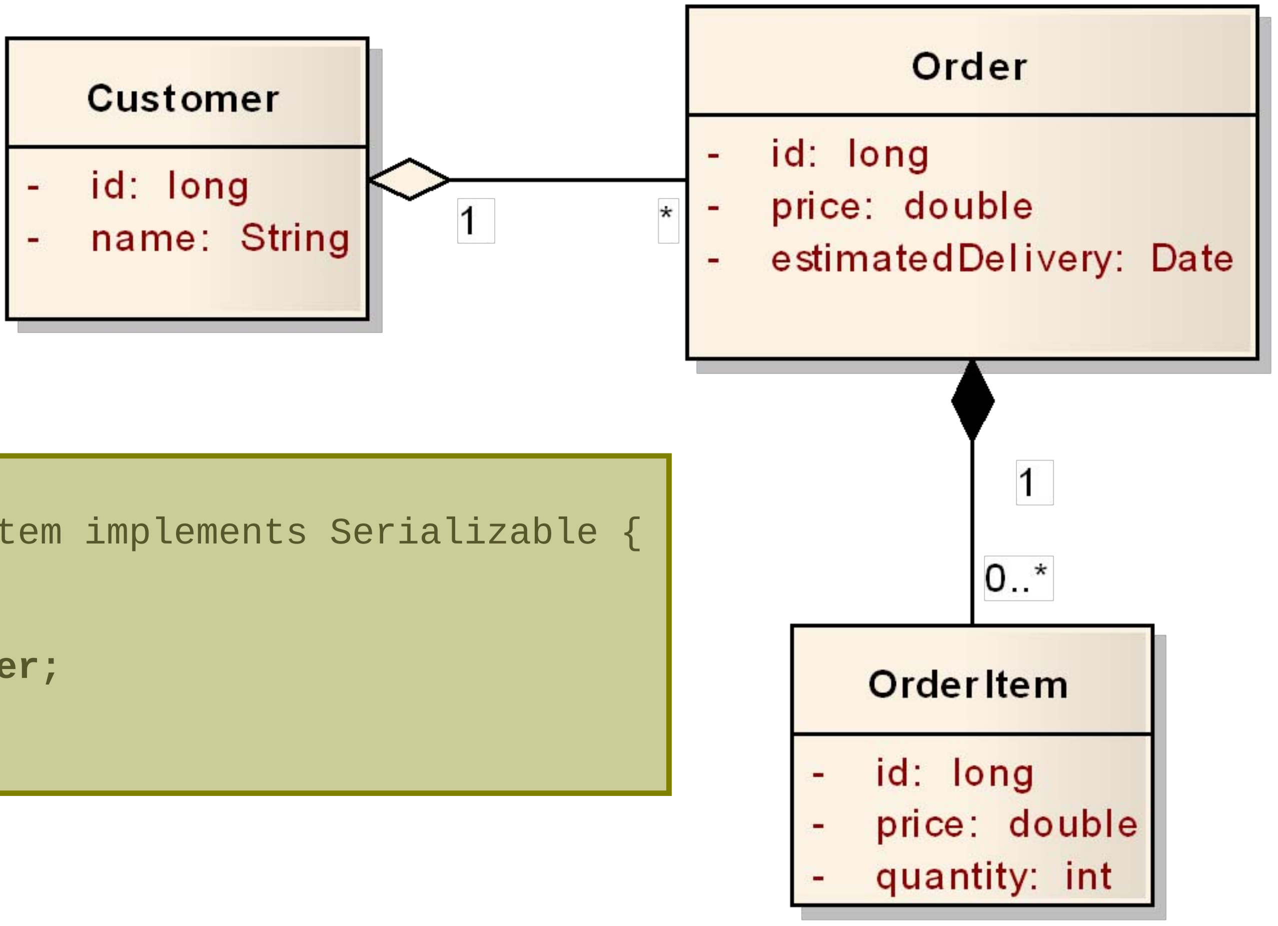
CALLISTA

Managing Relationships: Mapping (contd)



```
@Entity
public class Order implements Serializable {
  …
  @ManyToOne
  private Customer customer;

  @OneToMany(mappedBy = "order",
             fetch = FetchType.EAGER)
  private List<OrderItem> orderItems;
  …
}
```

Managing Relationships: Mapping (contd)



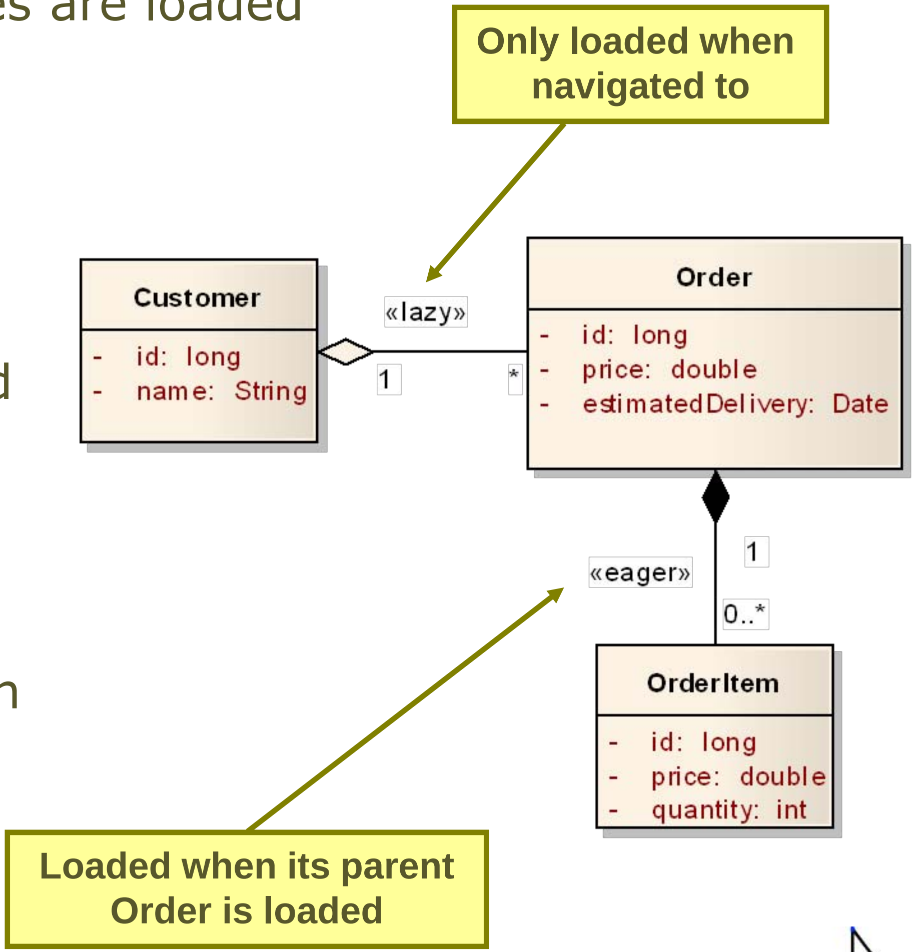
```
@Entity
public class OrderItem implements Serializable {
  …
  @ManyToOne
  private Order order;
  …
}
```

Relationships – Lazy and Eager loading

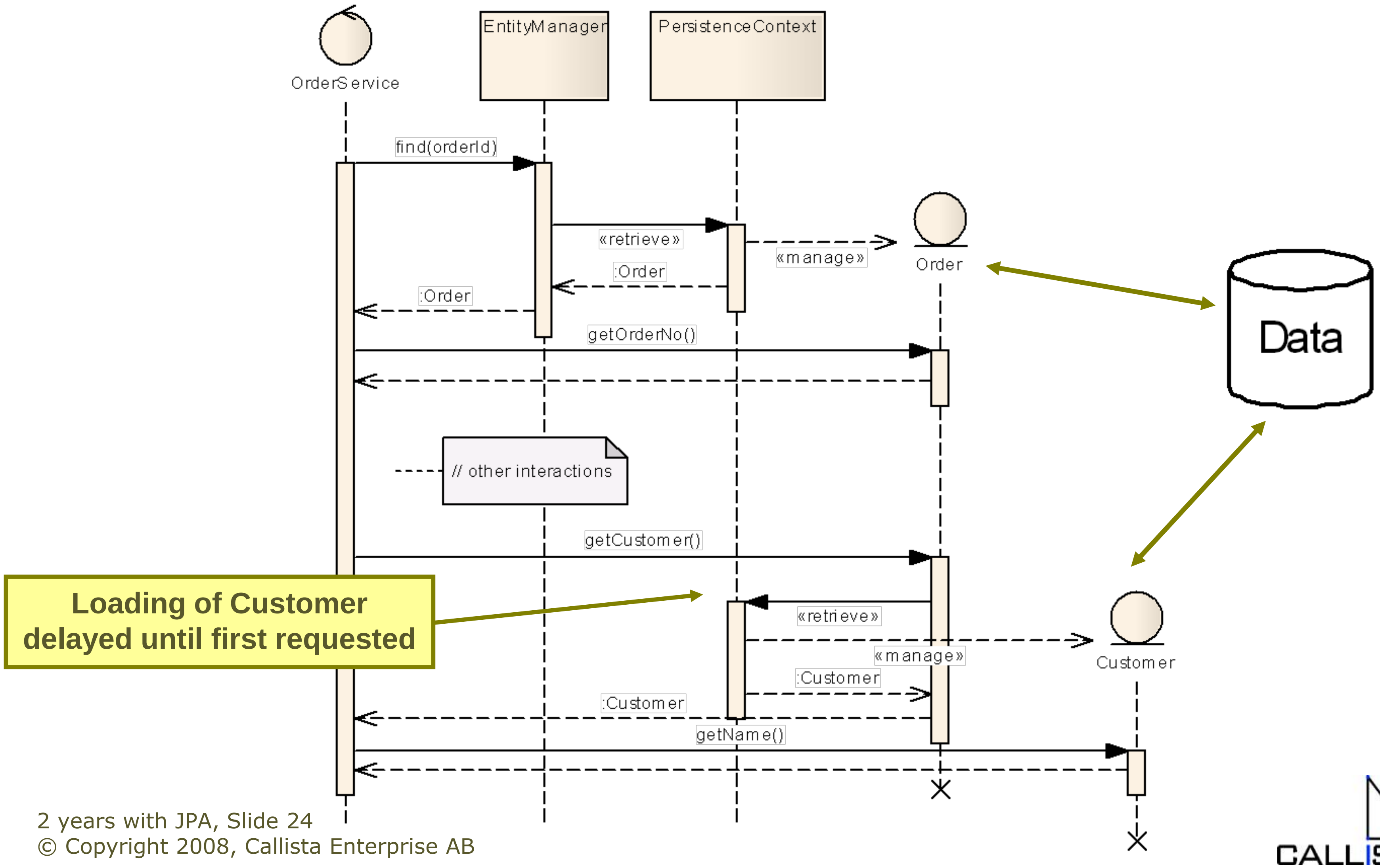
- Controlling when related entities are loaded
  - *Eager*

    Load related entities when the “parent” entity is loaded
  - *Lazy*

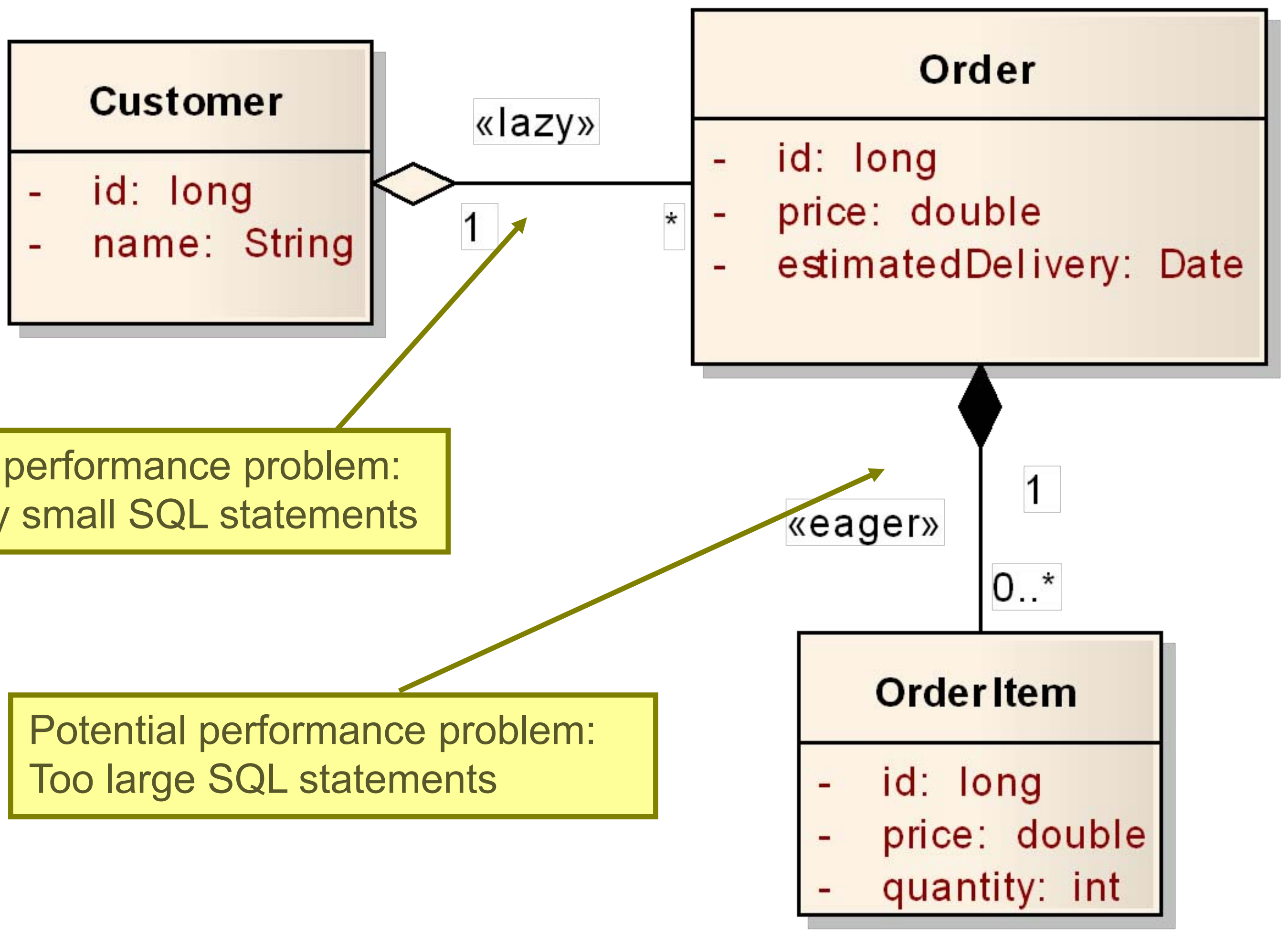
    Load related entities if/when they are navigated to



Example Lazy Loading scenario



Relationships – Lazy and Eager loading



Potential performance problem:
Too many small SQL statements

Potential performance problem:
Too large SQL statements

Static vs. Dynamic Lazy/Eager Loading

- Eager Loading is the default for OneToOne relations, whereas Lazy Loading is the default for OneToMany and ManyToMany relations
- Can be statically overridden using annotation attributes or DD metadata:

```
@Entity
public class Order implements Serializable {
   …
   @ManyToOne
   private Customer customer;

   @OneToMany(mappedBy = "order", fetch = FetchType.EAGER)
   private List<OrderItem> orderItems;
   …
}
```

- JPQL Queries can be used to dynamically change Lazy into Eager loading:

```
SELECT o FROM Order o JOIN FETCH o.customer WHERE ...
```

Experiences from the trenches

- Excellent developer productivity!
  - Order of magnitude more productive than JDBC
  - Much shorter start-up time compared to previous, model-based code generation approach
  - Simpler, less error-prone development tools
  - Portability across JPA vendors and DB vendors
- Adequate expressive power and flexibility of Mapping mechanism
  - Handles most cases
- Performance equal to or better than previous JDBC-based framework
  - Need to fine-tune balance between Lazy versus Eager loading

CALLISTA

The Bad – Not that transparent in reality …

- Lifecycle of Persistence Context governs detachment, which in turn affects
  - lazy loading
  - updates and merges
- Not that simple to foresee and comprehend!

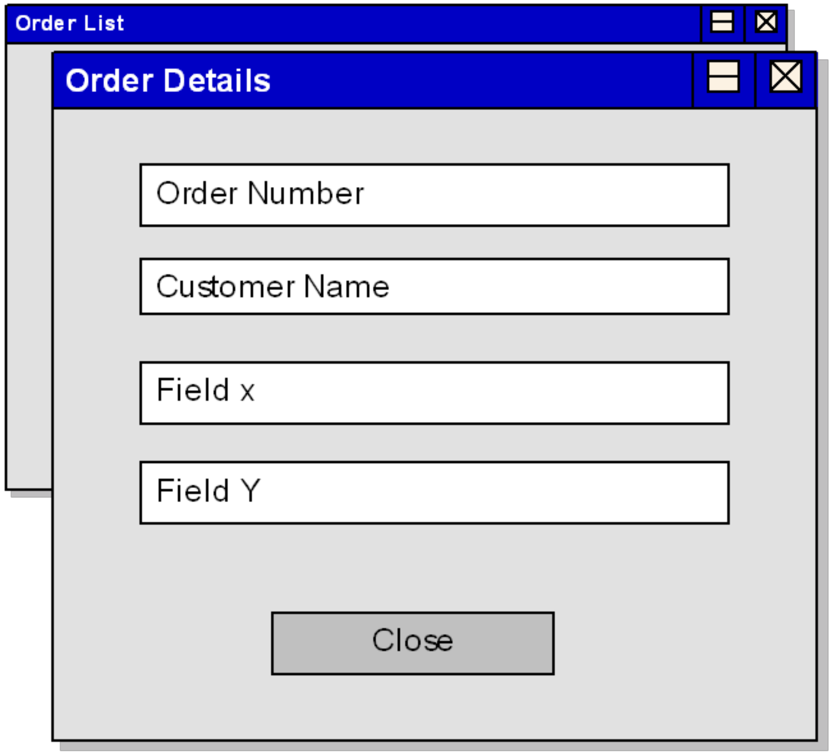
CALLISTA

Lazy Loading and Detachment

- Lazy loaded relations are fetched transparently when needed, *as long as the entity is managed*.
- When an entity has become detached, any lazy loaded relations *must not be accessed*.
- A *lazy load problem* occurs when trying to retrieve a lazily loaded related entity from a *detached* entity, i.e. when the persistence context is already closed.
- Lazy load problems are highly elusive and difficult to guard against!
  - Experience from the trenches: >75% of reported JPA-related defects due to Lazy Load problems

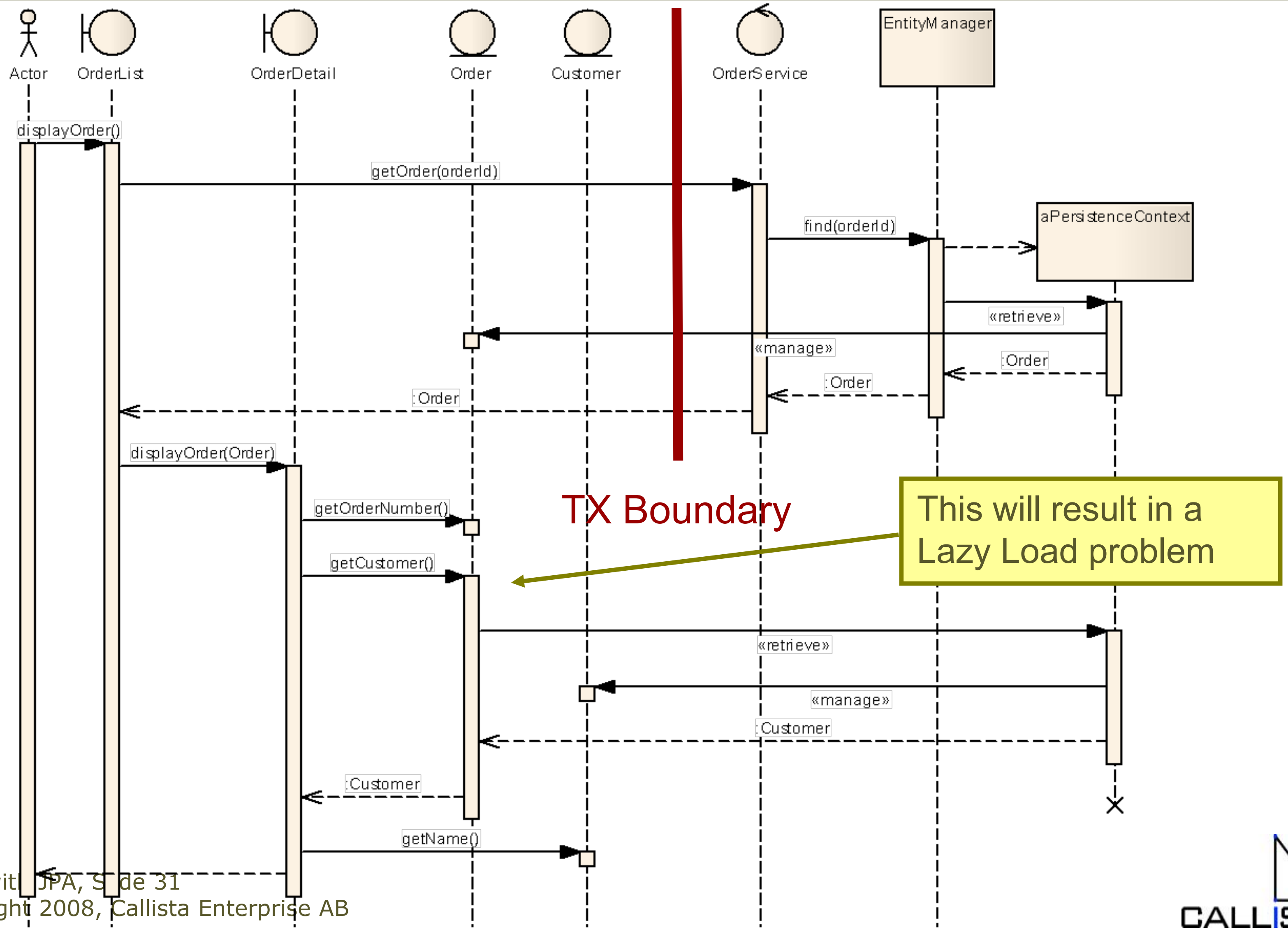
Typical Web Request-Response sequence

- **Scenario:** Web application +
  Container Managed and Transaction-Scoped Persistence Context



CALLISTA

Typical Web Request-Response sequence – Lazy Loading



CALLISTA

Preparing for Detachment

- Since a Transaction-Scoped Persistence Context is automatically closed at the transaction boundary, any entities that are passed out of the transaction boundary must be *prepared for detachment*:
  - All relations and/or attributes that is of interest to the consumer of the entity must be already loaded.
- Can be achieved by accessing the relation programmatically (i.e. calling the getter):

```
public Order getOrder(long orderId) {

    Order o = entityManager.find(orderIt,
                                 Order.class);
    ...
    o.getCustomer();
    return o;
}
```

- Or by using a **`fetch join`** JPQL query

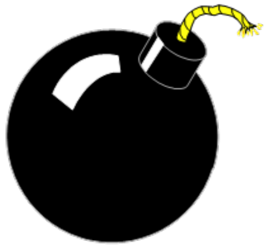
Preparing for Detachment will most likely affect your Service interface …

```
public interface OrderServices {

   public Order getOrder();

   public Order getOrderWithCustomer();

   public Order getOrderWithOrderItems();

   public Order getOrderWithCustomerAndOrderItems();

   public Order getOrderWithOrderItemsAndArticles();

    ...

}
```

CALLISTA

Detecting Lazy Load problems

- The JPA 1.0 specification does not clearly state how a lazy load problem should be handled and signalled by the JPA implementation
- Several different behavious have be observed between JPA providers:
  - No lazy problem occurs, because the JPA implementation implements the lazy relationship eagerly
  - No lazy problem occurs, because the JPA implementation fetches the related entities from the database even though the entity manager is closed
  - A vendor-specific exception is thrown



CALLISTA

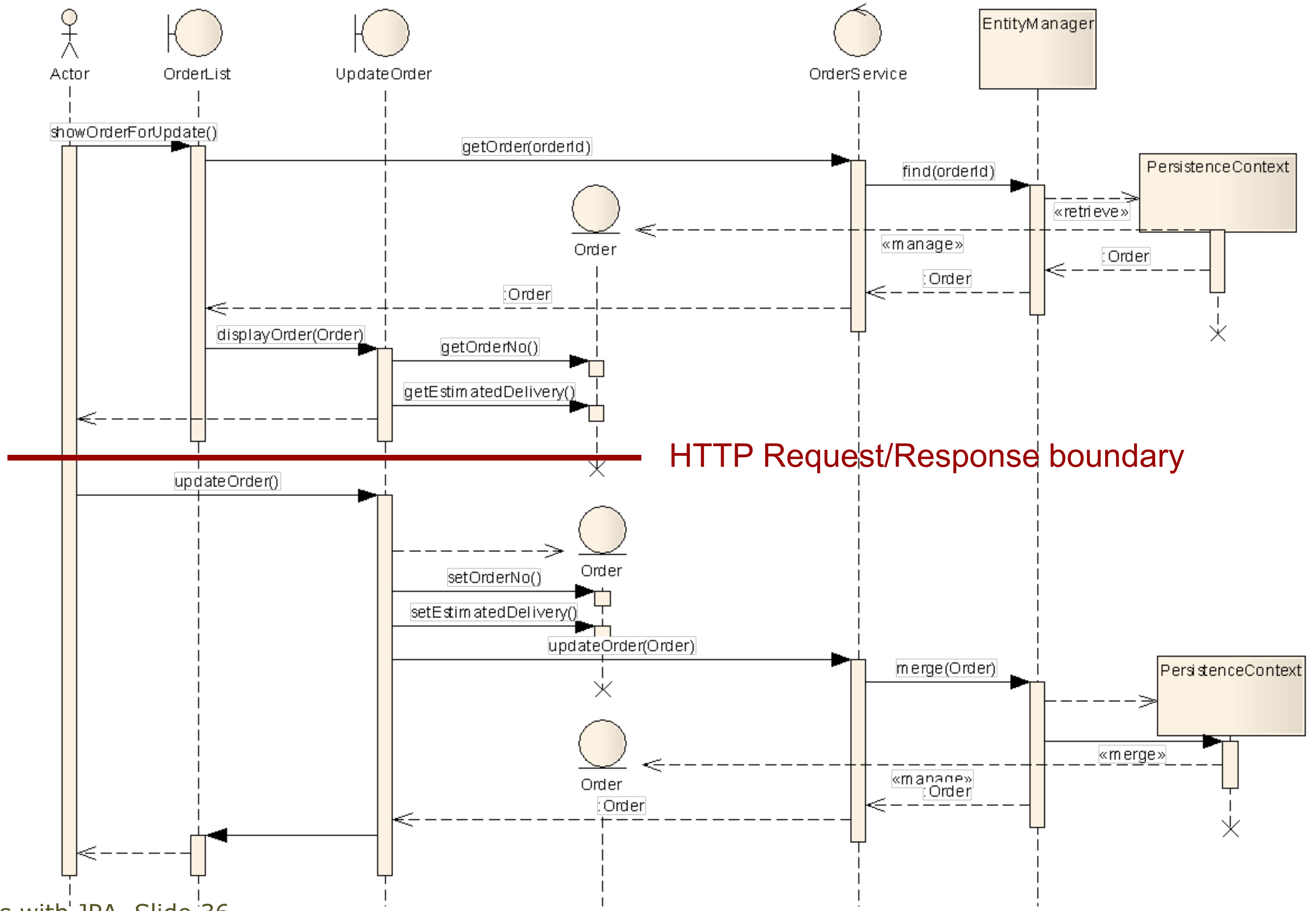
Updating Entities in a Stateless Web setting

- **Scenario:** Web application +
  Container Managed and Transaction-Scoped Persistence Context

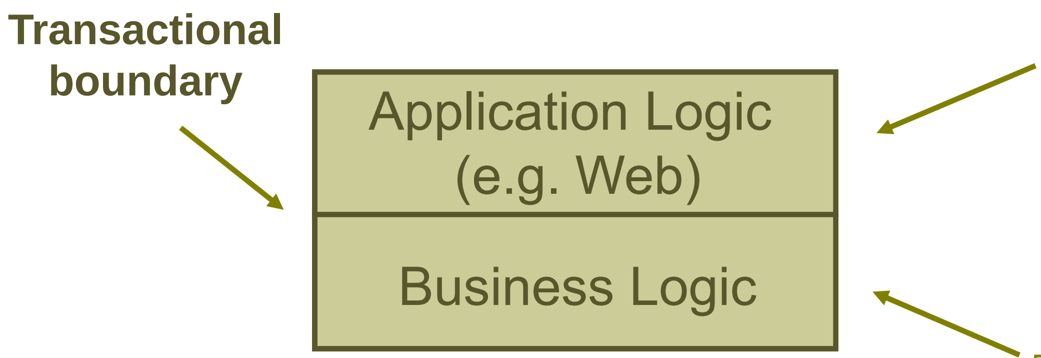
Update Order
OrderNo
Estimated Delivery
Update
Cancel

CALLISTA

Naive Approach: Merge “new” entity



Naive Approach: Sample code

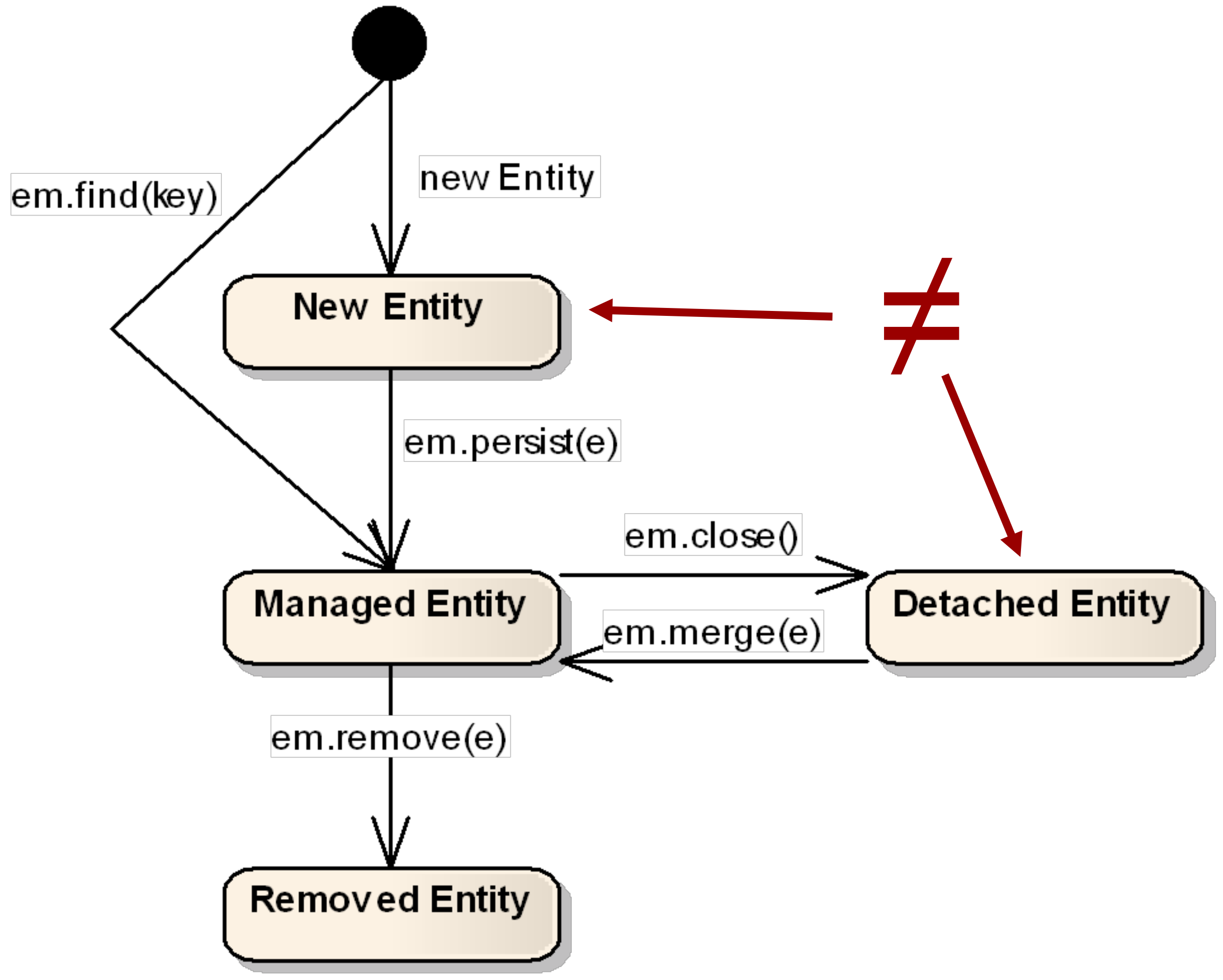


```
Order o = new Order();
o.setOrderNo(orderNo)
o.setVersion(version);
o.setEstimatedDelivery(delivery);
...
o = service.updateOrder(o);
```

```
public Order updateOrder(Order o) {
  return em.merge(o);
}
```

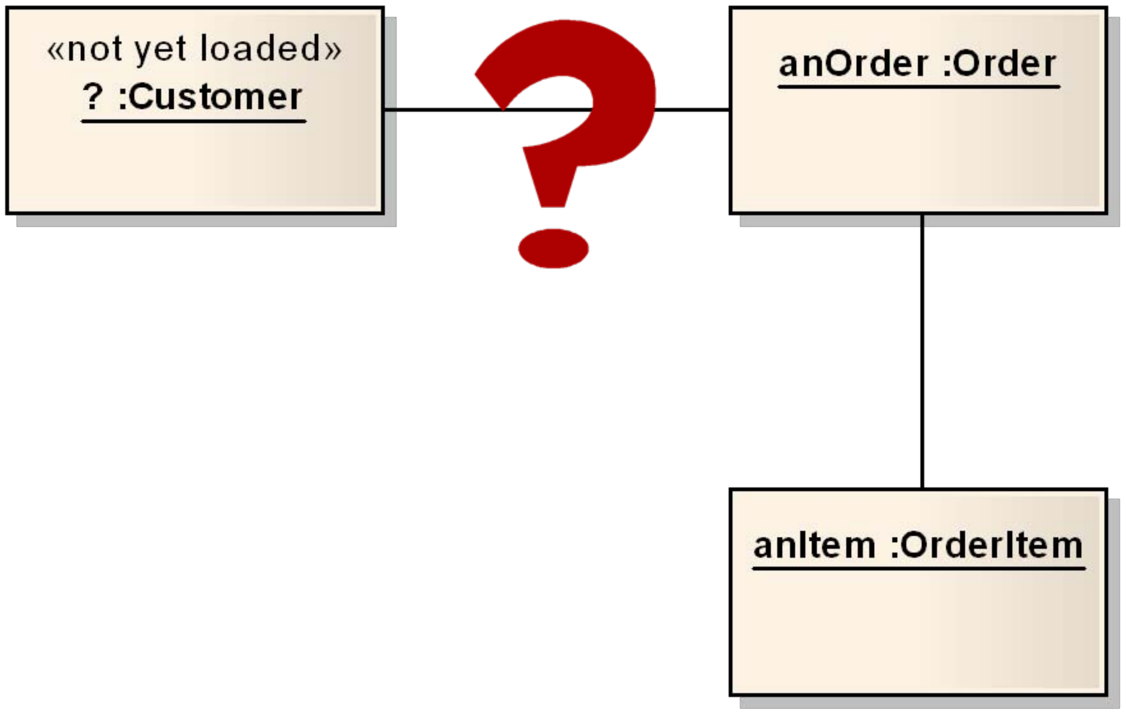
- Requires setting of all attributes
- Poorly documented behavior in spec
- You’re up for a nasty surprise!

Remember the difference between New and Detached Entities?

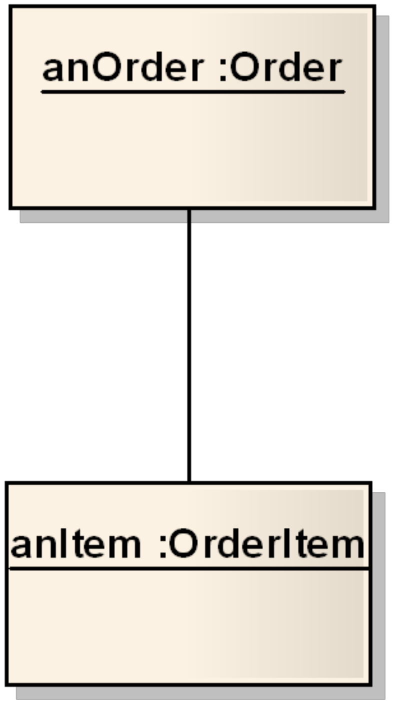


CALLISTA

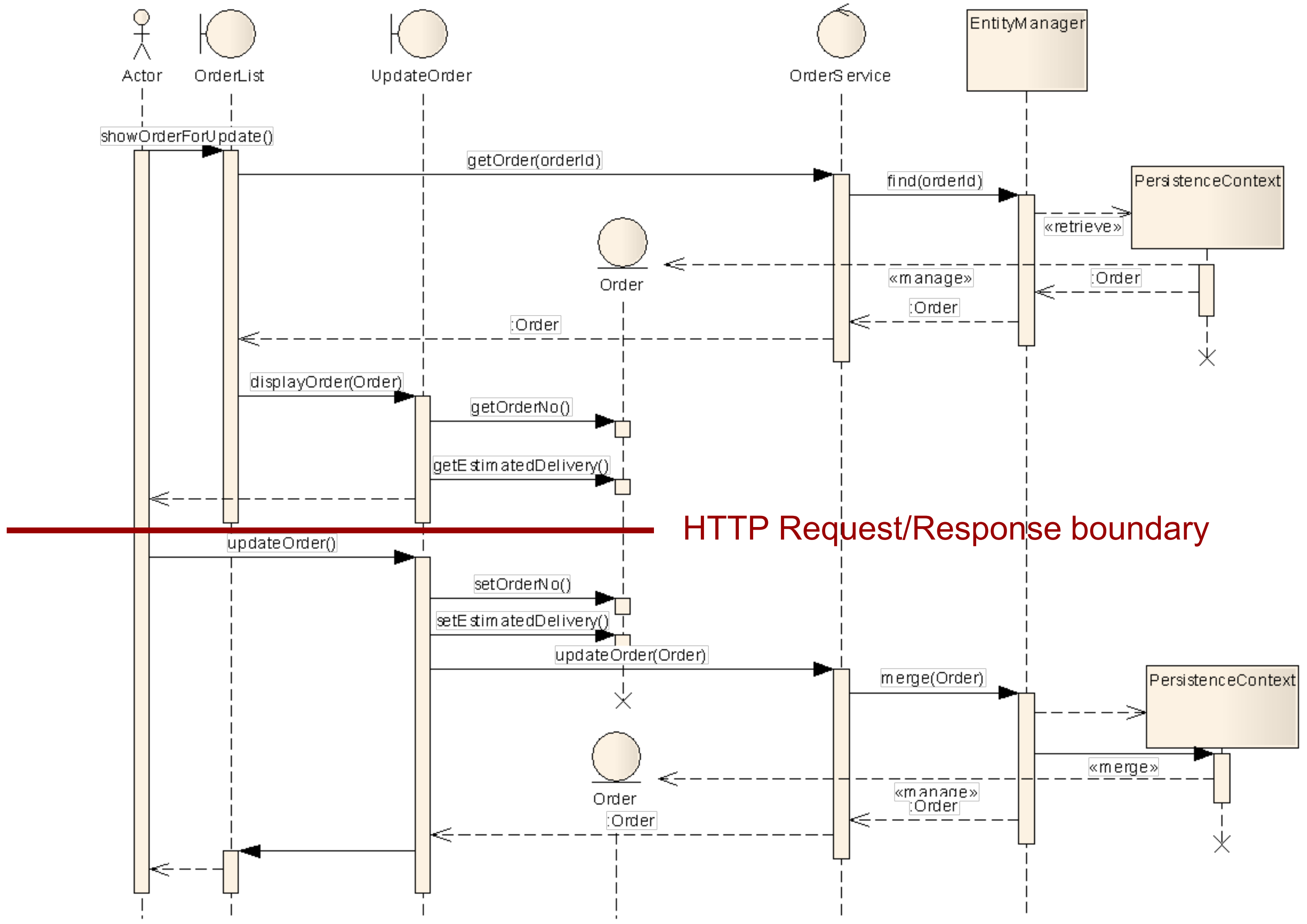
A subtle difference between …



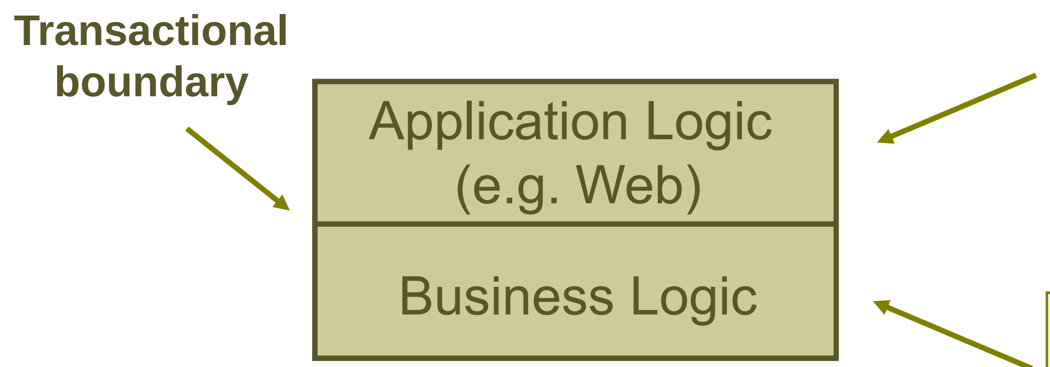
… and



Alternative #1: update and merge detached entity



Alternative #1: Sample code



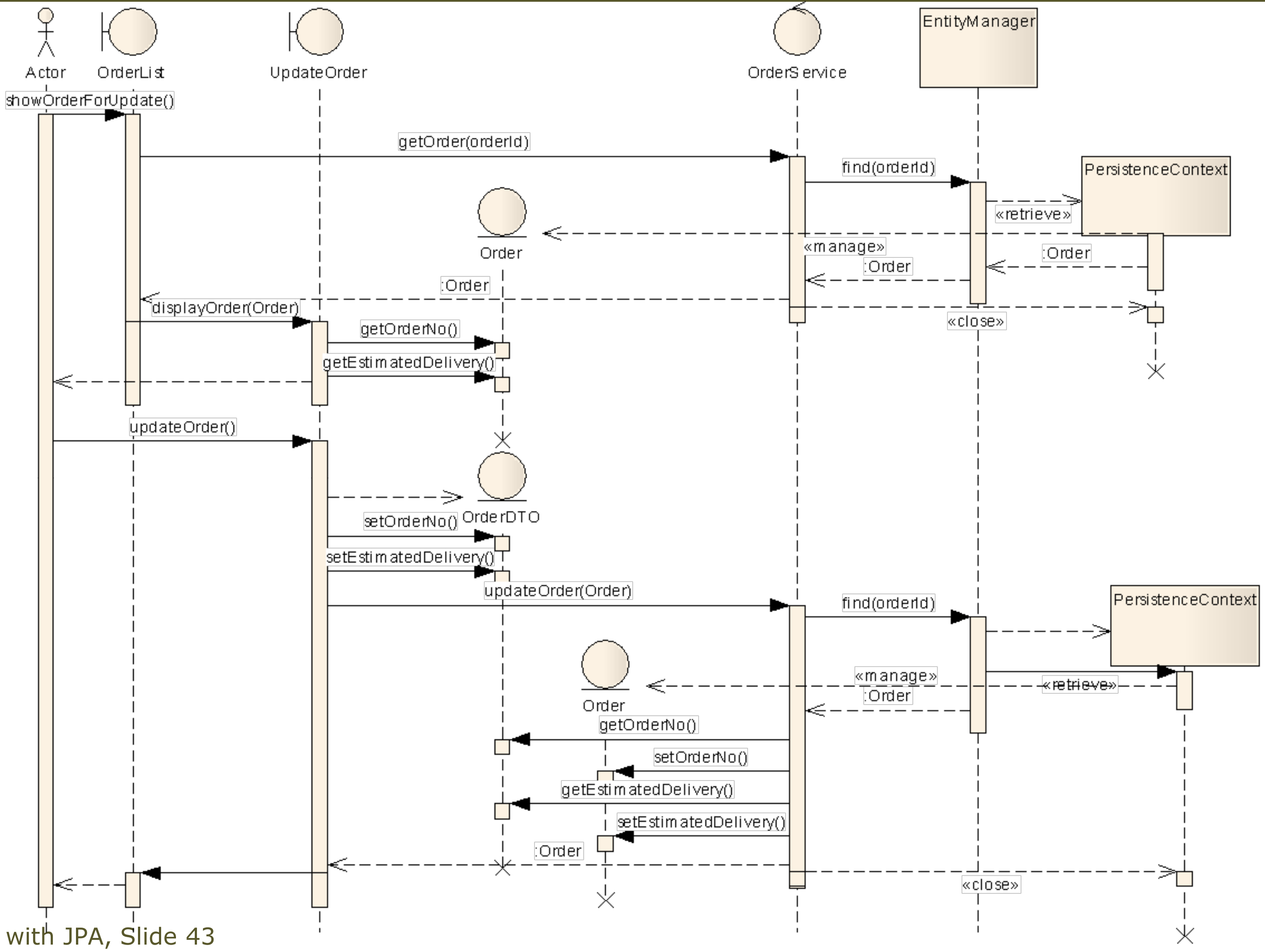
```
Order o = (Order)session
  .getAttribute("Order");
o.setEstimatedDelivery(delivery);
o = service.updateOrder(o);
```

```
public Order updateOrder(Order o) {
  return em.merge(o);
}
```

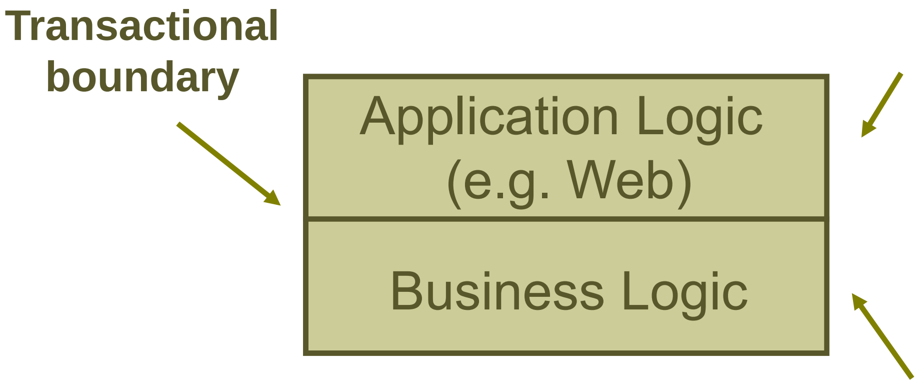
Requires explicit use of the HTTP Session

- HTTP Session must be correctly initialized from the previous page
- The detached entity must be removed explicit from the HTTP Session
- How large can a serialized detached entity become?

Alternative #2: Re-read and update



Alternative #2: Sample code



```
Order o = new Order();
o.setOrderNo(orderNo)
o.setEstimatedDelivery(delivery);
...
o = service.updateOrder(o);
```

```
public Order updateOrder(Order o) {
  Order o = service.find(o.getOrderNo(),
                         Order.class);
  o.setEstimatedDelivery(delivery);
  return o;
}
```

- Requires extra SQL SELECT round-trip
- Requires setting of all attributes in both application and business logic

CALLISTA

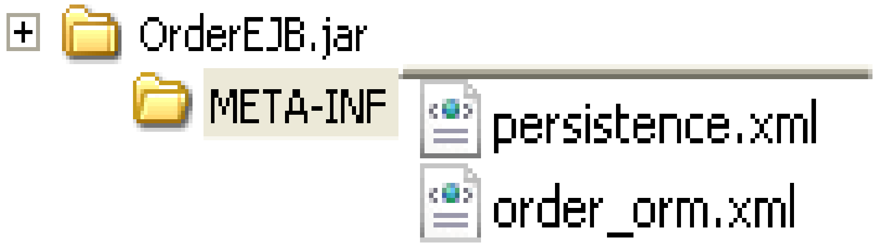
The Ugly

- The rigid JPA Deployment and Packaging model prevents testability
- Does not blend well with JavaEE Naming and separation of concerns between Developer and Deployer

CALLISTA

Persistence Archives and Persistence Units

- Unit of Persistence Packaging and Deployment
- Configuration for an EntityManagerFactory
- Object Relational Mapping Info
  - Annotation-based or Deployment Descriptor based
- Persistence ARchive: JAR archive containing a persistence.xml file placed in the META-INF folder



Persistence.xml and Pluggability

- The persistence.xml deployment descriptor specifies physical information, which cannot easily be changed in different contexts (e.g. in Unit Tests)

```
<persistence-unit name="OrderPU" transaction-type="RESOURCE_LOCAL">
  ...
  <properties>
    ...
    <property name="hibernate.connection.driver_class"
              value="org.apache.derby.jdbc.ClientDriver" />
    <property name="hibernate.connection.url"
              value="jdbc:derby://localhost:1527/Jee5TestDb_HB" />
    <property name="hibernate.connection.username"
              value="APP_HB" />
    <property name="hibernate.connection.password" value="APP" />
    <property name="hibernate.hbm2ddl.auto" value="update" />
    ...
  </properties>
</persistence-unit>
```

Custom Framework Solutions

- Gap filled by additional frameworks
  - Spring provides a custom JPA bootstrapping mechanism to allow configuration of Persistence Units
- Current project solution:
  - Specify multiple PersistenceUnits, choose which one to use based on runtime configuration

```
<!-- In-container persistence unit -->
   <persistence-unit name="OrderPU" transaction-type="JTA">
        ...
   </persistence-unit>

   <!-- Out-of-container Tests persistence unit -->
   <persistence-unit name="OrderPU_TEST" transaction-type="RESOURCE_LOCAL">
        ...
   </persistence-unit>
```

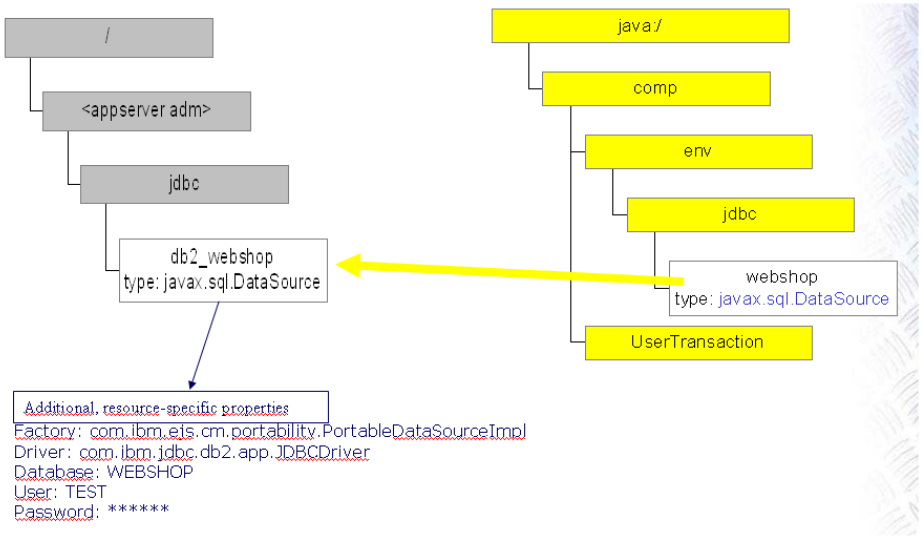
What does <jta-datasource> mean?

```
<persistence-unit name="PosPU" transaction-type="JTA">
  ...
  <jta-data-source>jdbc/PosDB</jta-data-source>
  ...
</persistence-unit>
```

- JNDI lookup string for Datasource?
- It’s not specified!

Naming: JavaEE Component Scope

- JavaEE requires the app server to support a ”logical” naming tree visible to components of the same enterprise application
  - Logical names are referenced through a standardized virtual sub-context: ”**java:comp/env**”
- The deployer maps the logical name to an external name visible to all clients of the network


/
<appserver adm>
jdbc
db2_webshop
type: javax.sql.DataSource
Additional, resource-specific properties
Factory: com.ibm.ejs.cm.portability.PortableDataSourceImpl
Driver: com.ibm.jdbc.db2.app.JDBCDriver
Database: WEBSHOP
User: TEST
Password: ******
java:/
comp
env
jdbc
webshop
type: javax.sql.DataSource
UserTransaction

Hence we should use a logical name in persistence.xml as well?

```
<persistence-unit name="PosPU" transaction-type="JTA">
  ...
  <jta-data-source>java:comp/env/jdbc/PosDB</jta-data-source>
  ...
</persistence-unit>
```

- But where should we place the resource ref?
  - The JPA entities are not JavaEE Components, hence they have no associated Component Scope
  - No standardized Application Scope exists (even though some App Server vendors allow configuration of resources in their proprietary deployment descriptors)
- On one of the components that uses the Persistence Unit?

Using the @Resource attribute

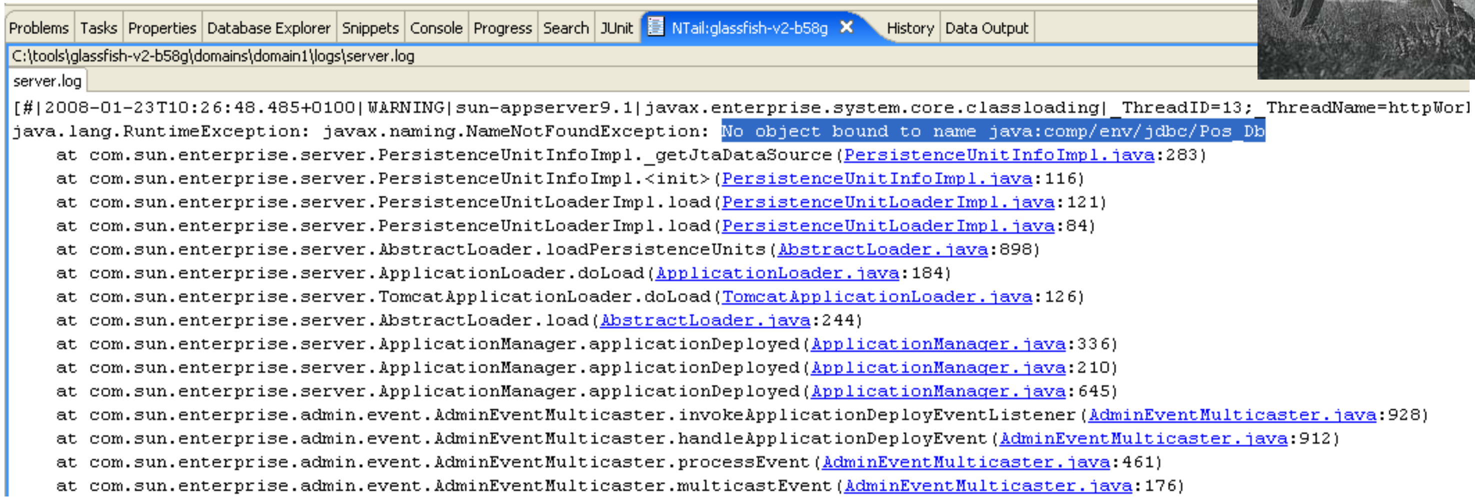
```
@Resource(name="jdbc/PosDB", mappedName="jdbc/PosDB_v1")
@Stateless
public class OrderServicesBean implements OrderServices {

  private EntityManager em = null;

  @PersistenceContext(unitName="PosPU")
  public void setEntityManager(EntityManager em) {
    this.em = em;
  }
}
```

But what if ... ?

```
@Resource(name="jdbc/PosDB", mappedName="jdbc/PosDB_v2")
@Stateless
public class CustomerServicesBean
  implements CustomerServices {

  private EntityManager em = null;

  @PersistenceContext(unitName="PosPU")
  public void setEntityManager(EntityManager em) {
    this.em = em;
  }
}
```

CALLISTA

And besides, it doesn’t work

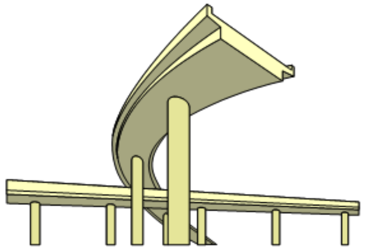


```
[#|2008-01-23T10:26:48.485+0100|WARNING|sun-appserver9.1|javax.enterprise.system.core.classloading|_ThreadID=13;_ThreadName=httpWorl
java.lang.RuntimeException: javax.naming.NameNotFoundException: No object bound to name java:comp/env/jdbc/Pos_Db
    at com.sun.enterprise.server.PersistenceUnitInfoImpl._getJtaDataSource(PersistenceUnitInfoImpl.java:283)
    at com.sun.enterprise.server.PersistenceUnitInfoImpl.<init>(PersistenceUnitInfoImpl.java:116)
    at com.sun.enterprise.server.PersistenceUnitLoaderImpl.load(PersistenceUnitLoaderImpl.java:121)
    at com.sun.enterprise.server.PersistenceUnitLoaderImpl.load(PersistenceUnitLoaderImpl.java:84)
    at com.sun.enterprise.server.AbstractLoader.loadPersistenceUnits(AbstractLoader.java:898)
    at com.sun.enterprise.server.ApplicationLoader.doLoad(ApplicationLoader.java:184)
    at com.sun.enterprise.server.TomcatApplicationLoader.doLoad(TomcatApplicationLoader.java:126)
    at com.sun.enterprise.server.AbstractLoader.load(AbstractLoader.java:244)
    at com.sun.enterprise.server.ApplicationManager.applicationDeployed(ApplicationManager.java:336)
    at com.sun.enterprise.server.ApplicationManager.applicationDeployed(ApplicationManager.java:210)
    at com.sun.enterprise.server.ApplicationManager.applicationDeployed(ApplicationManager.java:645)
    at com.sun.enterprise.admin.event.AdminEventMulticaster.invokeApplicationDeployEventListener(AdminEventMulticaster.java:928)
    at com.sun.enterprise.admin.event.AdminEventMulticaster.handleApplicationDeployEvent(AdminEventMulticaster.java:912)
    at com.sun.enterprise.admin.event.AdminEventMulticaster.processEvent(AdminEventMulticaster.java:461)
    at com.sun.enterprise.admin.event.AdminEventMulticaster.multicastEvent(AdminEventMulticaster.java:176)
```

- Currently no portable way exist to use a logical JNDI name in Persistence.xml
- Requires build-time manipulation of EAR and EJB-JAR files to provide physical details

To conclude

- JPA´s Transparent Persistency is a giant step forward ✓
  - Powerful
  - Good productivity
- But as (probably) any abstraction, JPA tends to leak ✗
  - Understanding the Persistence Context is critical
  - May rapidly affect the productivity equation
- Still immature in some aspects ✗
  - Issues with configuration pluggability (e.g. for testing)
  - Issues with JavaEE integration
- Looking ahead …



JPA 2.0

- JSR 317 Expert Group formed in mid 2007 under the lead of Linda DeMichiel
- First Public Review scheduled for Q2 2008
- Final Release scheduled for Q4 2008, with Reference Implementation late Q2 2009

JPA 2.0 Scope

- Expanded object/relational mapping functionality
- Additions to the Java Persistence query language
  - An API for "criteria" queries
- **Standardization of sets of "hints" for query configuration and for entity manager configuration**
- **Standardization of additional metadata to support DDL generation**
- Expanded pluggability contracts to support efficient passivation and replication of extended persistence contexts in Java EE environments
- **Standardization of additional contracts for entity detachment and merge, and persistence context management**
- Better support for validation

Time (?) for Questions!