Hugo Aguilera Sanz
Amaia Echenagusia Muñoz
Calixto Sánchez-Fresneda Redondo
Javier Ramírez de Andrés
Antonio Trenado Martín

Contenido

1. Estructura del proyecto

Para este proyecto debido a la naturaleza evolutiva del software, hemos usado un modelo de proceso evolutivo. Esto nos ha permitido construir el programa repitiendo una secuencia iterativa de pasos hasta acabar con el producto final. Gracias a esto pudimos obtener módulos operacionales definitivos de distintas partes del sistema.

Consideramos que para este proyecto no necesitaríamos prototipos ya que nos llevaría más tiempo para obtener el mismo resultado. Al prescindir de éstos, tuvimos una mayor comunicación con nuestro cliente, adaptándonos a todas sus directrices y cambios de requisitos del producto.

Aunque las fechas de entrega para este modelo de proyecto son más bien orientativas, nos hemos adaptado de manera estricta a la impuesta por el cliente, el día 3 de mayo de 2023.

Dentro del proceso evolutivo del software nos decidimos por la Espiral de Boston:

- La comunicación con el cliente.
- La planificación.
- El análisis de riesgo.
- Las representaciones de la aplicación.
- La construcción y adaptación.
- La evaluación por el cliente.

Un factor importante también ha sido el equipo ya que al no conocernos todos y no saber mucho sobre la asignatura uno de los miembros tubo que tomar las riendas del equipo convirtiéndose en el “jefe” ya que era repetidor y tenia mas conocimientos sobre la asignatura. También se organizó una reunión para dividirse el trabajo llegando al consenso de que se formarían dos equipos, uno de código formado por Antonio, Hugo y Javier; y otro de diagramas formando por Calixto y Amaia. Esta división no supuso que la gente que hacía código no tocase los diagramas de hecho los primeros modelos de diagramas de clase fueron hechos por Javier y Antonio. Posteriormente se creo un servidor en la aplicación Discord para poder comunicarse a la hora de realizar el código y los diagramas.

2. Modelo de Requisitos

Cuyo objetivo es delimitar y capturar la funcionalidad que debe ofrecer desde la perspectiva del usuario. Fue el primer modelo que se desarrolló y nos sirvió de base para el resto de los modelos. Este modelo contiene diagramas abstractos que representan el funcionamiento de la aplicación.

2.2 Modelos de comportamiento

Se refiere a los casos de uso y describen la funcionalidad que ofrece el sistema desde el punto de vista del usuario. Se encuentra dentro del documento SRS, un documento actualizado y corregido con respecto a la anterior entrega.

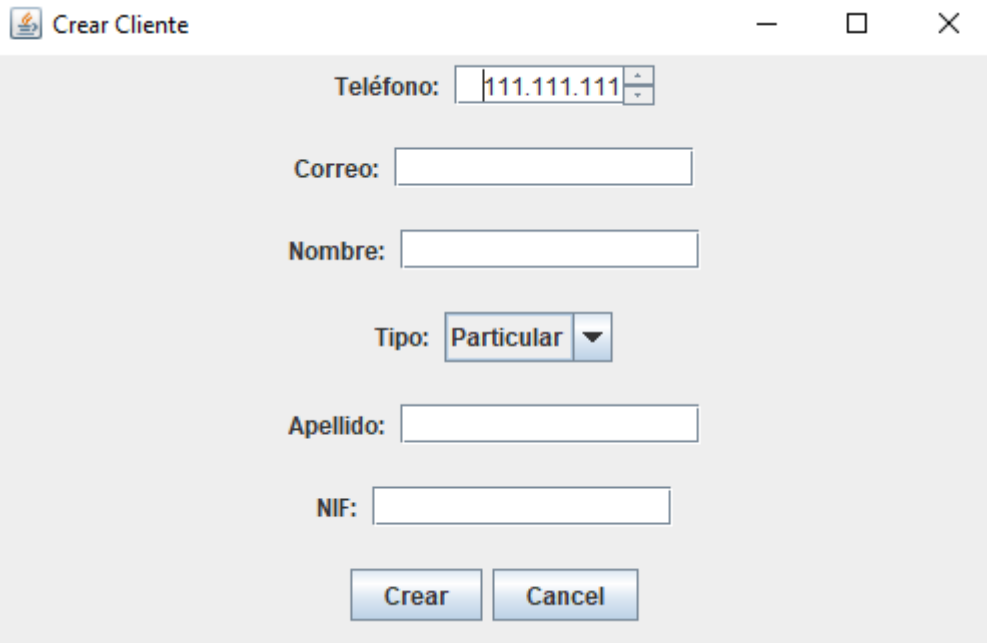
2.2 Modelos de presentación

Especifica la interacción del sistema con los actores externos al ejecutar los casos de uso, cómo se verán las interfaces gráficas y qué funcionalidad ofrecerá cada una de ellas.

Inicio:

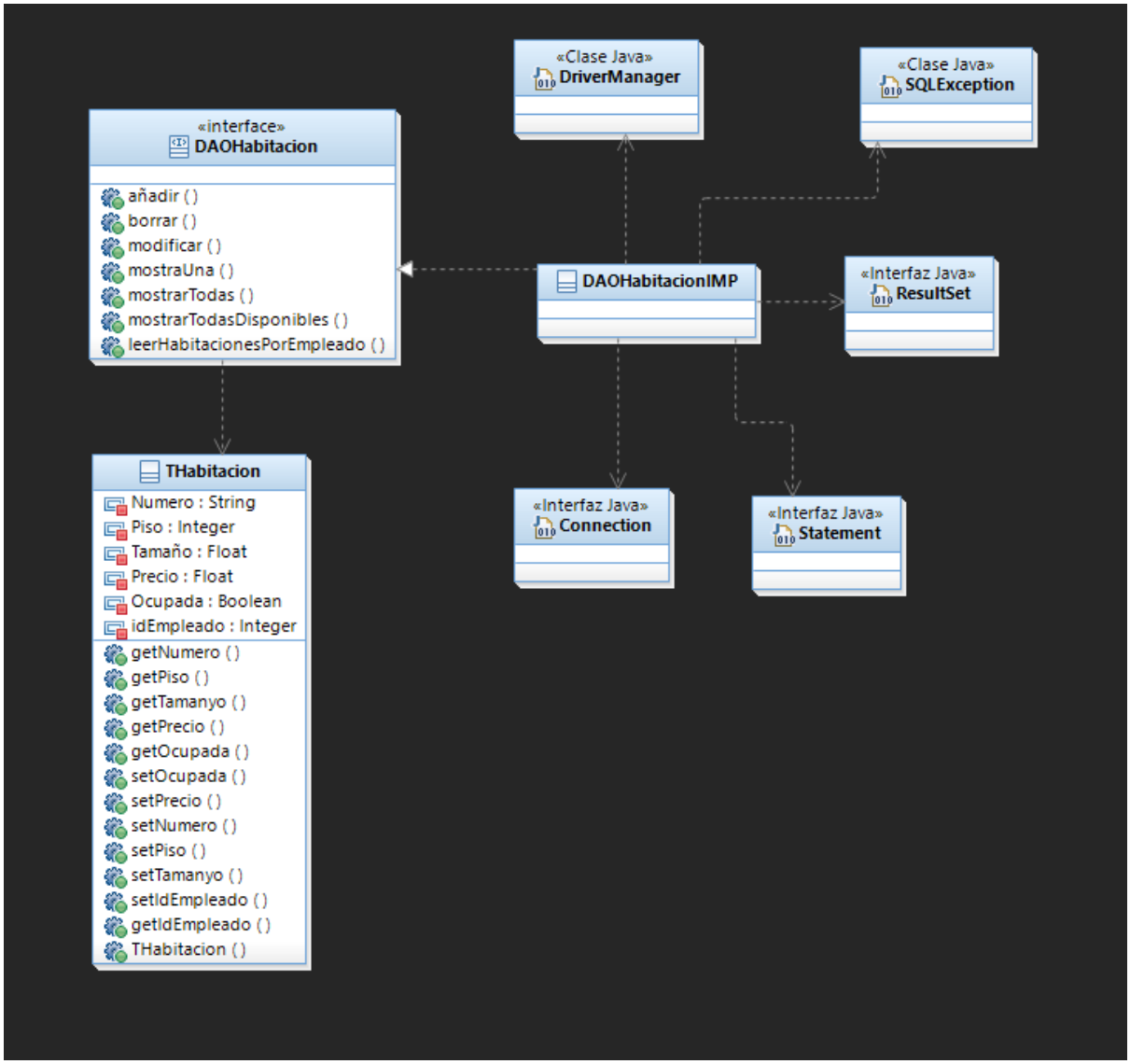
Una vista concreta:

La interfaz de una operación:



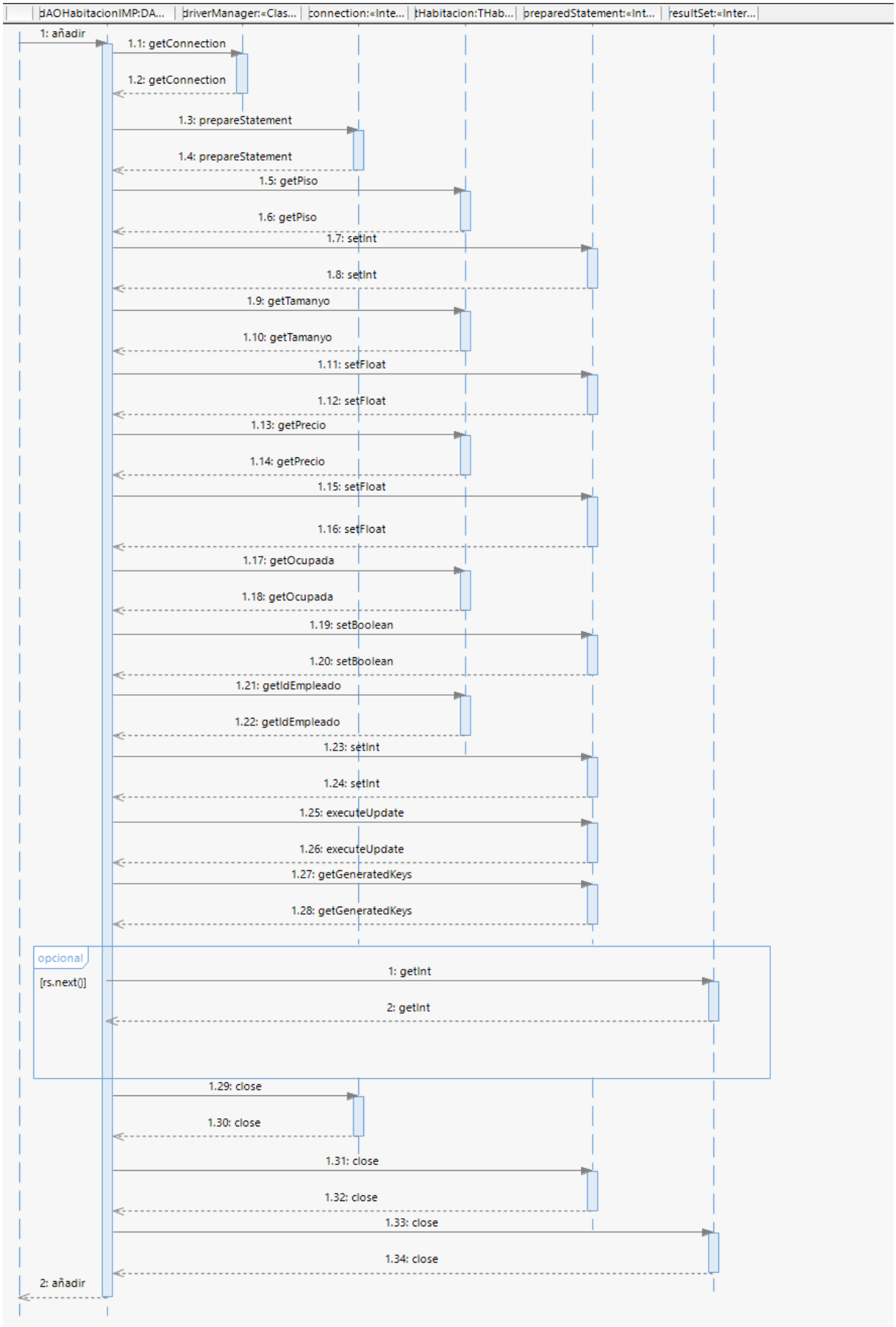
2.2 Modelos de información

Trata los aspectos estructurales del sistema con los diagramas de clase que se encuentran dentro del modelo.



3. Modelo de Diseño

Se encarga de mostrar, de forma gráfica, el comportamiento de cada caso de uso con la ayuda de los diagramas de secuencia, permitiendo así modelar la interacción entre objetos del sistema. Nos proporciona una visión más estrecha que define en detalle los requisitos para implementar el sistema.


dAOHabitacionIMP:DA...
driverManager:«Clas...
connection:«Inte...
tHabitacion:THab...
preparedStatement:«Int...
resultSet:«Inter...
1: añadir
1.1: getConnection
1.2: getConnection
1.3: prepareStatement
1.4: prepareStatement
1.5: getPiso
1.6: getPiso
1.7: setInt
1.8: setInt
1.9: getTamanyo
1.10: getTamanyo
1.11: setFloat
1.12: setFloat
1.13: getPrecio
1.14: getPrecio
1.15: setFloat
1.16: setFloat
1.17: getOcupada
1.18: getOcupada
1.19: setBoolean
1.20: setBoolean
1.21: getIdEmpleado
1.22: getIdEmpleado
1.23: setInt
1.24: setInt
1.25: executeUpdate
1.26: executeUpdate
1.27: getGeneratedKeys
1.28: getGeneratedKeys
opcional
[rs.next()]
1: getInt
2: getInt
1.29: close
1.30: close
1.31: close
1.32: close
1.33: close
1.34: close
2: añadir

4. Codificación del proyecto

Al habernos divido en dos equipos por falta de tiempo no pudimos tener los modelos para pasarlos a código así que mientras los programadores iban codificando los que hacían diagramas tenían que ir adaptándose a las nuevas funciones y parámetros de los transfers que se creaban.

Nuestra aplicación no podía ser sin una base de datos donde se guarde toda la información necesaria para su funcionalidad. Escogimos trabajar con las bases de datos de XAMMP que se ejecuta en local y se conecta de manera sencilla con la capa de integración desde el puerto 3306. Con esta base de datos MySQL podíamos realizar consultas de dos vías: desde la aplicación que estábamos desarrollando (ya conectada en los DAOs) y desde phpMyAdmin. El código desarrollado se divide en integración, negocio y presentación, cada uno con un patrón.

Sistema de aplicación, crear cliente:

```
public Integer crear(TCliente entradaCliente) {

    if(entradaCliente.getTelefono() == null){
        return -5;
    }

    String numero = Integer.toString(entradaCliente.getTelefono());
    // Patrón para validar el email
    Pattern pattern = Pattern.compile("^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@" + "[A-Za-z0-9-]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$");
    Matcher mather = pattern.matcher(entradaCliente.getCorreo());


    if(entradaCliente.getCorreo().trim().equals("")  || entradaCliente.getNombre().trim().equals("")){
        return -5; // los datos introducidos estan vacios mostrar complete los campos
    }
    if(entradaCliente.getApellidos() != null){
        if(entradaCliente.getApellidos().trim().equals("")){
            return -5;
        }
    }
    if(numero.length() != 9){
        return -6; // numero de longuitdud mayor
    }
    if(!mather.find()){
        return -7; //correo no valido
    }
    if(entradaCliente.getCIF() != null ){
        if(!CIFValido(entradaCliente.getCIF())){
            return -8; // cif invalido
        }
    }
    if(entradaCliente.getNIF() != null){
        if(!NIFValido(entradaCliente.getNIF())){
            return -9; // nif invalido
        }
    }


    //Creamos el dao
    DAOCliente daoCliente = FactoriaIntegracion.getInstance().newDAOCliente();

    if(entradaCliente.getId() != null){
        TCliente tCliente = daoCliente.MostrarUno(entradaCliente.getId());

        if(tCliente != null){
            daoCliente.modificar(entradaCliente);

            return tCliente.getActivo() ? -2 : entradaCliente.getId();
        }

    }

    System.out.println("llega al sa");

    return daoCliente.crear(entradaCliente);
}
```

5. Patrones empleados en el proyecto

5.1 Controlador

Este patrón es muy cercano al servicio de aplicación y se centra en la lógica de negocio, su implementación controla la gestión de los errores entre otras cosas que aparecen en la interfaz.

```java
public abstract class Controller {
    private static Controller instance;

    public static Controller getInstance() {
        if (instance == null) {
            instance = new ControllerImp();
        }
        return instance;
    }

    public abstract void carryAction(int event, Object data);
}
```

5.2 Patrón SA

Servicio de Aplicación, se centraliza en la lógica de negocio y la aplicamos a los objetos de negocio, conecta las capas de presentación e integración, mandando los fallos que recibe el SA si algo no está correcto.

```java
import java.util.Collection;

public interface SAEmpleado {

    public Integer crear(TEmpleados empleado);

    public Integer modificar(TEmpleados empleado);

    public Integer eliminar(int idEmpleado);

    public TEmpleados mostrarUno(int idEmpleado);

    public Collection<TEmpleados> mostrarTodos();

    public Collection<TEmpleados> mostrarPorDepartamento(Integer idDepartamento);

    public Integer vincular(TTareasDelEmpleado tTareasDelEmpleado);

    public Integer desvincular(TTareasDelEmpleado tTareasDelEmpleado);

    public Collection<TTareas> LeerLineasPedidoPorTareas(Integer idTareas);

    public Collection<TEmpleados> LeerLineasPedidoPorEmpleado(Integer idEmpleado);
}
```

5.3 Patrón DAO

Data Access Object, se utiliza para acceder a la base de datos. Proporciona acceso para el manejo de las peticiones de la capa de presentación y lo utilizamos en la capa de integración.

```java
import Negocio.Clientes.TCliente;

public interface DAOCliente {

    public Integer crear(TCliente tCliente);

    public Integer borrar(Integer id);

    public Integer modificar(TCliente tCliente);

    public TCliente MostrarUno(Integer id);

    public Collection<TCliente> MostrarTodos();

    public Collection<TParticular> MostrarParticular();

    public Collection<TEmpresa> MostrarEmpresa();

}
```

5.4 Patrón Transfer

Independiza el cambio de datos entre capas y evita que una capa tenga que conocer la representación de las entidades

```java
package Negocio.Habitaciones;

public class THabitaciones {

    private Integer numero;
    private Integer piso;
    private Integer tamaño;
    private Float precio;
    private Boolean ocupada;
    private Integer id_empleado;

    public THabitaciones(Integer numero, Integer piso, Integer tamaño, Float precio, Boolean ocupada, Integer id_empleado){ //tamaño = numero personas

        this.numero = numero;
        this.piso = piso;
        this.tamaño = tamaño;
        this.precio = precio;
        this.ocupada = ocupada;
        this.id_empleado = id_empleado;

    }

    public Integer getNumero() {
        return numero;
    }

    public Integer getPiso() {
        return piso;
    }

    public Integer gettamanyo() {
        return tamaño;
    }

    public Float getPrecio() {
        return precio;
    }

    public void setPrecio(Float precio) {
        this.precio = precio;
    }

    public Boolean getOcupada() {
        return ocupada;
    }

    public void setOcupada(Boolean ocupada) {
        this.ocupada = ocupada;
    }

    public Integer getId_empledo() {
        return id_empleado;
    }

    public void setNumero(Integer numero) {
        this.numero = numero;
    }

    public void setPiso(Integer piso){
        this.piso = piso;
    }

    public void setTamanyo(Integer tamaño){
        this.tamaño= tamaño;
    }

    public void setIdEmp(Integer id_empl){
        this.id_empleado=id_empl;
    }


}
```

5.5 Patrón Singleton

Ayuda a tener controlado el número de instancias sobre una clase concreta. Se obtiene un punto global sobre esta clase.

```java
package Negocio.NegocioFactory;
import Negocio.Clientes.SACliente;
public abstract class SAFactory {

    private static SAFactory instance;

    public static SAFactory getInstance() {
        if (instance == null) {
            instance = new SAIMPFactory();
        }
        return instance;
    }

    public abstract SAHabitacion newSAHabitaciones();

    public abstract SATarea newSATarea();

    public abstract SADepartamento newSADepartamento();

    public abstract SACliente newSACliente();

    public abstract SAReserva newSAReserva();

    public abstract SAEmpleado newSAEmpleado();
}
```

5.6 Modelo Vista Controlador

Modulariza la gestión de acciones y vistas. Permite tener múltiples vistas para un mismo modelo.

5.7 Factoría Abstracta

Lo utilizan el DAO y el SA. Genera familias de objetos relacionados. Crea objetos y devuelve interfaces.

```java
import Negocio.Clientes.SACliente;

public class SAIMPFactory extends SAFactory {

    @Override
    public SAHabitacion newSAHabitaciones() {
        return new SAHabitacionIMP();
    }

    @Override
    public SATarea newSATarea() {
        return new SATareaIMP();
    }

    @Override
    public SADepartamento newSADepartamento() {
        return new SADepartamentoIMP();
    }

    @Override
    public SACliente newSACliente() {
        return new SAClienteIMP();
    }

    @Override
    public SAReserva newSAReserva() {
        return new SAReservaIMP();
    }

    @Override
    public SAEmpleado newSAEmpleado() {
        return new SAEmpleadoIMP();
    }
}
```

6. Herramientas de control de versiones

Debido a las restricciones puestas por el profesor de la asignatura el equipo tuvo que usar GitHub como repositorio, haciendo dos repositorios distintos, uno para código y otro para diagramas. Al comienzo de la asignatura se creó un proyecto, pero al ir a ponerse a trabajar sobre él, pero equipo se dio cuenta de que estaba corrupto teniendo que crearlos nuevo para el correcto funcionamiento de este. Este repositorio es muy útil pues te permite ver los commits que ha hecho cada persona de forma independiente, además la forma de controlar los conflictos era muy sencilla pues nos dividimos de tal forma que en ningún momento se estuvieran dos personas tocando la misma clase java, y si daba algún conflicto solo tenias que escoger con cual de los cambios que daban conflicto te quedabas para subirlo al repositorio.

Si queríamos volver a alguna versión anterior por que habíamos tenido conflictos o algo no estaba bien ejecutado, solo teníamos que escogerla en el árbol de versiones de Git, dentro de IBM, para ir a esta.

7. Problemas durante el desarrollo del proyecto

A pesar de disponer de un plan de proyecto con un estudio de los riesgos, nos han surgido algunos problemas que no hemos tenido en cuenta.

1. **Mala planificación del proyecto**: Nos afectó ya que tuvimos un desliz a comienzo de curso y hasta que uno de los miembros tuvo tiempo para ponerse, y los demás al escasear en los conocimientos requeridos tampoco se pusieron hasta que este miembro se lo dijo, esto provoco ir ajustados de tiempo.
2. **Abandono de los integrantes del grupo**: Aunque no haya tenido ningún efecto negativo sobre el desarrollo del software o diagramas, a mediados de curso cuando se exigió hacer cosas para poder sacar el proyecto uno de los miembros del equipo decidió abandonar por el estrés ocasionado repentinamente, además al final del curso otro de los miembros decido abandonar también puesto que no había hecho nada del trabajo.
3. **Poco conocimiento del negocio**: Nos ha afectado debido a que solo uno de los miembros ha llevado acabo esta parte del código puesto que los demás no tenían conocimientos, aunque al final de la elaboración del proyecto algún otro miembro ha tocado esas partes que infundían más respeto para los poco experimentados.
4. **Incompatibilidad en las herramientas de software (java) o entre versiones de Windows**: A pesar de haber estado trabajando con distintas versiones de Windows entre los integrantes e incluso distintos sistemas operativos (Windows, MacOS), no hemos tenido problemas de incompatibilidad de las herramientas software.
5. **No asimilar los conocimientos técnicos de la asignatura**: No nos ha afectado, y se ha solucionado de manera rápida después de ayudar al compañero a comprender estos conocimientos sobre la asignatura.
6. **Función muy importante de la aplicación empieza a dar errores**: Con consecuencias críticas hemos tenido que solucionar este error en más de una ocasión después de dedicarle muchas horas de esfuerzo. Nos ha pasado con el controlador, los DAOs, las factorías y la vista principal.
7. **Función poco importante de la aplicación empieza a dar errores**: Error recurrente en zonas como el controler al olvidar poner un brake o en los DAOs haciendo cosas innecesarias como nos indicó el profesor.
8. **Sobrecarga de trabajo**: Tuvimos una sobrecarga al empezar en semana santa el proyecto pues veíamos que no llegábamos a la entrega, pero esto al final se soluciono con esfuerzo y horas de trabajo.