Transaction management in Spring/EJB3

Guido Anselmi

Argonet Srl

supergum@gmail.com

Agenda

- Transaction definitions
- Choosing appropriate isolation level
- Optimistic locking and lost update
- Transaction models
- Local transaction Model
- Programmatic transaction Model in Spring and EJB3
- Declarative transaction Model in Spring and EJB3
- Enterprise transaction patterns

Choosing Isolation Level

The ANSI SQL defines four differents Isolation levels

Phenomena / Isolation level	Dirty Reads	Non-repeatable Reads	Phantom Reads
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable Read	N	N	Y
Serializable	N	N	N

Dirty reads

Dirty read occurs when a transaction reads data from a row that has been modified by another transaction, but not yet committed.

Transaction 1	Transaction 2
SELECT stipendio FROM impiegato WHERE id = 1;	
/* java code */	UPDATE impiegato set stipendio = stipendio * 1,2 WHERE id = 1;
SELECT stipendio FROM impiegato WHERE id = 1;	
generaBustaPaga(rst.getFloat("stipendio"));	ROLLBACK;

Non-repeatable reads

Non-repeatable reads happen when a query returns data that would be different if the query were repeated within the same transaction. Non-repeatable reads can occur when other transactions are modifying data that a transaction is reading.

Transaction 1	Transaction 2
SELECT stipendio FROM impiegato WHERE id = 1;	
	UPDATE impiegato set stipendio = stipendio * 1,2 WHERE id = 1; **COMMIT;**
SELECT stipendio FROM impiegato WHERE id = 1;	

Phantom reads

Phantom reads can occur when other transactions insert rows that would satisfy the WHERE clause of another transaction's statement..

Transaction 1	Transaction 2
SELECT * FROM impiegato WHERE cognome = 'ROSSI';	
	INSERT INTO impiegato (cognome,....) VALUES('ROSSI',.....); **COMMIT;**
SELECT * FROM impiegato WHERE cognome = 'ROSSI';	

Choosing Isolation Level
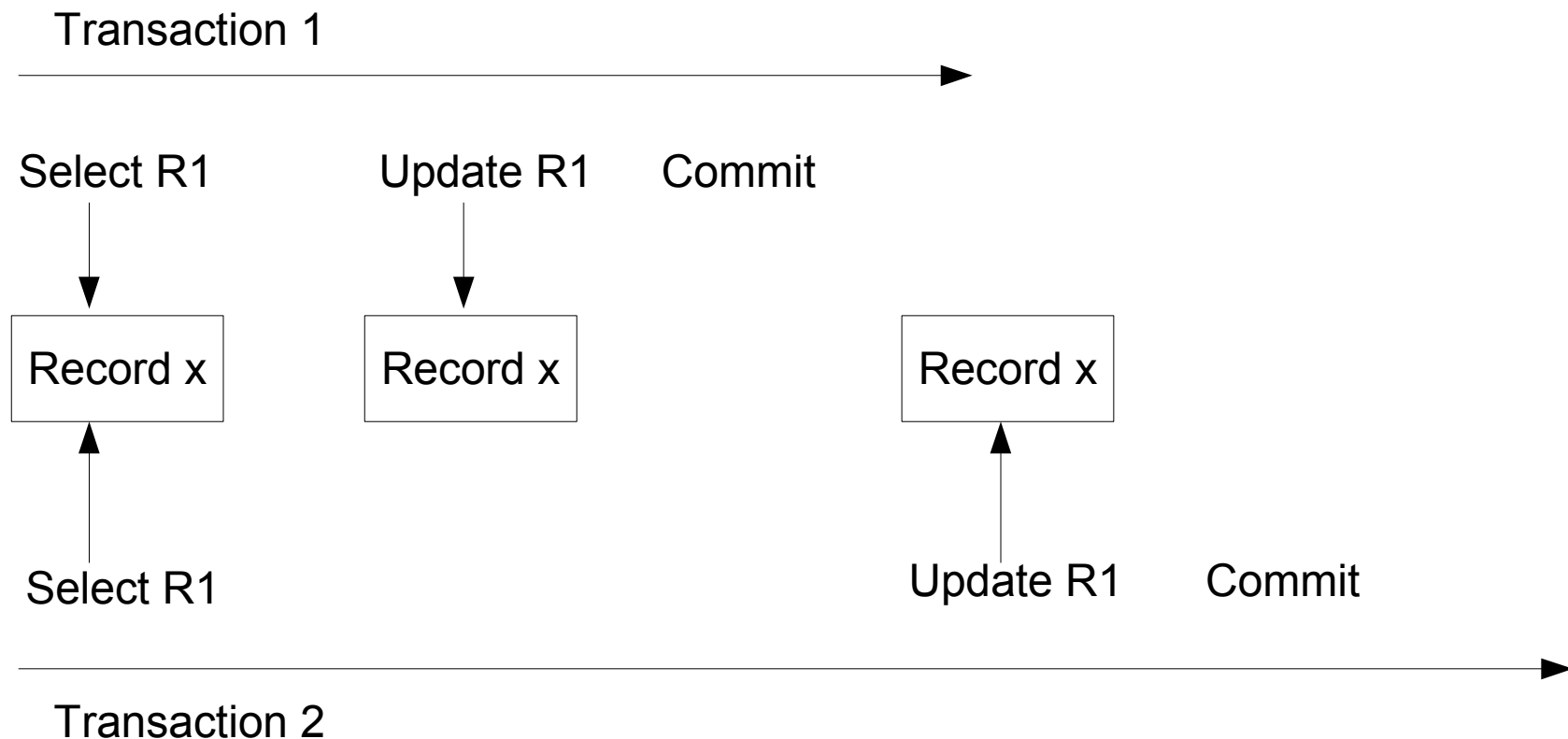
The ANSI SQL defines four differents Isolation levels

Phenomena / Isolation level	Dirty Reads	Non-repeatable Reads	Phantom Reads
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable Read	N	N	Y
Serializable	N	N	N

Optimistic locking and Lost update

Another different problem in concurrent data access is known as *Lost Update*
To understand this kind of problem, imagine that two transactions read the same record from the database, and both modify it.
Thanks to the read-committed isolation level of the database connection, neither transaction will run into dirty reads.

Transaction 1

Select R1   Update R1   Commit

| Record x | | Record x | | Record x |

Select R1   Update R1   Commit

Transaction 2

Optimistic locking and Lost update

The changes made in transaction 1have been lost, and (potentially worse) modifications of data committed in transaction 2 may have been based on stale information.
We have basically two choices for how to deal with lost updates in these second transactions

Last commit wins—Both transactions commit successfully, and the second commit overwrites the changes of the first. No error message is shown.

First commit wins—The first transaction is committed, and the user committing the transaction in second transaction gets an error message.

An optimistic approach always assumes that everything will be OK and that conflicting data modifications are rare. Optimistic concurrency control raises an error only at the end of a unit of work, when data is written.

Optimistic locking and Lost update

Enabling optimistic locking in JPA/Hibernate is very easy.

Hibernate provides automatic versioning. Each entity instance has a version, which can be a number or a timestamp. Hibernate increments an object's version when it's modified, compares versions automatically, and throws an exception if a conflict is detected.

In JPA syntax, this can be done as follows:

```
@Entity
public class Utente {
...
        @Version
        @Column(name = "USR_VERSION")
        private int version;
        ...
}
```

Local transaction model

An example from jdbc era is enough...

```java
public void updateUser(User user) throws Exception {
    DataSource ds = (DataSource) (new InitialContext()).lookup("jdbc/DS");
    Connection conn = ds.getConnection();
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    String sql = "update user ... ";
    try {
        stmt.executeUpdate(sql);
        conn.commit();
    } catch (Exception e) {
        conn.rollback();
        throw e;
    } finally {
        stmt.close();
        conn.close();
    }
}
```

Programmatic transaction model

The Programmatic transaction model is similar to local transaction model, with one BIG difference:

the developer manages transactions, not connections.

In this approach, the developer is responsible for starting and terminating the transaction.

In EJB this involves **JTA** and **UserTransaction**.

In Spring we have different possibles techniques, using **PlatformTransactionManager** or **TransactionTemplate**.

Programmatic transaction model EJB3

In EJB3, programmatic transaction model, or bean managed transaction, is not the default.
We must explicitly tell the container we want to use it.

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class UserServiceBean implements UserService
{
        @Resource
        Private SessionContext context;

        .........................
}
```

In a such EJB, we can (must) define transaction boundaries by code.

Programmatic transaction model EJB3

This is the tipycal approach:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class UserServiceBean implements UserService
{

    @Resource
    private SessionContext context;


    public void updateUserSalary(User user, Float salary) throws Exception {
        UserTransaction txn = sessionCtx.getUserTransaction();
        txn.begin();
        try {
            dao.updateUsersalary(user, salary);
             txn.commit();
        } catch (CheckedApplicationException e) {
            txn.rollback();
            throw e;
        }
    }
}
```

Programmatic transaction model EJB3

Checked Exception must be managed carefully.
A common mistake is to forget to handle them.

```
public void updateUserSalary(User user, Float salary) throws Exception {
    UserTransaction txn = sessionCtx.getUserTransaction();
    txn.begin();
    dao.updateUsersalary(user, salary);
    txn.commit();
}
```

If we get a checked exception in the code above, the following exception would be returned:

java.lang.Exception: [EJB:011063] Stateless session beans with bean-managed transactions must commit or rollback their transactions before completing a business method.

Programmatic transaction model EJB3

Another frequent mistake is to forget to commit the transaction

```
public void updateUserSalary(User user, Float salary) throws Exception {
    UserTransaction txn = sessionCtx.getUserTransaction();
    txn.begin();
    try {
        dao.updateUsersalary(user, salary);

    } catch (CheckedApplicationException e) {
        txn.rollback();
        throw e;
    }
}
```

Runnig this method, we would get a Runtime Exception

java.lang.Exception: [EJB:011063] Stateless session beans with bean-managed transactions must commit or rollback their transactions before completing a business method.

Programmatic transaction model Spring

Spring abstracts a general set of transaction facilities from different transaction management APIs.

It's possible ti utilize Spring's transaction facilities without having to know much about the underlying transaction APIs.

The transaction management code will be independent of any specific transaction technology.

Spring's core transaction management abstraction is PlatformTransactionManager It provides provides three methods for working with transactions:

- TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException

- void commit(TransactionStatus status) throws TransactionException;

- void rollback(TransactionStatus status) throws TransactionException;

Programmatic transaction model Spring
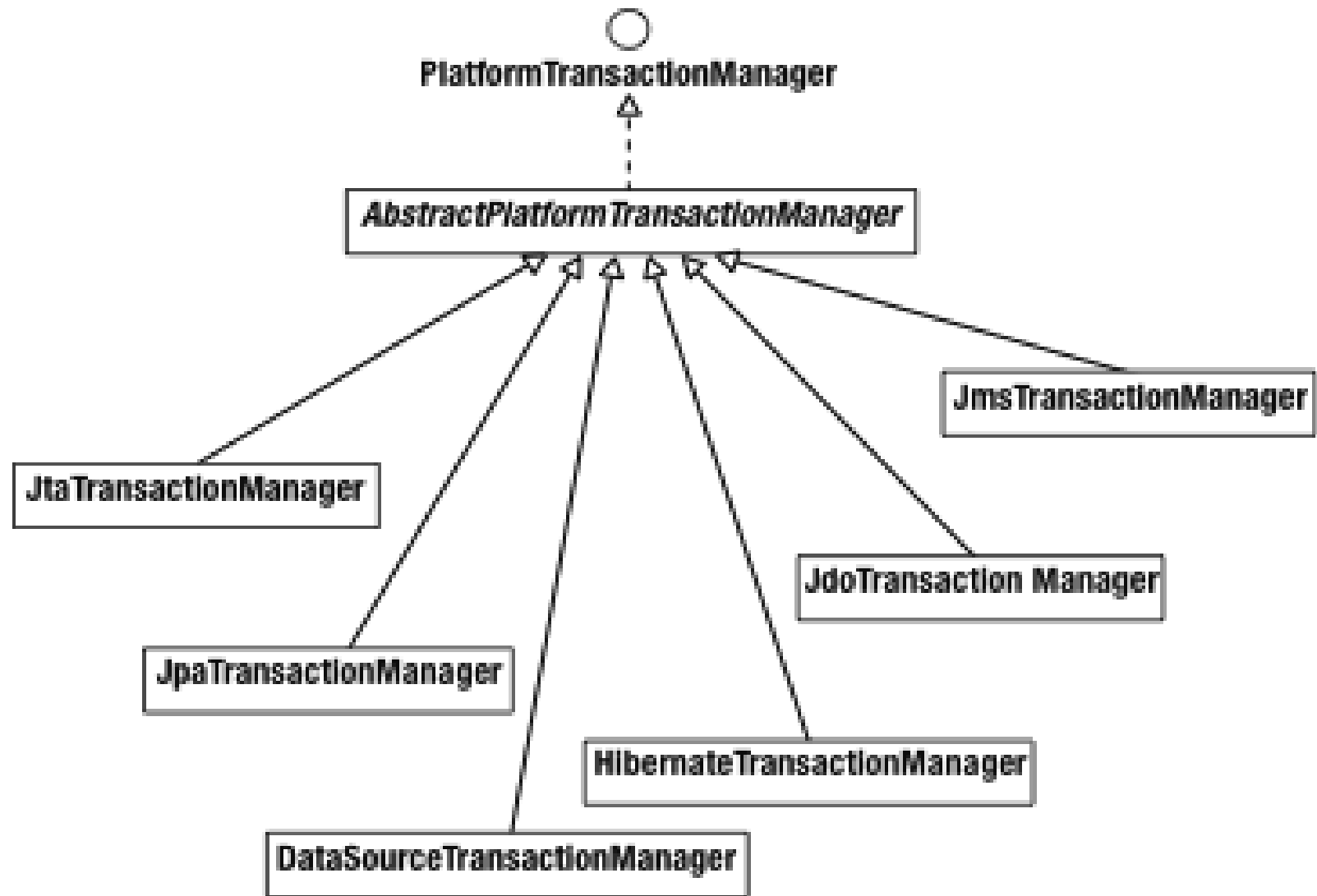
Programmatic transaction model Spring

Depending on the underliyng data access technology, we have to choose an appropriate implementation.

For example:

```xml
<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

And the code may use the transaction manager, without to worry about JPA.

```java
TransactionDefinition def = new DefaultTransactionDefinition();
TransactionStatus status = transactionManager.getTransaction(def);
try {
    .............
    userDao.save(utente);
    transactionManager.commit(status);
} catch (DataAccessException de) {
    transactionManager.rollback(status);
}
```

This DAO may use JPATemplate to simplify EntityManager management and to obtain an unified exception hierarchy

Programmatic transaction model Spring

In Spring some boilerplate operations in programmatic transaction management can be delegated to a transactionTemplate.

```java
public void saveUtente(Utente utente) {
    TransactionDefinition def = new DefaultTransactionDefinition();
    TransactionStatus status = transactionManager.getTransaction(def);
    try {
        jpaTemplate.merge(utente);
        transactionManager.commit(status);
    } catch (DataAccessException de) {
        transactionManager.rollback(status);
    }
}
```

Programmatic transaction model

There is a serious architectural limitation in the Programmatic Transaction Model.

We cannot pass a transaction context from one bean using programmatic transactions into another bean using programmatic transactions.

Consider the scenario where we have two Stateless Session-Beans, EmployeeService and DeptService.

The EmployeeService bean handles all functionality related to an employee (such as hiring an employee or giving an employee a raise). The DeptService bean handles all functionality relating to departments (adding, removing, and assigning employees).

Both of these EJBs use programmatic transactions. Furthermore, assume both of these EJBs have corresponding DAOs.

Programmatic transaction model

The following figure illustrates this scenario:
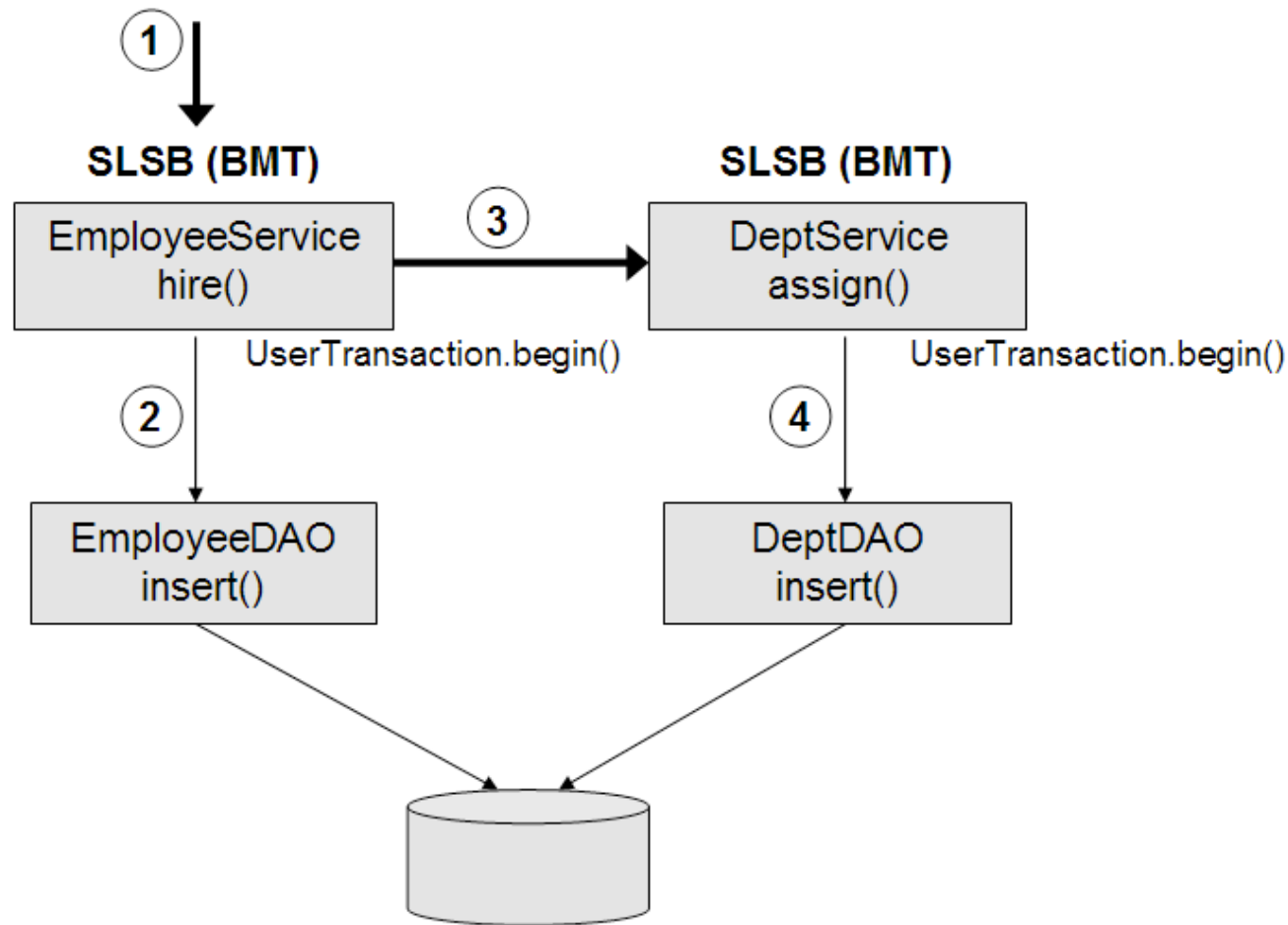
Declarative transaction model

With the Declarative Transaction Model the container manages the transaction, meaning that the developer does not have to write Java code to start or commit a transaction.
The developer must just tell the container *how* to manage the Transaction.

This can be done via Annotation, with both Spring and EJB3.
The approach is very similar, we have to define the transaction attribute of each business method that has to be sorrounded by a transaction.

 We can also define a global transaction attribute, and *override* it in the single methods.

```
@Stateless
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class TradingServiceBean implements TradingService
{
        @TransactionAttribute(TransactionAttributeType.REQUIRED)
        public void aMethod()  {
                .........
        }
}
```

Declarative transaction model

Transaction Attributes

When using declarative transactions we must tell the container how it should manage the transaction.
When should the container start a transaction?
What methods require a transaction?
Should the container start a transaction if one doesn't exist?

This beahviour can be controlled setting the appropriate attibute.

Spring and EJB have 6 commons attibutes, with the same semantyc.

- **Required**
- **Mandatory**
- **RequiresNew**
- **Supports**
- **NotSupported**
- **Never**

Declarative transaction model

Required Attribute

The Required attribute tells the container that a transaction is needed for the particular method. If there is an existing transaction context the container will use it; otherwise it will start a new transaction.

Mandatory Attribute

The Mandatory attribute tells the container that a transaction is needed for a particular method. However, unlike the Required attribute, this attribute will never start a new transaction. When using this transaction attribute, a prior transaction context *must* exist when the method is invoked. If a transaction has not been started prior to a method invocation, an exception wil be thrown.

Supports Attribute

If there's an existing transaction in progress, the current method can run within this transaction. Otherwise, it is not necessary to run within a transaction.

Declarative transaction model

RequiresNew Attribute

The current method must start a new transaction and run within its own transaction. If there's an existing transaction in progress, it should be suspended. This attribute is very useful for an activity that must be committed independent of the outcome of the surrounding transaction. One example of the use of this attribute is audit logging.

NotSupported Attribute

The NotSupported attribute tells the container that the method being invoked does not use a transaction. If a transaction has already been started it will be suspended until the method completes. If no transaction exists the container will invoke the method without starting a transaction.

Never Attribute

The Never attribute tells the container that the method being invoked *cannot* be invoked with a Transaction. If there's an existing transaction in progress, an exception will be thrown.

Declarative transaction model

Only in Spring, we can use another transaction attibute.

Nested Attribute (only Spring)

If there's an existing transaction in progress, the current method should run within the nested transaction of this transaction. Otherwise, it should start a new transaction and run within its own transaction. The behavior is useful for situations such as batch processing, in which you've got a long running process and you want to chunk the commits on the batch.

Declarative transaction model

1) If the method annotated with transactional who started the transaction ends normally, all job done in db will be committed.

2) If a RuntimeException has beeen thrown somewhere during execution, everything will be rollbacked.

3) If an Applcation exception (checked) has been thrown, the container doesn't rollback automatically.

```
@Stateless
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class TradingServiceBean implements TradingService
{
        @TransactionAttribute(TransactionAttributeType.REQUIRED)
        public void firstMethod()  {
                someDAO.someMethod();
                secondTransactionalMetod();
                thirdTransactionalemethod();
        }
}
```

If all these methods have been designed to throws RuntimeException, everything goes as expected.

Declarative transaction model

The scenario changes if we have to manage checked exceptions.

```
@Stateless
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class TradingServiceBean implements TradingService
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void firstMethod()  throws Exception {
        someDAO.someMethod();
        secondTransactionalMethod();
        thirdTransactionalMethod();
    }
```

These methods now throws checked exceptions

This is not a good solution: we propagate the exception to the caller, but we don't instruct the container to rollback. A more correct approach is the following

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void firstMethod()  throws Exception {
    try {
        someDAO.someMethod();   secondTransactionalMethod();
        thirdTransactionalMethod();
    } catch (ApplicationException e)  {
        ctx.setRollbackOnly();
    }
}
```

Declarative transaction model

Spring allows a more flexible approach than EJB3.0, allowing to choose which checked exceptions must be rollbacked.

```java
@Transactional(
    propagation=Propagation.REQUIRED,
    rollbackFor= {ApplicationException1, ApplicationException2 },
    noRollbackFor= { ArithmeticException }
)
public void transactionalMethod()  {
    someDAO.someMethod();
    secondTransactionalMethod();
    thirdTransactionalMethod();
}
```

Otherwise, we must inject and use a TransactionManger to manage the rollback.

DAO Pattern Revisited

In conjuction with declarative transaction model and ORM, it's very common the following implementation of classic DAO pattern.

Only EJB

```
@Local
public interface GenericDAO<T, ID extends Serializable> {

    T findById(ID id);

    List<T> findAll();

    T makePersistent(T entity);

    void makeTransient(T entity);

    void flush();

    void clear();
    ...........
}
```

DAO Pattern Revisited

More interesting are the implementation of the generic DAO interface.

```java
public class GenericDAOJpa <T, ID extends Serializable> implements
GenericDAO<T, ID> {

    @PersistenceContext(unitName = PERSISTENCE_UNIT_NAME)
    protected EntityManager entityManager;
    private Class<T> persistentClass;

    public GenericDAOJpa(final Class<T> persistentClass) {
        this.persistentClass = persistentClass;
    }
    public T findById(ID id) {
        T entity = (T) this.entityManager.find(this.persistentClass, id);
        if (entity == null) {
            throw new EntityNotFoundException(msg);
        }
        return entity;
    }
    public T makePersistent(T entity) {
        return entityManager.merge(entity);
    }

    ...............
```

Abstraction of CRUD methods for generic entity T

DAO Pattern Revisited

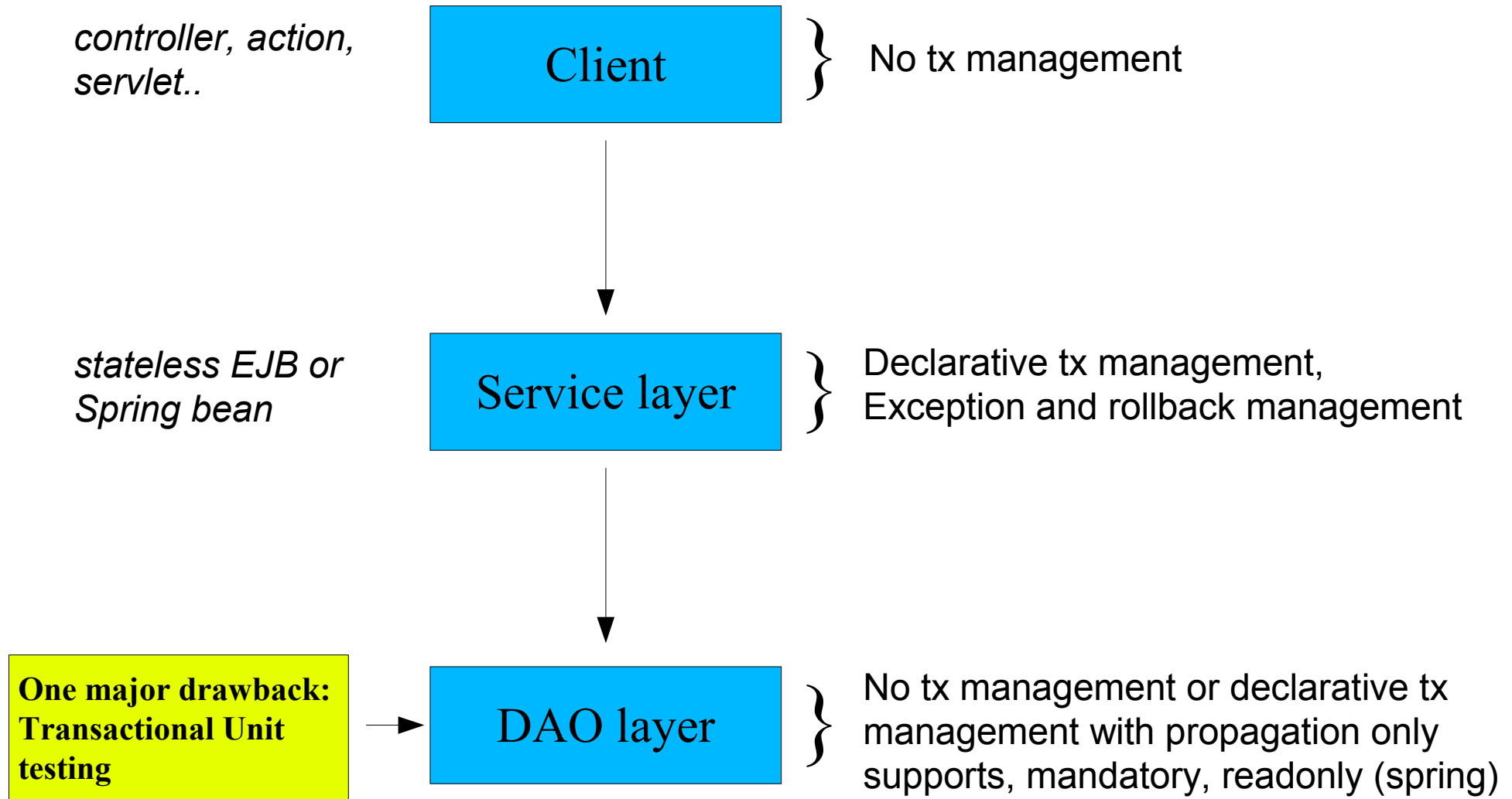With this superclass available, the CRUD for all entities are free.

`@Remote` ← Only EJB

```
public interface UtenteDAO extends GenericDAO<Utente, Long>{
    // sub class specific methods
    public Utente findByCredentials(String userName, String password);
}


@Stateless                            Only EJB
@Repository("utenteDAO")              Only Spring
public class UtenteDAOJpa extends GenericDAOJpa <Utente, Long> implements
UtenteDAO {

    public UtenteDAOJpa() {
        super(Utente.class);
    }


    public Utente findByCredentials(String userName, String password) {
        Map<String,Object> parameters = new HashMap<String,Object>();
        parameters.put("p1", userName);
        parameters.put("p2", password);
        Query query = createQuery("findByCredentials", parameters);
         return (Utente)query.getSingleResult();
    }
}
```

DAO Pattern Revisited

controller, action, servlet..

Client } No tx management

stateless EJB or Spring bean

Service layer } Declarative tx management, Exception and rollback management

One major drawback: Transactional Unit testing → **DAO layer** } No tx management or declarative tx management with propagation only supports, mandatory, readonly (spring)

DAO Pattern Revisited

Both the DAO implementations, Spring and EJB, use the @PersistenceContext annotation.
But how this annotation links transactions and persistence context (EntityManager or hibernate session)? There are three simple rules:

1) If a container-provided (through injection or obtained through lookup) EntityManager is invoked for the first time, a persistence context begins. If no system transaction is active at that time, the persistence context is short and serves only the single method call. Any SQL triggered by any such method call executes on a database connection in autocommit mode. All entity instances that are (possibly) retrieved in that EntityManager call become detached immediately.

2) If a stateless component(or Spring bean) is invoked, and the caller doesn't have an active transaction or the transaction isn't propagated into the called component (because the methods don't require or support a transaction), a new persistence context is created when the EntityManager is called inside the stateless component. In other words, no propagation of a persistence context occurs if no transaction is propagated.

3) If a stateless component (or Spring bean) is invoked, and the caller has an active transaction *and* the transaction is propagated into the called component (because the component methods require or support transactions), any persistence context bound to the transaction is propagated with the transaction.