CS544
Enterprise Architecture
Midterm December 2016

Name_____

Student ID _____

NOTE: This material is private and confidential. It is the property of MUM and is not to be disseminated.

1. [10 points] **Circle w**hich of the following is TRUE/FALSE concerning Spring Inversion of Control/Dependency Injection:

   T **F** Only Managed Beans can be injected in Spring, a POJO or JavaBean cannot.

   EXPLAIN:____If a POJO or JavaBean can be a Spring Managed bean so they can be injected.

   T **F** @Autowired works only on interfaces. It cannot work directly on classes.

   EXPLAIN:__ It can work on classes. However you lose some of the value, testing; changing implementations

   T F In practice, IoC container is not exactly the same as Dependency Injection as it involves a discovery step concerning the dependency.

   EXPLAIN:__ IoC involves a "look up the dependency" step before injecting it. Spring has declarative configuration that identifies "where to find" the resource to inject.

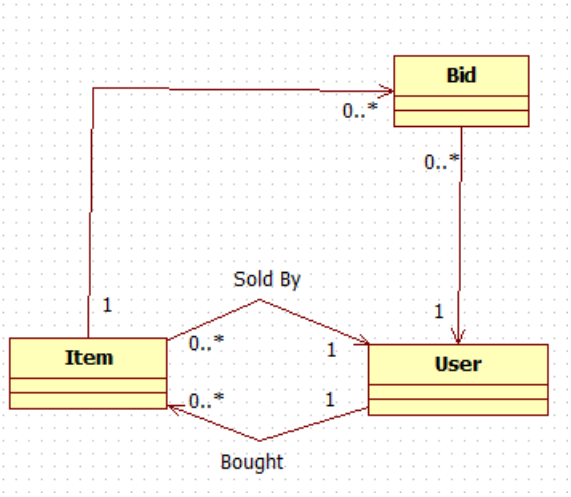
   T **F** A domain object needed in a @Service class is usually a good candidate for Dependency Injection

   EXPLAIN:____ DI is mainly used for "cross" layer transitions, access plumbing resources, etc. NOT for passing business data around…

   T **F** In Spring, DI can be done through either XML or through Annotations. They are mutually exclusive. That means, if you use XML for DI for one bean, they you should use it, exclusively for all beans.

   EXPLAIN:____ You could inject a DAO[memberDAO in memberService] through XML & inject another DAO[productDAO in productService] through @Autowired. Remember, XML configuration is the final word – if you configure the same bean twice, the XML confguration will take precedence.

2. [15 points] For the following relationships implement a SubSelect that fetches all items with their corresponding collection of bids.



What performance problem[s] does the SubSelect fetch address?
How does it work? – Explain the "algorithm" based on a universe of 10 Items each with a collection of 5-10 Bids.
Compare it to Join Fetch.

In Item.Java

```
public class Item {

	@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
	@Fetch(FetchMode.SUBSELECT)
	 private List<Bid> bids = new ArrayList<Bid>();
```

In ItemServiceImpl.Java

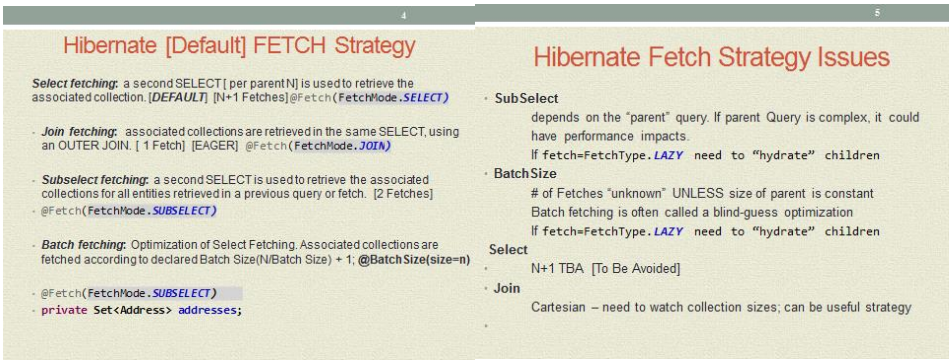
```
	public List<Item> findbySubSelect() {

		List<Item> items = (List<Item>)this.findAll();
		// hydrate since LAZY load
		items.get(0).getBids().get(0);

		return items;
	}
```

Subselect does ONE fetch for the Items and ONE fetch for ALL collections. Therefore the number of Items & Bids in the collections does not matter,
It solves the Cartesian issue && the N+1 issue.
If the collection is LAZY loaded, then doing a get all for the "parent" entities [Item as in above] will get the "parent"/Item list. Within the same transaction, a reference to ONE of the Bid collections will fetch ALL the collections.

The Join Fetch, on the other hand will get ALL the Bids AND Items in ONE Select/fetch.
The Join Fetch, however suffers from the Cartesian product issue [ItemsXBids] which means that more than one copy of each Item/Bid pair will be included in the fetch. The number of copies depends on the number of Bids in the collection. So if Item A has 3 bids, 3 copies of Item A will be present.



3. [15 points] The Core J2EE DAO pattern is fundamental to a well-organized ORM application.
   Explain the pattern, what is for, how it works. Include in the explanation, the sequence of interactions between the user, a DAO and the database. Include an explanation [& UML diagram] of the Generic DAO design.
   Be specific, give examples.

11

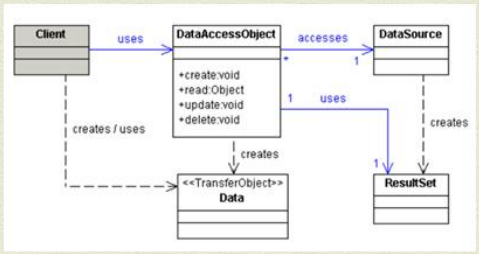
CRUD Services
Core J2EE DAO pattern

Data Access Object (DAO)

- Manage the connection with the data source
- Abstract and encapsulate access to data source
- Provides CRUD access:
  - Create
  - Read
  - Update
  - Delete
- Hides the data source implementation details
- Interface allows for different storage schemes
- Adapter between the client and the data source.

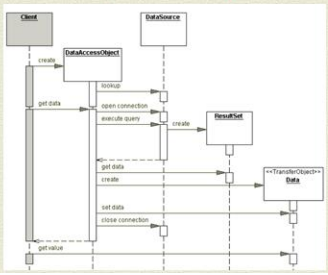
12

DAO Class Diagram



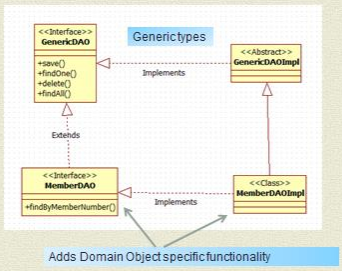
1

DAO Sequence Diagram



1

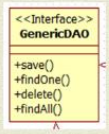
"Classic" ORM GenericDAO



15

Generic DAO Interface

```
public interface GenericDao<T> {
    void save(T t);
    void delete(Long id);
    T findOne(Long id);
    T update(T t);
    List<T> findAll();
```



16

Generic DAO Implementation

```
public abstract class GenericDaoImpl<T> implements GenericDao<T> {

@PersistenceContext
    protected EntityManager entityManager;
    protected Class<T> daoType;

     public void setDaoType(Class<T> type) {
           daoType = type;
     }
    @Override
    public void save( T entity ){
        entityManager.persist( entity );
     }
    public void delete( T entity ){
        entityManager.remove( entity );
```

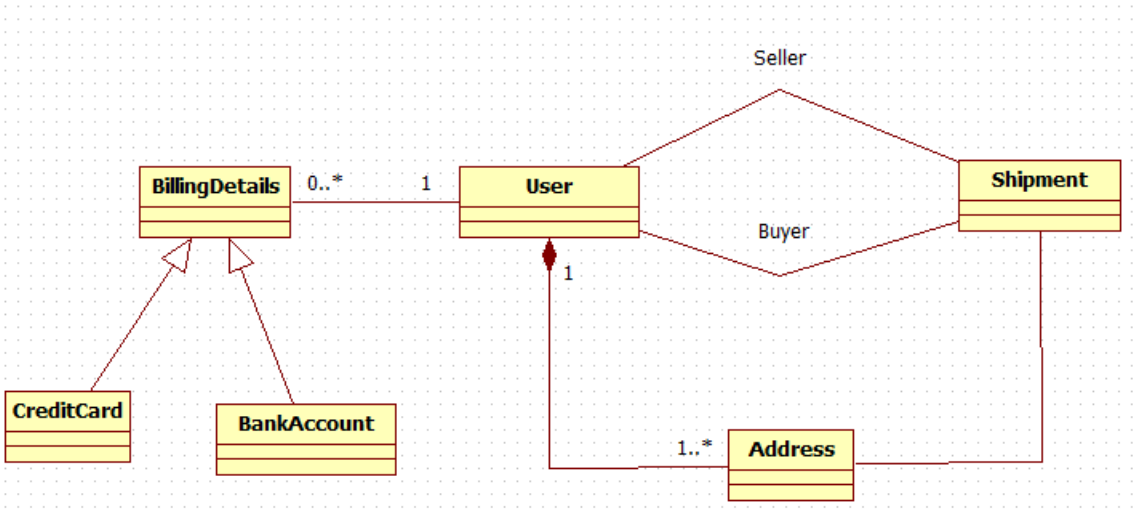
17

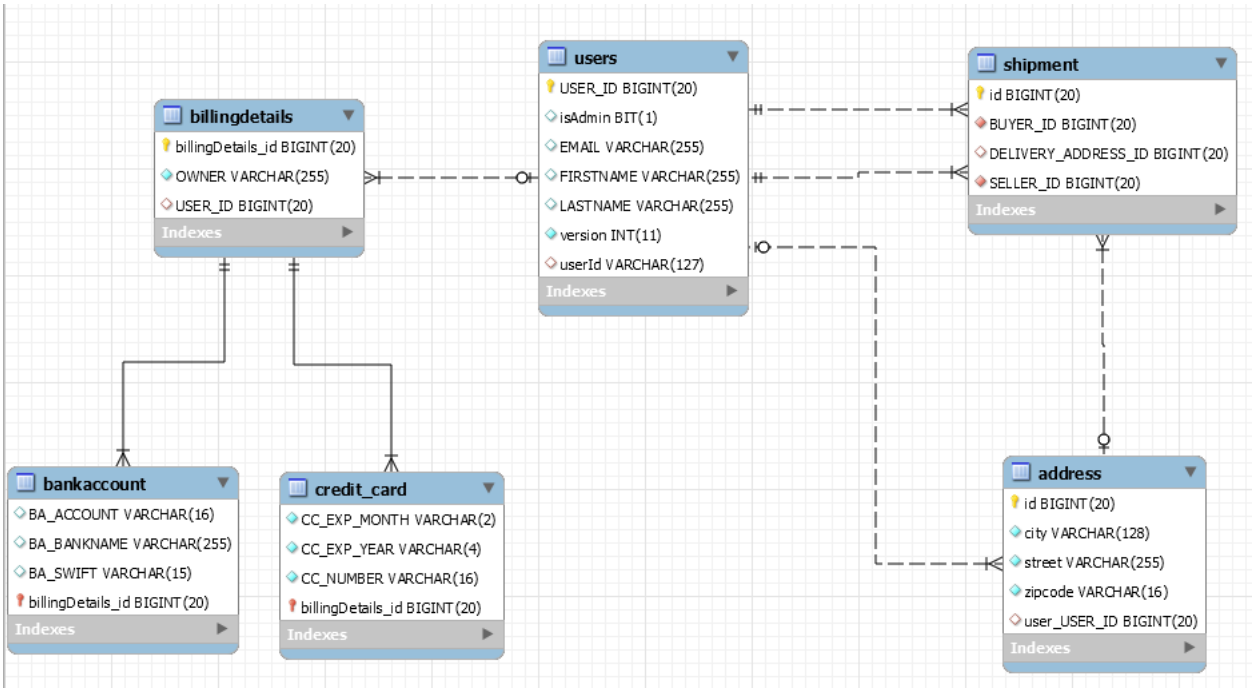
Domain Class specific DAO

```
public interface MemberDao extends GenericDao<Member> {
    public Member findByMemberNumber(Integer number);
```


```
public class MemberDaoImpl extends GenericDaoImpl<Member> implements MemberDao
public MemberDaoImpl() {
super.setDaoType(Member.class);
}
 public Member findByMemberNumber(Integer number) {
   Query query = entityManager.createQuery("select m from MEMBER m
                              where m.memberNumber =:number");
   return (Member) query.setParameter("number", number).getSingleResult();
}
```

4. [20 points] Annotate the Domain Objects based on the Domain Model and Entity Relationship Diagram provided. NOTE: All the Domain Objects are not listed. All the fields are not listed. Only annotate the objects and fields that are listed.





```java
@Entity
@Table(name = "USERS")
 public class User implements Serializable  {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "USER_ID")
    private Long id = null;

    @Column(name = "FIRSTNAME")
    private String firstName;

    @Column(name = "LASTNAME")
    private String lastName;

    @Column(name = "EMAIL")
    private String email;

     @Column(name = "isAdmin")
    private boolean admin = false;

    @OneToOne(fetch=FetchType.EAGER,  cascade = CascadeType.ALL)
    @JoinColumn(name="userId")
    private UserCredentials userCredentials;

       @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="user")
         private Set<Address> addresses = new HashSet<Address>();

       @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="buyer")
         private Set<Shipment> buyShipments = new HashSet<Shipment>();

       @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="seller")
         private Set<Shipment> sellShipments = new HashSet<Shipment>();

       @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="user")
      private Set<BillingDetails> billingDetails = new HashSet<BillingDetails>();
```

```java
 @Entity
@Table(name = "SHIPMENT")
public class Shipment {

    @Id @GeneratedValue
    private Long id = null;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="DELIVERY_ADDRESS_ID")
     private Address deliveryAddress;

     @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="BUYER_ID")
    private User buyer;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="SELLER_ID")
     private User seller;
```

```java
@Entity
public class Address implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
   private Long id = null;

    @Column(length = 255)
    private String street;

    @Column(length = 16)
    private String zipcode;

    @Column(length = 128)
    private String city;

    @ManyToOne(fetch=FetchType.LAZY)
    private User  user;
```

```java
@Entity
@Table(name = "billingDetails")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails  {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "billingDetails_id")
    private Long id = null;


    @Column(name = "OWNER")
    private String owner;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "USER_ID")
     private User user;
```

```java
@Entity
public class BankAccount extends BillingDetails {

    // Id Shared with Superclass - BillngDetails

    @Column(name = "BA_ACCOUNT",length = 16)
    private String account;

    @Column(name = "BA_BANKNAME",  length = 255)
    private String bankname;

    @Column(name = "BA_SWIFT", length = 15)
    private String swift;
```

5. [10 points] Explain the concept of locking. Include the definition of the two strategies covered in class. Give the details of Version-Based Optimistic Concurrency [Locking], how it relates to isolation levels, how it is implemented in JPA.

1

Lock Mode for Data Consistency

- Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used.

Locking strategies

- **Optimistic Lock**
- **Concurrent transactions can complete without affecting each other,**
- **Transactions do not need to lock the data resources that they affect.**

- **Pessimistic Lock**
- Concurrent transactions will conflict with each other
- Transactions require that data is locked when read and unlocked at commit.

14

Version-Based Optimistic Concurrency[Locking]

- High-volume systems
- No Connection maintained to the Database [Detached Objects]
- Effective in ***Read-Often Write-Sometimes*** scenario

- Uses **read committed isolation level**
  Guarantees **repeatable read isolation level**
  An exception is thrown if a conflict occurs

- **@Version Long version;**
- When entity is updated - version field is incremented.

15

Version-Based Optimistic Concurrency

- Optimistic concurrency assumes that lost update conflicts generally don't occur
  - Keeps version #s so that it knows when they do
  - Guarantees best performance and scalability
  - The default way to deal with concurrency
  - There is no locking anywhere
  - It works well with very long conversations, including those that span multiple transactions
- First commit wins instead of last commit wins
  - An exception is thrown if a conflict would occur
    - ObjectOptimisticLockingFailureException

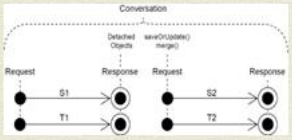
16

Optimistic Locking [Cont.]

- It works well with long conversations, including those that span multiple transactions

- **LONG CONVERSATION Use Case:**
  A multi-step dialog, for example a wizard dialog interacts with the user in several request/response cycles.

- ***session-per-request-with-detached-objects***
-
- @Version Locking manages
- Detached object consistency



17

JPA Optimistic Locking

JPA version-based concurrency control

- *Simply add @Version to an Integer field – JPA takes care of the rest*

```
@Id
 @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id = null;
   @Version
   @Column(name = "version")
   private int version = 0;
Hibernate:
    update purchaseOrder
    set orderNumber=?,version=?
    where id=? and version=?
```

@Version property is incremented on every DB write

Checked for consistency on every update

6. [15 points] Implement a JQPL query that looks up a User by email who bought an Item with a shipping address that has a specific zip code. [Reference UML in problem #4]
   For instance:

Find User who has an email address of JohnDoe@mail.com who bought an Item that was shipped to a zip code equal to 52556

OR

Find User who has an email address of JBean@post.com who bought an Item that was shipped to a zip code equal to 12345.

The Query should be a parameterized query. Also identify all the classes in the specific packages that need to be modified to adhere to the N-Tier architecture convention.

ANSWER:

edu.mum.dao. UserDao

public User findByBoughtItemShippedZip(String email,String zipCode);

edu.mum.dao.impl. UserDaoImpl

```
public User findByBoughtItemShippedZip(String email,String zipCode) {

        Query query = entityManager.createQuery("select u from User u,Shipment s
                     where u.email =:email and s.buyer = u "
                     + " and s.deliveryAddress.zipcode = :zipCode");
        return (User) query.setParameter("email", email)
                .setParameter("zipCode", zipCode).getSingleResult();
}
```

edu.mum.service. UserService

public User findByBoughtItemShippedZip(String email,String zipCode);

edu.mum.service.impl. UserServiceImpl

```
public User findByBoughtItemShippedZip(String email,String zipCode) {
    return  userDao. findByBoughtItemShippedZip (email,zipCode);

}
```