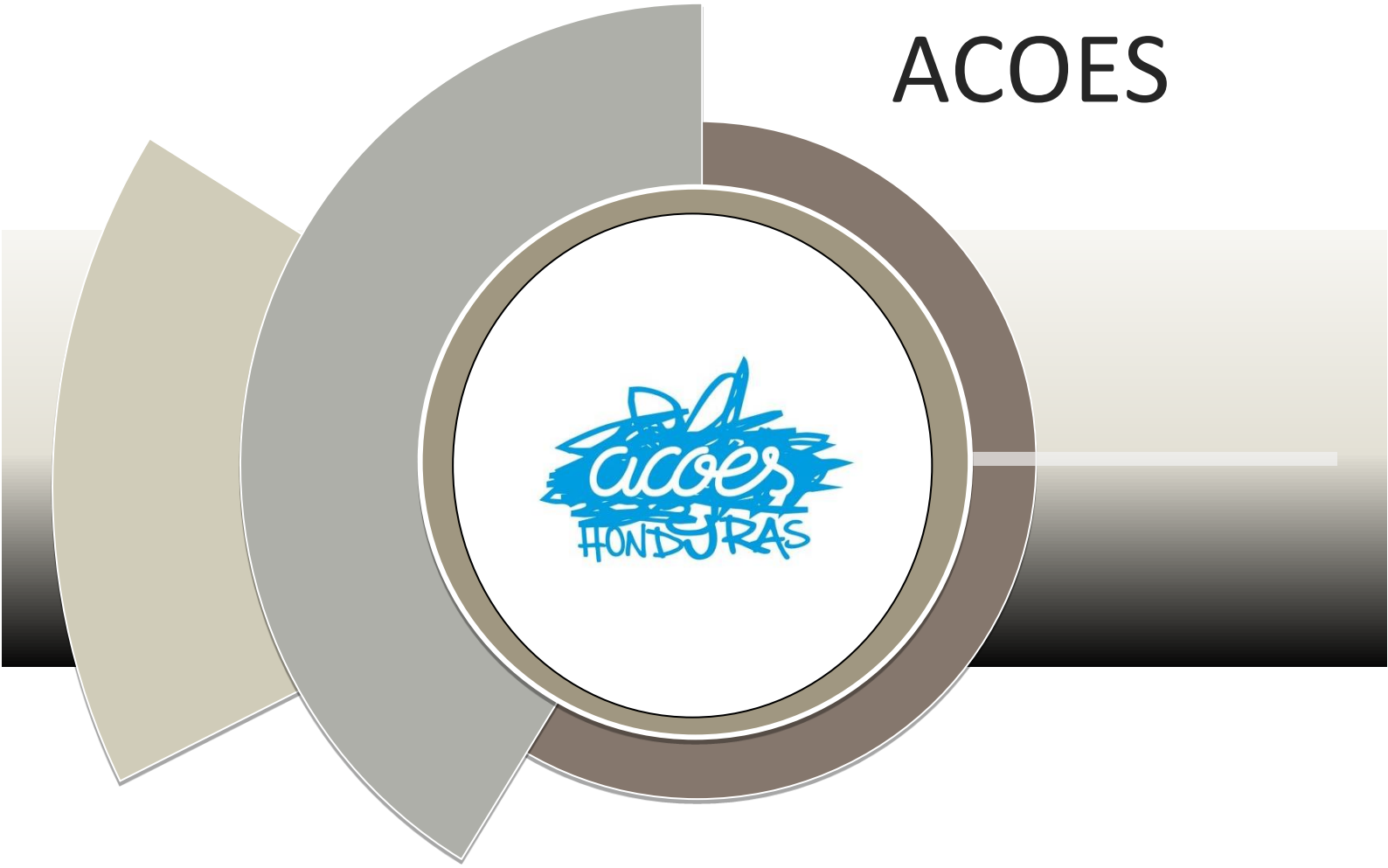
ACOES



PROYECTO FINAL ACOES

SISTEMAS DE INFORMACIÓN PARA INTERNET

ACOES

Contenido

MIEMBROS DEL GRUPO

Andrés Valentín Suárez Mediavilla - 77425032V

Sergio González Sicilia - 77225653W

Ignacio Pascual Gutiérrez - 54236255R

Katia Moreno Berrocal - 79038916Z

URL Repositorio (los archivos finales se encuentran en la carpeta: Tarea 3):

https://github.com/Lilithsoul/ACOES

Commit final del proyecto:

https://github.com/Lilithsoul/ACOES/commit/b4ef0d0bab9c65d01e64a93e66fc87a22b964e9e

Dicho commit es la última modificación del proyecto, la carpeta del proyecto ACOES también se encuentra dentro de la tarea 3, llamada ACOES.

INTRODUCCIÓN.

Este documento está elaborado con el propósito de actualizar y presentar toda la información referente al proyecto ACOES, incluyendo modificaciones respecto a las tareas anteriores.

Dichas modificaciones son respecto a la anterior entrega, situada en Tarea 2.

Cambios respecto a las entregas anteriores.

Modificación de los requisitos:

Se encuentran en el DGR dentro de Tarea 3. Commit correspondiente:

https://github.com/Lilithsoul/ACOES/commit/9135560198f78eb1113a8cd8cde7d645e481aaf4

Modificación de las entidades JPA.

Entidad: USUARIO

```
@Entity
@Inheritance( strategy = InheritanceType.JOINED )
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class USUARIO implements Serializable {

    @XmlTransient
    private static final long serialVersionUID = 1L;
    @Id
    private String User_name;

    @Column (nullable = false)
    private String Nombre;
    @Column (nullable = false)
    private String Apellido1;
    private String Apellido2;
    @Column (nullable = false)
    private String email;
    private String Telefono;
    @Column (nullable = false)
    private String Password;
    @Column (nullable = false)
    private String NIF;
    @Temporal(TemporalType.DATE)
    @Column (nullable = false)
    private Date FechaNacimiento;

    private String num_cuenta_bancaria;
    private String direccion;
    @Column (nullable = false)
    private int apadrinados;
    private Role role = Role.USUARIO;
    @OneToMany(mappedBy="socio_envio")
    private List<ENVIOS> envios_enviados;
    @OneToMany(mappedBy="socio_apadrina")
    private List<HISTORIAL_APADRINAMIENTO> historiales_pert;
```

```
 */
public enum Role {
    USUARIO,
    SOCIO,
    ADMINISTRADOR
}
```

La entidad Usuario ha sido modificada debido a que el grupo ha elegido eliminar la entidad de ADMINISTRADOR y SOCIO. Estas dos últimas entidades extendían de USUARIO.

En lugar de usar este método, hemos optado por tener un Enumerado llamado Role compuesta por los siguientes roles: USUARIO, SOCIO y ADMINISTRADOR.

La entidad ADMINISTRADOR tenía información sobre el número de seguridad social, fecha cuando empezó a trabajar y la nómina que percibía. Pensando lo que se pedía en el trabajo, estos campos eran totalmente innecesarios dado que en esta aplicación se nos pedía un gestor de niños, apadrinamiento y escolarización, no una aplicación de recursos humanos.

Estas dos entidades son las que usamos en la entrega anterior y han sido cambiadas.

Entidad: ENVIOS

```
@Entity
public class ENVIOS implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String Descripcion;

//   @Column(nullable=false)
    private Estado Estado;

    @ManyToOne
    private JOVEN_NIÑO niño_envio;

    @ManyToOne  //
    private USUARIO socio_envio;
```

```
public enum Estado {
    PENDIENTE,
    RECIBIDO,
    ENVIADO
}
```

Al igual que como nos pasaba en la entidad de usuario, hemos tenido que crear otro Enumerado llamado Estado para poder controlar cómo está actualmente.

Modificación de las vistas XHTML

En esta entrega hemos visto necesario añadir las diferentes vistas:

Vistas por cada excepción existente.

Dado que pueden existir distintas excepciones, ej: apadrinar un niño cuando no hay disponibles, error eliminando un ccj, etc. Hemos visto necesario cambiar el mensaje que aparecía en mitad de la pantalla a algo un poco más estético, una vista a la que se te redirige cuando se intenta usar un método y da fallo.

Vista: error<nombre>.xhtml

Así mismo, como pueden surgirmás errores como errorApadrinamiento, añadimos las siguientes vistas: errorApadrinamiento.xhtml, errorEliminaciónCCJ.xhtml, errorEliminaciónNinio.xhtml y errorEliminaciónPopulorum.xhtml.

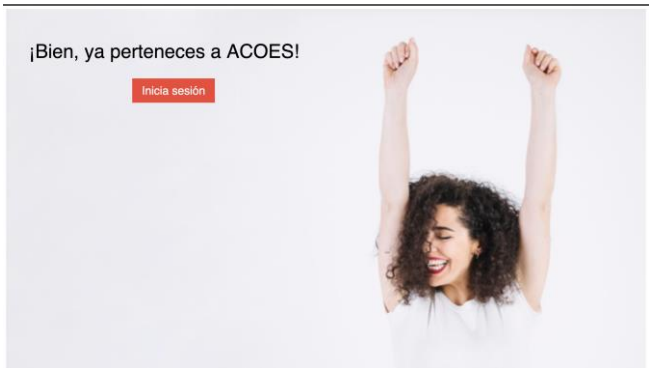
Vista: error errorApadrinamiento.xhtml.

Ninios.xhtml

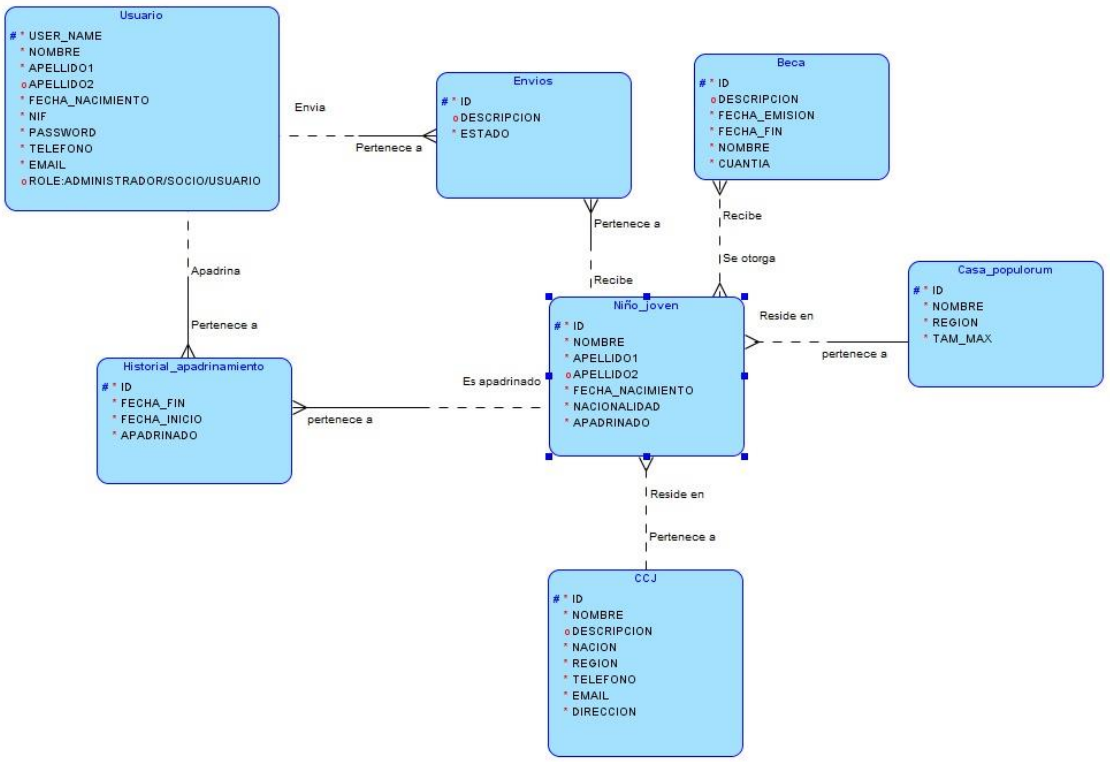
Viendo que era un caos el uso de varias tablas dentro de una misma, decidimos que en la vista ninios.xhtml aparezca una información resumida de los niños que están siendo apadrinamos por un socio en ese momento. Además, si dicho socio desea ver más información del niño en cuestión, solamente tendrá que pulsar sobre el botón: detalles. Detalles redirige a una página con la información del niño en cuestión y además podrá realizar el envío desde ahí.

Vista: exitoRegistro.xhtml

Añadimos esta vista para que el usuario se cerciore de que ha sido registrado con éxito en la web. Antes redirigía a la página Home-xhtml.



El esquema lógico resultante sería:



Modificación de los controladores de las vistas

Hemos optado por un modelo-vista-controlador. Los controladores de las vistas son las siguientes:

Becas.java

Esta clase contiene todos los controladores relacionado con el manejo de las becas. Sus métodos son: añadir becas, eliminar becas, refrescar la página, ejecutar acción becas (añadir, modificar) para los botones, comprobar si el rol del que intenta acceder una acción es autorizado.

CCJs.java

Al igual que la clase anterior, contiene métodos idénticos para los CCJs.

Control.java

Su función principal consiste en devolver las páginas que se pidan. Por ejemplo:

```
public String ccj(){
    return "CCJ.xhtml";
}
```

En este caso nos devolvería la página de acoes cuando lo llamemos mediante algún action="…" en la vista pertinente.

Envios.java

Contiene todos los métodos necesarios para interactuar con envíos, alguno de los métodos son: eliminar, modificar, insertar envío, refrescar la página (por si añadimos alguno y no aparece), modificar los estados del envío (solo disponibles para administrador), etc.

Historiales.java

Se dedica a implementar todos los métodos relacionados con apadrinar, tanto del manejo del historial (lista de apadrinamiento), como la obtención de los padrinos.

InfoSesion.java

Su función es devolver los datos que se le piden, como: historiales, usuarios, etc.

Login.java

Comprueba que el usuario introducido sea el correcto.

Ninios.java

Es idéntica a CCJ y Becas, pero usando métodos especializados a los niños.

Populorum.java

Como lo dicho anteriormente con ninios.java, usa los mismos métodos, pero personalizados en los populorum.

Registro.java

Se dedica a comprobar que los campos introducidos sean correctos y completos. Comprueba que las contraseñas coincidan (contraseña y repetir contraseña) y que el usuario no esté registrado ya. El método registerr() añade el usuario a la base de datos. Finalmente registrar socio añade un socio a partir de un usuario cuando se añaden los campos cuenta bancaria y dirección.

Descripción de los EJB.

A continuación, vamos a describir los EJB implementados en la capa de negocio, la cual nos sirve para conectar la capa de acceso a datos (JPA) y la capa de presentación (JSF). Para ello tenemos una interfaz Negocio y una clase NegocioImpl que implementa los métodos que se definen en Negocio, los cuáles son:

```
@Override
public void registrarUsuario(USUARIO u) throws ACOESException {
    USUARIO user = em.find(USUARIO.class, u.getUser_name());
    if (user != null) {
        // El usuario ya existe
        throw new CuentaRepetidaException();
    }

    em.persist(u);

}
```

Este método registra un usuario haciendo un em.persist() del usuario pasado por parámetro en el caso de que no exista en la BD. En el caso de que el método em.find() encuentre el Usuario (es decir, que esté registrado) saltará una excepción.

```
@Override
public void compruebaLogin(USUARIO u) throws ACOESException {
    USUARIO user = em.find(USUARIO.class, u.getUser_name());
    if (user == null) {
        throw new CuentaInexistenteException();
    }

    if (!user.getPassword().equals(u.getPassword())) {
        throw new ContraseniaInvalidaException();
    }
}
```

Este método comprueba que el usuario ha introducido sus credenciales correctamente (buscando en la BD al Usuario por su clave primaria (user_name en este caso)). En el caso de no haberlo encontrado, saltará una excepción. Además, comprueba que la contraseña introducida coincide con la guardada en la BD, en caso contrario, saltará otra excepción. Este método sirve para garantizar la integridad a la capa de negocio, por ello lo llamamos en otros métodos.

```
public void compruebaLoginAdmin(USUARIO u) throws ACOESException;
public void compruebaLoginSocio(USUARIO u) throws ACOESException;
public void compruebaLoginSocioAdmin(USUARIO u) throws ACOESException;
```

Al igual que compruebaLogin, implementamos estos tres métodos para comprobar cuando un usuario está logeado como Admin, Socio y Admin o Socio respectivamente.

```
@Override
public USUARIO refrescarUsuario(USUARIO u) throws ACOESException {
    compruebaLogin(u);
    USUARIO user = em.find(USUARIO.class, u.getUser_name());
    em.refresh(user);
    return user;

}
```

A este método se le pasa por parámetro el usuario de la sesión, comprueba que existe en la BD y le hace un refresh();

```
@Override
public void usuarioSocio(USUARIO u) throws ACOESException {
    compruebaLogin(u);
    em.merge(u);
}
```

Mediante la variable 'u' se llama a compruebaLogin para certificar que el usuario está logeado actualmente en la página. A partir de dicho Usuario se accede a la base de datos y se actualiza el valor del Usuario asociado.

```
    @Override
public USUARIO getSocioID(String socio) {
    USUARIO res = em.find(USUARIO.class, socio);
    if(res != null){
    return res;
    }
    return null;
}
```

A partir de la clave primaria del SOCIO, en este caso el username de su sesión, se obtiene el SOCIO asociado a dicho username. En caso de no encontrar ningún SOCIO con el username asociado, se devuelve null.

Métodos asociados a los CCJ:

```
@Override
public void modificarCCJ(CCJ c, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.merge(c);
}
```

A partir de una variable CCJ se accede a la base de datos y se actualiza el valor del CCJ asociado. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public void insertarCCJ(CCJ c, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.persist(c);
}
```

A partir de una variable CCJ se accede a la base de datos y se inserta el valor del CCJ. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public void eliminarCCJ(CCJ c, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.remove(em.merge(c));
}
```

A partir de una variable CCJ se accede a la base de datos y se elimina el valor asociado a este. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public CCJ getCCJID(Long id) {
    CCJ res = em.find(CCJ.class, id);
    if(res != null){
    return res;
    }
    return null;
}
```

A partir de la clave primaria de CCJ se obtiene el CCJ asociado dentro de la base de datos. Si el valor no se encuentra en la base de datos se devuelve null.

```
@Override
public List<CCJ> obtenerCCJs() throws ACOESException{
    return em.createNamedQuery(CCJ.FIND_ALL, CCJ.class).getResultList();
}
```

Devuelve todos los CCJ almacenados en la base de datos mediante una consulta JPQL. Este método es utilizado para mostrar al administrador todos los CCJ.

```
@Override
public List<CCJ> CcjNinio(USUARIO u){
    List<HISTORIAL_APADRINAMIENTO> h = u.getHistoriales_pert();
    List<CCJ> c = new ArrayList<>();
    for (HISTORIAL_APADRINAMIENTO hist : h) {
        if(hist.getNiño_apadrinado().getCcj() != null){
            if(!c.contains(hist.getNiño_apadrinado().getCcj())){
                c.add(hist.getNiño_apadrinado().getCcj());
            }
        }
    }
    if(c.isEmpty()){
        return null;
    }
    return c;

}
```

Este método devuelve una lista de CCJ asociadas a los niños apadrinados por el usuario que se le pasa por parámetro, evitando así que tenga acceso a todas las casas ccj existentes y solamente permitiendo su visualización a las que son de su interés. Creamos un arrayList de CCJ c, y para mostrar ccjs sin repeticiones añadimos a la lista solamente los que no estén ya.

Métodos asociados a las Casas Populorum:

```
@Override
public void modificarPopulorum(CASA_POPULORUM c, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.merge(c);
}
```

A partir de una variable CASA_POPULORUM se accede a la base de datos y se actualiza su valor. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public void insertarPopulorum(CASA_POPULORUM c, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.persist(c);
}
```

A partir de una variable CASA_POPULORUM se accede a la base de datos y se inserta el valor de CASA_POPULORUM. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public void eliminarPopulorum(CASA_POPULORUM c, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.remove(em.merge(c));
}
```

A partir de una variable CASA_POPULORUM se accede a la base de datos y se elimina el valor de CASA_POPULORUM. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public List<CASA_POPULORUM> obtenerPopulorums() throws ACOESException{


    return em.createNamedQuery(CASA_POPULORUM.FIND_ALL, CASA_POPULORUM.class).getResultList();
}
```

Se llama a la base de datos y se obtienen todas las CASAS POPULORUM dentro de ella mediante una consulta JPQL. Este método es utilizado para mostrar al administrador todas las casas populorum.

```
@Override
public CASA_POPULORUM getCPID(Long populorum) {
    CASA_POPULORUM res = em.find(CASA_POPULORUM.class, populorum);
    if(res != null){
    return res;
    }
    return null;
}
```

A partir de la clave primaria de CASA_POPULORUM (id), se accede a la base de datos y se obtiene la CASA_POPULORUM asociada. Si el valor no se encuentra se devuelve null. Se utiliza en los registros de niños para, a partir del id, obtener la casa populorum asociada.

```
@Override
public List<CASA_POPULORUM> PopulorumNinio(USUARIO u){
    List<HISTORIAL_APADRINAMIENTO> h = u.getHistoriales_pert();
    List<CASA_POPULORUM> c = new ArrayList<>();
    for (HISTORIAL_APADRINAMIENTO hist : h) {
        if(hist.getNiño_apadrinado().getCp() != null){
            if(!c.contains(hist.getNiño_apadrinado().getCp())){
                c.add(hist.getNiño_apadrinado().getCp());
            }
        }
    }
    if(c.isEmpty()){
        return null;
    }
    return c;

}
```

Este método devuelve una lista de CASA_POPULORUM asociadas a los niños apadrinados por el usuario que se le pasa por parámetro, evitando así que tenga acceso a todas las casas populorum existentes y solamente a las que son de su interés. Creamos un arrayList de CASA_POPULORUM c, y para mostrar casas populorum sin repeticiones añadimos a la lista solamente los que no estén ya.

Métodos asociados a las Becas:

```
@Override
public void modificarBeca(BECA c, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.merge(c);
}
```

A partir de una variable BECA se accede a la base de datos y se actualiza el valor de BECA asociado. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public void insertarBeca(BECA c, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.persist(c);
}
```

A partir de una variable BECA se accede a la base de datos y se inserta el valor de BECA. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public void eliminarBeca(BECA c, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.remove(em.merge(c));
}
```

A partir de una variable BECA se accede a la base de datos y se elimina el valor de BECA. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public List<BECA> obtenerBecas() throws ACOESException{

    return em.createNamedQuery(BECA.FIND_ALL, BECA.class).getResultList();
}
```

Se llama a la base de datos y se obtienen todas las BECAS mediante una sentencia JPQL. Este método se utiliza para que al administrador se le muestren todas las becas.

```
@Override
public BECA getBecasID(Long beca) {
    BECA b = em.find(BECA.class, beca);
    if(b != null){
    return b;
    }
    return null;
}
```

A partir de la clave primaria de BECA (id), se accede a la base de datos y se obtiene la BECA asociada. Si el valor no se encuentra se devuelve null. Se utiliza en los registros de niños para a partir del id obtener la beca.

Métodos asociados a los Niños o Historiales:

```
@Override
public void modificarNinio(JOVEN_NIÑO ninio, USUARIO u) throws ACOESException {

    compruebaLoginAdmin(u);

    em.merge(ninio);


}
```

A partir de una variable JOVEN_NIÑO se accede a la base de datos y se actualiza el valor del JOVEN_NIÑO asociado. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public void insertarNinio(JOVEN_NIÑO ninio, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.persist(ninio);

}
```

Gracias al persist tomamos la instancia de JOVEN_NIÑO pasada por parámetro, se añade al contexto de persistencia y hace que la instancia sea manejada.

--

```
@Override
public void eliminarNinio(JOVEN_NIÑO ni, USUARIO u) throws ACOESException {

    compruebaLoginAdmin(u);
    em.remove(em.merge(ni));

}
```

A partir de una variable JOVEN_NIÑO se accede a la base de datos y se elimina el valor de JOVEN_NIÑO. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
public List<JOVEN_NIÑO> obtenerNinios(){

    return em.createQuery("SELECT c FROM JOVEN_NIÑO c", JOVEN_NIÑO.class).getResultList();
}
```

Creamos una Query seleccionando a todos los JOVEN_NIÑO dentro de la BD y hacemos un return de dicha lista. Dicho método sirve para que el administrador pueda ver a todos los niños en su pestaña correspondiente dentro de la página web y así administrarlos.

```
@Override
public List<HISTORIAL_APADRINAMIENTO> obtenerNiniosSocio(USUARIO u) {
    USUARIO user = em.find(USUARIO.class, u.getUser_name());

    return user.getHistoriales_pert();
}
```

Buscamos al Usuario pasado por parámetro dentro de la BD gracias al método em.find() y devolvemos las filas pertenecientes a su Historial como SOCIO.

Dicho método sirve para que el SOCIO pueda ver a todos los niños apadrinados y no apadrinados en su pestaña correspondiente dentro de la página web y así administrarlos.

```
@Override
public JOVEN_NIÑO getNinioID(Long ninio){
    JOVEN_NIÑO res = em.find(JOVEN_NIÑO.class, ninio);
    if(res != null){
    return res;
    }
    return null;
}
```

A partir de la clave primaria del JOVEN_NIÑO, se obtiene el niño asociado a ella. En caso de no encontrar ningún JOVEN_NIÑO con el id asociado, se devuelve null.

```
@Override
public void insertarHistorial(HISTORIAL_APADRINAMIENTO historial) throws ACOESException {
    try{
    Query selectQuery = em.createQuery("select q from JOVEN_NIÑO q where q.apadrinado = 0", JOVEN_NIÑO.class);
    selectQuery.setMaxResults(1);
    JOVEN_NIÑO n = (JOVEN_NIÑO)selectQuery.getSingleResult();

    historial.setNiño_apadrinado(n);
    n.setApadrinado(true);

    USUARIO u = historial.getSocio_apadrina();
    u.setApadrinados(u.getApadrinados()+1);

    Calendar cal = Calendar.getInstance();
    Date fechaInicio = cal.getTime();
    cal.add(Calendar.MONTH, 6);
    Date fechaFin = cal.getTime();
    historial.setFecha_inicio(fechaInicio);
    historial.setFecha_fin(fechaFin);
    historial.setApadrinado(true);
    em.merge(u);
    em.merge(n);
    em.persist(historial);
    }catch (Exception e){
        try {
            throw new Exception(e.getCause());
        } catch (Exception ex) {
            Logger.getLogger(NegocioImpl.class.getName()).log(Level.SEVERE, null, ex);
        }

    }
}
```

Este método es llamado cada vez que un usuario quiera apadrinar a un niño. El método hará una consulta JPQL de los niños que no estén apadrinados, y mediante el uso del método query.setMaxResults(1) obtendremos un único resultado, el cual se almacenará en un nuevo JOVEN_NIÑO. Se realizan las actualizaciones correspondientes, en el historial se introduce el niño obtenido, el niño se actualiza a 'apadrinado' y el número de niños apadrinados del socio que apadrina se incrementará en 1. Aparte, para poder establecer el resto de los componentes de un historial, obtenemos como fecha de inicio de apadrinamiento la del sistema y la fecha fin serán 6 meses más que la fecha de inicio. Se realizan los merge correspondientes al usuario, al historial y al niño.

```
@Override
public String getPadrinoNegocio(JOVEN_NIÑO n) {
    if(n.isApadrinado()){
    Query selectQuery = em.createQuery("select q from HISTORIAL_APADRINAMIENTO q "
        + "where q.niño_apadrinado.id = :id and q.apadrinado = 1",
        HISTORIAL_APADRINAMIENTO.class);

    selectQuery.setParameter("id", n.getId());
    HISTORIAL_APADRINAMIENTO res = (HISTORIAL_APADRINAMIENTO)selectQuery.getSingleResult();
    USUARIO u = res.getSocio_apadrina();

    return u.getUser_name();

    }
    return "No";
}
```

Este método obtendrá el username del padrino asociado a un JOVEN_NIÑO que se le pasa por parámetro. Primero se comprobará que el niño este apadrinado para poder realizar la consulta JPQL. Nos sirve a la hora de mostrar los niños en el rol ADMINISTRADOR, pues así reflejaremos

el padrino del niño en caso de estar apadrinado, y en otro caso se mostrará “No” para indicar que no tiene padrino.

```
@Override
public void refrescarApadrinamientos(){
    List<HISTORIAL_APADRINAMIENTO> h = em.createQuery("select q from HISTORIAL_APADRINAMIENTO q ",
    HISTORIAL_APADRINAMIENTO.class).getResultList();

    Calendar cal = Calendar.getInstance();
    Date fechaHoy = cal.getTime();

    for (HISTORIAL_APADRINAMIENTO hist : h) {
            if(fechaHoy.compareTo(hist.getFecha_fin())>0){
                hist.setApadrinado(false);
                hist.getNiño_apadrinado().setApadrinado(false);
                hist.getSocio_apadrina().setApadrinados(hist.getSocio_apadrina().getApadrinados()-1);
                em.merge(hist.getNiño_apadrinado());
                em.merge(hist.getSocio_apadrina());
                em.merge(hist);

            }
    }
}
```

Este método, llamado a través de un botón ‘refrescar apadrinamientos’, se encarga de modificar los parámetros correspondientes cuando un apadrinamiento se ha terminado, es decir, comprobando si la fecha_fin del apadrinamiento es menor a la del sistema, el niño apadrinado dejará de estarlo, el socio tendrá 1 apadrinado menos y estos cambios se verán reflejados en dicho historial de apadrinamiento.

```
@Override
public List<HISTORIAL_APADRINAMIENTO> getHistorialesTotales() {
    Query selectQuery = em.createQuery("select q from HISTORIAL_APADRINAMIENTO q where q.apadrinado = 1", HISTORIAL_APADRINAMIENTO.class);
    return selectQuery.getResultList();
}
```

Este método, el cual será llamado desde un rol ‘ADMINISTRADOR’, consulta todos los historiales de apadrinamiento existentes en la BD. Se ha implementado mediante una consulta JPQL.

Métodos asociados a los Envíos:

```
@Override
public void modificarEnvio(ENVIOS e, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.merge(e);
}
```

A partir de una variable ENVIOS se accede a la base de datos y se actualiza el valor de ENVIOS asociado. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public void insertarEnvio(ENVIOS e, USUARIO u) throws ACOESException{

    compruebaLoginSocioAdmin(u);
    e.setEstado(Estado.PENDIENTE);
    em.persist(e);
}
```

A partir de una variable ENVIOS se le fija el valor de estado de forma predeterminada y se añade a la base de datos. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public void eliminarEnvio(ENVIOS e, USUARIO u) throws ACOESException{

    compruebaLoginAdmin(u);
    em.remove(em.merge(e));


}
```

A partir de una variable ENVIO se accede a la base de datos y se elimina el valor de ENVIO. Mediante la variable 'u' se llama a compruebaLoginAdmin para certificar que el usuario que esta interactuando posee el rol ADMINISTRADOR.

```
@Override
public List<ENVIOS> obtenerEnviosSocio(USUARIO u) {
    USUARIO user = em.find(USUARIO.class, u.getUser_name());

    return user.getEnvios_enviados();
}
```

Este método obtendrá todos los envíos realizados por un usuario (SOCIO) y nos servirá para poder mostrarle sus envíos en la pestaña de Niños apadrinados, evitando así que tenga acceso a todos los envíos de la base de datos y solamente a los de su interés.

```
@Override
public List<ENVIOS> obtenerEnviosTotales() {
    Query selectQuery = em.createQuery("select q from ENVIOS q ", ENVIOS.class);
    return selectQuery.getResultList();

}
```

Este método, el cual será llamado desde un rol 'ADMINISTRADOR', consulta todos los envíos existentes en la BD. Se ha implementado mediante una consulta JPQL.

```
@Override
public List<JOVEN_NIÑO> obtenerNiniosSinPadrino() {
    return em.createQuery("SELECT c FROM JOVEN_NIÑO c where c.apadrinado = 0", JOVEN_NIÑO.class).getResultList();
}
```

Este método nos es útil para comprobar que, a la hora de apadrinar un nuevo niño, existan niños sin apadrinar disponibles en la base de datos y así poder asignar un nuevo apadrinamiento a un usuario.