



Obligatorio 1 - Diseño de aplicaciones 2

Evidencia de Clean Code y de la aplicación de TDD

Agustín Hernandorena (233361)
Joaquín Lamela (233375)

<https://github.com/ORT-DA2/233375-233361Obl>

Aplicación de TDD	3
Estrategia de TDD seguida	3
Cobertura de las pruebas	4
BusinessLogic	5
BusinessLogicException	5
BusinessLogicTest	5
DataAccess	5
DataAccessTest	6
Domain	7
DomainException	7
Filters	7
Models	7
Repository Exception	8
WebApi	8
WebApiTest	8
Clean Code	9
Nombres nemotécnicos	9
Nombre de clases y métodos	9
Funciones	10
Parámetros de las funciones	10
Comentarios	11
Formateo	11
Horizontal	11
Vertical	12
Ley de Demeter	13
Manejo de errores	14
Pruebas unitarias	14
Clases	15

Aplicación de TDD

Con respecto a los tests, la funcionalidades de la aplicación se desarrollaron utilizando *TDD* (*Test Driven Development*). Realizamos *tests* para cada una de las capas de nuestra aplicación, es decir, tenemos un paquete de tests para la *WebApi*, otro para la *BusinessLogic* y otro para el paquete *DataAccess*.

A su vez, dentro de cada uno de estos paquetes de *tests* tenemos diferentes clases, que se utilizan para probar cada una de las entidades. En el paquete *WebApiTest*, tenemos una clase de prueba por cada controller de la *WebApi*, en el de *BusinessLogicTest* consideramos apropiado realizar una clase de test para cada clase manejadora del paquete *BusinessLogic*, y de la misma forma, en el paquete *DataAccessTest* tenemos una clase de Test por cada clase del paquete *DataAccess*. Esto lo hicimos con el objetivo de que los tests que prueban operaciones de una determinada clase estén todos juntos, y que sean independientes de otros que prueban otros tipos de objetos.

En particular se tomó esta decisión para seguir las características que deben poseer las pruebas unitarias, conocido como el principio *FIRST*, definido por el autor Robert Cecil Martín¹.

El principio *FIRST*, es el acrónimo de las cinco características que deben tener nuestros tests unitarios para ser considerados tests con calidad. Siendo las características:

1. F (fast-rápido): posibilidad de ejecutar un gran número de tests en cuestión de segundos.
2. I (independent-independiente): Todas las pruebas unitarias deben de ser independientes de las otras. En el momento que un test falla por el orden en el que se ha ejecutado, este test está mal desarrollado. El resultado no debe verse alterado ejecutando los tests en un orden y otro o incluso de forma independiente.
3. R (repeatable-repetible): El resultado de las pruebas debe ser el mismo independientemente del pc/servidor/terminal en el que se ejecute.
4. S (self-validating- auto evaluable): La ventaja de las pruebas automatizadas es que podemos ejecutarlas simplemente al pulsar un botón o incluso hacer que se ejecuten de forma automática tras otro proceso.
5. T (timely-oportuno): Las pruebas unitarias deben escribirse justo antes del código de producción que las hace pasar. Si se escriben pruebas después del código de producción, se puede encontrar que el código de producción es difícil de probar.

Estrategia de TDD seguida

La estrategia de *TDD* utilizada es *Inside-Out (bottom - up)*, es decir, comenzamos en el nivel de componente / clase (adentro) y agregamos pruebas a los requisitos. A medida que el código evoluciona (debido a las refactorizaciones), aparecen nuevos colaboradores, interacciones y otros componentes. El diseño fue guiado en absoluto por *TDD*.

¹ Interpretado de: Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education. Página: 132-133.

Al tener la lógica y el modelo de dominio bastante claros desde un comienzo, nos pareció oportuno comenzar desde “adentro”, es decir empezar por los componentes de más bajo nivel e ir evolucionando a medida que se introducen más pruebas y refactorizaciones.

Una ventaja importante que tuvimos al centrarnos primero en las partes individuales del sistema fue la capacidad de trabajar en incrementos pequeños. Esto nos permitió que el trabajo de desarrollo se pueda paralelizar dentro de nuestro equipo de desarrollo.

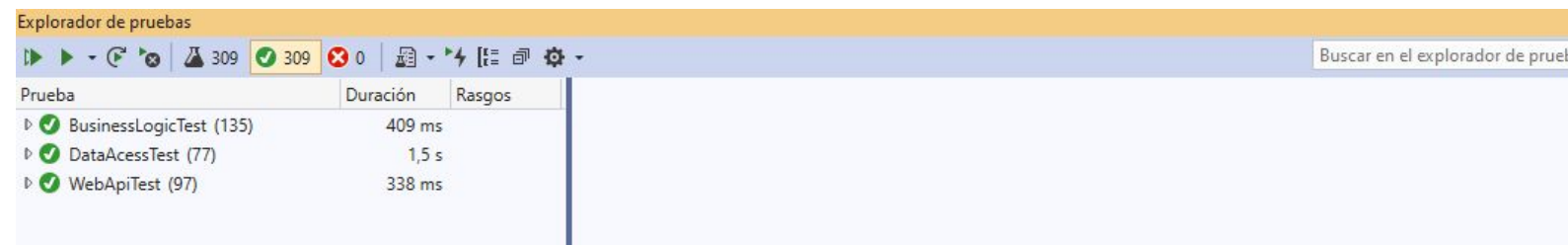
Otra ventaja que pudimos notar, fue la gran cobertura de código que obtuvimos al trabajar con este enfoque, esto debido a que cómo se trabaja con incrementos pequeños y además se sigue el proceso de TDD, esto conlleva al final y al cabo a una excelente cobertura de las líneas de código desarrollando, debido a que se pone un fuerte énfasis en el desarrollo guiado por las pruebas y en iteraciones pequeñas, lo cual forman un gran complemento el uno del otro, y dan como resultado la cobertura de pruebas obtenidas.

Cobertura de las pruebas

Utilizando el explorador de pruebas, podemos apreciar que se realizaron un total de 309 tests, obteniendo un porcentaje de pruebas exitosas igual al 100 %.

Esos 309 *tests* se dividen de la siguiente forma:

- 135 pertenecen al paquete *BusinessLogicTest*.
- 77 pertenecen al paquete *DataAccessTest*.
- 97 pertenecen al paquete *WebApiTest*.



Prueba	Duración	Rasgos
✔ BusinessLogicTest (135)	409 ms	
✔ DataAccessTest (77)	1,5 s	
✔ WebApiTest (97)	338 ms	

La aplicación fue desarrollada siguiendo *TDD (Test Driven Development)*. Esta práctica consiste en escribir antes la prueba que la funcionalidad. Para esto, en primer lugar, se escribe una prueba y se verifica que la nueva prueba falle. Luego, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito.

Utilizando esta práctica, se alcanza un porcentaje de cobertura de pruebas unitarias muy bueno. Si pasamos a ejecutar el analizador de cobertura de código, obtenemos finalmente que el porcentaje total de cobertura para toda la solución es igual al **98,05 %**.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Agustin Hernandorena_DESKTOP...	222	1,95 %	11161	98,05 %
businesslogic.dll	0	0,00 %	469	100,00 %
businesslogicexception.dll	0	0,00 %	17	100,00 %
businesslogictest.dll	102	1,98 %	5052	98,02 %
dataaccess.dll	35	7,22 %	450	92,78 %
dataacesstest.dll	41	4,31 %	911	95,69 %
domain.dll	15	2,91 %	500	97,09 %
domainexception.dll	0	0,00 %	13	100,00 %
filters.dll	0	0,00 %	23	100,00 %
model.dll	29	4,83 %	571	95,17 %
repositoryexception.dll	0	0,00 %	13	100,00 %
webapi.dll	0	0,00 %	399	100,00 %
webapitest.dll	0	0,00 %	2743	100,00 %

Ahora realizaremos un análisis de los resultados de cobertura obtenidos para cada paquete.

BusinessLogic

Pasando a analizar el paquete *BusinessLogic*, podemos notar que el porcentaje de cobertura de líneas de código del mismo es igual al **100 %**, esto quiere decir, que cada línea de las clases pertenecientes a BusinessLogic fueron testeadas.

businesslogic.dll	0	0,00 %	469	100,00 %
BusinessLogic	0	0,00 %	469	100,00 %
CategoryManagement	0	0,00 %	57	100,00 %
LodgingManagement	0	0,00 %	101	100,00 %
RegionManagement	0	0,00 %	19	100,00 %
ReserveManagement	0	0,00 %	64	100,00 %
ReserveManagement....	0	0,00 %	4	100,00 %
ReserveManagement....	0	0,00 %	4	100,00 %
TouristSpotManagem...	0	0,00 %	113	100,00 %
UserManagement	0	0,00 %	107	100,00 %

BusinessLogicException

En este paquete, podemos notar que el porcentaje de cobertura es igual al **100 %**, es decir, todas las excepciones de las reglas de negocio fueron testeadas por completo y de forma exitosa.

businesslogicexception.dll	0	0,00 %	17	100,00 %
BusinessLogicException	0	0,00 %	17	100,00 %
ClientBusinessLogicE...	0	0,00 %	6	100,00 %
DomainBusinessLogic...	0	0,00 %	4	100,00 %
MessageExceptionBu...	0	0,00 %	1	100,00 %
ServerBusinessLogicE...	0	0,00 %	6	100,00 %

BusinessLogicTest

Podemos apreciar que el porcentaje de cobertura de *BusinessLogicTest* es igual a **98.02 %**. Este análisis lo hacemos básicamente para saber si tenemos ciertas líneas en los tests que no son necesarias dejarlas porque no se están ejecutando, en este caso, al ingresar a revisar qué es lo que está sucediendo, notamos que no es que existan líneas que no se están ejecutando, sino que considera a las llaves "{" y "}" como parte del código y no lo está contabilizando cuando analiza la cobertura.

businesslogictest.dll	102	1,98 %	5052	98,02 %
{ } BusinessLogicTest	102	1,98 %	5052	98,02 %
CategoryTest	9	2,26 %	389	97,74 %
LodgingTest	24	2,16 %	1087	97,84 %
RegionTest	4	2,80 %	139	97,20 %
ReserveTest	15	1,76 %	839	98,24 %
TouristSpotTest	27	1,77 %	1502	98,23 %
UserTest	23	2,06 %	1096	97,94 %

Esto se puede apreciar en la siguiente imagen, donde claramente se puede observar que la llave que cierra al método de testeo, no está siendo contabilizada por el analizador de código.

```
[TestMethod]
[ExpectedException(typeof(ClientBusinessLogicException))]
0 referencias | Agustín Hernandorena, Hace 4 días | 1 autor, 1 cambio
public void ClientErrorInGetLodgingTest()
{
    var lodgingRepositoryMock = new Mock<ILodgingRepository>(MockBehavior.Strict);
    lodgingRepositoryMock.Setup(m => m.Get(It.IsAny<Guid>())).Throws(new ClientException());
    LodgingManagement lodgingLogic = new LodgingManagement(lodgingRepositoryMock.Object);

    Lodging resultOfGetLodging = lodgingLogic.GetLodgingById(lodging.Id);
}
```

Cómo se puede observar, lo que se muestra en rojo, son líneas las cuales no son alcanzadas por el analizador de cobertura, de tal manera que nos está queriendo decir que esa llave no llega a ser probada, lo cual no es cierto debido a que el test es ejecutado con normalidad. Por lo cual, ese resultado de cobertura es un tanto engañoso debido a que todos los test se ejecutan correctamente y cubren todas las líneas de los mismos.

DataAccess

Analizando el porcentaje de cobertura del paquete *DataAccess*, se obtiene que el mismo es igual a **92,78 %**. Analizando los métodos particulares en que la cobertura no llega al 100 %, notamos que esto se debe a que dentro de estos métodos se está capturando una *Exception* con el fin de que si ocurre un error interno en la base de datos, como puede ser una desconexión, el sistema pueda capturar esa excepción e informar al usuario que es lo que ha sucedido. Lo que sucede es que este tipo de excepciones, al surgir por un error inesperado de la base de datos, no es posible probarlas desde los métodos de *tests* (a no ser que se pruebe con mocking, pero no es adecuado para las pruebas de acceso a datos), y eso implica que el porcentaje de cobertura descienda.

Cabe destacar que aquí, se utilizó el tag `[ExcludeFromCodeCoverage]` con el fin de que no se tomen en cuenta las migraciones a la hora de realizar el análisis de cobertura de código.

dataaccess.dll	35	7,22 %	450	92,78 %
{ } DataAccess	35	7,22 %	450	92,78 %
BaseRepository<T>	6	8,96 %	61	91,04 %
CategoryRepository	2	9,52 %	19	90,48 %
ContextFactory	0	0,00 %	9	100,00 %
ContextObl	10	6,85 %	136	93,15 %
LodgingRepository	4	6,45 %	58	93,55 %
TouristSpotRepository	8	10,00 %	72	90,00 %
TouristSpotRepository...	0	0,00 %	2	100,00 %
TouristSpotRepository...	1	12,50 %	7	87,50 %
UserRepository	4	7,41 %	50	92,59 %
UserSessionRepository	0	0,00 %	36	100,00 %

En esta imagen, se puede apreciar como el *catch* de la *Exception*, no se encuentra testado por lo descrito previamente.

```

17 references | 15/15 passing | Agustin Hernandorena, 3 days ago | 1 author, 1 change
public Lodging GetLodgingByNameAndTouristSpot(string lodgingName, Guid touristSpotId)
{
    try
    {
        Lodging lodgingObtained = context.Set<Lodging>().Where(x => x.Name.Equals(lodgingName) && x.TouristSpotId == touristSpotId).FirstOrDefault();
        return lodgingObtained;
    }
    catch (Exception e)
    {
        throw new ServerException(MessagesExceptionRepository.ErrorObtainedLodgingByNameAndTouristSpotId, e);
    }
}

```

DataAccessTest

Analizando la cobertura del paquete *DataAccessTest* notamos que la misma es igual al **95.69 %**. La razón por la cual no se cubre un 100 % se debe a la misma razón que la del paquete *BusinessLogicTest* (está contabilizando como que las "{" y "}" no se están ejecutando).

dataacesstest.dll	41	4,31 %	911	95,69 %
{ } DataAccessTest	41	4,31 %	911	95,69 %
CategoryDATest	7	5,07 %	131	94,93 %
LodgingDATest	9	5,88 %	144	94,12 %
RegionDATest	4	3,70 %	104	96,30 %
ReserveDATest	4	3,01 %	129	96,99 %
TouristSpotDATest	10	4,48 %	213	95,52 %
UserDATest	4	4,30 %	89	95,70 %
UserSessionDATest	3	2,88 %	101	97,12 %

Esto se puede apreciar en la siguiente imagen, donde claramente se puede observar que la llave que cierra al método de testeo, no está siendo contabilizada por el analizador de código.

```

[TestMethod]
[ExpectedException(typeof(ServerException))]
public void TestRemoveCategoryInvalid()
{
    ContextObl context = ContextFactory.GetMemoryContext(Guid.NewGuid().ToString());
    ICategoryRepository categoryRepo = new CategoryRepository(context);

    Category categoryToAdd = new Category()
    {
        Id = Guid.NewGuid(),
        Name = "Playa"
    };

    categoryRepo.Remove(categoryToAdd);
}

```


De tal forma como se puede observar, lo que se muestra en rojo, son líneas las cuales no son alcanzadas por el analizador de cobertura, de tal manera que nos está queriendo decir que esa llave no llega a ser probada, lo cual no es cierto debido a que el test es ejecutado con normalidad. Por lo cual, ese resultado de cobertura es un tanto engañoso debido a que todos los test se ejecutan correctamente y cubren todas las líneas de los mismos.

Domain

Analizando la cobertura del paquete *Domain* podemos observar que la misma es igual al **97,09 %**. La razón por la cual no se llega al 100 % de cobertura, se basa básicamente es que no fueron probados, algunas secciones de métodos equals, específicamente en aquellos en que se recibe un objeto *null* por parámetro, luego restan probar algunos métodos get y set principalmente referidos a las clases *CategoryTouristSpot* y *LodgingPicture*, que son clases para representar las relaciones entre *CategoryTouristSpot*, y *Lodging* y *Picture* respectivamente, entonces los atributos de id, no son muy utilizados en las pruebas a nivel de lógica de negocios o dominio, debido a que el propósito que tienen dichos atributos es referido al nivel de base de datos.

domain.dll	15	2,91 %	500	97,09 %
{ } Domain	15	2,91 %	500	97,09 %
Category	1	4,17 %	23	95,83 %
CategoryTouristSpot	2	20,00 %	8	80,00 %
Lodging	2	1,54 %	128	98,46 %
LodgingPicture	3	37,50 %	5	62,50 %
Picture	2	33,33 %	4	66,67 %
Region	1	3,70 %	26	96,30 %
Reserve	2	1,31 %	151	98,69 %
TouristSpot	0	0,00 %	76	100,00 %
TouristSpot.<>c	0	0,00 %	2	100,00 %
User	1	1,49 %	66	98,51 %
UserSession	1	8,33 %	11	91,67 %

DomainException

Analizando la cobertura de *DomainException*, se observa que la misma es igual a **100 %**, es decir, que todas las excepciones relativas al dominio fueron testeadas.

domainexception.dll	0	0,00 %	13	100,00 %
{ } DomainException	0	0,00 %	13	100,00 %
CategoryException	0	0,00 %	2	100,00 %
LodgingException	0	0,00 %	2	100,00 %
MessageExceptionDo...	0	0,00 %	1	100,00 %
ReserveException	0	0,00 %	2	100,00 %
SearchException	0	0,00 %	2	100,00 %
TouristSpotException	0	0,00 %	2	100,00 %
UserException	0	0,00 %	2	100,00 %

Filters

Se puede apreciar que la cobertura del paquete *Filters* es igual al **100 %**, concluyendo entonces, que el *AuthorizationFilter* (único filtro presente), ha sido testeado por completo.

filters.dll	0	0,00 %	23	100,00 %
Filters	0	0,00 %	23	100,00 %
AuthorizationFilter	0	0,00 %	23	100,00 %

Models

Analizando la cobertura del paquete *Models*, se puede observar que la misma es igual a **95,17 %**. Básicamente este porcentaje se debe a que no todos los métodos *equals* se encuentran probados por completo, faltan testear algunos *gets* y *sets* de algunas *properties* en particular.

model.dll	29	4,83 %	571	95,17 %
Model	2	3,23 %	60	96,77 %
Model.ForRequest	4	2,45 %	159	97,55 %
Model.ForResponse	20	6,17 %	304	93,83 %
Model.ForResponseAndR...	3	5,88 %	48	94,12 %

Repository Exception

Se puede apreciar que el porcentaje de cobertura del paquete *RepositoryException* es igual a **100 %**. A partir de esto podemos indicar que todas las excepciones que fueron creadas para ser lanzadas cuando ocurre un error en la base de datos, están probadas por completo.

repositoryexception.dll	0	0,00 %	13	100,00 %
RepositoryException	0	0,00 %	13	100,00 %
ClientException	0	0,00 %	6	100,00 %
MessagesExceptionRe...	0	0,00 %	1	100,00 %
ServerException	0	0,00 %	6	100,00 %

WebApi

Analizando la cobertura de *WebApi* podemos observar que la misma asciende al **100 %**, es decir, que todos los controllers de la *API* fueron probados por completo. Cabe aclarar que en este paquete se utilizó el *tag* [ExcludeFromCodeCoverage] con el fin de que no se tomen en cuenta las clases *Program* y *Startup* a la hora de realizar el análisis de cobertura de código.

webapi.dll	0	0,00 %	399	100,00 %
WebApi.Controllers	0	0,00 %	399	100,00 %
CategoryController	0	0,00 %	41	100,00 %
LodgingController	0	0,00 %	75	100,00 %
RegionController	0	0,00 %	26	100,00 %
ReserveController	0	0,00 %	51	100,00 %
SearchOfLodgingCon...	0	0,00 %	28	100,00 %
TouristSpotController	0	0,00 %	84	100,00 %
UserController	0	0,00 %	94	100,00 %

WebApiTest

Analizando la cobertura del paquete *WebApiTest* podemos ver que la misma es igual al **100 %**, esto nos indica que todas las líneas de los *tests* de este paquete están siendo ejecutadas, y que ninguna es innecesaria.

webapitest.dll	0	0,00 %	2743	100,00 %
{ } WebApiTest	0	0,00 %	2743	100,00 %
▶ AuthorizationFilterTest	0	0,00 %	80	100,00 %
▶ CategoryControllerTest	0	0,00 %	214	100,00 %
▶ LodgingControllerTest	0	0,00 %	537	100,00 %
▶ RegionControllerTest	0	0,00 %	126	100,00 %
▶ ReserveControllerTest	0	0,00 %	382	100,00 %
▶ SearchOfLodgingCon...	0	0,00 %	220	100,00 %
▶ TouristSpotController...	0	0,00 %	604	100,00 %
▶ UserControllerTest	0	0,00 %	580	100,00 %

Clean Code

Es muy importante que el código del proyecto cumpla con las condiciones mencionadas en el libro de *Clean Code*. Esto es debido a que el código será mantenido en el futuro en la gran mayoría de los proyectos y para poder mantener el código, este debe ser entendible y claro, y si se siguen las pautas de *Clean Code* esto se logra de forma fácil. Según el libro de *Clean Code*, hasta el mal código puede funcionar, pero que si no está limpio puede dar mucho trabajo a la hora de editarlo, mantenerlo y reorganizarlo. Todos los años se pierden muchas horas por culpa del código sucio. Robert C. Martin, alias Uncle Bob, escribió este libro para que mantener código no sea un trabajo difícil sino que sea tan fácil como agradable de hacer. En el libro, los primeros 10 capítulos nos cuentan sobre las principales pautas a seguir. En esta sección mostraremos ejemplos de nuestro código como forma de evidencia que el desarrollo fue realizado siguiendo clean code.

Nombres nemotécnicos

Clean Code nos dice que la elección de los nombres es claramente importante para el trabajo y el entendimiento del código. Deben revelar la intención de las cosas.

La nomenclatura se definió para que al leer el nombre de métodos, variables, entre otros, se pueda comprender fácilmente cuál es su función a cumplir y que no haya lugar a confusiones o ambigüedades, evitamos dar nombres que den ideas erradas y contribuyan a la desinformación, y los nombres utilizados son sencillos y pronunciables.

```
public List<Category> GetAssociatedCategories(List<Guid> categoriesId)
```

```
public double PricePerNight { get; set; }
```

Nombre de clases y métodos

Para los nombres de las clases, no utilizamos verbos, ni palabras no significativas o genéricas.

```
C# Region.cs
C# Reserve.cs
C# TouristSpot.cs
C# User.cs
C# UserSession.cs
```

En los nombres de los métodos siempre incluimos un verbo, e intentamos ser consistentes en el sentido de que por ejemplo, todos los métodos que se encargan de obtener elementos poseen la palabra *Get*.

```
public Lodging GetLodgingById(Guid lodgingId)
private void VerifyIfLodgingExist(Lodging lodging, Guid touristSpotId)
```

Funciones

Las funciones que desarrollamos, están enfocadas a factores que hacen que la función sea fácil de leer y de entender, y para que a futuro se entienda bien cómo están hechas, y cómo modificarlas.

En general las funciones desarrolladas son pequeñas, hacen una sola cosa (no están divididas en secciones), utilizan un solo nivel de abstracción, es decir, por ejemplo en una función perteneciente a una clase de la capa de las reglas de negocios no introducimos operaciones relativas al acceso a la base de datos. Esto lo logramos como consecuencia de la clara independencia de capas la cual se encuentra justificada con más detalle en la sección de *Descripción de Diseño*.

Otro aspecto a destacar, es que dentro de las funciones, los bloques de *if*, *else* y *while*, poseen una línea de código dentro de los mismos, que generalmente es una llamada a otra función.

Siguiendo los lineamientos de *Clean Code*, podemos afirmar que las funciones poseen el nivel de indentación correcto, y en la gran mayoría de los casos el mismo no es mayor a uno o dos.

También, podemos destacar que debido a la forma en que están implementadas las funciones, es posible leer el código from *top* to *bottom*, lo que significa que debajo de una función, se encuentran las funciones que son utilizadas por ella.

Un ejemplo de función mostrando algunos de los aspectos mencionados previamente:

```
public Lodging Create(Lodging lodging, Guid touristSpotId, List<Picture> pictures)
{
    try
    {
        VerifyIfLodgingExist(lodging, touristSpotId);
```

Se puede observar como la función *VerifyIfLodging* es utilizada en la función *Create*, y se encuentra ubicada debajo de la misma, permitiendo una lectura del código from *top* to *bottom*.

```

private void VerifyIfLodgingExist(Lodging lodging, Guid touristSpotId)
{
    try
    {
        Lodging lodgingObteined = lodgingRepository.GetLodgingByNameAndTouristSpot(lodging.Name, touristSpotId);
        if (lodgingObteined != null)
        {
            throw new DomainBusinessLogicException(MessageExceptionBusinessLogic.ErrorLodgingAlredyExist);
        }
    }
    catch (ServerException e)
    {
        throw new ServerException("No se puede crear el hospedaje debido a que ha ocurrido un error.", e);
    }
}

```

Parámetros de las funciones

Clean Code nos indica que una función debe tener pocos parámetros, y que idealmente no debería tener ningún parámetro.

Las funciones deben tener entre cero y dos argumentos, tres en casos excepcionales, y más de tres es inaceptable.

En nuestro caso, la mayor cantidad de parámetros que una función tiene es igual a 3 (porque analizando la situación, vimos que no había otra solución posible en esos casos), pero por lo general el número de parámetros de las funciones desarrolladas oscila entre cero, uno y dos.

Las que reciben un parámetro, por lo general son funciones que toman un elemento que viene en la request, y terminan invocando por lo general a alguna función de la capa de acceso a datos.

```

public Lodging GetLodgingById(Guid lodgingId)

public List<Lodging> GetAvailableLodgingsByTouristSpot(Guid touristSpotId)

```

Las que reciben dos o tres parámetros, por lo general los necesitan para formar un punto, es decir, para formar un único objeto a partir de los parámetros recibidos, esta idea de formar un punto *Clean Code* la acepta (actúan como un único parámetro).

Uno de estos casos es que el se muestra en la siguiente imagen:

```

public TouristSpot Create(TouristSpot touristSpot, Guid regionId, List<Guid> categoriesId)

```

Este es un claro ejemplo de lo mencionado anteriormente, donde se pasa un objeto el cual está pre-armado, y luego con los restantes parámetros se termina de crear completamente, esto no significa que sean los parámetros aquellos elementos que se utilicen directamente, sino que se pueden utilizar para operaciones de obtener otros elementos a partir de ellos, y así completar el objeto pre-armado. Esto para que quede consistente con los datos que tiene que tener, ya que luego de esto se verifica si el formato del objeto es válido o no.

Otro aspecto a destacar es que evitamos el uso de banderas booleanas, que a menudo causan ruido y bajan el entendimiento de los métodos.

Comentarios

Siguiendo los principios de *Clean Code*, que nos indica que el código idealmente no debería estar comentado, nuestro código no posee ningún comentario.

No posee comentarios porque consideramos que el código es autoexplicativo. Si tuviéramos que escribir un comentario, significa que el código no es suficientemente autoexplicativo, y en ese sentido deberíamos poder mejorarlo en lugar de tapar la falencia con comentarios.

Los comentarios que son tolerables para *Clean Code* son aquellos que refieren a aspectos legales, aclaraciones que el código no puede ser modificado por terceros, entre otros. Estos tipos de comentarios no consideramos apropiados incorporarlos a nuestra solución porque no aportan valor agregado alguno en nuestro caso.

Formateo

Al comenzar el desarrollo, nos inclinamos por escoger un grupo de reglas simples que se encuentren en todo el formato del código, para mantener una determinada consistencia.

Horizontal

En este sentido, se refiere a cómo está organizado el código, es decir, el largo de los archivos, el ancho de los archivos, la buena indentación entre otras cosas. En la próxima imagen podemos ver el formato de un archivo, el cual no es excesivamente grande horizontalmente:

```
public class Category
{
    0 references
    public Guid Id { get; set; }

    3 references
    public string Name { get; set; }

    1 reference
    public virtual List<CategoryTouristSpot> ListOfTouristSpot { get; set; }

    0 references
    public Category() {
        ListOfTouristSpot = new List<CategoryTouristSpot>();
    }

    0 references
    public void VerifyFormat()
    {
        if (string.IsNullOrEmpty(Name))
        {
            throw new CategoryException(MessageExceptionDomain.ErrorIsEmpty);
        }
    }

    0 references
    public override bool Equals(object obj)
    {
        return obj is Category category &&
            Name.Equals(category.Name);
    }
}
```

Vertical

El formateo vertical hace referencia a diferentes conceptos. Uno de ellos es la llamada *metáfora del periodico*, que indica que el nombre de la clase debe ser simple pero explicativo, y se incrementará en los detalles mientras se recorre el código hacia abajo. Como se mostró previamente, los nombres son simples, y para obtener más detalles de la clase, se va recorriendo el código hacia abajo y se los va obteniendo.

Otro aspecto que se destaca en este punto, es que *Clean Code* recomienda que debemos declarar las variables lo más cerca posible de su uso.

Un ejemplo de aplicación de este concepto es el siguiente:

Se puede apreciar como se define una lista de tipo *Category*, inmediatamente previo a su uso.

```
public List<Category> GetAssociatedCategories(List<Guid> categoriesId)
{
    List<Category> listOfCategoriesToAssociated = new List<Category>();

    foreach (Guid identifierCategory in categoriesId)
    {
        Category category = GetById(identifierCategory);
        listOfCategoriesToAssociated.Add(category);
    }
    return listOfCategoriesToAssociated;
}
```

Otra lineamiento que se describe en este sentido, es que las instancias de las variables, deben estar al principio de la clase. Esto no debe ser impedimento a la distancia de su uso, ya que en una clase bien ideada, se usan las variables en la mayoría (si no en todos), los métodos de la misma.

En la siguiente imagen tenemos una de las clases de nuestra aplicación, en donde se puede observar claramente que las instancias de las variables están ubicadas al principio de la misma.

```
public class Lodging
{
    2 references
    public Guid Id { get; set; }

    6 references
    public string Name { get; set; }

    8 references
    public int QuantityOfStars { get; set; }

    6 references
    public string Address { get; set; }

    4 references
    public string Description { get; set; }

    3 references
    public virtual List<LodgingPicture> Images { get; set; }

    5 references
    public double PricePerNight { get; set; }

    2 references
    public bool IsAvailable { get; set; } = true;

    1 reference
    public virtual TouristSpot TouristSpot { get; set; }
}
```

También, como se mostró previamente, se respetó el ordenamiento vertical, en el sentido que las funciones auxiliares se encuentran debajo de los métodos en donde son llamadas.

Ley de Demeter

Esta ley, nos indica que no se debe conocer la lógica interna de los objetos que se manipulan.

En otras palabras, significa hablar con los conocidos y no con los extraños.

Se deben evitar las llamadas en cadena, y es mejor sustituirlo por llamadas verticales en los que haya de por medio definición de variables auxiliares.

```
public void VerifyFormat()
{
    if (IsValidNameOrAddressOrDescription())...
    if (IsValidQuantityOfStars())...
    if (IsValidPricePerNight())...
    if (IsValidListOfPictures())...
    TouristSpot.VerifyFormat();
}
```

En esta imagen, tenemos una función llamada *VerifyFormat()* que se encuentra dentro de la clase *Lodging* y cuya responsabilidad es la de verificar si el hospedaje que está siendo creado es válido, es decir, si todos los campos requeridos tienen el formato válido. Lo que sucede es que un hospedaje tiene un punto turístico asociado, y debemos validar que el formato del mismo sea correcto, ahí podemos ver que estamos verificando la validez de un punto turístico simplemente con una llamada a un método de esta clase, que no conocemos la estructura interna del mismo. Una violación a este principio hubiera sido desde esta clase, ir accediendo a cada uno de los atributos del punto turístico y verificar que los mismos sean válidos.

Manejo de errores

Las recomendaciones que se establecen para el manejo de errores es utilizar excepciones y no retornar códigos de error. Esto es recomendable, ya que al usuario se le avisa al momento, que se está haciendo algo mal, y no se espera a otro comportamiento.

Es importante definir excepciones en términos de las necesidades de los clientes, y en ese sentido, decidimos 3 paquetes que manejan excepciones: uno para las excepciones del dominio, otro para las de las reglas de negocio, y otro para la capa de acceso a datos. De esta forma, podemos crear clases dentro de estos paquetes que sean específicas y acordes a las necesidades de cada cliente. Una explicación más en detalle del manejo de excepciones se puede observar en la documentación de la *Descripción del Diseño*.

Un ejemplo de una clase de excepción implementada en nuestra solución es la siguiente:

```
namespace DomainException
{
    3 references
    public class TouristSpotException : Exception
    {
        0 references
        public TouristSpotException() : base() { }

        0 references
        public TouristSpotException(String message) : base(message) { }

        0 references
        public TouristSpotException(String message, Exception exception) : base(message, exception) { }
    }
}
```

La misma nos permite manejar todas las excepciones relativas a los puntos turísticos, y lanzar excepciones específicas para estos clientes.

Pruebas unitarias

Robert Martin habla de las tres leyes de las pruebas unitarias:

1. No escribirás código de producción hasta haber escrito una prueba fallida.
2. No escribirás más sobre una prueba unitaria que lo suficiente para que falle.
3. No escribirás más código de producción que el necesario para que pase la prueba.

No podemos ofrecer una evidencia de completa del cumplimiento de estas leyes, pero por medio de los *commits* en el repositorio en *GitHub*, se puede visualizar que la aplicación fue desarrollado siguiendo *TDD (Test Driven Development)*, debido a que no hay commits que contengan lógica que no esté acompañada por sus respectivos *tests*.

Clean Code también nos indica que las pruebas al tener que evolucionar al mismo ritmo que el código, deben ser igualmente mantenibles y respetar las mismas reglas de código limpio.

Está permitido hacer más de un *assert* en una prueba, pero sí se debe cumplir que el número de *asserts* sea mínimo. Lo que sí se debe cumplir siempre es que solo se prueba una cosa en cada *test*.

Nuestros métodos de *tests* en su totalidad incluyen un único *assert*, afirmando el concepto de que los mismos prueban una única cosa.

En la siguiente imagen podemos observar un método de *test* que cumple con lo mencionado previamente.

También, es importante notar que siguiendo *Clean Code*, intentamos evitar métodos de *test* muy largos con todos los detalles de implementación, intentando mostrar claramente la estructura *Arrange-Act-Assert* de las pruebas escondiendo los detalles en métodos.

```
[TestMethod]
0 references
public void LoginOkTest()
{
    var userMock = new Mock<IUserManagement>(MockBehavior.Strict);
    userMock.Setup(m => m.Login(aLoginModel.Email, aLoginModel.Password)).Returns(aUserSession);
    UserController userController = new UserController(userMock.Object);
    var result = userController.Login(aLoginModel);
    var createdResult = result as OkObjectResult;
    var userModelResult = createdResult.Value as UserModelForResponse;
    userMock.VerifyAll();
    Assert.AreEqual(aUserModel, userModelResult);
}
```

Clases

Las clases deberían ser pequeñas principalmente, y no en líneas de código sino que en responsabilidades. Lo ideal es que las clases tengan una responsabilidad única. Esto es el *Single Responsibility Principle (SRP)*. Nosotros intentamos hacer las clases lo más chicas posibles, agregando clases si es necesario para dividir las responsabilidades. Esto se puede observar en el paquete *BusinessLogic*, en el cual podemos observar que el problema está dividido en varias clases. Esto ayuda en muchos sentidos, por ejemplo aumenta la cohesión y baja el acoplamiento, y divide las responsabilidades entre varias clases.

