Parking App

Obligatorio 2 - Diseño de Aplicaciones 1 - M5A
Universidad ORT

Profesores: Nicolás Fornaro, Dan Blanco, Yani Kleiman

Sebastián Bañales 202711
Romina Rodríguez 209234

Contenidos del documento

Descripción del trabajo

Introducción

Este trabajo se enmarca en un proyecto para implementar un sistema que permita la compra de tiempo de estacionamiento mediante el uso de mensajes de texto (SMS) y que todos los datos del sistema sean persistidos en una base de datos. El objetivo final es que los usuarios de una empresa telefónica puedan, mediante el envío de mensajes de texto, comprar minutos de estacionamiento. A su vez, inspectores de tránsito podrán realizar diferentes consultas, sobre los datos almacenados, dependiendo cual sea su intención.

Implementación

El proyecto fue desarrollado en el lenguaje de programación C# usando el framework .NET (Versión 4.6.1) proveído por Microsoft Visual Studio.
Para la parte de mapeo de entidades y relaciones con una base de datos se utilizó el ORM de Microsoft, Entity Framework 6 (EF6) Version: 6.3.0.
Como servicio de SQL, se utilizó Microsoft SQL Server 2014.
Se utilizó un repositorio en GitHub para el control de versiones, y Git Flow como técnica de trabajo si bien no era obligatorio.
El proyecto fue creado a partir de la metodología TDD para la parte de la lógica de negocio, cumpliendo lo mejor posible por las buena prácticas de Clean Code. Y también tomando en cuenta criterios de usabilidad para la interfaz de usuario.

Funciones no implementadas

Se pudieron implementar todas las funciones requeridas.

Bugs

No se detectaron bugs en el proyecto al día de entrega.

Supuestos efectuados

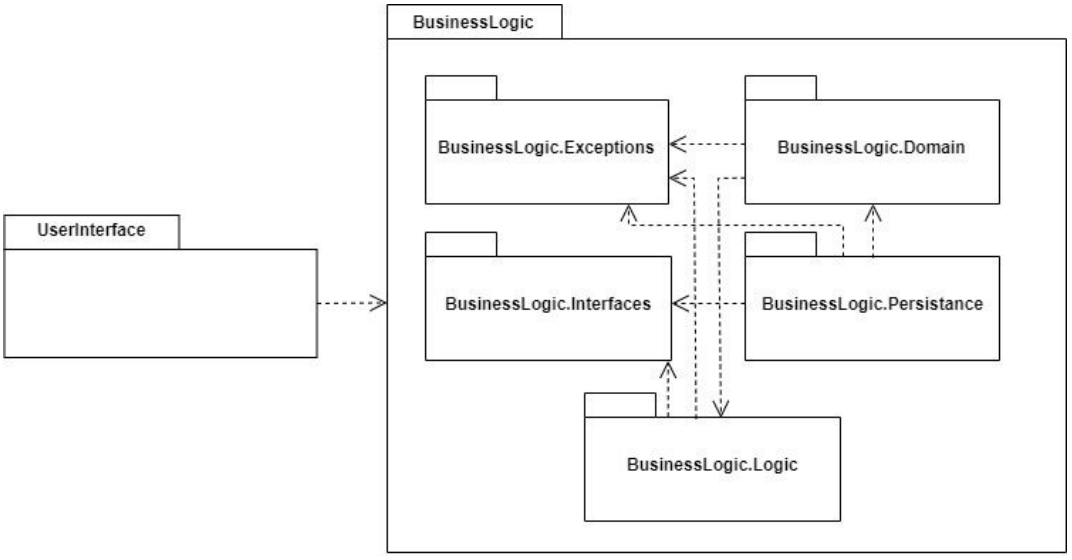
- Se supone que el valor del costo por minuto no va a exceder el valor máximo predefinido *Int32.MaxValue*.
- Se supone que el valor del costo por minuto, multiplicado por la cantidad de minutos comprados, no va a exceder el valor máximo predefinido *Int32.MaxValue*.
- El valor del costo por minuto para todos los países está seteado en 1 la primera vez que se inicia la aplicación.

Descripción y justificación del diseño

Costo de instalación

Para poder ejecutar la aplicación se debe crear la conexión a la base de datos, ir a UserInterface.config y cambiar el el Servidor por el nombre que se le haya otorgado a la conexión. Luego de guardar los archivos, se ejecuta la aplicación (UserInterface.exe) y se generará la base de datos con el nombre definido en UserInterface.config. Cabe destacar que previo a todo lo mencionado anteriormente hay que asegurarse que los servicios SQL Server (SQLEXPRESS), SQL Server Browser (SQLEXPRESS) estén corriendo y el tipo de arranque de SQL Server Agent seteado como automático.

Descripción de paquetes y contenido



El sistema fue dividido en tres proyectos:

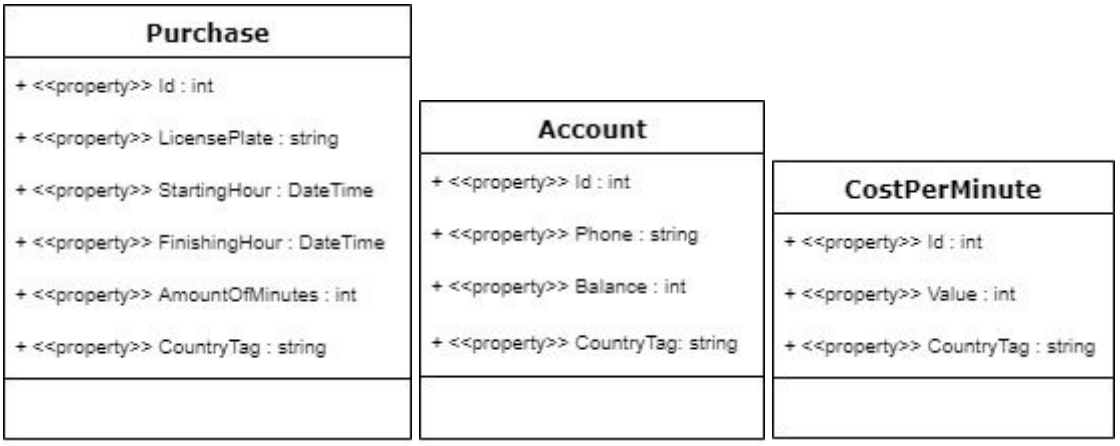
- Proyecto de la lógica del negocio y persistencia: BusinessLogic
- Proyecto de pruebas de la lógica del negocio y de base de datos: BusinessLogic.Test (no presente en diagrama de acuerdo a lo especificado en la letra, pero si en aplicación).
- Proyecto de interfaz gráfica: UserInterface

En el paquete BusinessLogic se pueden encontrar cinco carpetas:

- Domain
- Exceptions
- Interfaces
- Logic
- Persistance

BusinessLogic.Domain, fue realizado completamente con TDD. En él se encuentran todas las clases, cuyas funciones satisfacen los requerimientos de negocio. Dichas clases serán las entidades a persistir en la base de datos y no contienen lógica, solo properties. Se identifican 3 clases:

- Account: Modela la cuenta de un usuario cuyos atributos son número de teléfono, saldo, etiqueta de país y un identificador.
- Purchase: Modela la compra de tiempo de estacionamiento que un usuario podrá efectuar, teniendo como atributos la matrícula del auto a ocupar el estacionamiento, la hora de inicio de la reserva (implementada como un tipo DateTime), la hora de finalización de la reserva (implementada como un tipo DateTime), la cantidad de minutos, etiqueta de país y un identificador.
- CostPerMinute: Modela el costo por minuto de un país teniendo como atributos un valor, etiqueta de país y un identificador

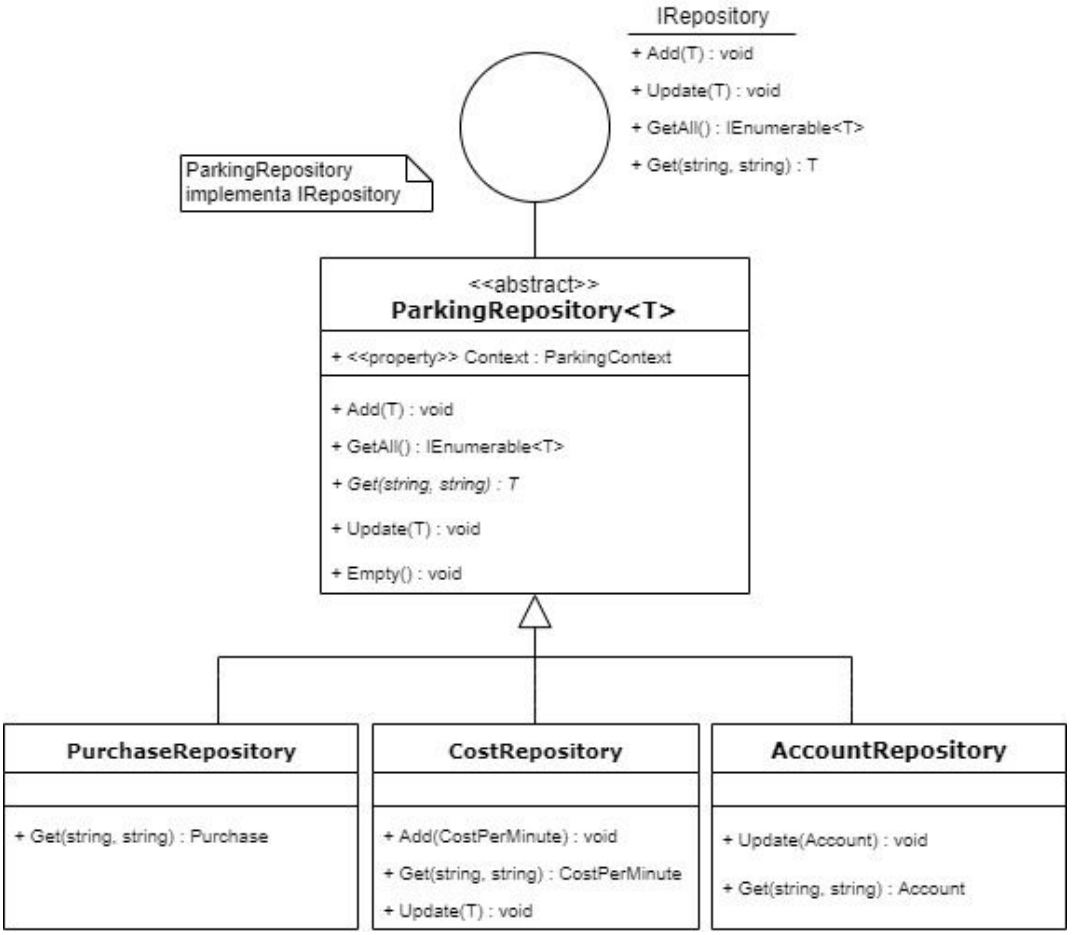


En BusinessLogic.Exceptions se encuentran 2 clases, BusinessException y DatabaseException, la primera es una clase de excepción creada por nosotros, la cual extiende de la clase Exception, para el manejo de los errores que el usuario pueda comer. Ese manejo se da de tal forma que un error cometido por el usuario no daña el flujo ni la integridad del programa, sino que le notifica al usuario y le permite recuperación instantánea. La clase tienen 3 constructores, 2 por los cuales es posible escribir un mensaje el cual luego será mostrado por pantalla en la interfaz como una ventana emergente. Si bien la letra sugería mensajes de errores dependiendo de la situación, consideramos que eran en muchos casos muy poco específicos y por lo tanto no indicaban bien al usuario el error cometido y la solución posible a dicho error, bajando mucho la facilidad de uso de la aplicación y por lo tanto afectando la usabilidad seriamente. Es por esto que se decidió poner mensajes de error específicos al error cometido.

La segunda clase presente en el paquete, DatabaseException es otra clase de excepción creada por nosotros, la cual extiende de la clase Exception, para el manejo de los errores que puedan ocurrir con las operaciones de interacción con la base de

datos. Al igual que para BusinessException la clase tienen 3 constructores, 2 por los cuales es posible escribir un mensaje el cual luego será mostrado por pantalla en la interfaz como una ventana emergente.

En BusinessLogic.Interfaces se encuentran 2 clases. La primera, IRepository, interfaz que contiene las firmas para las distintas operaciones de interacción con la base de datos requeridas y por lo tanto definidas por las reglas del negocio de la aplicación. Las operaciones utilizan una entidad genérica T, la cual puede ser sustituida por cualquiera de las entidades a persistir (presentes en BusinessLogic.Domain). La segunda es ParkingRepository, clase abstracta que toma un tipo T e implementa la interfaz IRepository y sus operaciones algunas ya implementadas en dicha clase, algunas con posibilidad a ser sobreescritas por las clases que hereden de esta y otras declaradas como abstract obligando a dichas clases a implementarlas.

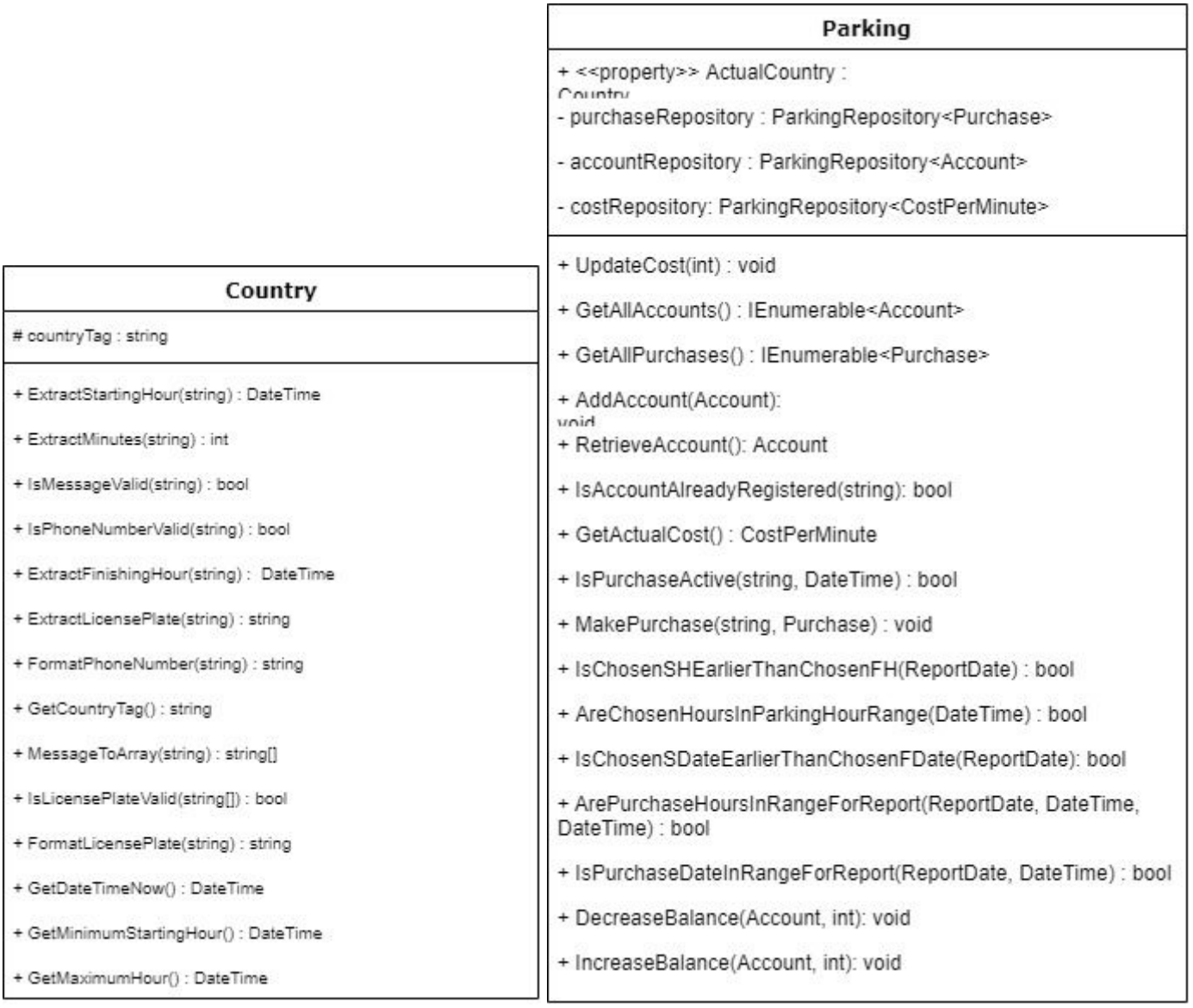


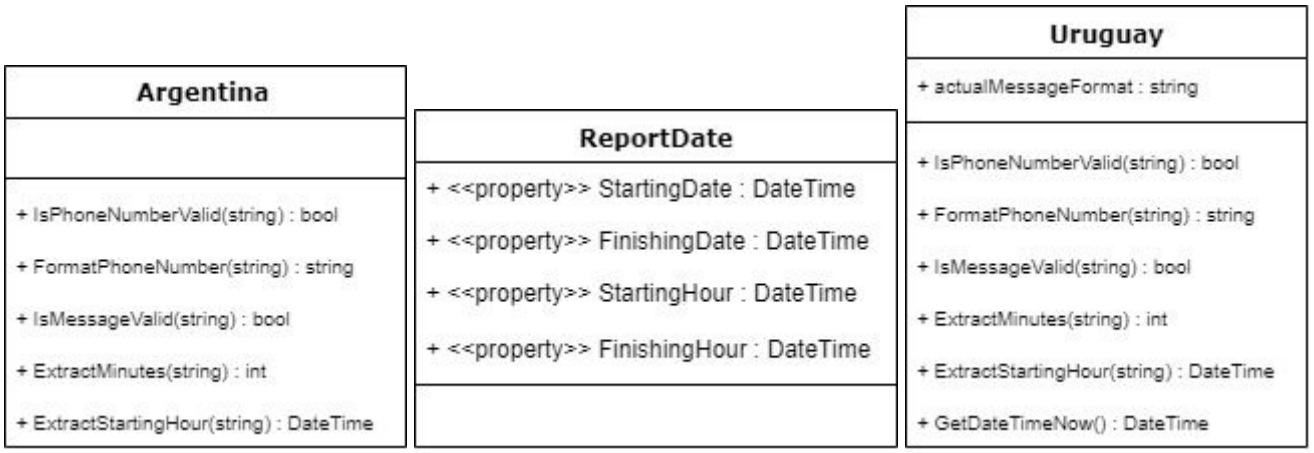
En BusinessLogic.Logic se encuentran las clases relacionadas con la lógica del negocio. Realizan las validaciones y operaciones necesarias. Se identifican 5 clases:

- Country: Clase abstracta la cual contiene métodos de validación, parseo y formateo de mensajes y números de teléfono.
- Argentina: Clase que hereda de Country, la cual utiliza métodos definidos en Country, sobre escribiendolos cuando es necesario para cumplir con el los

estándares de la región (Ejemplo: validar números de teléfono con formato de argentina) o creando métodos propios.

- Uruguay: Clase que hereda de Country, la cual utiliza métodos definidos en Country, sobre escribiendolos cuando es necesario para cumplir con el los estándares de la región (Ejemplo: validar números de teléfono con formato de uruguay) o creando métodos propios.
- ReportDate: Clase creada para modelar un rango de fechas y horas para los reportes.
- Parking: Clase que tiene una property de Country, y atributos privados de ParkingRespository, uno por cada entidad del dominio. Esta clase es la intermediaria entre la UI y las clases que implementan las operaciones de la base de datos, las cuales heredan de ParkingRepository.

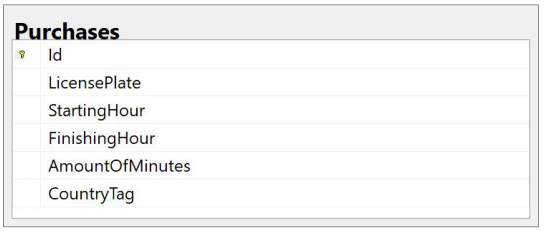




En BusinessLogic.Persistance se encuentran las clases relacionadas con la persistencia de objetos e implementación de operaciones con la base de datos. Se identifican 4 clases:

- ParkingContext: Clase que sirve de contexto la cual hereda de DbContext que mantiene las listas por tipo de las entidades del dominio las cuales serán almacenadas en la base de datos.
- CostRepository: Clase que hereda de ParkingRepository, y por lo tanto implementa IRepository. Se encarga de manejar las operaciones de base de datos definidas en la interfaz, para la entidad CostPerMinute y su lista Costs en ParkingContext.
- PurchaseRepository: Clase que hereda de ParkingRepository, y por lo tanto implementa IRepository. Se encarga de manejar las operaciones de base de datos definidas en la interfaz, para la entidad Purchase y su lista Purchases en ParkingContext.
- AccountRepository: Clase que hereda de ParkingRepository, y por lo tanto implementa IRepository. Se encarga de manejar las operaciones de base de datos definidas en la interfaz, para la entidad Account y su lista Accounts en ParkingContext.

Modelo de tablas de la estructura de la base de datos, en el diagrama podemos ver las clases del paquete BusinessLogic.Domain representadas como tablas en la base de datos:



En el paquete BusinessLogic.Test se encuentran 3 carpetas de pruebas unitarias. DomainTests, DatabaseTests y LogicTests las cuales contienen clases de prueba para las clases de cada carpeta en el dominio con el mismo nombre. Se intentó que estas tuvieran el mismo nivel de código que las clases dominio, poniendo en cada caso nombres de métodos claros y específicos referentes al caso de prueba que se está probando y los datos con los que se va a trabajar.

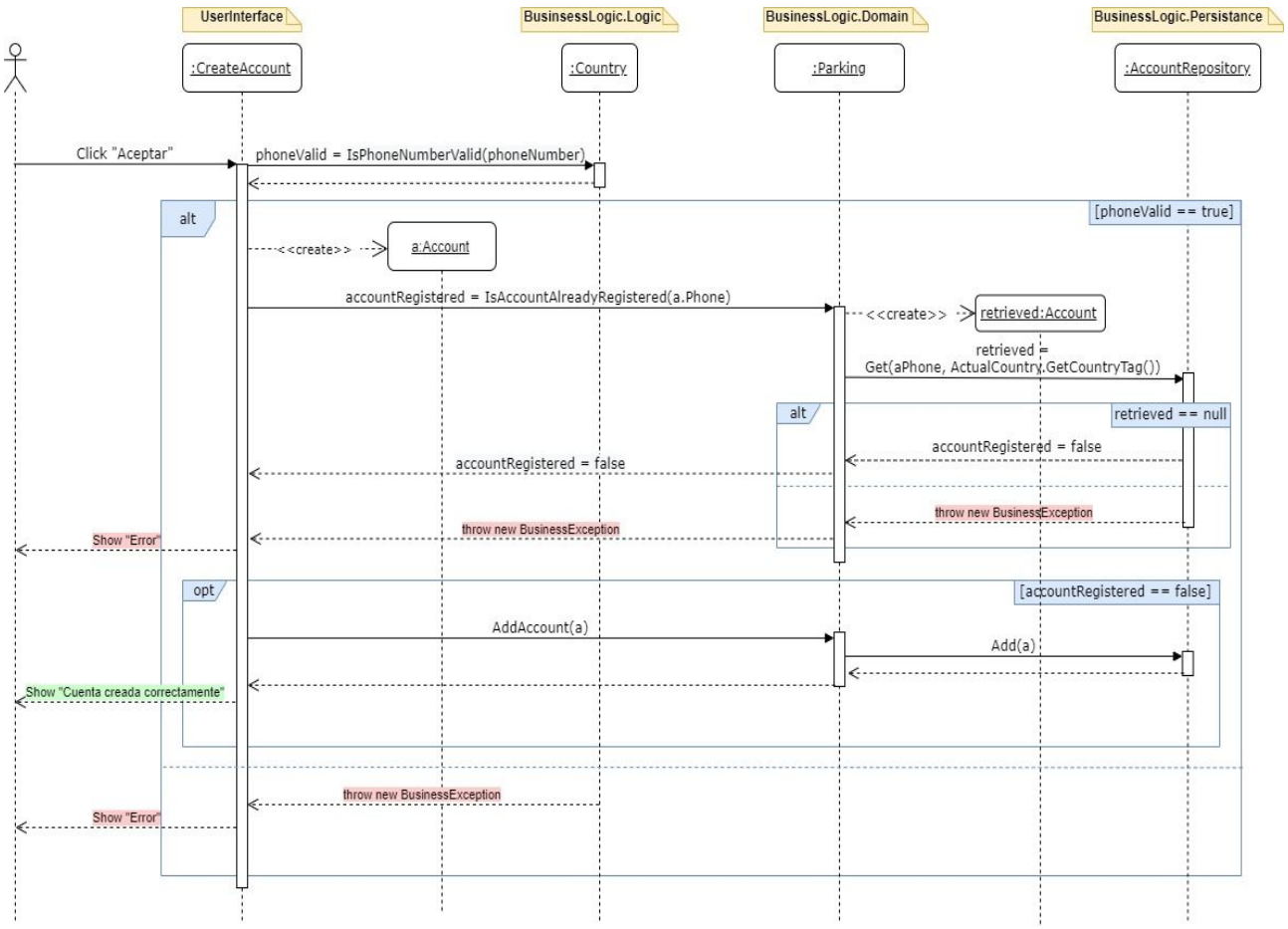
En un momento de la etapa de desarrollo del proyecto, se decidió cambiar los métodos de prueba relacionados con horas, implementando el uso de Mocks para modelar las mismas y no depender de la hora local de la computadora en la que se está probando. Así se pudo evitar que algunas pruebas fallen al intentar acceder a la hora actual, haciendo uso del método DateTime.Now.

En el paquete UserInterface, se encuentra un Form llamado UserInterface y un UserControl por cada operación posible excepto cambiar país la cual se realiza en UserInterface, no tiene una ventana propia. Se puede ver el país para el que se encuentra la aplicación indicado en una etiqueta que se muestra en todas las ventanas todo el tiempo. En UserInterface hay un panel en el cual se van cargando los user controls correspondientes a las distintas funciones que la aplicación ofrece. Por medio de un MenuStrip el usuario accede a estas funciones.

Diagrama de secuencias

Para modelar brevemente el funcionamiento de una función y la interacción con el usuario, modelamos dos diagramas de secuencia.

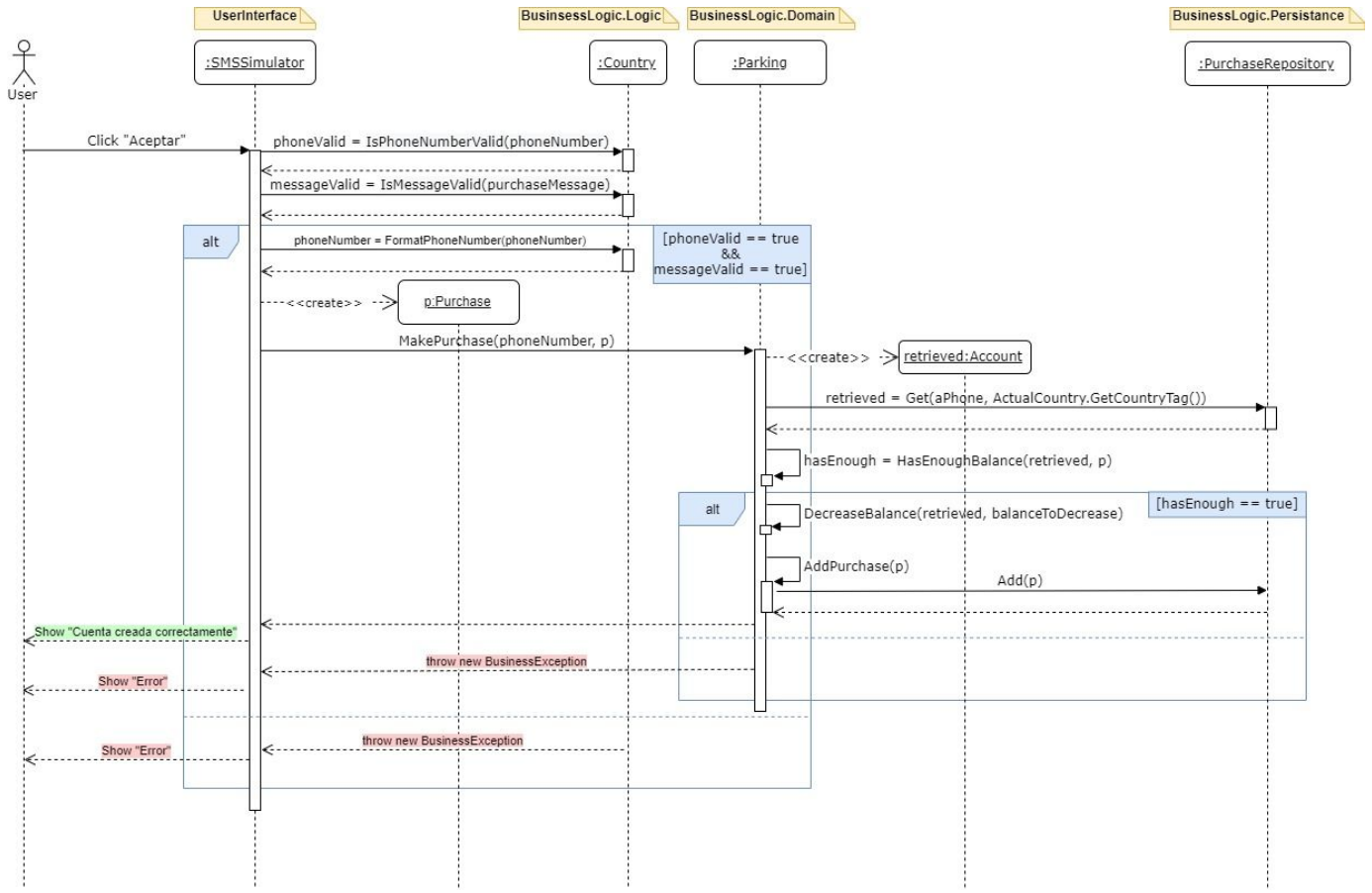
En la siguiente imagen mostramos el diagrama de la acción Crear Cuenta. La misma inicia luego de que el usuario presiona el botón Aceptar.



Allí se muestra el proceso que hace internamente el programa y las devoluciones que hace al usuario. Cómo para distintos casos de uso se retornan distintas cosas y cómo interactúan internamente los objetos.

Se ve que luego de presionar en Aceptar, la ventana pide al país que pertenece validar si el número ingresado es correcto. De serlo continúa con la creación de un objeto Account la cual luego, mediante distintas interacciones con Parking y la base de datos, pregunta si existe en el sistema y de no existir se añade. En todos los demás casos que no siga el curso normal de Crear Cuenta, se muestra un error por pantalla diciendo qué está mal. Por ejemplo, que el teléfono no sea válido, que la cuenta ya esté registrada, etc.

Luego se hizo otro diagrama para mostrar el caso de uso del Registrar Compra. El mismo comienza de la misma manera.



Este método ha de ser un poco más complejo ya que las validaciones y las interacciones dentro del mismo son en mayor cantidad, pero el razonamiento es el mismo.

Para crear una compra, el usuario debe ingresar un número de teléfono y un mensaje el cual incluye matrícula, minutos que desea comprar, hora de inicio y país, el cual determina si alguna de las anteriores es opcional o no.

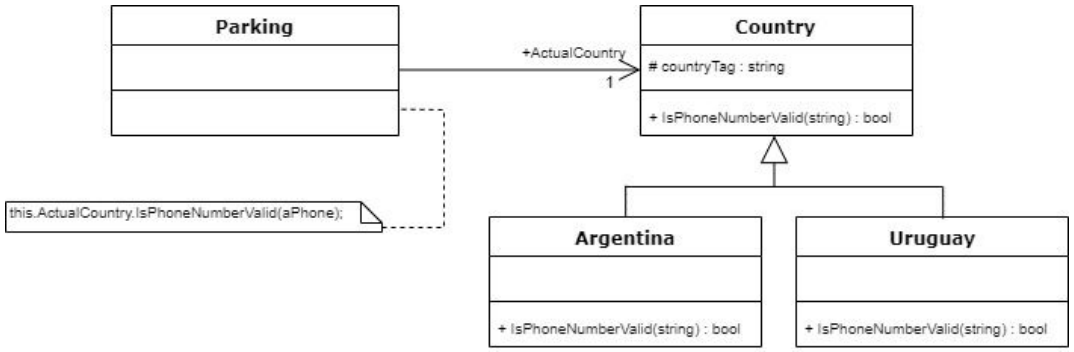
Luego de apretar Aceptar, se verifica que los datos ingresados sean válidos. De serlo, se crea una compra y se obtiene desde la base de datos la cuenta con la cual se creó. De ella se verifica si el saldo es suficiente y se efectúa la compra, decrementando el saldo y agregando la misma a la base de datos.

Todos los demás cursos alternativos terminan en un error que es distinto dependiendo en la etapa que se encuentren y el tipo del mismo. Para mostrar generalizado se puso Show “Error”, pero ese depende del mensaje que traiga la exception.

Principales decisiones de diseño tomadas y mecanismos generales

El problema central de este obligatorio, era implementar una solución que contemplase el uso de distintos medios de validación dependiendo del país en el que se fuese a usar la aplicación. También era sabido que el software iba a ser llevado a otros países lo que establecía que el problema debía ser resuelto de manera flexible para así facilitar el agregado de nuevos países a los programadores que fuesen a trabajar en el código en un futuro. Para modelar esto se utilizó el patrón de comportamiento visto en el curso conocido como Strategy. Dicho algoritmo permite a los algoritmos variar independientemente de las clases que los utilizan. Debido a que como requisito se pedía poder cambiar de país en tiempo de ejecución, este patrón provee una solución óptima para resolver el problema.

Se aplicó de la siguiente forma, una clase abstracta Country contiene métodos para validar mensajes de compra y número de teléfono, parsear mensajes, formatear números de teléfono y matrículas. Los métodos con visibilidad pública en Country modelan las necesidades de la clase sin importar el tipo de país que se requiera, ya que son las clases que hereden de Country (los países), las que implementaran, o sobre escriban de ser necesario dichos métodos. Al tener estos métodos visibilidad pública, pueden ser llamados desde afuera de la clase, al existir una instancia de Country en Parking se pueda acceder a ellos mediante una llamada polimórfica. Así al cambiar de país solo se genera una nueva instancia de la clase país y que se precise y se setea el atributo ActualCountry en Parking, pero la llamada a los métodos no cambia. Es independiente del país que se esté utilizando. Para la solución actual existen 2 países, Argentina y Uruguay. El tipo de diseño aplicado a su vez provee fácil implementación de un nuevo país, teniéndose que crear una nueva clase que herede de Country e implemente y sobre escriba las operaciones necesarias y/o también modifique operaciones en Country si se desean aplicar nuevas validaciones o se requiere modificar los datos a validar, parsear o extraer por país.



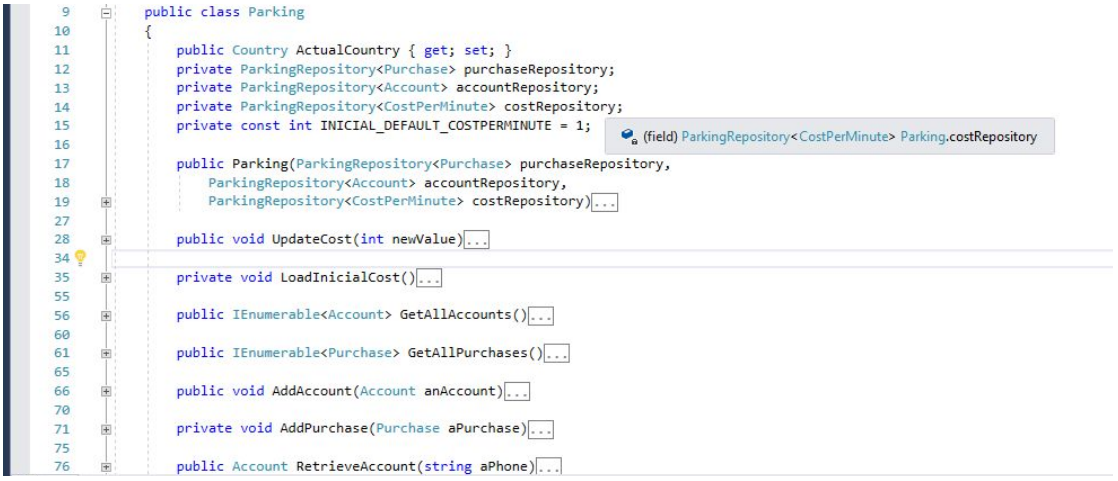
Como fue mencionado previamente Parking tiene como atributos a 3 objetos de ParkingRepository que varían según entidad del dominio que manejan. Ya que un Parking no podría crearse sin tener estos atributos que a su vez tienen cada uno la misma instancia de ParkingContext, se optó por tener un constructor de Parking que tome 3 argumentos (uno por cada implementación de repositorio) y así aplicar lo que se conoce como inyección de dependencias. Por más que Clean Code establece que se debe tener como máximo 2 parámetros por métodos, en este caso consideramos que era necesario para que se realice la inyección correctamente ya que dichas implementaciones de repositorio son esenciales para que la aplicación funcione correctamente y tampoco creemos que no se entienda que hace la función (constructor en este caso) como establece Clean Code que sucede al exceder la cantidad máxima de argumentos por parámetro.

```
ParkingContext context = new ParkingContext();
ParkingRepository<Purchase> purchaseRepository =
    new PurchaseRepository(context);
ParkingRepository<Account> accountRepository =
    new AccountRepository(context);
ParkingRepository<BusinessLogic.Domain.CostPerMinute> costRepository =
    new CostRepository(context);

MyParking = new Parking(purchaseRepository,
    accountRepository, costRepository);
```

Se aplicó el principio de diseño conocido como DIP, (Dependency Inversion Principle) o principio de inversión de dependencias el cual establece que módulos de alto nivel no deben depender de módulos de bajo nivel, sino que ambos deben depender de interfaces. En nuestro caso se da en que UserInterface (alto nivel), depende de BusinessLogic.Logic ya que en él está Parking, clase que tienen como atributos a diferentes instancias de ParkingRepository (clase abstracta que implementa la interfaz IRepository), y son las implementaciones de esta última clase las que interactúan con la base de datos pero al ser hijas de ParkingRepository, implementan y dependen por lo tanto de IRepository, que define las operaciones que los hijos de ParkingRepository deberán implementar. Mirar Diagrama de paquetes para entender mejor, cómo UserInterface termina dependiendo de IRepository, y a su vez las clases que implementan las operaciones en BusinessLogic.Persistance dependen de IRepository presente en BusinessLogic.Interfaces

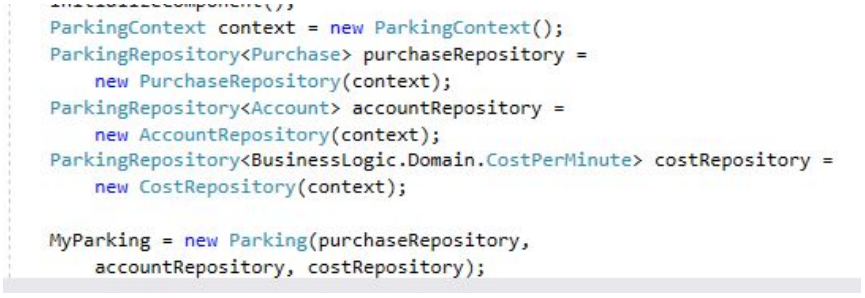
Por último, la otra consideración importante que se hizo fue la creación de una clase abstracta ParkingRepository con un tipo genérico T, que implemente la interfaz IRepository, de la cual los hijos serán los que implementen, sobre escriban o utilizen del padre, las operaciones con la base de datos definidas en la interfaz. La otra alternativa es que, no hubiera clase abstracta sino una única implementación de una clase que implemente la interfaz y las las operaciones, pero consideramos que sería una mala decisión de diseño ya que tendría muy baja cohesión. Ahora si separamos las clases por tipo de entidad que implementen la interfaz, se repetiría código en muchos casos. Por lo que se decidió que una clase abstracta con un tipo de entidad (clase del dominio) genérico T y que algunos métodos que no varían según entidad esten ya implementados, otros abstractos (obligados a implementar por los hijos) y otros sobre escribibles en el caso de que sea necesario. A su vez esto genera que si se desea persistir un nuevo tipo de dato (clase en el dominio), la implementación de sus operaciones sea más fácil de lo que sería si se tuviera que realizar por completo (si no existiera la clase abstracta de tipo genérico T) .

Ejemplo de uso de ParkingRepositry en Parking, y su instanciación.



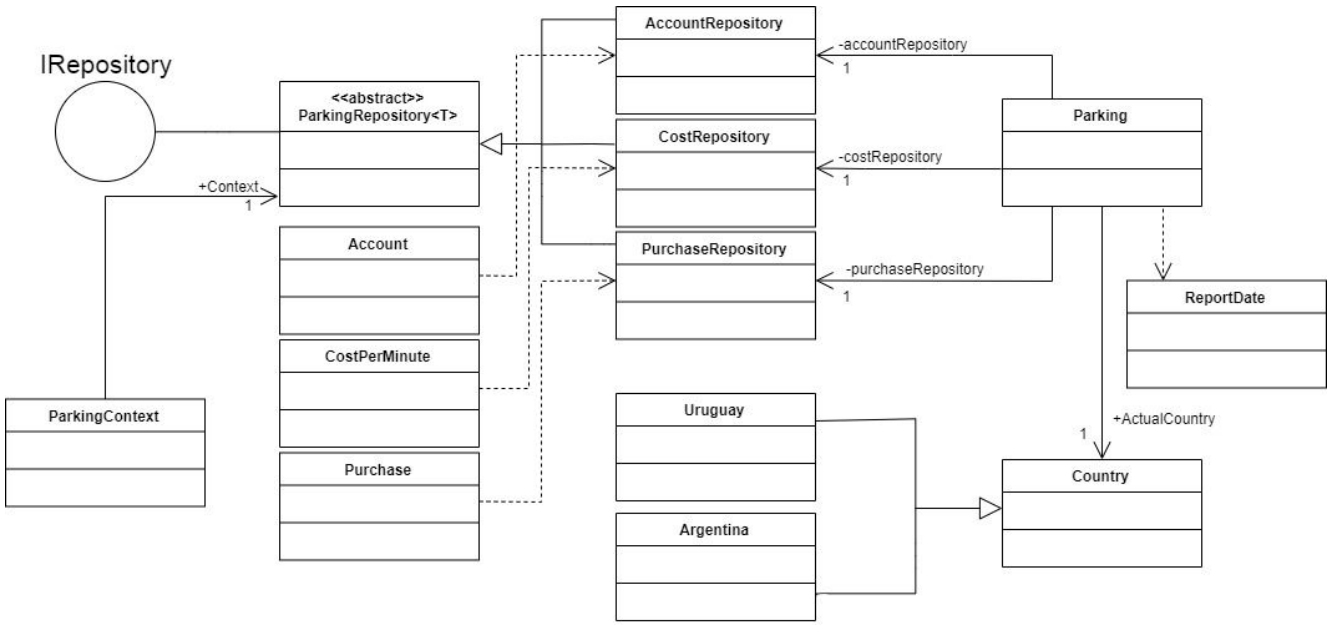
```
public class Parking
{
    public Country ActualCountry { get; set; }
    private ParkingRepository<Purchase> purchaseRepository;
    private ParkingRepository<Account> accountRepository;
    private ParkingRepository<CostPerMinute> costRepository;
    private const int INICIAL_DEFAULT_COSTPERMINUTE = 1;

    public Parking(ParkingRepository<Purchase> purchaseRepository,
        ParkingRepository<Account> accountRepository,
        ParkingRepository<CostPerMinute> costRepository)...

    public void UpdateCost(int newValue)...

    private void LoadInicialCost()...

    public IEnumerable<Account> GetAllAccounts()...

    public IEnumerable<Purchase> GetAllPurchases()...

    public void AddAccount(Account anAccount)...

    private void AddPurchase(Purchase aPurchase)...

    public Account RetrieveAccount(string aPhone)...
```

(field) ParkingRepository<CostPerMinute> Parking.costRepository



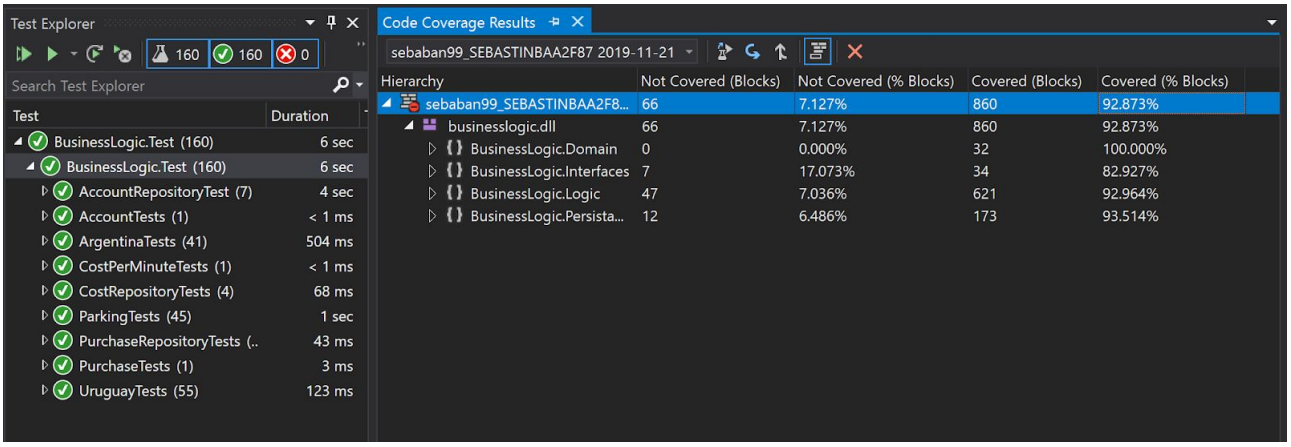
```
ParkingContext context = new ParkingContext();
ParkingRepository<Purchase> purchaseRepository =
    new PurchaseRepository(context);
ParkingRepository<Account> accountRepository =
    new AccountRepository(context);
ParkingRepository<BusinessLogic.Domain.CostPerMinute> costRepository =
    new CostRepository(context);

MyParking = new Parking(purchaseRepository,
    accountRepository, costRepository);
```

UML general


IRepository
<<abstract>>
ParkingRepository<T>
+Context
1
ParkingContext
Account
CostPerMinute
Purchase
AccountRepository
CostRepository
PurchaseRepository
-accountRepository
1
-costRepository
1
-purchaseRepository
1
Parking
ReportDate
+ActualCountry
1
Country
Uruguay
Argentina

Cobertura de las pruebas unitarias

La cobertura de código se realizó con la herramienta para analizar código presente en Visual Studio 2019 Enterprise Edition. La cobertura solo debía realizarse sobre el dominio por lo cual se incluyó la etiqueta [ExcludeFromCodeCoverage] encima del nombre de las clases pertenecientes al paquete de pruebas y las clases de la carpeta BusinsessLogic.Exceptions. El resultado fue de 92,87%. Resultado el cual podrán verificar si corren el mismo análisis sobre la solución final. A continuación una foto de la salida resultante del análisis. Como especifica la letra, si para algún caso no se llegó a superar 90% se debe explicar porqué. El único caso en que no se llegó al 90% fue en la clase Parking Repository. Esto es debido a que no se realizaron pruebas para los bloques catch(DbException) ya que estos están para manejar problemas de conexión con la base de datos. Se realizaron pruebas funcionales para estos bloques catch, apagando el servicio SQL Server Express en medio de la ejecución. El comportamiento fue el esperado.



Link al repositorio

https://github.com/ORT-DA1/202711-209234.git