Universidad ORT Uruguay

Diseño de Aplicaciones 2

Obligatorio 2

Emiliano Russo 227209

Índice

Contenido

Repositorio

https://github.com/Emiliano-Russo/Obligatorio

Descripción general del trabajo

La solución presentada para esta segunda instancia conserva todas las funcionalidades de la primera parte, más las nuevas que fueron solicitadas, a excepción de la carga de registros automática mediante json/xml. Con esto se hace hincapié a las implementaciones nuevas:

- Reporte tipo A
- Nuevo tipo de huésped (Jubilado)
- Reseñas para un hospedaje

Se implemento todo el front-end en el framework angular, como single-page-application, dividido en componentes reutilizables, con su correcto uso y conexión a los controladores del back-end. También se aplicaron estilos css para su correcta visualización de interfaz de uso. Para los Administradores de la pagina se implemento un panel donde puedan operar con sus tareas correspondientes. Todas las funcionalidades de la parte administrativa se conservan y queda exclusivamente habilitadas para ellos, con un sistema de login previamente implementado. La funcionalidad “Reporte tipo A” está disponible en la sección del panel para el administrador.

Las nuevas funcionalidades fueron desarrolladas siguiendo la metodología TDD, al igual que en la primera instancia de trabajo, esto nos aporta una seguridad en cuanto a menos probabilidad de bugs desconocidos (no se conoce ninguno que tenga la aplicación).

Diagrama de descomposición de paquetes

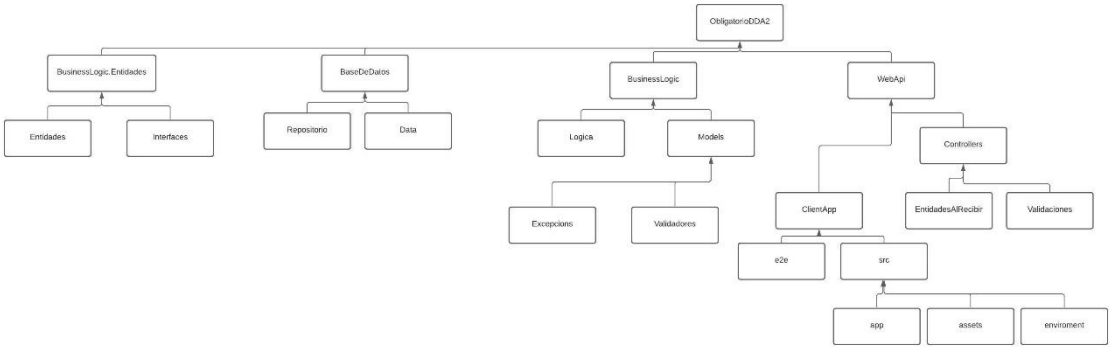


Diagrama General de paquetes

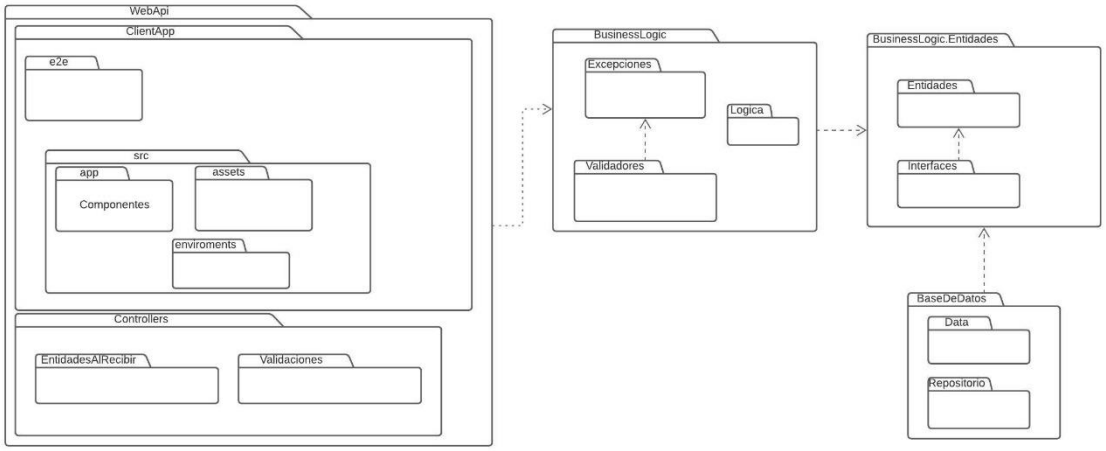


Diagrama de clases

Diagrama de clases BusinessLogic

La lógica de negocio es el núcleo de todas las funcionalidades solicitadas. Su entrada es la clase “Sistema”, para cumplir el patrón “Facade”. Tiene una fuerte dependencia con el paquete “entidades”

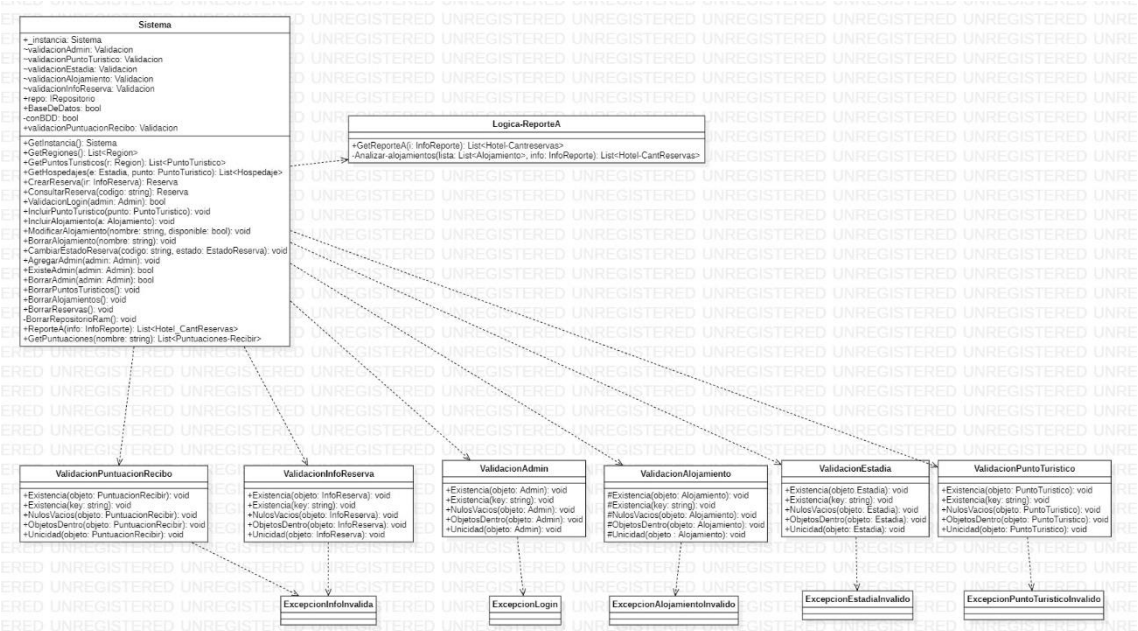


Diagrama de clases BusinessLogic.Entidades

Este paquete contiene todas las clases que representan el dominio del sistema, no tiene dependencia externa con ningun otro paquete de la solucion presentada. Contiene interfaces para representar un repositorio y reglas de validacion.

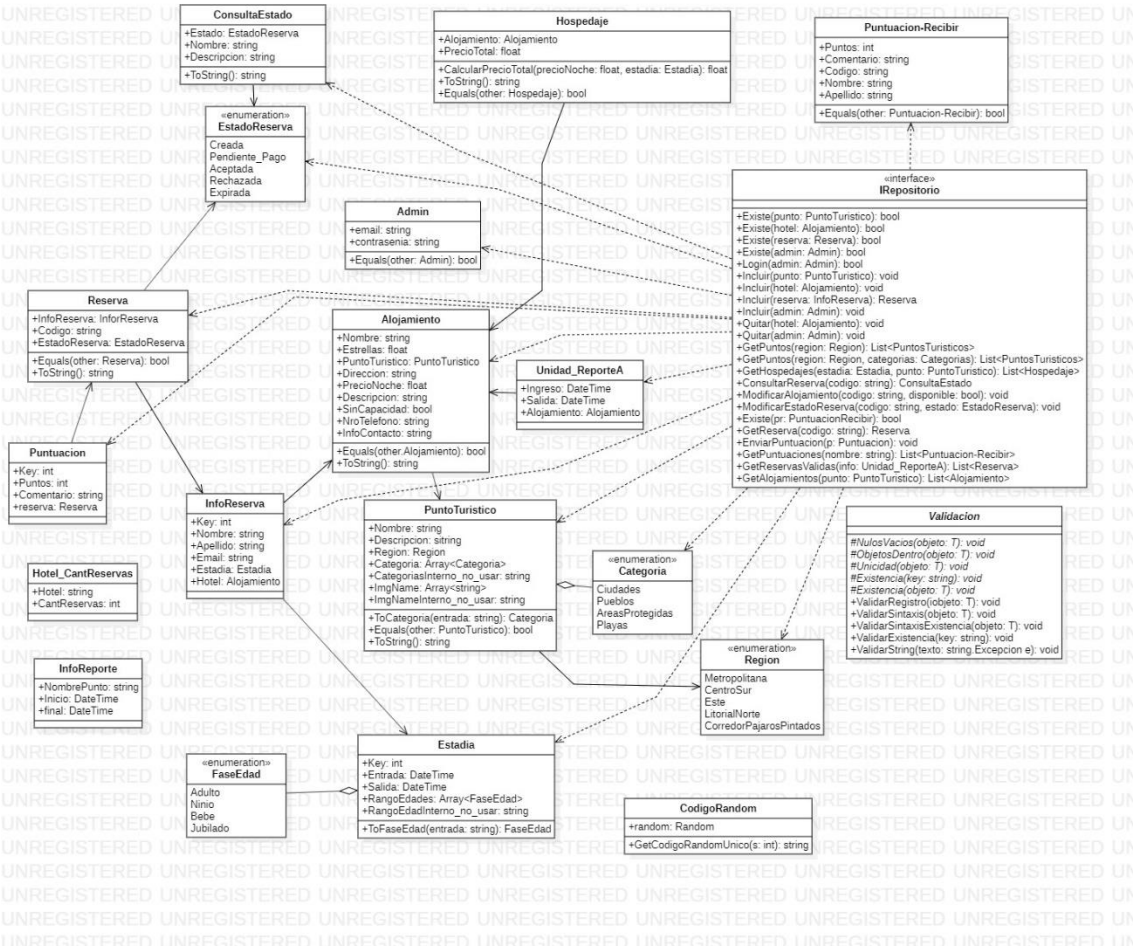


Diagrama de clases Base de datos

Este paquete maneja el repositorio de la solucion, aquí se presentan implementaciones de la interfaz prestada de entidades “IRepositorio”, para establecer los metodos a escribir.

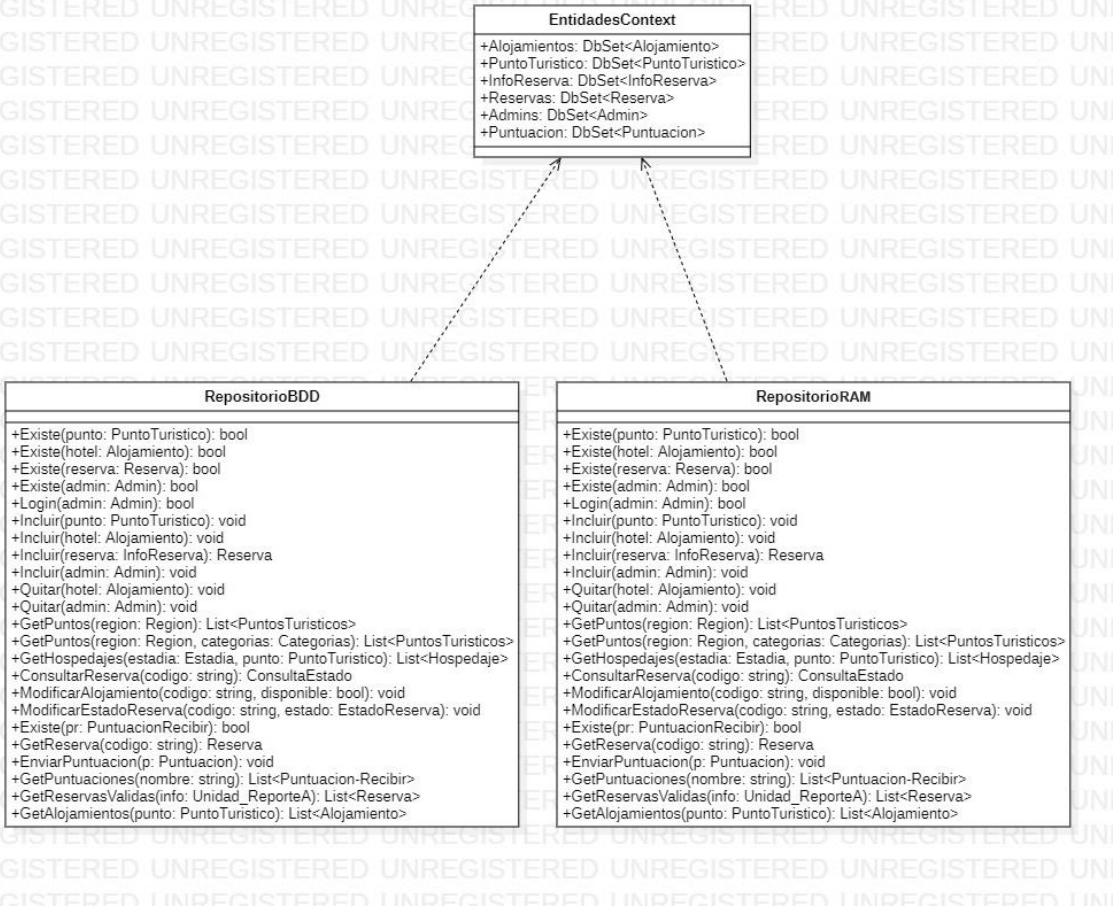
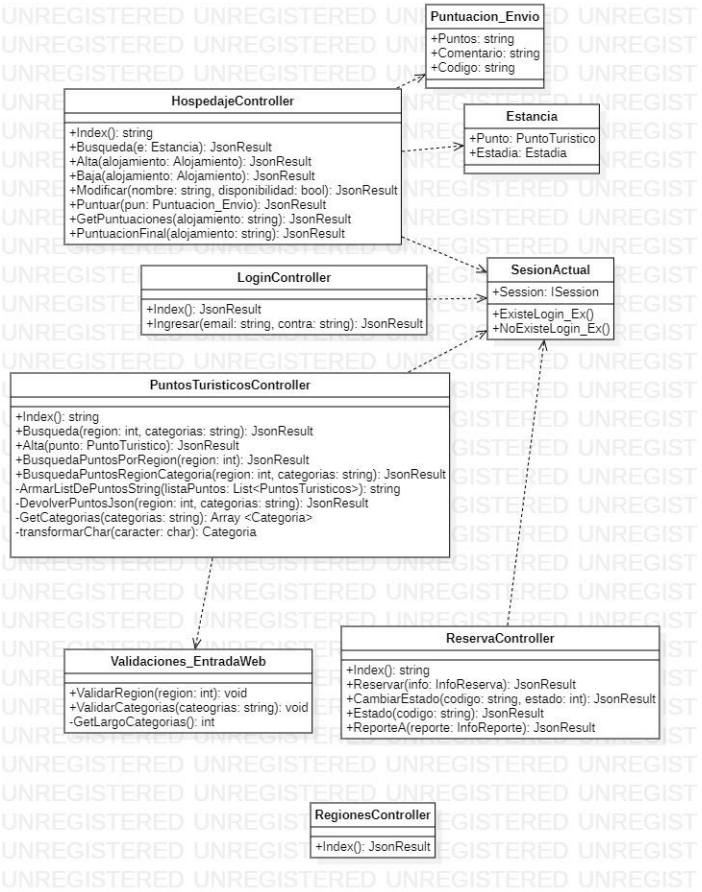


Diagrama de clases WebApi

La misma sostiene todos los controladores (puntos de entrada) de la aplicación, por la cual el cliente va a recibir y solicitar datos.

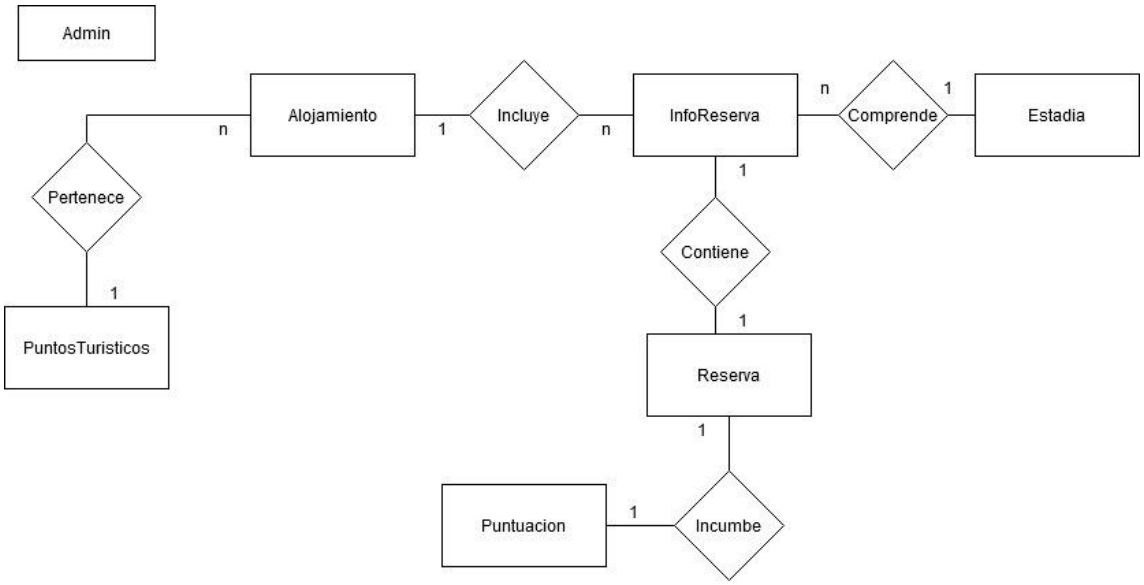


Descripción de herencias utilizadas

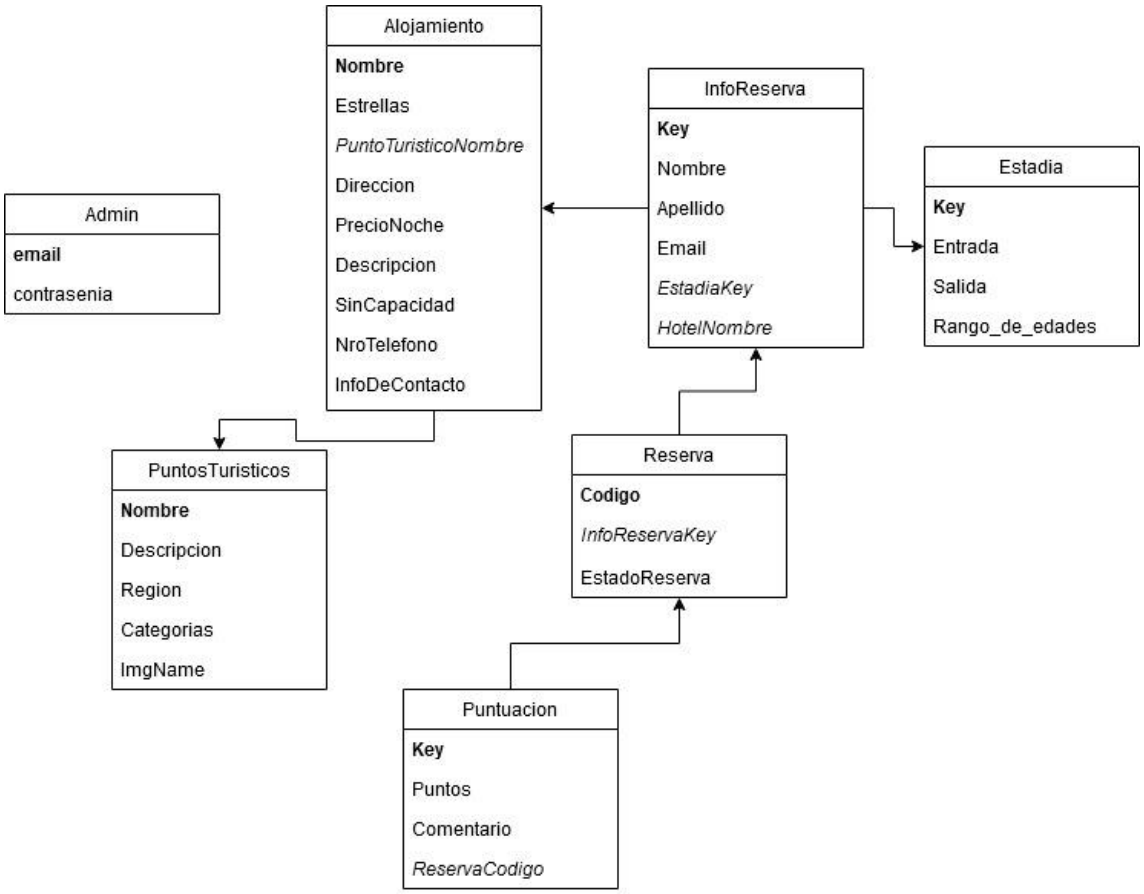
Clase Abstracta Validación: fue creada con un tipado genérico, para que cualquier clase del dominio pueda implementar sus validaciones. La misma tiene métodos abstractor protected, y tiene sus algoritmos originales como públicos ya implementados, así cumplimos con el patrón de diseño template method. Cada entidad que necesita una validación creara su clase y heredara de la anterior mencionada, Ej: Validacion<PuntoTuristico>.

Interfaz IRepositorio: fue creada para implementar libremente cualquier tipo de repositorio. En el código actual existen 2 implementaciones, uno para las pruebas que utiliza la memoria RAM y otro para la base de datos, que trabaja con EntityFramework.

Modelo de entidades Base de datos



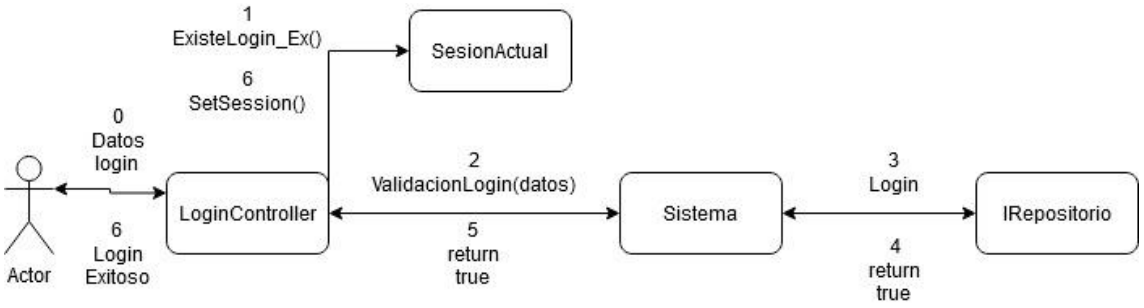
Estructura de la base de datos



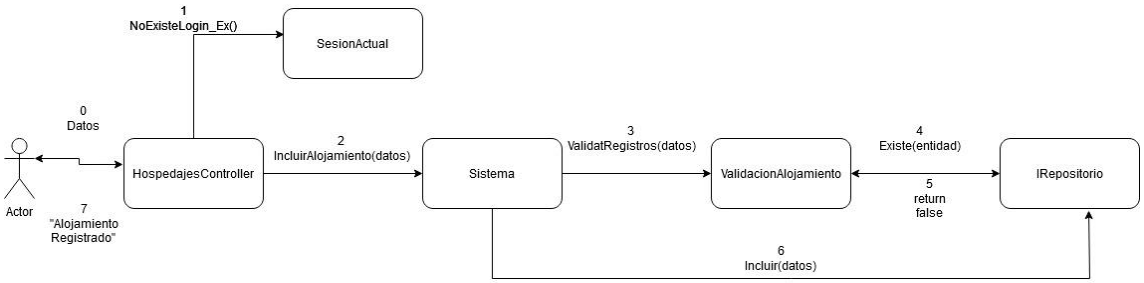
Diagramas en funcionalidades claves

1. Diagrama Colaboración

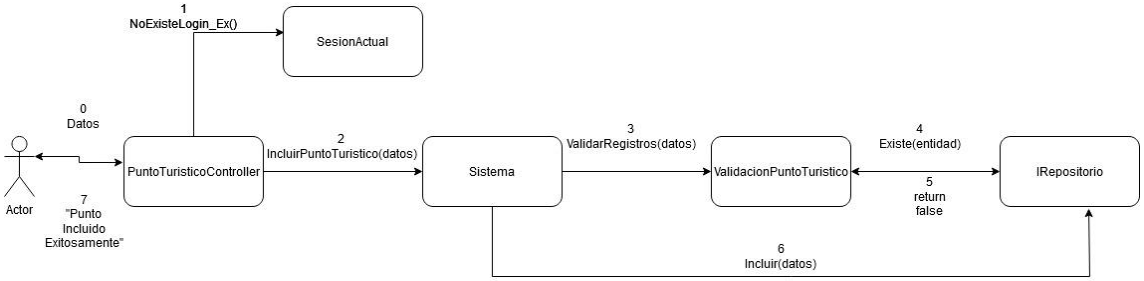
Colaboración Login



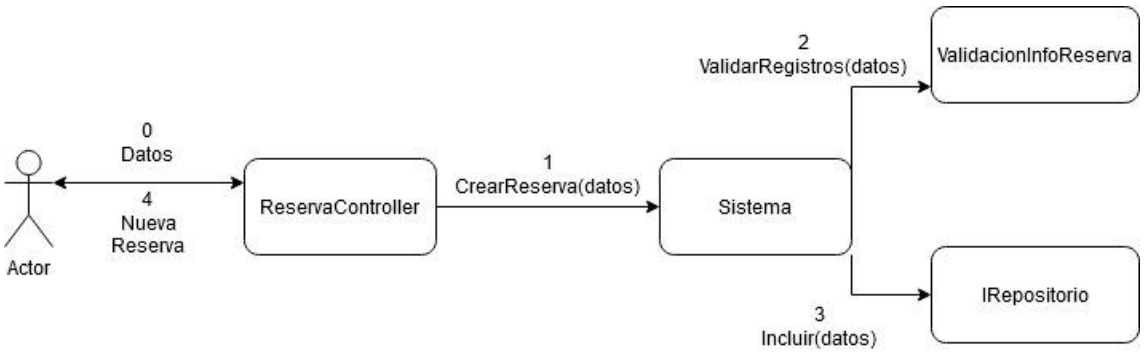
Colaboración Incluir Alojamiento



Colaboración Incluir Punto Turístico

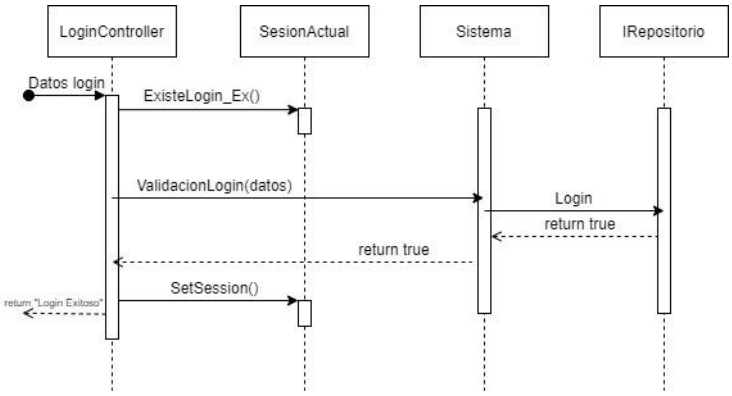


Colaboración Crear Reserva

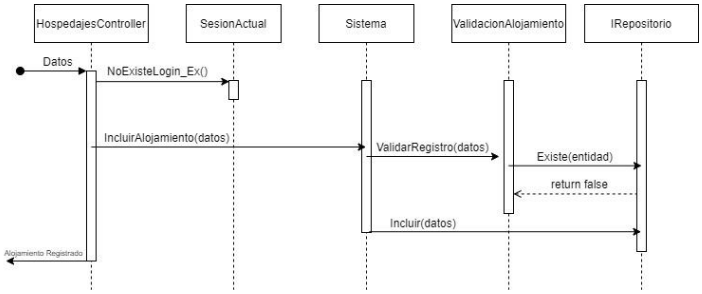


2. Diagrama de Secuencia

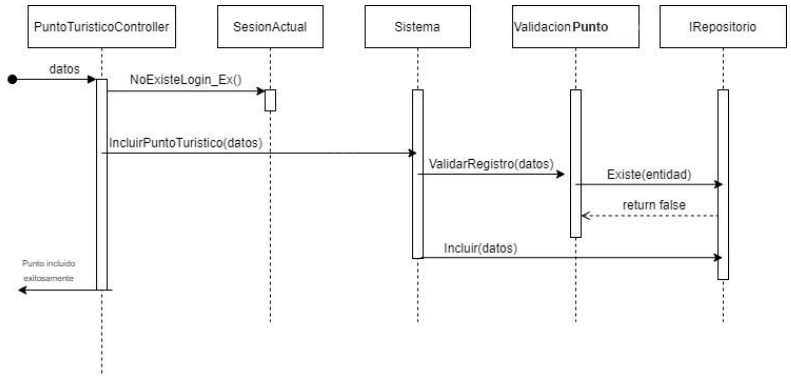
Secuencia de Login



Incluir Alojamiento



Incluir Punto Turístico



Crear Reserva

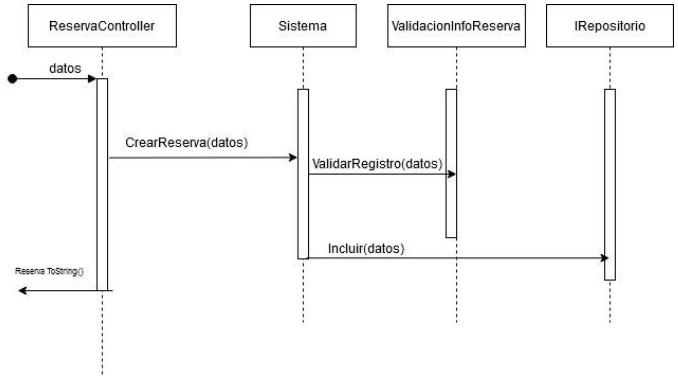
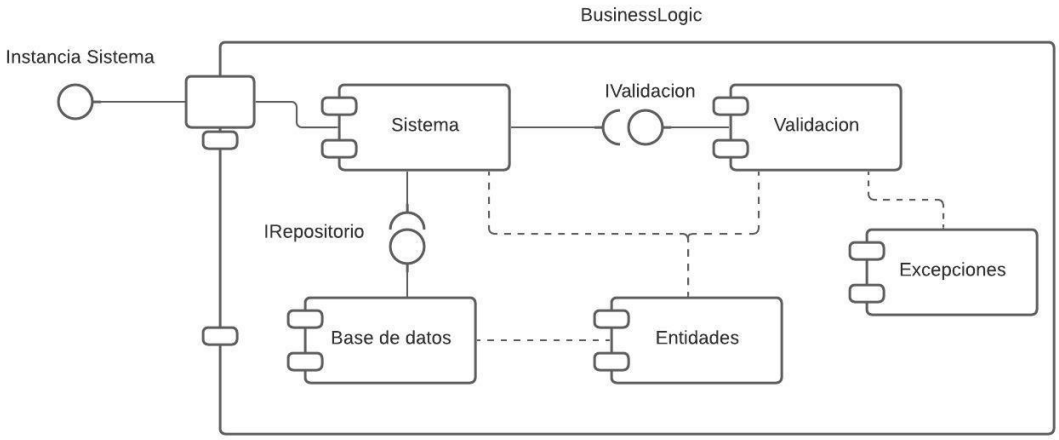


Diagrama de componentes

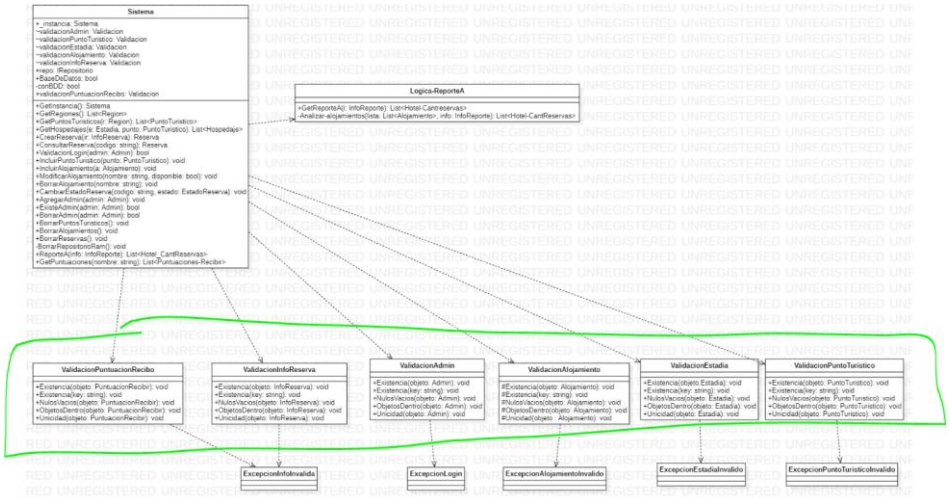


Se dividió en estos componentes, porque eran los que mas se ajustaban a nivel de diseño para esta instancia de trabajo, separando la lógica, la persistencia, la excepciones y la validación.

Justificación del diseño

Uso de patrones y principios

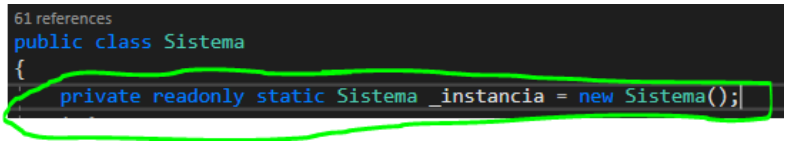
Clase Abstracta Validación: fue creada con un tipado genérico, para que cualquier clase del dominio pueda implementar sus validaciones. La misma tiene métodos abstractor protected, y tiene sus algoritmos originales como públicos ya implementados, así cumplimos con el patrón de diseño template method. Cada entidad que necesita una validación creara su clase y heredara de la anterior mencionada, Ej: Validacion<PuntoTuristico>. Esto nos permite si el día de mañana queremos una implementación de validación diferente a la actual, tenemos facilidad al cambio, gracias a los principios que se cumplen (OCP,SRP).



Las clases señaladas heredan de la clase abstracta no tipada “Validación”, que posteriormente sistema las utiliza como objetos.

Interfaz IRepositorio: fue creada para implementar libremente cualquier tipo de repositorio. En el código actual existen 2 implementaciones, uno para las pruebas que utiliza la memoria RAM y otro para la base de datos, que trabaja con EntityFramework. Al disponer de esta interfaz cumplimos con el principio de abierto cerrado. Cabe destacar que con esta interfaz (que se encuentra en el paquete entidades) estamos eliminando la posible dependencia entre los paquetes BaseDeDatos y BusinessLogic, al depender de abstracciones y no de implementaciones.

Sistema: fue creada para disponer de un único punto de entrada a la businesslogic, y realizar todas las peticiones desde esa clase. La misma cumple con el patrón singleton, para el fácil acceso desde cualquier clase y no dejar más de una instancia sistema para la solución.



La estructura de los paquetes y las clases también cumple con el principio de responsabilidad única. Al separar la abstracción de la implementación, se pudo llegar a la inversión de dependencias con el paquete de la BaseDeDatos. La principal mejora de diseño fue disminuir el acoplamiento a nivel de paquetes en la solución.

***Principio Clausura Común**:* viendo los componentes en su diagrama, podemos denotar que por su composición (Validación, Excepciones, Base de datos, Entidades), cumple con este principio porque cada uno de estos componentes tiene una única razón para cambiar, por ejemplo si hacemos un enfoque en validación, tenemos la herencia abstracta que al modificar alguna regla de negocio a la hora de validar, afectaría a cada una de las clases que la implementan, ósea a todo el componente, eso afirma la existencia de "un sola razón para cambiar". Esto nos aporta mantenibilidad en nuestro código.

Principio Reutilización Común: este principio nos dice que: las clases de un componente se reutilizan juntas. Si reutiliza una de las clases en un componente, las reutiliza todas. Nuestro código cumple con este principio, una justificación con ejemplificación: seria casi imposible reutilizar solo una clase del componente entidades, imaginemos que quisiéramos llevaron la clase "InfoReserva", esta tiene como property la clase "Estadía" y "Alojamiento", a su vez la clase alojamiento tiene como property "PuntoTuristico", la clase estadía tiene como property a la enumeración "FaseEdad". Como se puede ver, seria un nudo largo y difícil de romper.

Principio de Dependencias Estables: este principio nos dice que: la métrica I de un paquete debe ser mayor que las métricas I de los paquetes de los que él depende, esto es la métrica I debe decrecer en la dirección de la dependencia.

- Formula: $I = C_e/(C_e + C_a)$

Ca = número de clases externas al paquete que dependen de clases internas al paquete.

Ce= número de clases internas que dependen de clases externas.

BaseDeDatos

Ca = 0 | Ce = 3 => I = 3 / (3+0) => I = 1

BusinessLogic.Entidades

Ce = 0 => I = 0

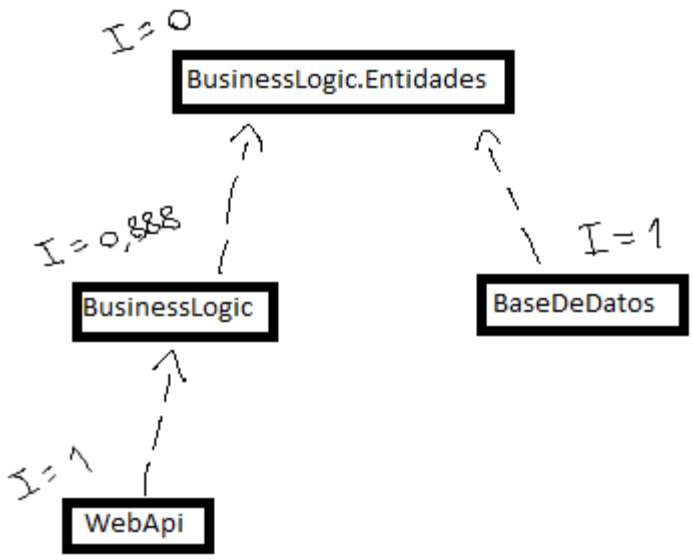
BusinessLogic

Ca = 1 | Ce = 8 => I = 8/(8+1) => I = 0.888

WebApi

Ca = 0 | Ce = 7 => I = 1

Con estos cálculos se puede ver que la dirección de la dependencia WebApi -> BusinessLogic, disminuye el grado de inestabilidad, por consiguiente, cumple el principio de dependencias estables.



Principio de Abstracciones Estables: Los paquetes más estables (dependen de él y él no depende) deben tender a ser abstractos (con interfaces o clases abstractas)

- Formula: $A=N_a / N_c$

Na =Cantidad de clases abstractas e interfaces en el paquete

Nc = Cantidad de clases concretas, abstractas e interfaces del paquete.

BusinessLogic

A = 0/n => A = 0

BusinessLogic.Entidades

A = 2 / 20 => A = 0.1

BaseDeDatos

A = 0 / n => A = 0

WebApi

A = 0 / n = 0

Sección Abstracción y Estabilidad

$$D' = |A + I - 1|$$

BusinessLogic

D = ABS (0+0.888-1)

D = 0.112

Nos encontramos con una distancia tendiendo a 0, en el lugar [A: 0, I:0.888], Se encuentra muy cerca de la secuencia principal, esto quiere decir que tiene unos valores de estabilidad y abstracción bastante equilibrados.

WebApi

ABS (0 + 1 - 1)

D = 0

Nos encontramos con una distancia 0 a la recta ideal, en el lugar [A:0, I:1], este paquete tiene unos valores de estabilidad y abstracción equilibrados.

BusinessLogic.Entidades

D = ABS (0.1 + 0 - 1)

D = 0.9

Nos encontramos con una distancia 0.9 a la recta ideal, em el lugar [A:0.1, I:0], En este caso nos encontramos con un paquete cuyas clases no dependen de ninguna clase de otro paquete y que además pocas de sus clases son abstractas, vendría a hacer referencia a un paquete estable y concreto y por tanto rígido. Siendo un paquete a vigilar si el acoplamiento aferente es alto, debido a que las posibilidades de efectos colaterales pueden ser alta.

BaseDeDatos

D = ABS (0 + 1 - 1)

D = 0

Nos encontramos con una distancia 0, en el lugar [A: 0, I:1], esto quiere decir que tiene unos valores de estabilidad y abstracción equilibrados.

Cobertura Test Unitarios

No se incluyen los paquetes base de datos y testlogic

No se incluyen las clases startup y program

No se incluyen los atributos autogenerados para entityframework (como atributo key)