Parking App

Obligatorio 1 - 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). El objetivo final es que los usuarios de una empresa telefónica puedan, mediante el envío de mensajes de texto, comprar minutos de estacionamiento.

Implementación

El proyecto fue desarrollado en el lenguaje de programación C# usando el framework .NET (Versión 4.6.1) proveído por Microsoft Visual Studio.

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

Si se ingresa una compra y en el mismo minuto se consulta por su existencia, devuelve que la compra no está activa. Se intentó resolver el bug, pero por más que se solucionó, el método encargado de realizar la validación en cuestión. IsPurchaseInRange(Purchase, DateTime), se vio afectado y complejizado afectando otros métodos si se quisiera mantener el estándar de código limpio. Debido a la falta de tiempo se decidió reportar el bug y no ponerse a implementar cambios de de apuro sin el correspondiente chequeo y refactoreo.

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*.
- Se supone que el inspector, al consultar por la actividad de una compra, pregunta por momentos referidos al momento actual o momentos pasados. Si pregunta por una compra futura, mostrará un error.
- Se supone que el costo por minuto vale 1 al iniciar el programa.

Descripción y justificación del diseño

El sistema fue dividido en tres proyectos:

- Proyecto de la lógica del negocio: BusinessLogic
- Proyecto de pruebas de la lógica del negocio: BusinessLogic.Test
- Proyecto de interfaz gráfica: UserInterface

En el paquete BusinessLogic se pueden encontrar dos paquetes:

- Domain
- Exceptions

BusinessLogic.Domain, realizado completamente con TDD. En él se encuentran todas las clases, cuyas funciones satisfacen los requerimientos de negocio. Se identifican 3 clases:

- Cuenta: modela la cuenta de un usuario cuyos atributos son número de teléfono y saldo.
- Compra: 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) y la cantidad de minutos.
- Parking: modela el sistema total, contiene un entero que representa el costo del estacionamiento por minuto, una lista con las cuentas registradas y otra con las compras registradas (activas y pasadas).

En BusinessLogic.Exceptions se encuentra la clase BusinessException, 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.

En el paquete BusinessLogic.Test se encuentran 3 clases de pruebas unitarias. ParkingTests, AccountTests y PurchaseTests las cuales prueban cada clase correspondientemente y están asociadas también a la clase BusinessException.

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.

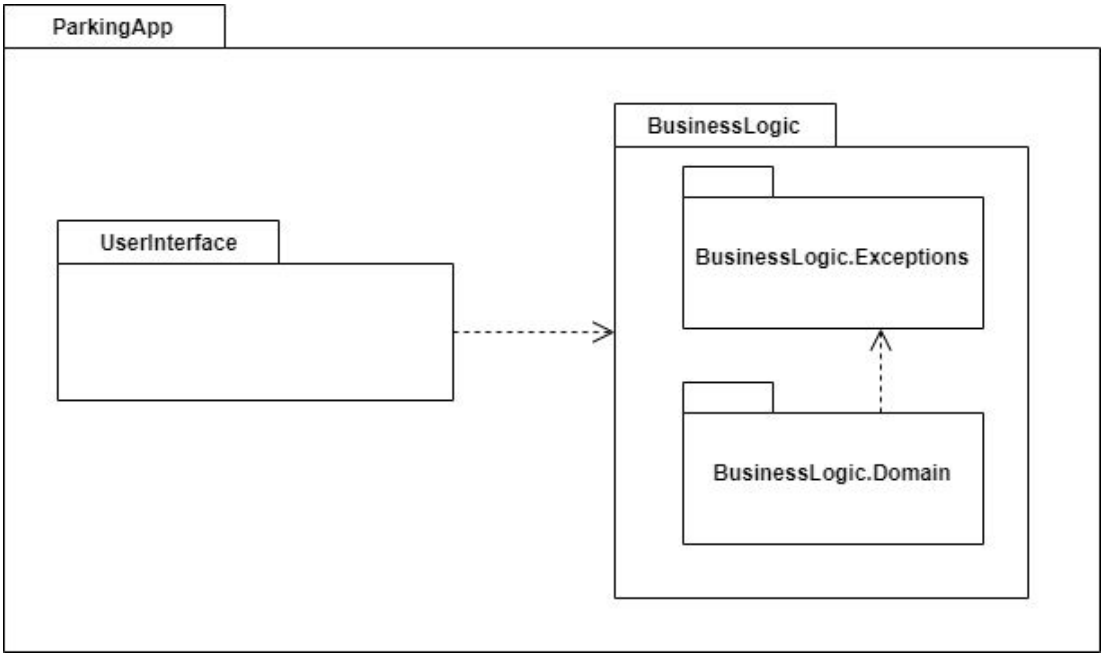
Ejemplo de una prueba mockeada:

```
[TestMethod]
public void IsPurchaseActiveTestIsActiveTrue()
{
    Account anAccount = new Account() { Phone = "098 740 956", Balance = 500 };
    aParking.AddAccount(anAccount);

    Mock<Purchase> mockedPurchase = new Mock<Purchase>();
    DateTime aDate = DateTime.Today;
    aDate = aDate.AddHours(13);
    mockedPurchase.Setup(m => m.GetDateTimeNow()).Returns(aDate);
    mockedPurchase.Object.SetPurchaseProperties("SBT 4505 120 13:00");

    aParking.MakePurchase("098 740 956", mockedPurchase.Object);

    DateTime chosenMoment = DateTime.Today;
    chosenMoment = chosenMoment.AddHours(13);
    chosenMoment = chosenMoment.AddMinutes(5);

    Assert.IsTrue(aParking.IsPurchaseActive("SBT 4505", chosenMoment));
}
```

Se decidió hacer más bloques de los recomendados para entender mejor la prueba.
Primero hay cuatro bloques los cuales setean el contexto de la prueba, uno para la cuenta, dos para la compra y uno para el chosenMoment.
En este ejemplo se utiliza el mock en compra, generando que esta prueba pase independientemente de la hora del sistema.
Una consideración de diseño que se tomó fue sacrificar un poco de prolijidad en la clase Purchase al hacer el método GetDateTimeNow() virtual y público (aunque no sea un método relacionado con la funcionalidad de la clase). Pero ganando más usabilidad al hacer la aplicación más testeable.
se tuvo que sacrificar un poco de diseño, ya que se hizo un método public virtual para poder sobreescribirlo y así lograr que las pruebas no fallen.
Luego está el bloque que contiene al assert, donde ahí se genera y verifica que la prueba sea correcta.

En el paquete UserInterface, se encuentra un Form llamado UserInterface. En él se cargan paneles correspondientes a las distintas funciones que la aplicación ofrece. Por medio de un MenuStrip el usuario accede a estos paneles dependiendo la función que quiera utilizar.

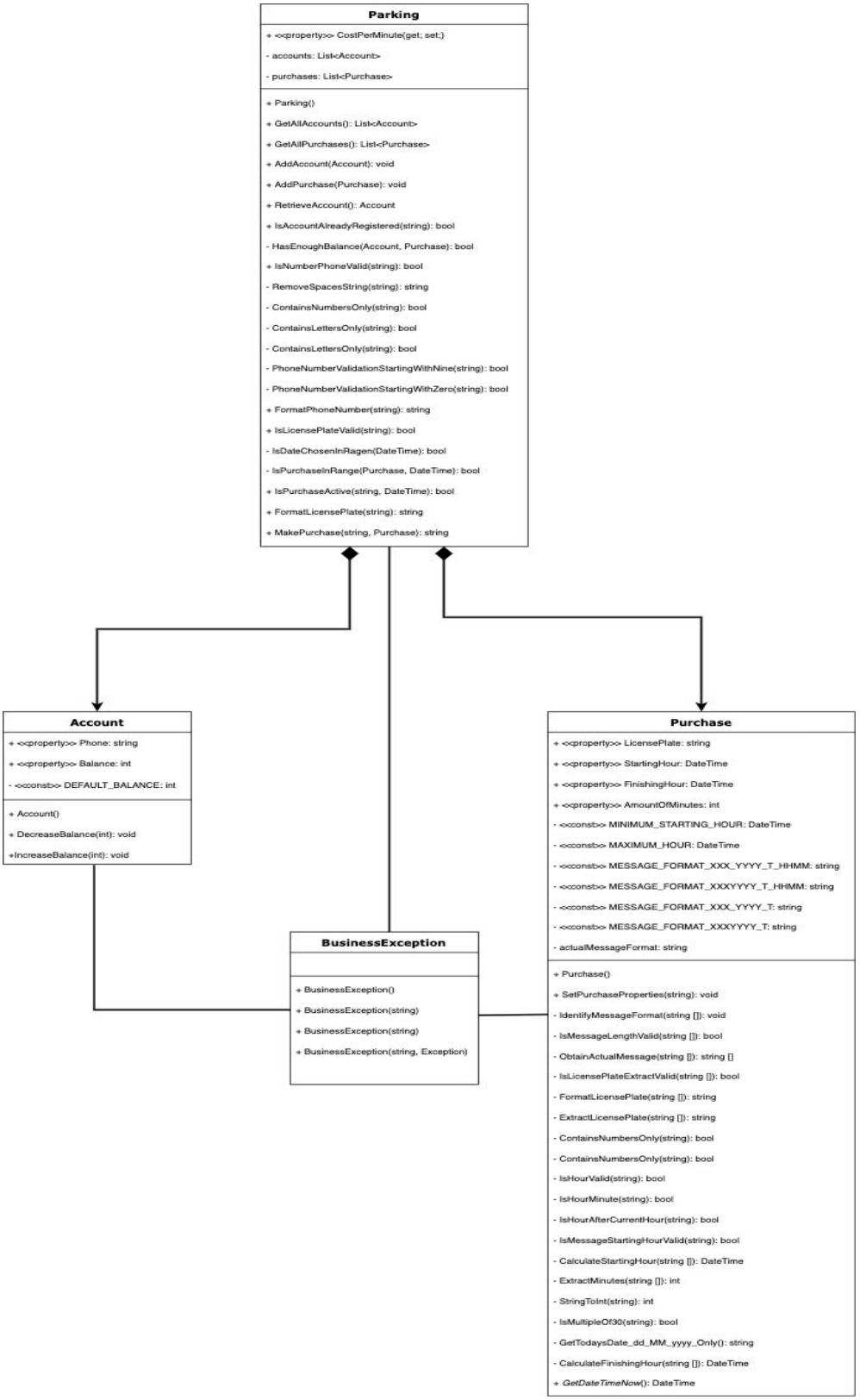
Diagrama de paquetes



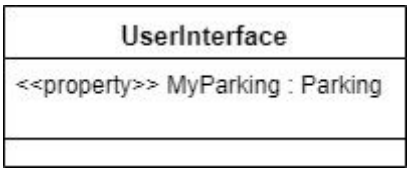
Aquí se muestran los paquetes del sistema, ignorando el paquete BusinessLogic.Test de acuerdo a lo especificado en la letra.

Diagrama de clases

Package BusinessLogic



Package UserInterface



Evidencia de Clean Code

Nos aseguramos de usar nombres nemotécnicos y representativos tanto para las variables como para los métodos implementados.

A lo largo de todo el proyecto se intentó mantener los mismos nombres para representar las mismas cosas.

Se evitó totalmente el uso de comentarios, ya que estos no se consideran necesarios ya que el código es fácil de comprender, gracias a la claridad de codificación.

En los casos que se consideró necesario, se extrajo condiciones dentro de sentencias “if” a métodos booleanos para agilizar la lectura y comprensión de las mismas.

En UserInterface (Form), también se extrajeron las sentencias que estaban dentro de los try para tener una mejor visualización y lograr un código menos extenso.

Se intentó hacer un uso extenso de encapsulación a lo largo de todo el dominio para que los métodos sean lo más claros posibles e incluso en algunos casos poder leerlos como un pseudocódigo.

Se evitó repetir código. Salvo en algunos casos que por tema de tiempo y complejidad, no pudieron ser resueltos.

Se cuidó específicamente, en toda la solución, que las funciones no recibieran más 2 parámetros.

Se creó una excepción propia la cual es invocada con mensajes de error descriptivos para informar al usuario lo que está sucediendo luego de que realiza una acción. Esto mejora la usabilidad de proyecto.

Se usó Pascal Case para los métodos, respetando el estándar de C#.

Se implementó el uso de constantes para mejorar la legibilidad del código.

```
public void MakePurchase(String aPhone, Purchase aPurchase)
{
    Account newAccount = RetrieveAccount(aPhone);

    if (HasEnoughBalance(newAccount, aPurchase))
    {
        this.AddPurchase(aPurchase);
    }
}
```

En este ejemplo del método MakePurchase, se pueden ver varios de los puntos mencionados anteriormente. Aquí se muestra como se invoca a distintos métodos, que se sabe qué hacen explícitamente por su nombre, y vemos claramente de que el método hace una única cosa. Realizar la compra. Valida mediante otros métodos y se mantiene concreto en lo que desea hacer.
En este ejemplo ya se asume que el telefono que se le pasa esta validado y la compra tambíen.

Diseño de casos de prueba

Pruebas funcionales

Se adjuntan dos videos mostrando la funcionalidad del sistema.

PruebasFuncionales1: https://youtu.be/s2jHFjPhOQE
En esta prueba se omitió probar la funcionalidad de consultar si una compra está activa, por lo tanto se adjunta en el video PruebasFuncionales2.

PruebasFuncionales2: https://youtu.be/w3w-v1KPNL0

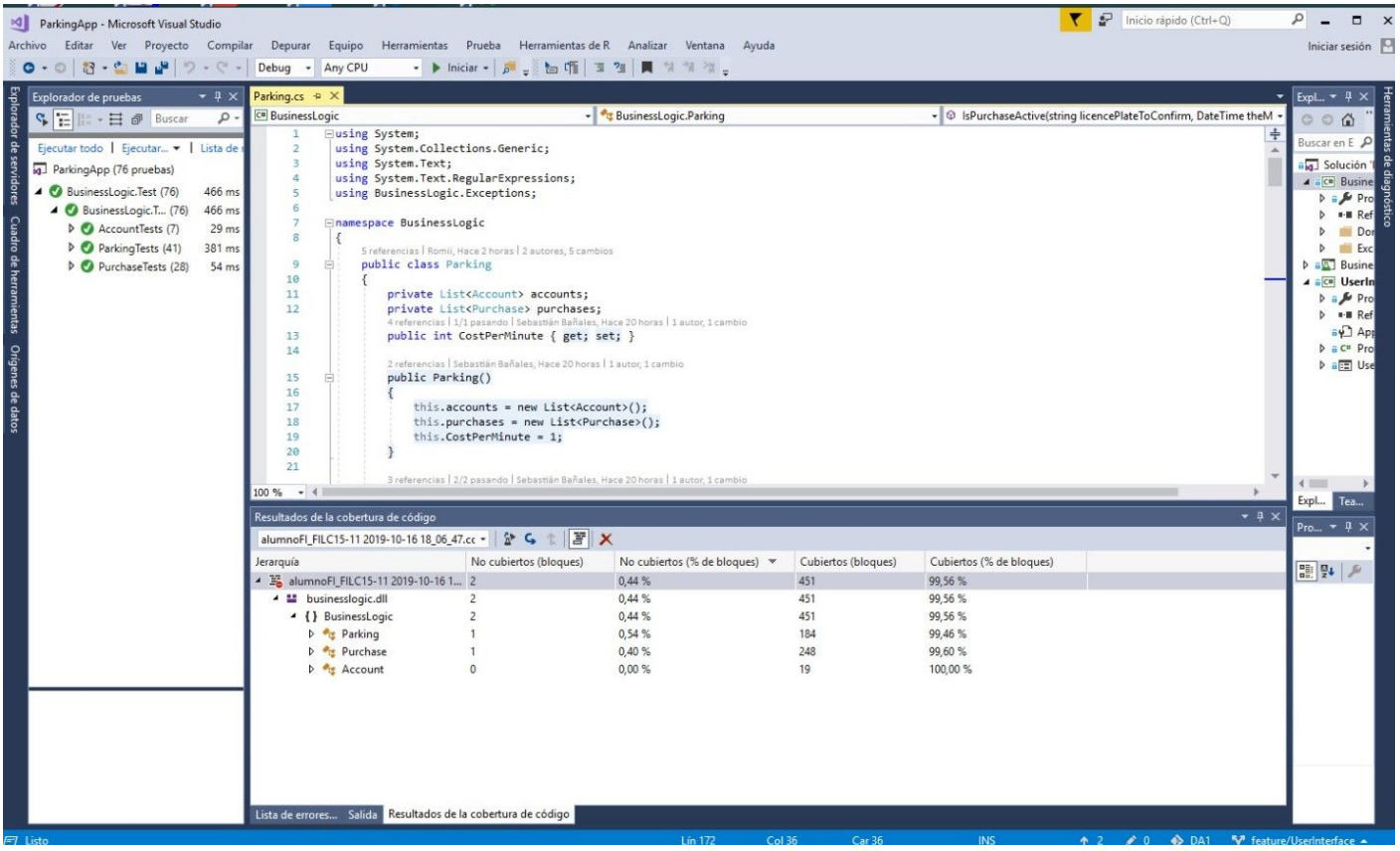
Pruebas unitarias

Se adjunta una carpeta mostrando algunas imágenes capturadas donde se puede observar el proceso de TDD.

https://drive.google.com/drive/folders/148VMHhIaaodEzTi3ylY6TJ64iyYOmsRO?usp=sharing

Cobertura de las pruebas unitarias

La cobertura de código se realizó con la herramienta para analizar código presente en Visual Studio 2017 Enterprise Edition, software presente en las computadoras de los laboratorios de la facultad. 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 a la clase BusinsessException. El resultado fue de 99,56%. Resultado el cual podrán verificar si corren el mismo análisis sobre la solución final. A continuación una foto de la salida resultante del análisis.



Link al repositorio

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