Descripción del diseño

Descripción general del trabajo

La solución presentada ofrece todo y cada uno de los requerimientos (a excepción de las imágenes, aunque si se guarda su ruta) planteados por la empresa “Uruguay Natural”, y no hay existencia de bugs, o al menos no se saben de los mismos. El proyecto fue desarrollado implementando la metodología TDD, la cual nos previene ya de entrada una cantidad de posibles bugs que se puedan cometer en el transcurso de desarrollo.

La descripción de los requerimientos funcionales son los siguientes

- Página principal Índex, para ver las regiones disponibles.
- Buscar puntos turísticos por región y filtrar dicha búsqueda por categoría
- Buscar hospedajes (con detalles y precio total incluido) para un cierto punto turístico con los parámetros especificados. los cuales son: Estadía y Punto Turístico.
- Realizar una reserva de un hospedaje (sin necesidad de login)
- Consultar el estado actual de una reserva dado su número

Requerimientos funcionales administrativos

- Iniciar sesión en el sistema usando su e-mail y contraseña
- Dar de alta un nuevo punto turístico, para una región existente
- Dar de alta un nuevo hospedaje o borrar uno existente, para un punto turístico existente.
- Modificar la capacidad actual de un hospedaje.
- Cambiar el estado de una reserva, indicando una descripción.
- Agregar Administrador (Solo BusniessLogic)
- Quitar Administrador (Solo BusinessLogic)

Diagrama de descomposición de los namespaces

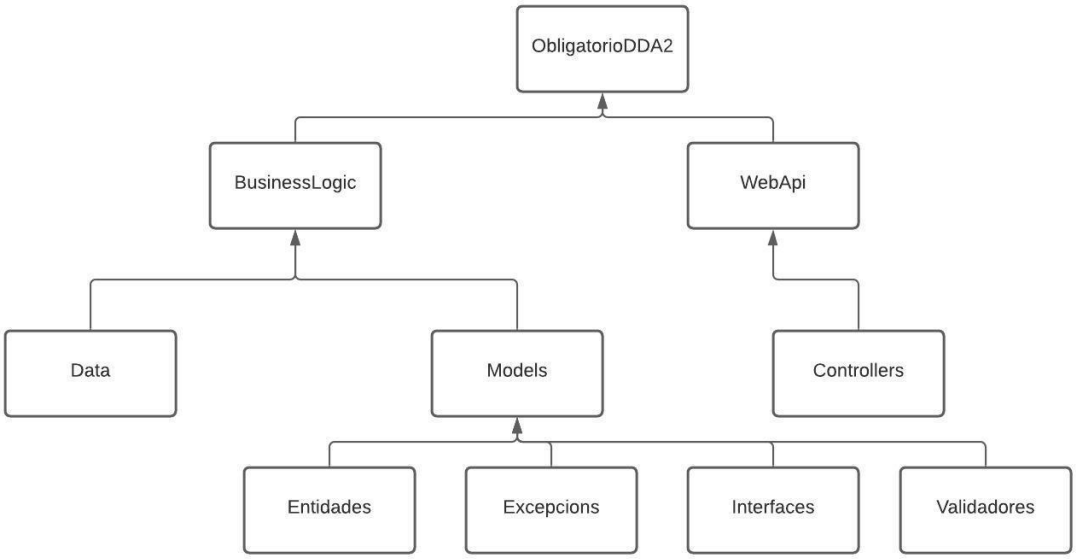


Diagrama general de paquetes (namespaces)

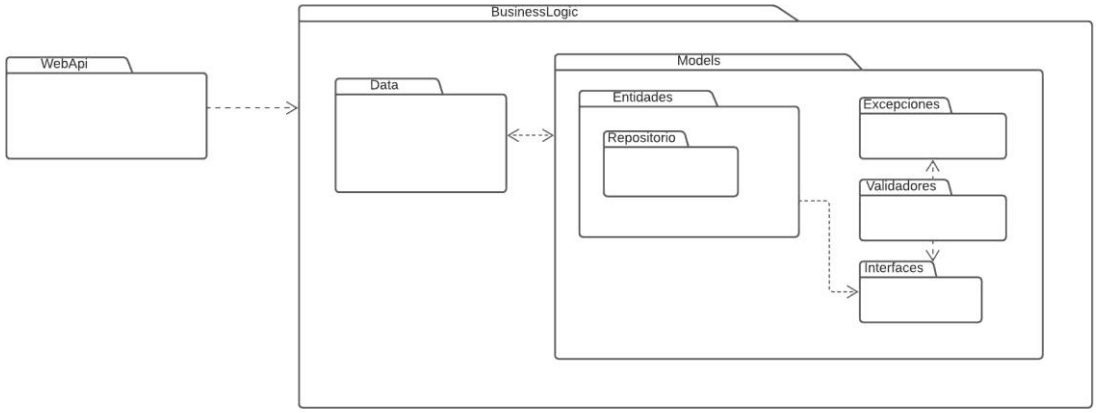
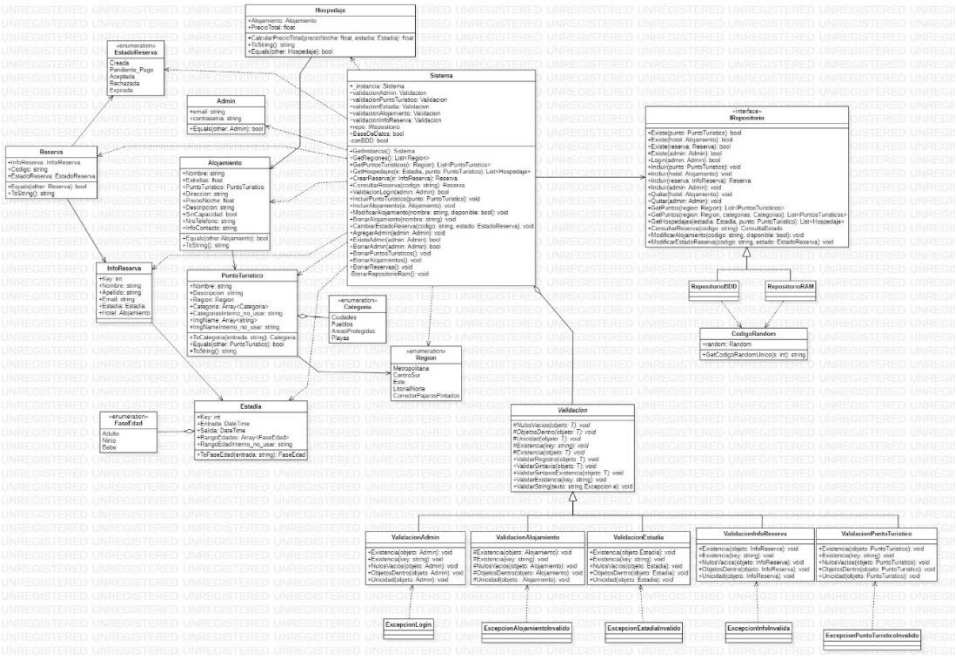


Diagrama de clases de los paquetes

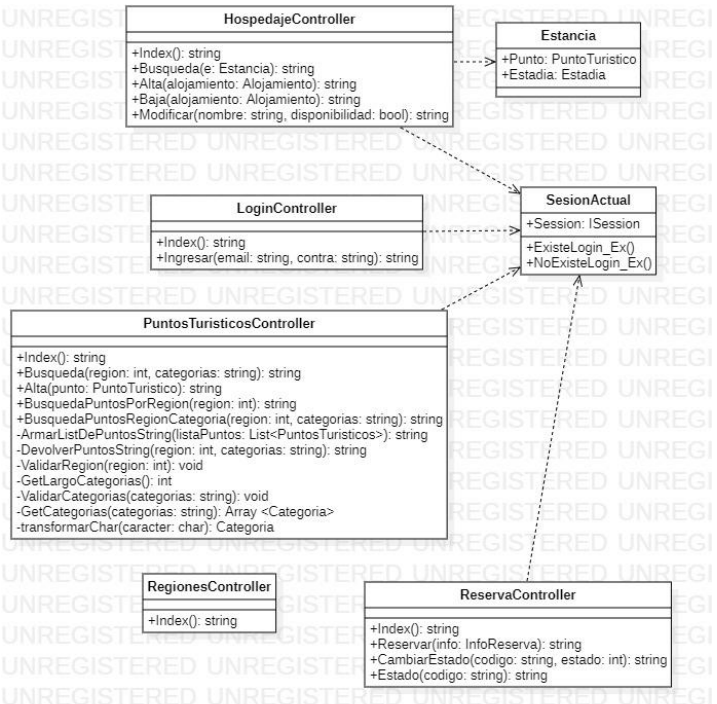
BusinessLogic

La lógica de negocio es el núcleo de todas las funcionalidades solicitadas. Su entrada es la clase “Sistema”, para cumplir el patrón “Facade”.



WebApi

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

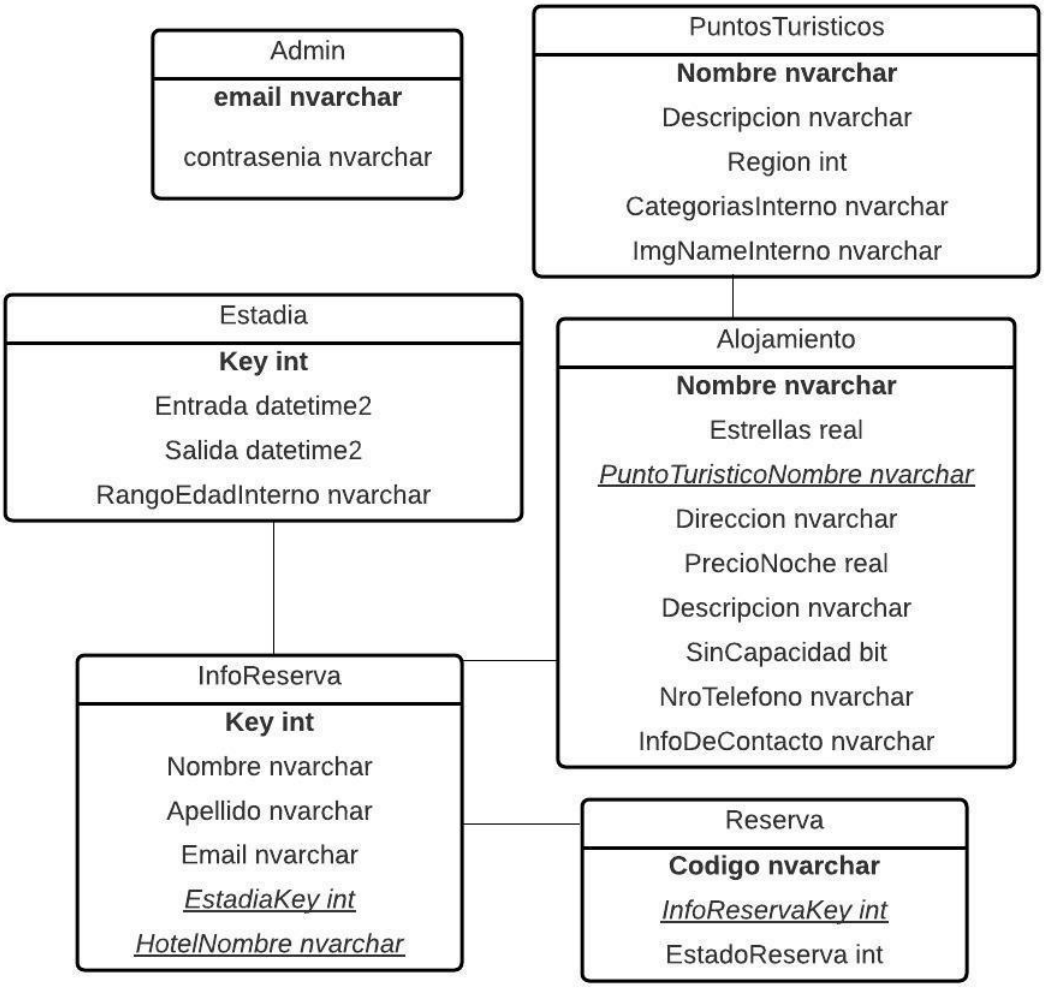


Descripción de jerarquías de herencia utilizadas

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>.

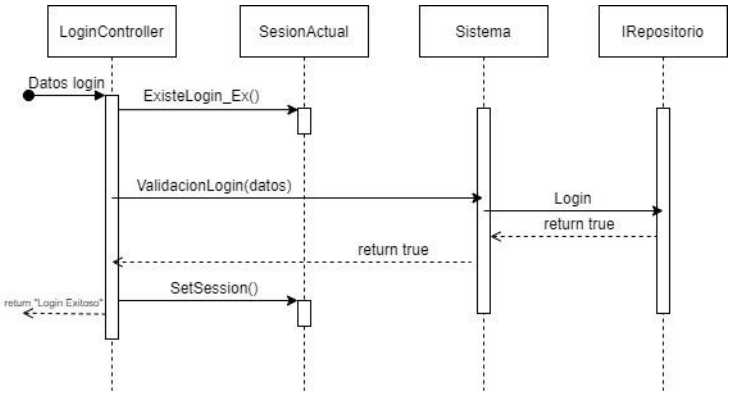
Interfaz IRepositorio: fue creada para implementar libremente cualquier tipo de repositorio. En el código actual existen 2 implementaciones, uno para las pruebas que utiliza la memoria RAM y otro para la base de datos, que trabaja con EntityFramework.

Estructura de la base de datos

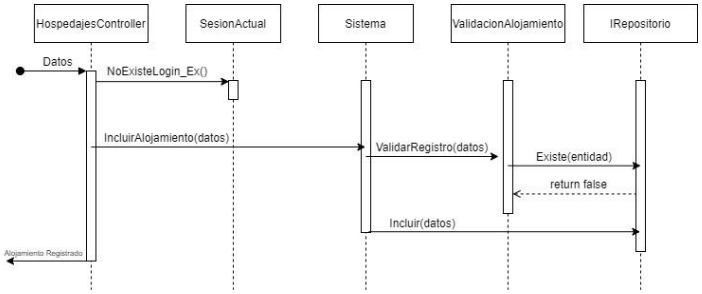


Diagramas de secuencia

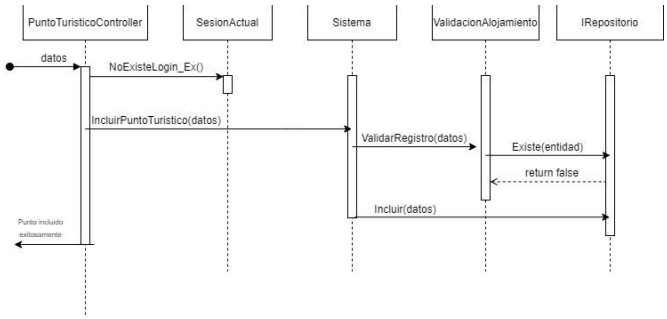
Secuencia de login



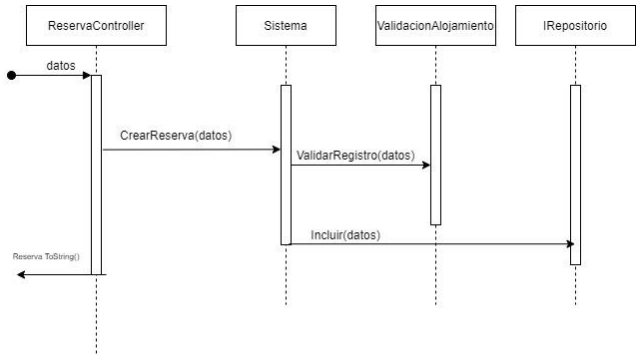
Incluir Alojamiento



Incluir Punto Turístico



Crear Reserva

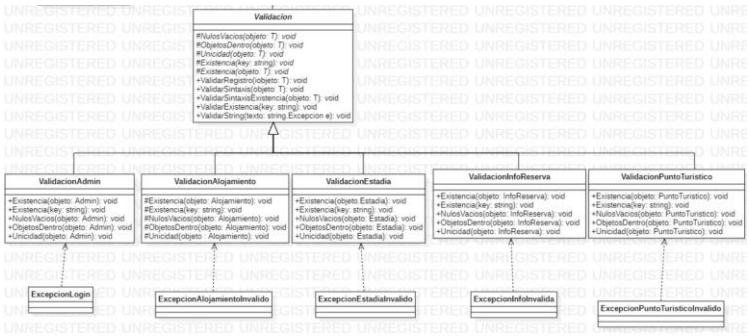


Justificación del diseño

En este apartado se describirá el diseño que tiene el paquete “Business Logic”.

El primer patrón de estructura que podemos destacar es el “Facade”, que fue implementada como singleton (para su fácil acceso y que solo exista una entidad de la misma), la clase sistema toma su rol de controlador para manejar las entradas de la lógica de negocio (su única utilidad), esto hace que bajemos el acoplamiento y aumentemos la cohesión. La misma también tiene properties para acceder al repositorio y a las validaciones de las entidades ingresadas. En conclusión, sería la clase conectora más importante, porque tiene la mayoría de las dependencias sobre ella.

El segundo patrón para destacar es el “template method”, que fue aplicado para darle una forma genérica al validar entidades. El mismo consta de una clase abstracta que tiene métodos abstractos y protegidos y otros públicos y ya implementados (que hacen uso de los abstractos).



Los métodos para implementar son, Existencia (objeto y key), NulosVacios, ObjetosDentro (Validación de estos), Unicidad. Métodos públicos combinan esos métodos de la siguiente forma:

Públicos => *Protected*

ValidarSintaxis => *NulosVacios, ObjetosDentro.*

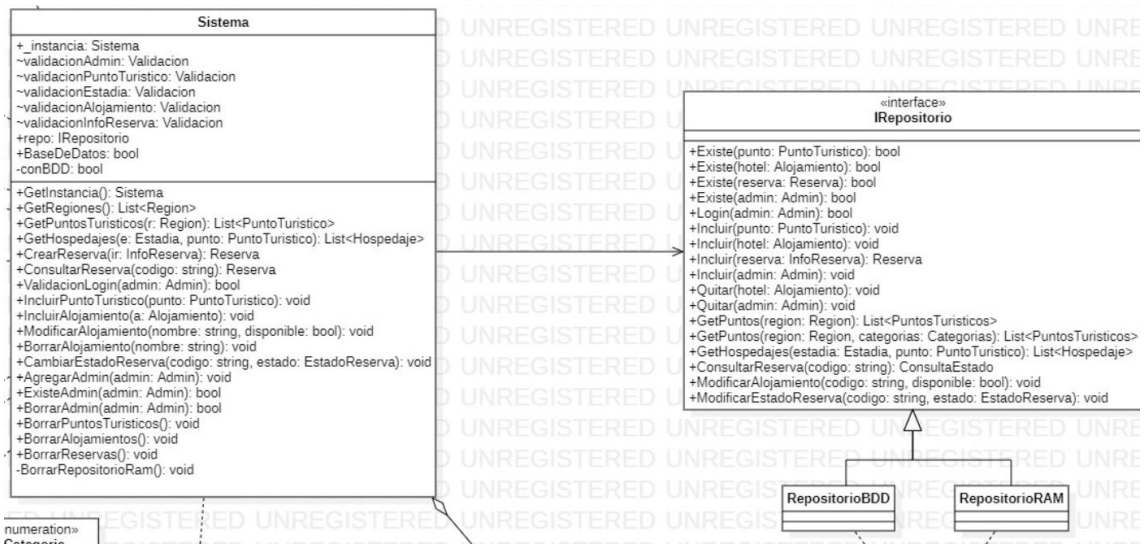
ValidarRegistro =>*ValidarSintaxis, Unicidad*

ValidarSintaxisExistencia => *ValidarSintaxis, Existencia*

ValidarExistencia => *Existencia*

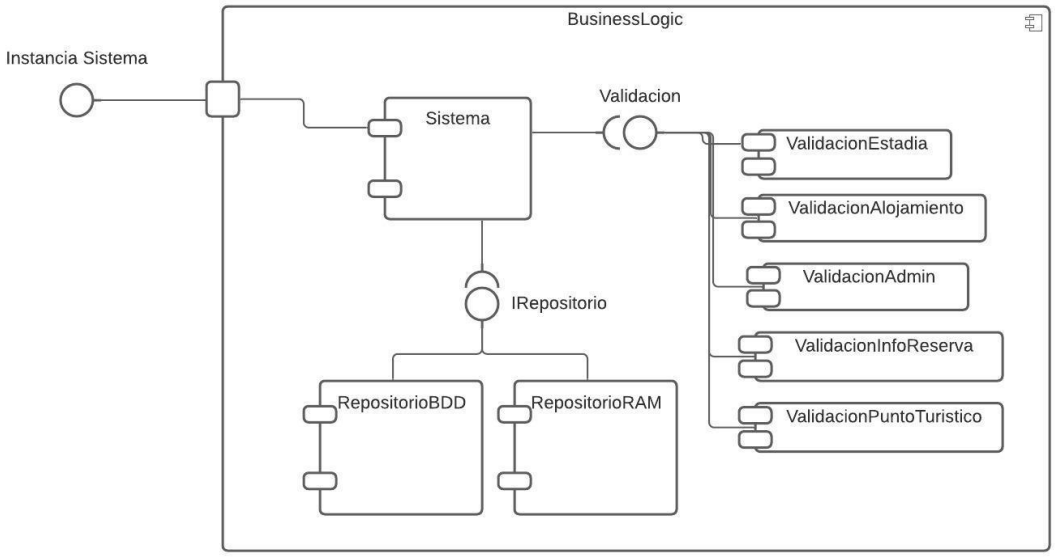
Estas clases largan las excepciones (solo a la entidad relacionada). Tienen que ser manejadas por la WebApi (controladores) que esta más cerca del cliente.

Tercer patrón aplicado, esta vez de comportamiento, es el strategy. La clase sistema hace uso de una interfaz IRepositorio para respetar los principios Open-Closet Principle (para sustituir por cualquier base de datos de preferencia rápidamente), Interface Segregation Principle (porque los métodos que se describen son los mínimos que se necesitan para que el sistema funcione), Dependecy inversión principle (para depender de abstracciones y no de implementaciones).



Este diseño facilito el desarrollo de test unitarios, que hacen uso del repositorio RAM y no del de base de datos, para su rápida ejecución y que no se vuelva una prueba de integración. En conclusión, este es nuestro sistema de acceso a datos, RepositorioBDD trabaja con EntityFramework.

Diagrama de implementación (componentes)



La justificación de porque se dividieron así los componentes es simplemente para aportar una mayor mantenibilidad al código. Sabemos que se podrían dar cambios a futuro, tanto de la base de datos como de validaciones para las entidades.