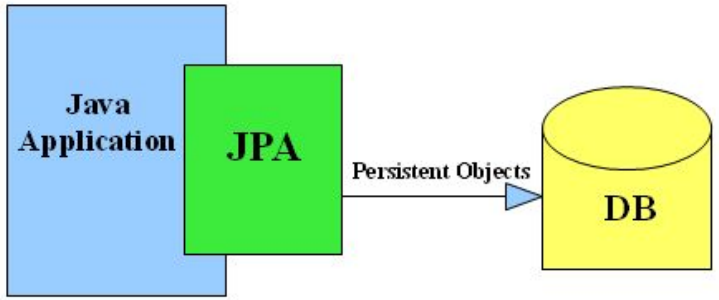
Arquitecturas Web

Persistencia, JDBC, Patrones, JPA

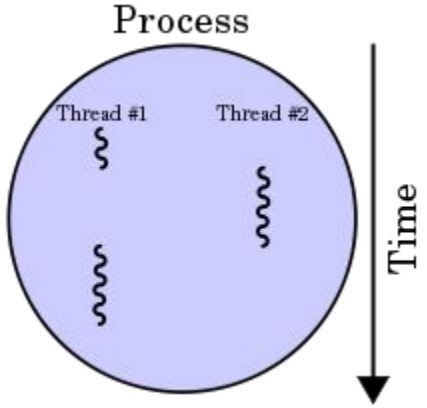
2022

Algunos conceptos previos

- Conexión a BD
- Transacción
- Thread
- Patrones de diseño (aplicados)
  - Factory Method y Abstract Factory
  - Singleton



Threads



- Un proceso “liviano”
  - Una secuencia de instrucciones de programa que puede ser manejada independientemente por un scheduler
  - La implementación depende del sistema operativo subyacente
  - Pueden existir varios threads dentro de un mismo proceso, ejecutando concurrentemente
  - Los threads comparten memoria, código ejecutable, u otro tipo de recursos
- Multi-threading → **Performance**
  - A fin de lograr aplicaciones más responsivas
  - Paralelismo
  - Mejor uso de recursos del sistema

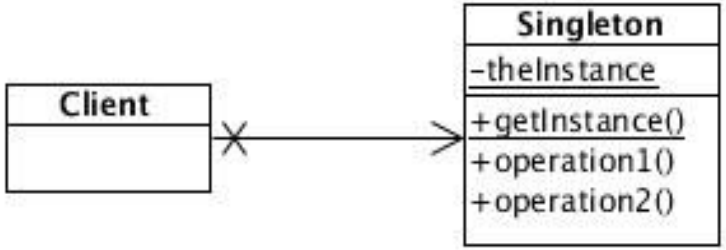
Patrón DAO: Data Access Object

- Intermediario entre la lógica del negocio y la BD
  - Proporciona una “interface abstracta” de la base de datos, y encapsula los mecanismos de acceso (por ej., gestión de conexión)
  - Implementa el principio de *Single Responsibility*
  - Puede cambiarse el mecanismo de persistencia en forma “controlada”
  - Minimizar los detalles (de persistencia) que se exponen
  - Generalmente asociado a Java EE, pero también puede implementarse en otros lenguajes (por ej., PHP)

- **Ventajas → Modificabilidad**
- **Desventajas → Performance?**

Repaso de patrones: Singleton

- Restringe la instanciación de una clase a una **única** instancia
  - Provee consistencia para el resto de los objetos del sistema
  - En Java, se implementa normalmente con métodos de clase



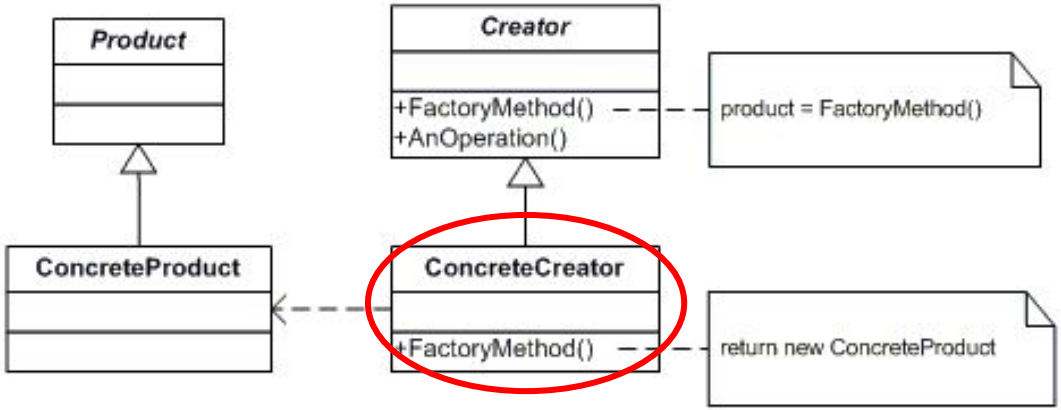
Ejemplo de Singleton (Java)

```
public class Coin {
    private static final int ADD_MORE_COIN = 10;
    private int coin;
    private static Coin instance = new Coin(); // Eagerly loading of singleton instance
    private Coin(){
        // private to prevent anyone else from instantiating
    }
    public static Coin getInstance(){
        return instance;
    }

    public int getCoin(){
        return coin;
    }
    public void addMoreCoin(){
        coin += ADD_MORE_COIN;
    }
    public void deductCoin(){
        coin--;
    }
}
```

Repaso de patrones: Factory Method

- Es un patrón creacional, que permite crear distintos “productos” (objetos), sin necesidad de especificar la clase exacta del objeto que necesita ser creado
  - Permite distintas creaciones de objetos, dependiendo del contexto
  - En Java, se implementa normalmente con una clase abstracta (para el creador)



Ejemplo de Factory Method (Java)

```
public abstract class Room {
    abstract void connect(Room room);
}
public class MagicRoom extends Room {
    public void connect(Room room) {}
}
public class OrdinaryRoom extends Room {
    public void connect(Room room) {}
}
```

```
public abstract class MazeGame {
    private final List<Room> rooms = new ArrayList<>();
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        rooms.add(room1);
        rooms.add(room2);
    }
    abstract protected Room makeRoom();
}
```

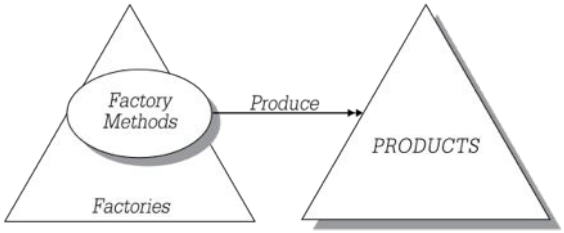
```
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}
```

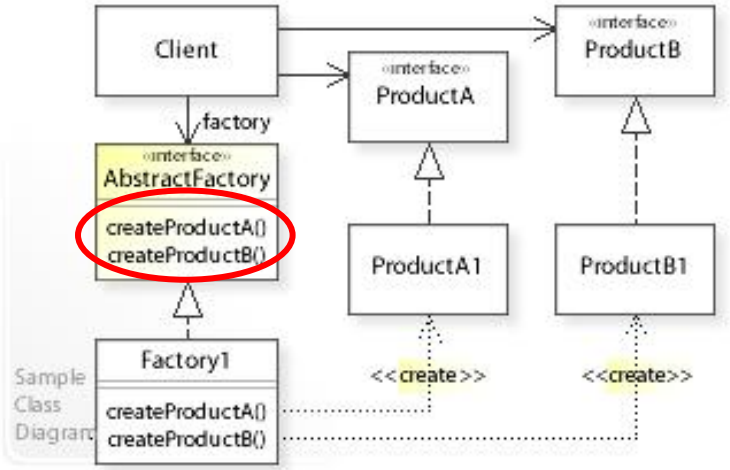
```
public class OrdinaryMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}
```

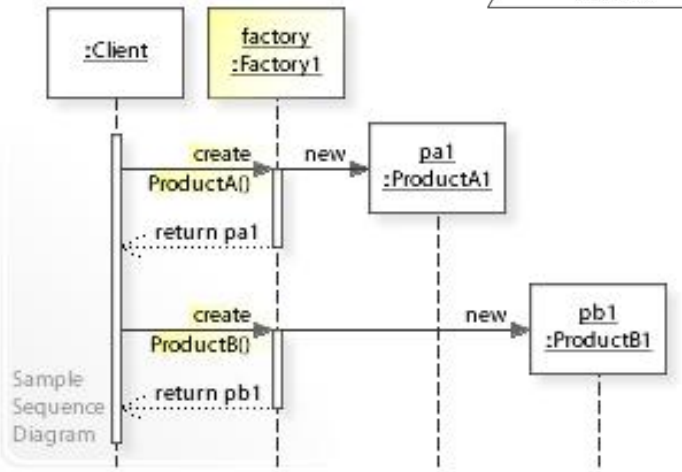
```
MazeGame ordinaryGame = new OrdinaryMazeGame();
MazeGame magicGame = new MagicMazeGame();
```

Repaso de patrones: Abstract Factory

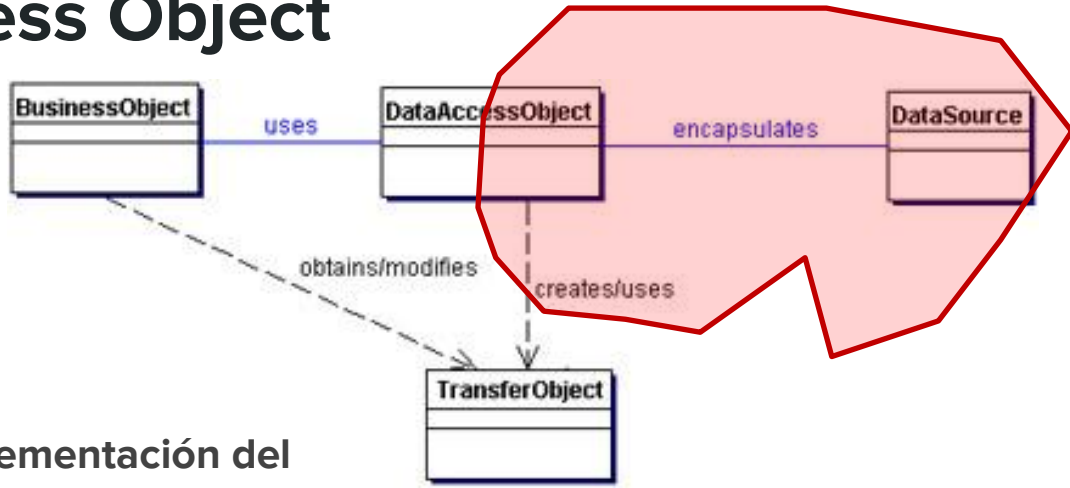
- Encapsula una estrategia de creación de factories de productos individuales
  - La estrategia de creación concreta puede ser un Singleton
  - Involucra varios Factory Method







Patrón DAO: Data Access Object

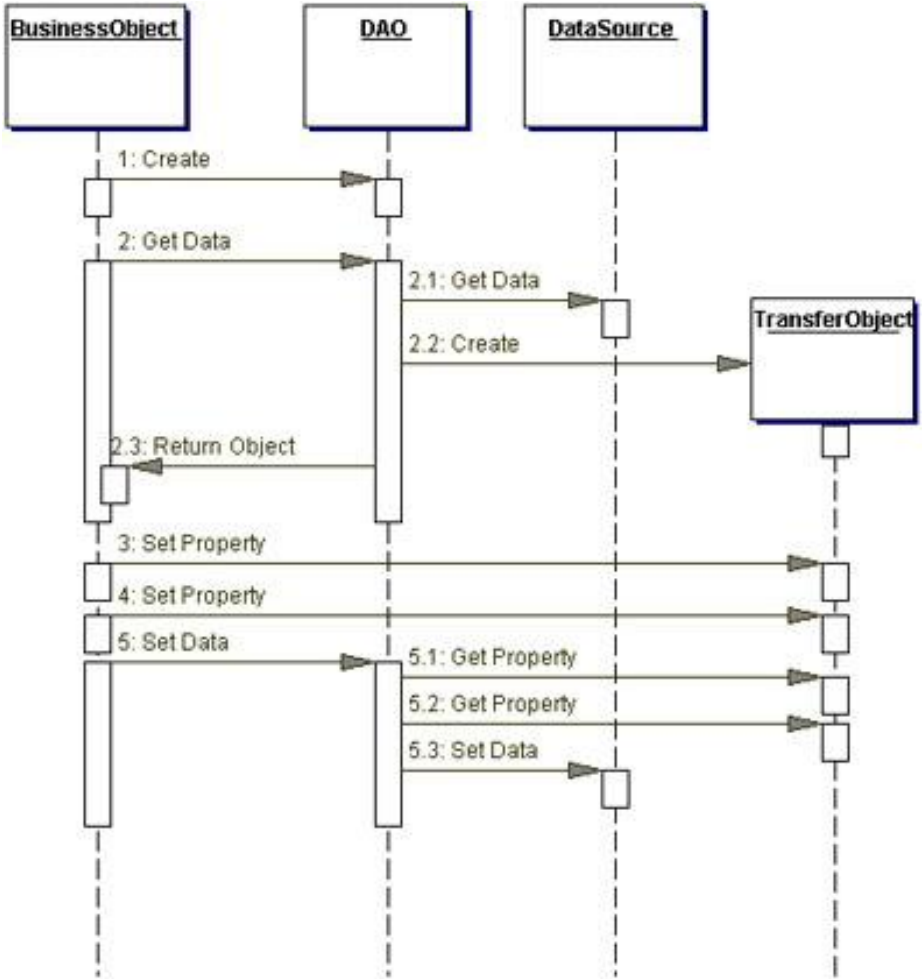


- BusinessObject
  - Representa el objeto que requiere los datos (o servicio) de la fuente de datos
- **DataAccessObject (DAO)**
  - Rol principal, que abstrae la **implementación del acceso a la fuente de datos** y permite un acceso transparente. *Encapsula conexión, carga y almacenamiento de datos*
- DataSource
  - Representa la implementación de la persistencia (base de datos o archivo)
- DataTransferObject (DTO)
  - Acarrea datos entre objetos

http://www.oracle.com/technetwork/java/dataaccessobject.html

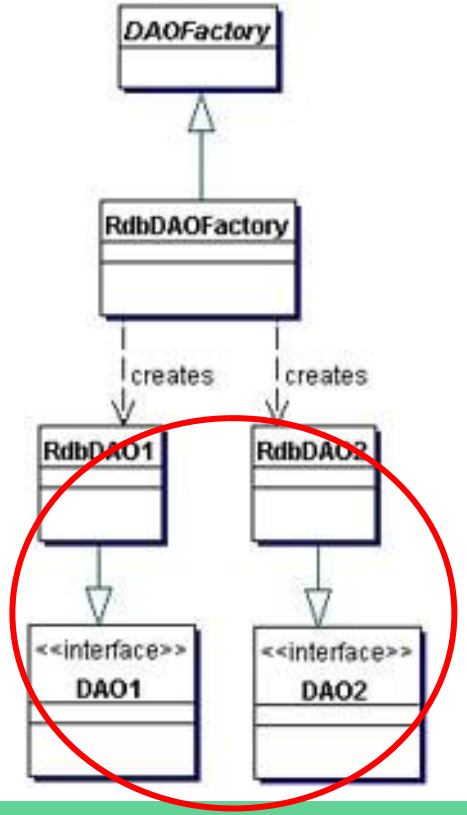
Patrón DAO

- Comportamiento típico



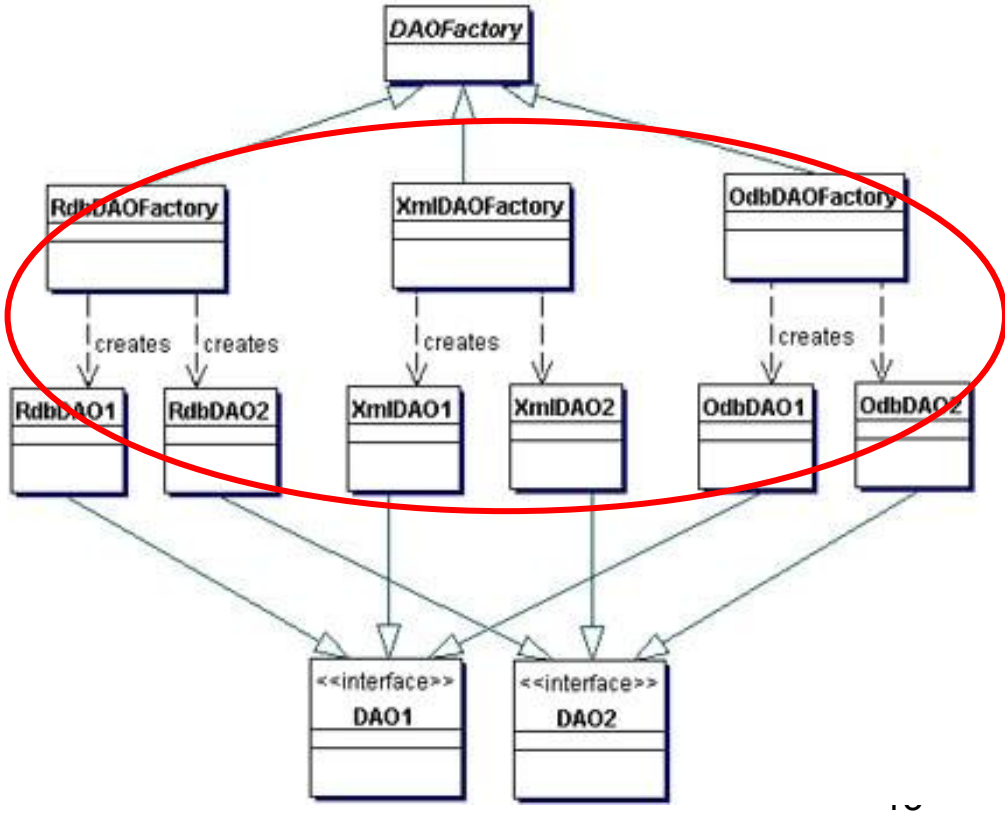
Patrón DAO: La base de datos no va a cambiar

- Alcanza con definir una serie de **factory methods**
- Para cada DAO es conveniente separar la interface de su implementación, y que dicha implementación gestione la conexión y operaciones con la BD

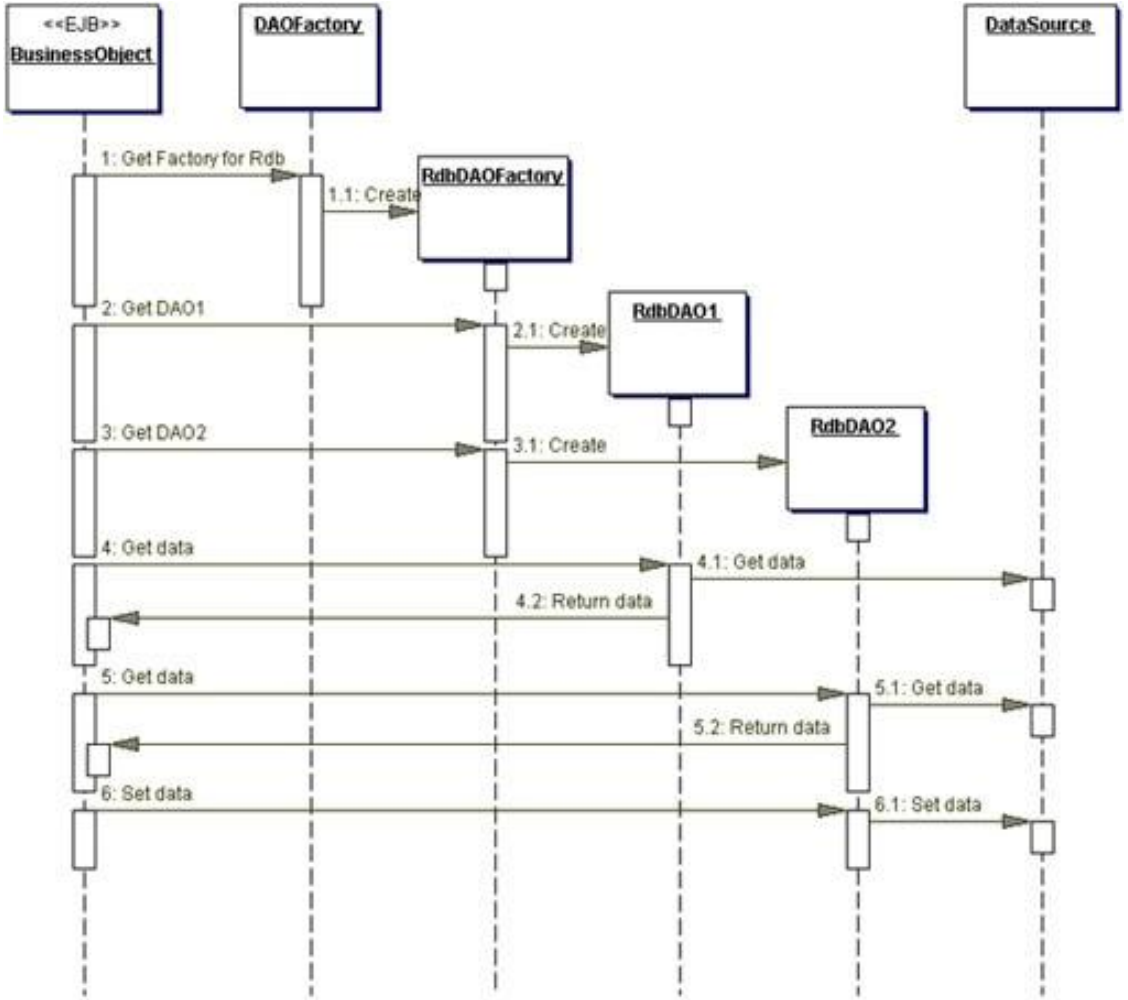


Patrón DAO: La base de datos puede cambiar

- Se incorpora un Abstract Factory, que luego puede ser extendido por cada una de las tecnologías de almacenamiento



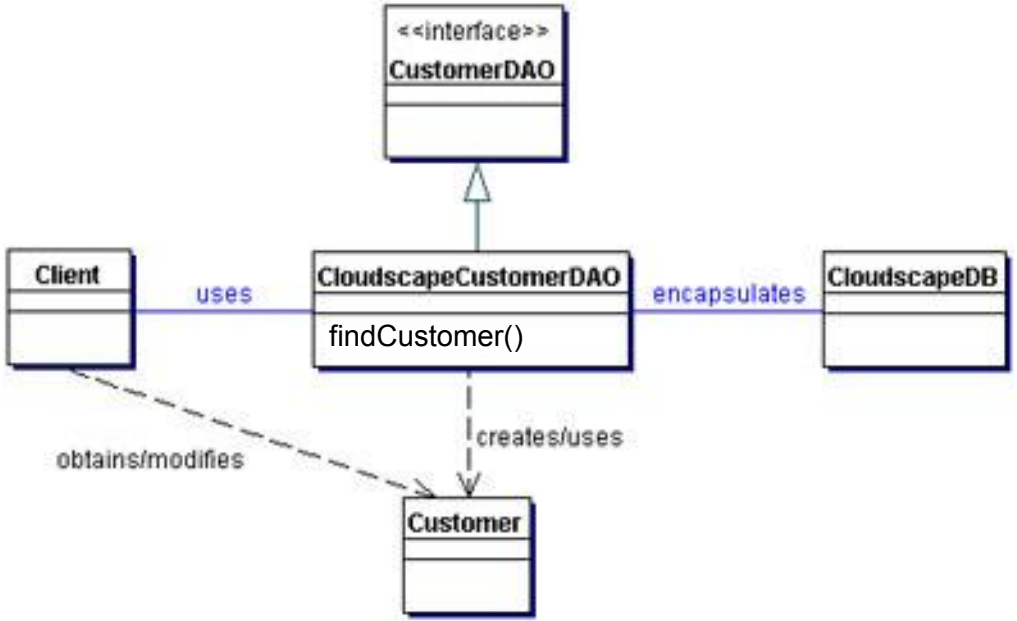
Patrón DAO: Esquema con Abstract Factory



Patrón DAO: Ejemplo de implementación

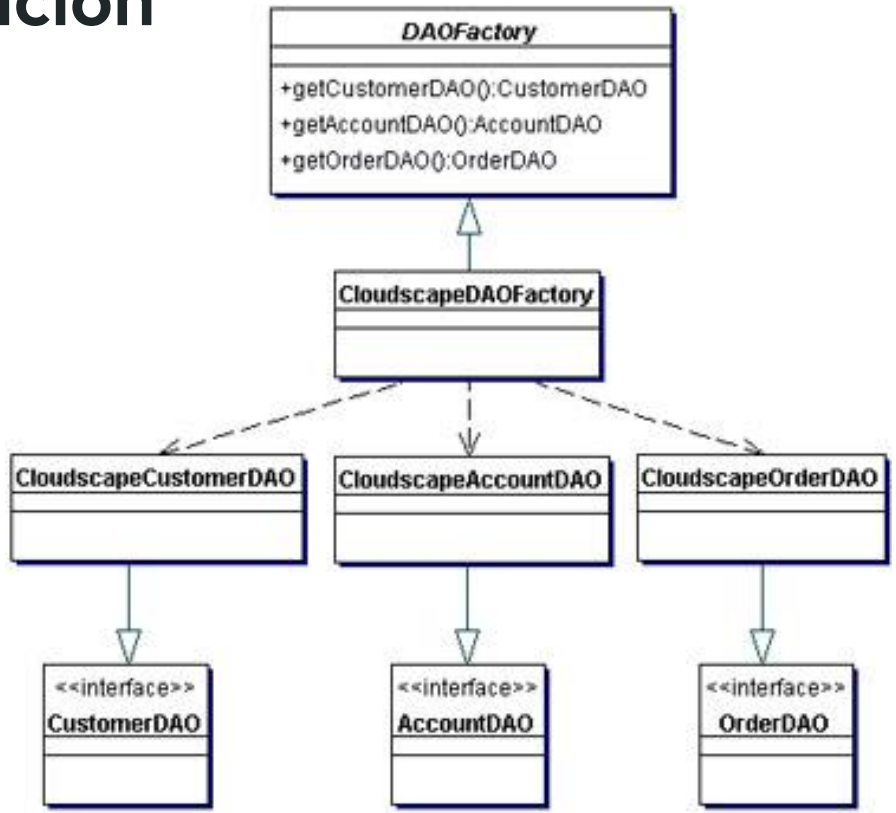
- Entidad **Customer**
- DB: **CloudscapeDB**

- El DAO crea un Customer (DTO) cuando se invoca el método *findCustomer()*



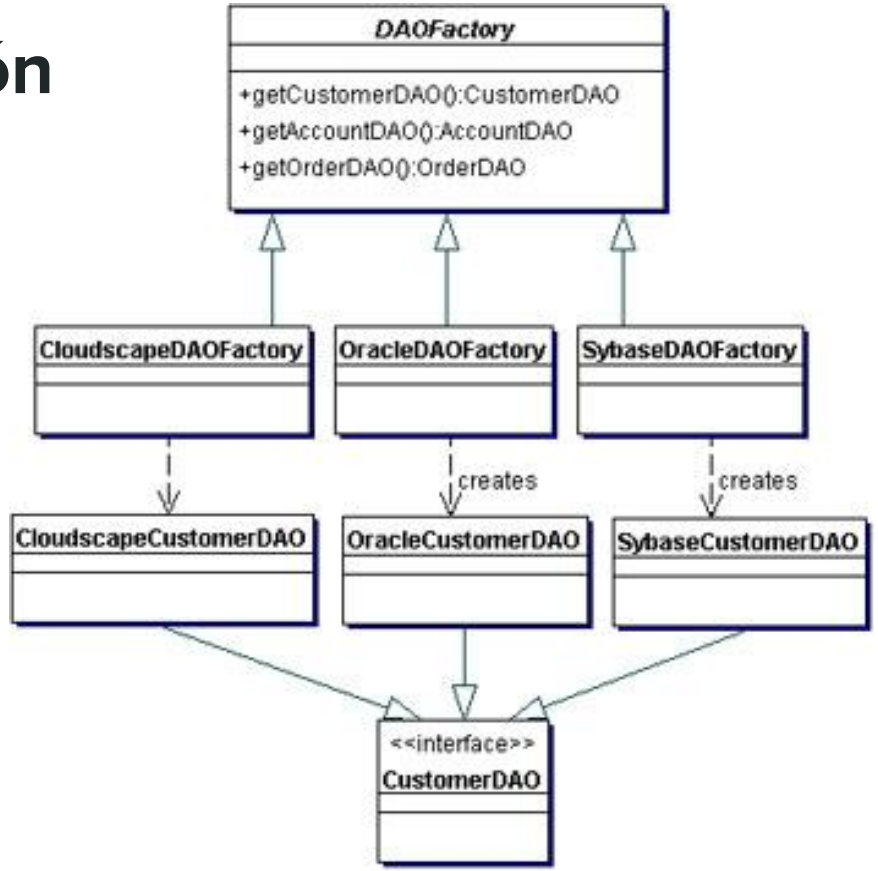
Patrón DAO: Implementación con Factory Method

- Un único mecanismo de persistencia (DB), con varios tipos de objetos DAO



Patrón DAO: Implementación con Abstract Factory

- Varios mecanismos de persistencia alternativos



Patrón DAO: Ejemplo - 1

- Combinación de Abstract Factory y FactoryMethod
- Permite gestionar distintos tipos de persistencia
- Hay que definir la inicialización de cada una de las bases de datos

```
public abstract class DAOFactory {

    public static final int MYSQL_JDBC = 1;
    public static final int DERBY_JDBC = 2;
    public static final int JPA_HIBERNATE = 3;

    public abstract CustomerDAO getCustomerDAO();

    public static DAOFactory getDAOFactory(int whichFactory) {
        switch (whichFactory) {
        case MYSQL_JDBC : return new MySqlJDBCDAOFactory();
        case DERBY_JDBC: return new DerbyJDBCDAOFactory();
        case JPA_HIBERNATE: …
        default: return null;
        }
    }
}
```

Patrón DAO: Ejemplo - 2

```
// MySQL concrete DAO Factory implementation
import java.sql.*;

public class MySQLDAOFactory extends DAOFactory {
  public static final String DRIVER= …
  public static final String DBURL= …
  // method to create DB connection
  public static Connection createConnection() {
    // Use DRIVER and DBURL to create a connection
   }
  public CustomerDAO getCustomerDAO() {
    // MySQLCustomerDAO implements CustomerDAO
    return new MySQLCustomerDAO();
  }
// Other DAOs
...
}
```

```
// Interface that all CustomerDAOs must support
public interface CustomerDAO {
  public int insertCustomer(...);
  public boolean deleteCustomer(...);
  public Customer findCustomer(...);
  public boolean updateCustomer(...);
  public RowSet selectCustomersRS(...);
  public Collection selectCustomers(...);
  ...

}

public class MySQLCustomerDAO implements
CustomerDAO {
 …
}
```

Patrón DAO: Ejemplo - 3

- Factory específico para MySQL
- En vez de crear un UserDTO, podría simplemente retornar strings

```
public class MySqlJDBCDaoFactory extends DAOFactory {

    // Inicializar y conectarse a la base de datos

    @Override
    public List<CustomerDTO> listAllCustomers() {
        //Connection = DriverManager ...
        //PreparedStatement ps =
        //connection.prepareStatement("select * from ...");
        //ResultSet = ps.executeQuery()
        // and so on...

      return null;
    }

}
```

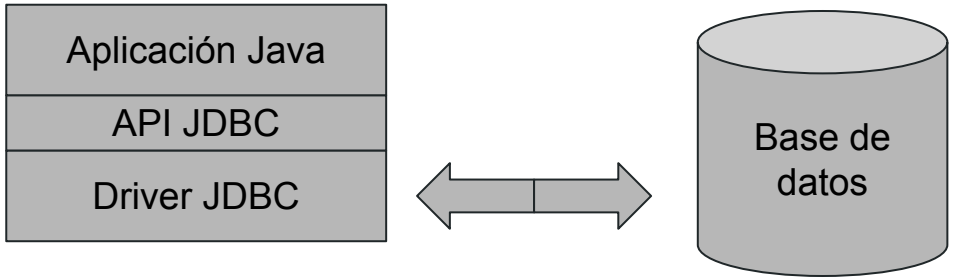
Patrón DAO: Ejemplo - 3

- Consulta desde la capa de servicio (o código cliente)

```
// create the required DAO Factory
DAOFactory mysqlFactory =  DAOFactory.getDAOFactory(DAOFactory. MYSQL_JDBC);
CustomerDAO custDAO = mysqlFactory.getCustomerDAO();

// create a new customer
int newCustNo = custDAO.insertCustomer(...);
// Find a customer object.
Customer cust = custDAO.findCustomer(...);
// modify the values in the Transfer Object and update it using the DAO
cust.setAddress(...);
cust.setEmail(...);
custDAO.updateCustomer(cust);
// select all customers in the same city
CustomerCriteria criteria=new CustomerCriteria();
criteria.setCity("New York");
Collection customersList = custDAO.selectCustomers(criteria);
```

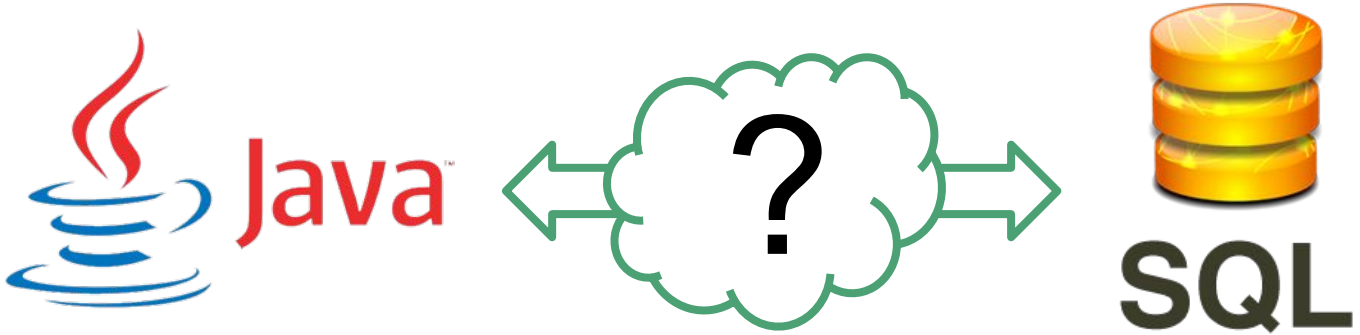
Hasta ahora: JDBC



- Algunos inconvenientes
  - Código Java y SQL mezclados
  - Actualizar manualmente el esquema de la base de datos
  - Dependencia de la sintaxis SQL con cada base de datos
  - Lidiar con aspectos de bajo nivel: conexiones, transacciones, caching de datos

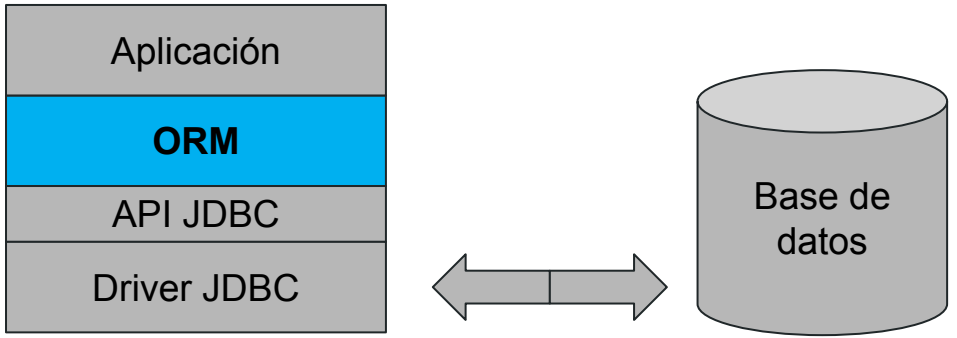
¿Qué desean los desarrolladores?

- El modelo de datos no debe restringir el modelo de objetos
- Acceso rápido a los datos, sin lidiar con aspectos de bajo nivel
- Separar código SQL de Java
- Consultas basadas en **Objetos** sin SQL
- Aislar la aplicación Web de los cambios que pueda haber en los esquemas de la base de datos.



Solución: Mapeo objeto-relacional (ORM)

- Reglas y técnicas de programación para convertir, de manera **transparente**, elementos entre el **modelo de objetos** de una aplicación y el **esquema relacional** de la base de datos.
- El desarrollador ...
  - “ve” una base de datos orientada a objetos
  - se “desliga” de los aspectos de bajo nivel y la tecnología subyacente de base de datos



Solución: Mapeo objeto-relacional (ORM)

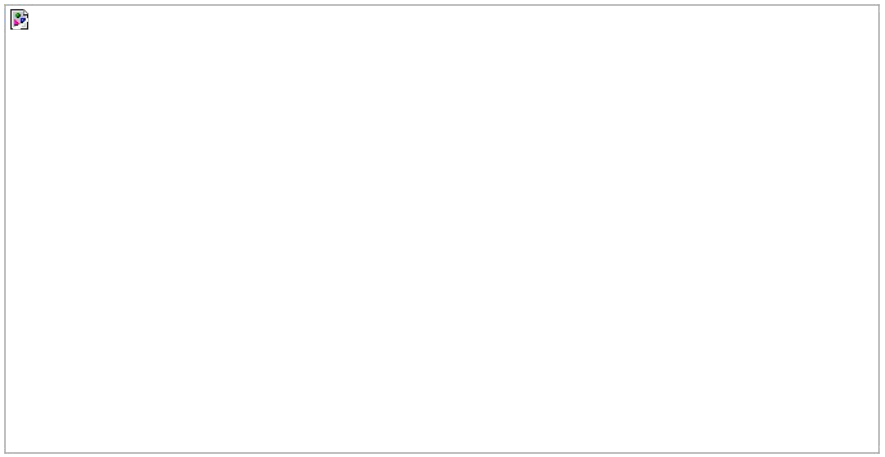
	Programación orientada a objetos	Base de datos relacionales
Elementos	Clases Objetos Atributos Métodos	Tablas Tuplas Atributos Stored procedures, Triggers
Relaciones	Referencias en memoria	Foreign Keys
Identificación	Identificadores de objetos	Primary Keys
Otros aspectos	Encapsulamiento, Herencia, Polimorfismo	Restricciones de integridad, Transacciones
Diseño	Diagrama de clases, secuencia	Diagrama de entidades y relaciones, Esquema relacional
Lenguajes	Java, C++, Python, C# ...	SQL

Diseño de modelo de objetos y modelo relacional

Diagrama de entidades y relaciones (DER)

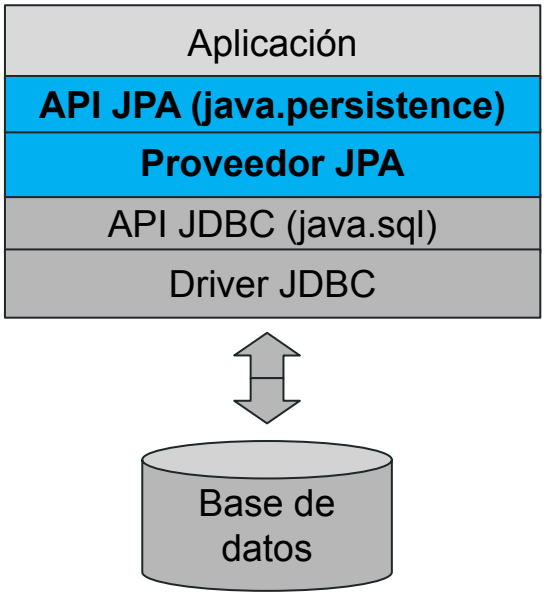
Esquema relacional

Diagrama de clases



JPA: Mapeo objeto-relacional en Java

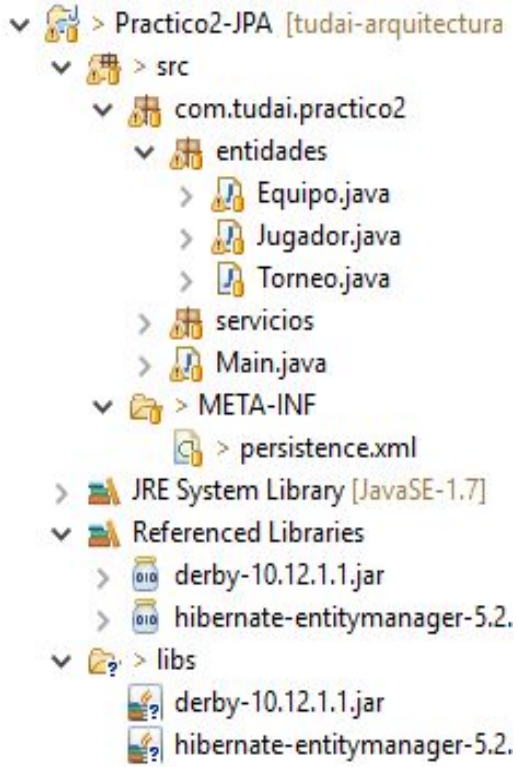
- JPA (Java Persistence API) es una API estándar de Java para implementar mapeo objeto-relacional.
- Es una **abstracción** sobre JDBC



- JPA es solo la API. Para usarla, se necesita una **implementación o "proveedor"** de la misma, como Hibernate, EclipseLink, OpenJPA, etc.
- Se configura a través de metadatos (archivos xml y/o anotaciones)

Proyecto con JPA: ¿Qué se necesita?

- **Driver JDBC + Base de datos**. Ej: Apache Derby
- **Proveedor JPA**. Ej: Hibernate
- **Archivo de configuración JPA** (persistence.xml)
- **Entidades**: clases Java extendidas con metadatos que describen el mapeo de sus atributos a tablas.
  - Mediante archivos de mapeo (entity.xml)
  - Mediante anotaciones (@Entity)
- Código de la aplicación, que manipula las entidades a través de un **EntityManager**



Archivo de configuración JPA

src/META-INF/persistence.xml

```
<persistence>
   <persistence-unit name="my_persistence_unit">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <properties>
         <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver" />
         <property name="javax.persistence.jdbc.url" value="jdbc:derby:MyDataBase;create=true" />
         <property name="javax.persistence.jdbc.user" value="root" />
         <property name="javax.persistence.jdbc.password" value="root" />
      </properties>
      <class>com.tudai.Jugador</class> <!-- <mapping-file>\META-INF\mapping.xml</class> -->
      <class>com.tudai.Equipo</class>
   </persistence-unit>
</persistence>
```

Proyecto con JPA: Entidades sin anotaciones

src/com/.../Jugador.java

```java
public class Jugador {

   private int dni;
   private String nombre;
   private String apellido;
   private int edad;
   private Equipo equipo;

   public Jugador() {
   }

   public int getDni( ) {
      return dni;
   }
   public void setDni(int dni) {
      this.dni = dni;
   }
   ...
}
```

src/META-INF/jugador.xml

```xml
<entity-mappings>
   <entity class="Jugador">
      <table name="TABLA_JUGADORES"/>
      <attributes>
         <id name="dni" />
         <basic name="nombre">
            <column nullable="false" />
         </basic>
         <basic name="apellido">
            <column nullable="false" />
         </basic>
         <basic name="edad" />
         <many-to-one name="equipo">
            <join-column nullable="false"/>
         </many-to-one>
      </attributes>
   </entity>
</entity-mappings>
```

JPA: Entidades con anotaciones

src/com/.../Equipo.java

```
@Entity
@Table(name=”TABLA_EQUIPOS”)
public class Equipo {
   @Id
   @Column(name=”idEquipo”)
   private int id;
   @Column(name=”nombre”,
    nullable = false)
   private String nombreEquipo;
   @OneToMany(mappedBy=”equipo”)
   private Set<Jugador> jugadores;

   public Equipo ( ) {
   }
   ...
}
```

src/com/.../Jugador.java

```
@Entity
@Table(name=”TABLA_JUGADORES”)
public class Jugador {

   @Id
   private int dni;
   @Column(nullable = false)
   private String nombre;
   @Column(nullable = false)
   private String apellido;
   private int edad;
   @ManyToOne
   @JoinColumn(nullable = false)
   private Equipo equipo;

   public Jugador( ) {
   }
   ...
}
```

JPA: Uso de EntityManager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("my_persistence_unit");
EntityManager manager = emf.createEntityManager();
entitymanager.getTransaction( ).begin( );

Equipo equipo = new Equipo( 1 , "River" );
entitymanager.persist( equipo );

Jugador jugador = new Jugador( );
jugador.setDni( 34287439 );
jugador.setNombre( "Emiliano" );
jugador.setApellido( "Sanchez" );
jugador.setEquipo( equipo );
entitymanager.persist( jugador );

entitymanager.getTransaction( ).commit( );
manager.close();
emf.close();
```

JPA: Otros métodos de EntityManager

```
...
entitymanager.getTransaction( ).begin( );

Jugador jugador= entitymanager.find(
      Jugador.class, 34287439 );
jugador.setEdad(28);
entitymanager.flush();

...
entitymanager.remove( otroJugador );


entitymanager.getTransaction( ).commit( );

...
```

Método	Descripcion
EntityTransaction **getTransaction**();	Accede al controlador de transacciones
T **find**(Class<T> entityClass, Object primaryKey);	Busca y devuelve una instancia (SELECT) a partir de su identificador.
void **persist**(Object entity)	Persiste una entidad (INSERT)
void **flush**();	Actualiza el estado de las entidades a la base de datos (UPDATE)
void **refresh**(Object entity);	Actualiza el estado de la entidad desde la base de datos (SELECT)
void **remove**(Object entity);	Remueve la entidad de la base de datos (DELETE)

JPA: ¿Cómo crear y mantener sincronizado el esquema de base de datos?

- A mano: “DROP TABLE … CREATE TABLE …”
- Usando el soporte de DDL automático de los proveedores JPA:

src/META-INF/persistence.xml

```
      ...
      <property name="javax.persistence.jdbc.url" value="jdbc:derby:MyDataBase;create=true" />
      <property name="hibernate.hbm2ddl.auto" value="create"/>
      <!-- <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/> -->
      <!-- <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema" /> -->
    </properties>
  </persistence-unit>
</persistence>
```

Patrón DAO con JPA?

- JPA simplifica el DAO y permite contar con DAOs “genéricos”

- Algunos casos especiales
  - Persistencia != Acceso a datos
  - Queries complejos
  - Sistemas legados (no compatibles con JPA)

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("my_persistence_unit");
EntityManager manager = emf.createEntityManager();
entitymanager.getTransaction( ).begin( );

Equipo equipo = new Equipo( 1 , "River" );
entitymanager.persist( equipo );

Jugador jugador = new Jugador( );
jugador.setDni( 34287439 );
jugador.setNombre( "Emiliano" );
jugador.setApellido( "Sanchez" );
jugador.setEquipo( equipo );
entitymanager.persist( jugador );

entitymanager.getTransaction( ).commit( );
manager.close();
emf.close();
```

Próximos pasos

- Completar el Práctico 1
  - Ejercicios de JDBC sobre Apache Derby y MySQL
  - Luego, abstraer y definir una solución basada en DAO

- Mayores detalles sobre el funcionamiento de JPA

Preguntas?

Patrón DAO Ejemplo - 4

- Factory específico para JPA

```
public class MySqlJpaDaoFactory extends DAOFactory {

    @Override
    public List<UserDTO> listAllUsers() {
      // Here I implement specific functionality
      // to retrieve data using JPA Framework
        //EntityManagerFactory emf = ...
        //EntityManager em = ...
        //List<UserDTO> list = em.get...();
        //return list;

 return null;
    }

}
```

Preguntas sobre DAO en JPA

- Quién gestiona las conexiones en JPA?
- Quién gestiona las transacciones en JPA?
- Es EntityManagerFactory thread-safe?
- Es EntityManager thread-safe?
- Al utilizar entityManager.getTransaction(), cual es el scope de la transaccion?
- Al utilizar entityManager.persist(), cómo se manejan las excepciones?
- Al utilizar entityManager.getTransaction().commit(), cómo se manejan las excepciones? Que sucede en un caso de rollback()?