BiciTron Class List

This document contains information about all the classes present in the project. Written by Daniel García Molero.

- integracion
  - bicicleta (DaoXXXImp is placed into an *imp* folder, but omitted here for the sake of simplicity.)
    - DaoBicicleta: Interface. Functions: Add, update, search, searchId, list. Transfer is usually the argument for the functions. (ID sometimes).
    - DaoBicicletaImp: Implementation. Each of the above functions has Statement (st), ResultSet (rs). [cn (Connection) is sometimes omitted but still necessary].
      cn = (Connection) TransactionManager.getInstance().getTransaccion().getResource();
      st = cn.createStatement(); [Establish connection, create statement.]
      rs = Query result. (st.executeUpdate(*"Query"*)); [rs is *int or ResultSet* depending on the expected Query result]
      Check conditions and return. *ResultSet* type contains Iterator.
  - cliente: Same as above.
    - DaoCliente
    - DaoClienteImp
  - compra: More complex logic but same as above.
    - DaoCompra
    - DaoCompraImp
  - factoria
    - FactoriaDAO: Abstract, not Interface. Singleton, has getInstance(). Atribute: *private static FactoriaDAO instance;* for Singleton.
      Has *public abstract DaoBicicleta createDAOBicicleta/Cliente/Compra()*.
    - FactoriaDAOImp: Extends. public DaoX createDAOX() { return new DaoXImp(); } DaoXImp() from the above modules.
  - query
    - factory
      - FactoriaQuery: Abstract, not Interface. Singleton, has getInstance(). Atribute: *private static FactoriaQuery instance;* for Singleton.
        Also has *public abstract Query getQuery (int event);*
      - FactoriaQueryImp: Extends. Query getQuery(int event) is a switch statement with each one of the possible queries. (Each implemented at the *imp* folder underneath.) int event is a constant from *ListaComandos*. It returns the actual Query object.
    - imp: Implemented queries from the Interface Query.
      - CantidadBicicletasQuery: Acts like a DAO. Has cn, st, rs (Connection, Statement and ResultSet) inside the implemented function (public Object execute (Object datos)). Returns whatever's needed (May or may not be a Transfer object, i.e. list of transfers, int…).
      - NumeroClientesQuery: Same as above.
    - Query: Interface. Only contains public abstract Object execute(Object datos);
  - transactionManager
    - Transaction: Abstract Class. Contains:
      public abstract void start() / void commit() / void rollback() / Object getResource();
    - TransactionFactory: Abstract, not Interface. Singleton, has getInstance(). Atribute: *private static TransactionFactory instance;* for Singleton. Also has another method:

public abstract Transaction crearTransaccion(); (Implemented on TransactionFactoryImp).

- TransactionFactoryImp: Only contains *public Transaction crearTransaccion()* { return new TransactionMySQL(); }
- TransactionManager: Another abstract, Singleton class. Contains: *public abstract boolean nuevaTransaccion() / Transaction getTransaccion() / Boolean eliminaTransaccion();*
- TransactionManagerImp: It contains a *ConcurrentHashMap<Thread, Transaction>* as attribute.
  nuevaTransaccion() searches for the current thread on the HashMap, if it is not contained there, it inserts the current thread into the HashMap, alongside with a new Transaction (TransactionFactory.getInstance().crearTransaccion()).
  getTransaccion() returns the current thread's transaction.
  eliminaTransaccion() searches for the current thread, and if it exists, it deletes it from the HashMap.
- TransactionMySQL: Extends Transaction. Implements its operations. *private Connection connection* as attribute!
  -Default constructor *public TransactionMySQL()* sets the attribute *connection* to DriverManager.getConnection(…); with the needed parameters.
  -void start(): *connection.setAutoCommit(false);*
  -void commit(): *connection.commit() + close();*
  -void rollback(): *connection.rollback() + close();*
  -Object getResource(): Returns the connection itself. (The attribute)

- main
  - Main: Simply invokes the main window.
    Controlador.getInstance().accion(ListaComandos.VENTANAPRINCIPAL, null);
- negocio
  - bicicleta (WITHOUT JPA!)
    - imp
      - ServicioAplicacionBicicletaImp: Implements the functions from the interface. Every function represents a transaction.
        altaBicicleta(TransferBicicleta t) calls the TransactionManager (Singleton call!) to create a new transaction.
        TransactionManager.getInstance().nuevaTransaccion();
        Then starts the transaction
        TransactionManager.getInstance().getTransaccion().start();
        In the case of adding a bike, it checks that a bike with the same name doesn't exist previously, calling
        FactoriaDAO.getInstance().createDAOBicicleta().search(t); where "t" is the Transfer object to be added, for it not to be added twice.
        Same thing to actually add the bike
        FactoriaDAO.getInstance().createDAOBicicleta().add(t); if the previous operation was successful.
        After performing the operation, either "commit" or "rollback" must be called, as well as deleting the Transaction:
        TransactionManager.getInstance().getTransaccion().commit()/rollback();
        TransactionManager.getInstance().eliminaTransaccion();
        The Query however is similar, but works in a different way.
        Creates Transaction, Starts it, pQuery (PareadoQuery object) contains the necessary information for the Query and then a call is made to:
        FactoriaQuery.getInstance().getQuery(ListaComandos.NEGOCIOCANTIDADBIC

ICLETASQUERY).execute(pQuery); [NOTE: The *"execute"* method is abstract and different for each Query].

- TransferBicicleta: Straightforward Transfer Object with *id, material, marca, precio and activo* attributes. +getters/setters.
- TransferBicicletaCarretera: Adds *velocidadMaxima* attribute. Constructor calls super() with every other attribute and manually adds the new one.
- TransferBicicletaMontana: Similar to above.

▪ ServicioAplicacionBicicleta: Interface containing the Application Service functions. *altaBicicleta, bajaBicicleta, modificarBicicleta, detalleBicicleta, listaBicicleta and numeroBicicletas(PareadoQuery pQuery).* To be implemented at the "imp" folder.

- cliente (WITHOUT JPA!)
  - imp
    - ServicioAplicacionClienteImp
    - TransferCliente
  - ServicioAplicacionCliente
- compra (WITHOUT JPA!) [This module has an associated class (LineaDeCompra), with attributes *cantidad, precio* and *precioUnitario*. Compra has *id, costeTotal, fecha* and *activo*. Bicicleta is on the other side of the relation. However, the Transfer Objects have more attributes.]
  - imp
    - ServicioAplicacionCompraImp:
      iniciarCompra(int id) returns a new TransferCompra. (Same Transaction scheme as before)
      Contexto agregarArticulo(Contexto articulo) checks if the article exists, and returns the Context as it was if correct, for it to be processed by the Command itself and the View. It doesn't add the articles!
      JFrameGestionCarrito does instead!
      The rest are straightforward. The current TransferCompra is stored at the VIEW.
    - TransferCompra: Contains *id, idCliente, costeTotal, fecha, lineaCompra (HashMap <Integer, Integer>), log* and *activo.*
    - TransferLineaCompra: Contains *idCompra, idArticulo, cantidad, cantidadNueva, activo.*
  - ServicioAplicacionCompra:
    TransferCompra iniciarCompra(int id); Contexto agregarArticulo(Contexto articulo); TransferCompra finalizarCompra(TransferCompra tCompra); int devolverArticulo(TransferLineaCompra tLineaCompra); TransferCompra detalleCompra(int id); ArrayList<TransferCompra> listaCompra;
- departamento (WITH JPA!)
  - imp
    - ServicioAplicacionDepartamentoImp: Contains the implementations for every function from the module (entity). Every operation follows a similar pattern:
      -EntityManager needs to be created as follows:
      EntityManager eM = Persistence.createEntityManagerFactory("BiciTronCod").createEntityManager();
      -The transaction must get started:
      eM.getTransaction().begin();
      -The Query can now be fixed, either a Named Query from the Entity or a different one created here.
      TypedQuery<Departamento> findByName = eM.createNamedQuery("negocio.entidadesJPA.Departamento.findByNomb

re", Departamento.class);
-In this case, the necessary parameters for the Query must be set as follows: ***findByName.setParameter("nombre", t.getNombre());*** [NOTE: t is a parameter from the function; TransferObject.]
-The Lock Mode must be set here as well before performing the operation: ***findByName.setLockMode(LockModeType.OPTIMISTIC);***
-*findByName* acts as the Query object itself and you can use functions like ***findByName.getResultList()*** [plus ***.size()*** if needed], *getSingleResult(), getFirstResult()*, among others.
-THE RESULTS' *RETURN TYPE* IS THE **TEMPLATE** GIVEN ABOVE, therefore, you can check the attributes from the returned object by calling the functions (getters/setters) from the Entity like so:
Departamento aux = findByName.getSingleResult(); [In case the given name was found]
if(!aux.getActivo()) aux.setActivo(true);
-In the case of this method, after checking that the Department didn't exist, it would be inserted, thus a new instance must be created (managed and persistent) by creating a new *Departamento* object with the default constructor and then calling the *persist(x)* function like so:
Departamento aux = new Departamento(t.getNombre(), t.getDescripcion(), true); [true represents the *activo* field on the constructor].
eM.persist(aux);
-As before, the transaction must be either confirmed or reverted, by calling ***eM.getTransaction().commit();/.rollback();*** and closed afterwards by calling ***eM.close();***

There are other types of Queries however, since every Entity has an ID, we can just make a Query like ***Departamento aux = eM.find(Departamento.class, id, LockModeType.OPTIMISTIC);*** where the first field is the Entity class to search, the second one is the ID to search (given as a parameter in this case) and the third one is the Lock Mode Type.

It is always necessary to commit/rollback before closing the EntityManager:
EntityManager eM = Persistence.createEntityManagerFactory("BiciTronCod").createEntityManager();
eM.getTransaction.begin();
Departamento aux = eM.find(Departamento.class, id); [This one doesn't lock]
eM.getTransaction().commit();
eM.close();

Also:
eM.getTransaction().begin();
List<Departamento> lista = eM.createQuery("SELECT obj FROM Departamento obj", Departamento.class).getResultList();
+commit +close

- TransferDepartamento: Normal Transfer object, contains: *id, nombre, descripcion* and *activo.*

- ServicioAplicacionDepartamento: Interface. Contains every function in the module. *altaDepartamento, bajaDepartamento, modificarDepartamento, detalleDepartamento, listaDepartamento* and *calcularNomina.*

- empleado
  - imp
    - ServicioAplicacionEmpleadoImp: Same as above, except for the fact that Empleado can be either Dependiente or Directivo. To check this, the Query is performed the same way, after getting the results returned, we must check whether the returned Entity is an instance of *Dependiente* or *Directivo.* (Both are entities as well!)
    - TransferDependiente: Inherits from TransferEmpleado.
    - TransferDirectivo: Inherits from TransferEmpleado.
    - TransferEmpleado
  - ServicioAplicacionEmpleado
- entidadesJPA: Entities contain the same functions as a Transfer object, but do NOT need a DAO.
  - Departamento: Starts with a header before the class declaration:
    @Entity
    @NamedQueries({
    @NamedQuery(name= "negocio.entidadesJPA.Departamento.findByid", query = "select obj from Departamento obj where obj.id = :id"),
    @NamedQuery(…) }) Every available Named Query is declared here.
    public class Departamento implements Serializable {
    private static final long serialVerisonUID = 0;
    @Id @GeneratedValue (strategy = GenerationType.IDENTITY) private int id; Entity ID.
    @OneToMany (mappedBy = "departamento") Relations between Entities are declared here.
    private ArrayList<Empleado> empleado;
    private string nombre, descripcion;
    private Boolean activo;
    private ArrayList<Presupuesto> presupuesto;
    @Version private int version; [Serializable version.]
    Methods are implemented underneath.
  - Dependiente
  - Directivo
  - Empleado: In this case, we need a *JOIN* for some queries, thus it must be specified by: ***@ManyToOne @JoinColumn (name = "departamento") protected Departamento departamento;*** and it's called on the Queries like any other attribute (obj.departamento). The inheritance must also be specified under the *@Entity* tag like so: ***@Inheritance (strategy=InheritanceType.JOINED)*** followed by the *@NamedQueries.*
  - Presupuesto: It has a compound Primary Key, therefore, we must have an *IdClass.* ***@IdClass(PresupuestoId.class)*** under the *@Entity* tag, ***@Id @ManyToOne private Departamento departamento; @Id @ManyToOne private Tienda tienda;***
  - PresupuestoId: This is the implemented *IdClass* for Presupuesto, it only has the 2 ID attributes (***private int departamento, private int tienda***) as Integers. It has ***@Embeddable*** instead of *@Entity.*
  - Tienda
- factoria
  - FactoriaServicioAplicacion: Singleton. Abstract class with *private static FactoriaServicioAplicacion factoriaSA;* as attribute.
    public static synchronized FactoriaServicioAplicacion getInstance() {
    if (factoriaSA == null) factoriaSA = new FactoriaServicioAplicacionImp();
    return factoriaSA;

Plus functions:
public abstract ServicioAplicacionBicicleta createSABicicleta();
…

- FactoriaServicioAplicacionImp: Each function simply returns the new object it's asked for: ***public ServicioAplicacionBicicleta createSABicicleta() { return new ServicioAplicacionBicicletaImp(); }*** although implemented.

- tienda
  - imp
    - ServicioAplicacionTiendaImp
    - TransferPresupuesto
    - TransferTienda
  - ServicioAplicacionTienda

- presentacion
  - command
    - factoria
      - FactoriaCommands: Singleton Class. Only one abstract function apart from getInstance(): *public abstract Command getCommand(int event);*
      - FactoriaCommandsImp: Implements the above function; returns the suitable command given the number of an event. Those numbers are on *ListaComandos.java.* It consists on a huge switch that simply returns the appropriate command.
    - imp: These are all implemented commands, they implement Command.java, that has a single abstract function: ***public abstract Contexto execute(Object datos);***
      - bicicleta
        - negocio
          - Negocio(Alta/Baja/Cantidad/Detalle/Lista/Modificar)BicicletaCommand: Every Command Class here contains an implementation of the above function *"execute"*. The action to perform on the first line and the return on the second line. The action to perform usually has *Object datos* as a parameter using *Casting* to make it the needed type of object. It always returns a Contexto, consisting on an event (Command from the list, usually view to show after the action) and whatever the action above returned as *Object Datos*, thus having a *Contexto(int event, Object datos);*
            Classes of the type NegocioXXXCommand return an Event of the type *"ListaComandos.MOSTRARXXX".*
        - vista
          - Vista(AltaBicicletaCarretera/AltaBicicletaMontana/BajaBicicleta/BotoneraBicicleta/CantidadBicicletasQuery/DetalleBicicleta/ListaBicicleta/ModificarBicicletaCarretera/ModificarBicicletaMontana)Command. Same as above, although it performs no action whatsoever. Returns the same *Object datos* from the function parameter without performing any action but adds an event to the *Contexto* object. In this case, the returned events are like *"ListaComandos.VENTANAXXX".*
      - cliente
        - negocio
          - Similar to above
        - vista
          - Similar to above
      - compra

        - negocio
            - Similar to above
        - vista
            - Similar to above
    - departamento
        - negocio
            - Similar to above
        - vista
            - Similar to above
    - empleado
        - negocio
            - Similar to above
        - vista
            - Similar to above
    - principal.vista
        - VistaBotoneraPrincipalCommand
    - tienda
        - negocio
            - Similar to above
        - vista
            - Similar to above
- Command: *public interface Command {* ***public abstract Contexto execute (Object datos);*** *}* each specific Command is specified on the implemented Commands.

- controlador
    - Contexto: Simple class, consisting on two attributes (*private int event* and *private Object datos*). Also has a constructor and two getters, one for the *event* and one for *datos*.
    - Controlador: Singleton class. Only one function, *public abstract* ***void accion(int event, Object datos);***
    - ControladorImp: It contains the implementation of the above function, it shows what should be done after each action performed by the user:
      public void accion(int event, Object datos) {
      FactoriaCommands factoriaComandos = FactoriaCommands.getInstance();
      Command commando = factoriaComandos.getCommand(event); [Returns an executable object (Command) given an event.]
      Contexto contexto = comando.execute(datos); [Executes the command (action or not) and returns a Contexto, containing an event and datos.]
      Dispatcher dispatcher = Dispatcher.getInstance();
      dispatcher.accion(contexto); [Performs the required action on the view, making it act as the Contexto dictates.]
    - PareadoQuery: Simple Query Transfer-like object, with 2 attributes (*private int primerObjeto, segundoObjeto*).
- dispatcher
    - Dispatcher: Singleton class, contains instance as attribute, *getInstance()* function and ***public abstract void accion(Contexto contexto);***
    - DispatcherImp: Implements the above function, it consists on a switch statement (***switch (contexto.getEvent())***) with the event. It will call the needed View (*VistaBicicleta* for instance) in the following way:
      case ListaComandos.VENTANAALTABICICLETACARRETERA:
      case ListaComandos.MOSTRARALTABICICLETA:
      …
      VistaBicicleta.getInstance().update(contexto); break;

It will only call the appropriate view in each situation for it to deal with the Context and update accordingly.

- vistas
  - vistaBicicleta
    - bicicleta
      - JFrame(AltaBicicletaCarretera/AltaBicicletaMontana/BajaBicicleta/BotoneraBicicleta/CantidadBicicletasQuery/DetalleBicicleta/ListaBicicleta/ModificarBicicletaCarretera/ModificarBicicletaMontana): ***Extends JFrame***. Contains the necessary elements to compose each window (*JTextField, JButton, JTextArea* mainly), and their layout, as well as the *cerrar()* function to close the window and an embedded class (*ActionListener* implementation) as follows:
        JButton enviar;
        JButton salir;
        enviar.addActionListener (new ActionListenerAltaBicicletaCarretera());
        ----
        salir.addActionListener (new ActionListener() {
        @Override
        public void actionPerformed (ActionEvent e) { cerrar() });
        ----
        public class ActionListenerAltaBicicletaCarretera implements ActionListener {
        public void actionPerformed(actionEvent arg0) {
        //Try-catch block with the actions to perform, usually calling an
        //action on the Controller:
        Controlador.getInstance().accion(ListaComandos.NEGOCIOALTABICICLETA, transferBicicleta); [Transfer object parsed from the fields]
        }
        Only one *ActionListener* is needed for *JFrameBotoneraXXX*. A different argument is used for each button and then captured on a switch statement inside the *ActionListenerBotoneraBicicleta* (embedded as well). For instance, button *JButton alta,* would work as follows:
        alta.addActionListener(new ActionListenerBotoneraBicicleta(X)); //X is an arbitrary number.
    - VistaBicicleta: Singleton class. Only contains one extra function:
      public abstract void update(Contexto contexto);
    - VistaBicicletaImp: Contains a different class attribute for each possible window (*JFrameDetalleBicicleta, JFrameBajaBicicleta…*), functions to initialize these (*new JFrameDetalleBicicleta;…* functions called in constructor), and an implementation of the *update(Contexto contexto)* function, modifying the windows as follows:
      -It is a *switch statement.* [***switch (contexto.getEvent())***]
      -If the event is a command of the type *ListaComandos.VENTANAXXX,* it doesn't take the *datos* field into account and only uses the *setVisible* function. However, if it is of the type *ListaComandos.MOSTRARXXX,* it does different things depending on the data given (Calls a function to print the result message given a code in the case of *MOSTRARBAJABICICLETA,* print the assigned ID or failure message in the case of *ALTA*, call to print the *ArrayList<TransferBicicleta>* in the case of Lista, etc.) and might show a MessageDialog (*JOptionPane.showMessageDialog(…)*) with the information in some cases.

- vistaCliente
  - cliente
    - JFrame(Same as above)
  - VistaCliente
  - VistaClienteImp
- vistaCompra
  - compra
    - JFrame(Same as above)
      JFrameGestionCarrito contains a TransferCompra object to store the "Shopping Cart" before inserting the items into the database!! This only happens after clicking on *JButton finalizarCompra*
  - VistaCompra
  - VistaCompraImp
- vistaDepartamento
  - departamento
    - JFrame(Same as above)
  - VistaDepartamento
  - VistaDepartamentoImp
- vistaEmpleado
  - empleado
    - JFrame(Same as above)
  - VistaEmpleado
  - VistaEmpleadoImp
- vistaPrincipal
  - principal
    - JFrame(Same as above)
  - VistaPrincipal
  - VistaPrincipalImp: Simple and clean class, to see and understand the others more easily. (*VistaXXXImp*) Shows the Main Screen after receiving the signal through the *update(Contexto contexto)* method: *principal.setVisible(true)*; inside the *update* function.
- vistaTienda
  - tienda
    - JFrame(Same as above)
  - VistaTienda
  - VistaTiendaImp

- ListaComandos: A class with every possible Command in the following format: ***public static final int NAMEOFTHECOMMAND = XXX*** (3 digit unique code) NAMEOFTHECOMMAND must start with either VENTANA-, NEGOCIO- or MOSTRAR-. (NEGOCIOALTABICICLETA)