

Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de aplicaciones 2

Obligatorio 1:

Evidencia de Clean Code y TDD

Juan Pablo Poittevin(169766)

Joselen Cecilia(233552)

Entregado como requisito de la materia Diseño de
aplicaciones 2

6 de mayo de 2021

Link al repositorio de GitHub

[https://github.com/ORT-DA2/Poittevin-169766-
-Cecilia-233552-](https://github.com/ORT-DA2/Poittevin-169766-Cecilia-233552-)

Declaraciones de autoría

Nosotros, Juan Pablo Poittevin y Joselen Cecilia, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos Diseño de Aplicaciones 2 ;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Índice general

1. Clean Code y TDD	3
1.1. Descripción de la estrategia de TDD seguida	3
1.2. Informe de cobertura para todas las pruebas desarrolladas	4
1.2.1. Cobertura general	4
1.2.2. Cobertura paquete Domain	4
1.2.3. Cobertura paquete WebAPI	5
1.2.4. Cobertura paquete BuisnessLogic	5
1.2.5. Cobertura paquete DataAccess	6
1.3. Evidencia de Clean Code	7
1.4. Nombres con sentido	7
1.4.1. Nombre de las variables	7
1.4.2. Nombres de los métodos	7
1.4.3. Nombres de las Clases	8
1.5. Formato	8
1.5.1. Formato variables y properties	9
1.5.2. Formato métodos	9
1.5.3. Formato clases	11
1.6. Indentación	11
1.7. Manejo de errores	11
1.8. Importaciones	13
1.9. Ley de demeter	13
1.10. Código duplicados	14

1. Clean Code y TDD

El código de la aplicación fue desarrollado en su totalidad usando TDD y siguiendo las buenas practicas establecidas por Clean Code.

1.1. Descripción de la estrategia de TDD seguida

Como se menciona anteriormente el obligatorio fue desarrollado utilizando TDD (Test-Driven Development), para esto se siguió el proceso de escribir primero los test y luego la implementación para que estos pasen, siguiendo los tres pasos o reglas:

Nunca escribir código si no hay una prueba unitaria.

Escribir el código mínimo y suficiente para que la prueba sea fallida.

Escribir sólo el código mínimo y suficiente para que la prueba pase.

Luego se realizo el refactor donde no solo se mejoro el código que se había escrito en las etapas anteriores, si no que también se arreglaron errores que se fueron encontrando en los test.

También en esta etapa se verifico que los test cumplieran con el principio FIRST:

F- FAST: Las pruebas se ejecutan rápido.

I- INDEPENDENT: La prueba no dependen del resultado de otra prueba.

R- REPEATABLE: Las pruebas son repetibles en cualquier ambiente.

S- SELF-VALIDATING: Las pruebas tienen una salida booleana. Ya sea que pasan o no.

T- TIMELY: Las pruebas son escritas antes que el código de producción que hace que pase dicha prueba.

Mas al final, cuando ya estábamos terminando el código del obligatorio, creamos ramas refactor para; agregar validaciones que habían faltado, mejorar las excepciones propias, borrar métodos que no se usaban, cambiar nombre (de métodos, variables y clases), separar métodos en partes mas pequeñas, mejorar el código de cada uno de estos para que sea lo mas entendible posible y eliminar código duplicado.

En esta etapa también se reviso la cobertura del código y en el caso donde se encontraron partes de código donde no se había aplicado cien por cien TDD, se agregaron los test que faltaban para mantener una alta cobertura.

La estrategia de TDD que se utilizo

1.2. Informe de cobertura para todas las pruebas desarrolladas

En este punto mostraremos evidencia de que utilizamos TDD para el desarrollo de la aplicación.

La evidencia de TDD se muestra mediante la creación de las clases de los tests y seguido la creación de las clases que estos prueban. Esto se puede ver reflejado en los commits del repositorio.

De estas clases luego se realiza el refactor y en caso de no haber cumplido cien por cien con TDD se agregan luego los test que faltan para cubrir todo el código con pruebas.

Este proceso nos permitió llegar a una cobertura total de 98 %.

1.2.1. Cobertura general

Para analizar la cobertura del código utilizamos dotCover en Rider. Se muestra en la siguiente imagen el porcentaje de cobertura general de cada paquete, como vemos todos los paquetes, a excepción del BuisnessLogic que tiene un porcentaje de 96 %, están en un porcentaje de cobertura de 99 %, porcentaje que creemos muy satisfactorio ya que muestra que el código esta probado casi en su totalidad.

Symbol	Coverage (%)	Uncovered/Total Stmt.
Total	98%	33/1910
> MSP.BetterCalm.DataAccess	99%	10/842
> MSP.BetterCalm.Domain	99%	4/304
> MSP.BetterCalm.WebAPI	99%	1/332
> MSP.BetterCalm.BusinessLogic	96%	18/432

1.2.2. Cobertura paquete Domain

Evidencia de TDD en el paquete Domain, como se ve en la imagen de abajo se logro casi un 100 % de cobertura en el paquete domain a excepción de la clase psychologist donde el método GetDayForNextMeetingOnWeek fue el que bajo el porcentaje de esta ya que quedo con 80 % de cobertura.

Symbol	Coverage (%)	Uncovered/Total Stmt.
> MSP.BetterCalm.DataAccess	99%	10/842
▼ MSP.BetterCalm.Domain	99%	4/304
▼ MSP.BetterCalm	99%	4/304
▼ BusinessLogic.Exceptions	100%	0/15
> InvalidAmountOfProblematicsError	100%	0/3
> InvalidDescriptionLength	100%	0/3
> InvalidDurationFormat	100%	0/3
> InvalidNameLength	100%	0/3
> InvalidUrl	100%	0/3
▼ Domain	99%	4/289
> Administrator	100%	0/23
> Audio	100%	0/60
> Category	100%	0/14
> Meeting	100%	0/23
> Patient	100%	0/29
> Playlist	100%	0/64
> Problematic	100%	0/14
> User	100%	0/4
> Psychologist	93%	4/58

1.2.3. Cobertura paquete WebAPI

Evidencia de TDD en el paquete WebAPI, como vemos en la imagen presentada los namespaces Filters y Controllers están con cobertura del 100 %, es decir todas sus líneas de código están probadas y en correcto funcionamiento. El namespaceDto esta con cobertura de 99 %, esto se debe a que el objeto AudioDto esta con 98 % de cobertura.

Symbol	Coverage (%)	Uncovered/Total Stmt.
> MSP.BetterCalm.Domain	99%	4/304
▼ MSP.BetterCalm.WebAPI	99%	1/332
▼ MSP.BetterCalm.WebAPI	99%	1/332
▼ Filters	100%	0/48
> FilterAuthentication	100%	0/15
> FilterExceptions	100%	0/33
▼ Controllers	100%	0/189
> AdministratorController	100%	0/24
> AudioController	100%	0/41
> CategoryController	100%	0/16
> LoginController	100%	0/7
> PatientController	100%	0/16
> PlaylistController	100%	0/45
> ProblematicController	100%	0/16
> PsychologistController	100%	0/24
▼ Dtos	99%	1/95
> ErrorDto	100%	0/15
> PlaylistDto	100%	0/16
> ScheduleMeetingDto	100%	0/4
> AudioDto	98%	1/60

1.2.4. Cobertura paquete BuisnessLogic

Evidencia de TDD en el paquete BuisnessLogic, como se ve en la imagen de abajo se logro un 96 % de cobertura en el paquete BuisnessLogic. La clases que quedaron con cobertura menor a 90 % fueron PatientService y AdministratorService

▼ MSP.BetterCalm.BusinessLogic	96%	18/432
▼ MSP.BetterCalm.BusinessLogic	96%	18/432
> AudioService	100%	0/99
> GuidService	100%	0/3
> PlaylistService	100%	0/142
> Exceptions	100%	0/33
> PsychologistService	96%	1/26
> CategoryService	95%	1/22
> ProblematicService	95%	1/22
> PatientService	85%	6/39
> AdministratorService	80%	9/46
> ManagerAdministratorRepository		0/0
> ManagerAudioRepository		0/0
> ManagerCategoryRepository		0/0
> ManagerMeetingRepository		0/0
> ManagerPatientRepository		0/0
> ManagerPlaylistRepository		0/0
> ManagerProblematicRepository		0/0
> ManagerPsychologistRepository		0/0

1.2.5. Cobertura paquete DataAccess

Evidencia de TDD en el paquete DataAccess, como se ve en la imagen de abajo se logro un 99 % de cobertura. Donde casi todas las clase están con 100 %, a excepción de PlaylistMapper, MeetingMapper y PsychologistMapper.

Symbol	Coverage (%) ▼	Uncovered/Total Stmts.
▼ Total	97%	143/4758
▼ MSP.BetterCalm.DataAccess	99%	10/842
▼ MSP.BetterCalm.DataAccess	99%	10/842
> ContextDB	100%	0/41
> AdministratorDto	100%	0/8
> AudioCategoryDto	100%	0/8
> AudioDto	100%	0/16
> CategoryDto	100%	0/8
> MeetingDto	100%	0/12
> PatientDto	100%	0/8
> PlaylistAudioDto	100%	0/8
> PlaylistCategoryDto	100%	0/8
> PlaylistDto	100%	0/12
> ProblematicDto	100%	0/6
> PsychologistDto	100%	0/12
> PsychologistProblematicDto	100%	0/8
> UserDto	100%	0/6
> AdministratorMapper	100%	0/18
> AudioMapper	100%	0/121
> CategoryMapper	100%	0/17
> PatientMapper	100%	0/38
> ProblematicMapper	100%	0/17

> ProblematicMapper	100%	0/17
> AdministratorRepository	100%	0/4
> AudioRepository	100%	0/4
> CategoryRepository	100%	0/4
> DataBaseRepository<D,T>	100%	0/76
> MeetingRepository	100%	0/4
> PatientRepository	100%	0/4
> PlaylistRepository	100%	0/4
> ProblematicRepository	100%	0/4
> PsychologistRepository	100%	0/4
> PlaylistMapper	99%	2/232
> MeetingMapper	96%	2/47
> PsychologistMapper	93%	6/83

1.3. Evidencia de Clean Code

Para facilitar la escritura, la lectura y la mantenibilidad del código aplicamos en el desarrollo de todo el sistema las buenas practicas especificadas por Clean code. En este punto mostraremos evidencia de la utilización de este.

1.4. Nombres con sentido

Todos los nombres tanto de variables, métodos, clases y paquetes son descriptivos de lo que hacen o para que sirven. Además son nombres pronunciables y “buscables”.

1.4.1. Nombre de las variables

Los nombres de todas las variables van en minúsculas y se declaran lo mas cerca posible de uso

```
Playlist playlistToUpdate = repository.Playlists.FindById(id);
Playlist playlist = CreateNewPlaylist(id, newPlaylist);
repository.Playlists.Update(playlistToUpdate, playlistToUpdate);
```

Los nombres de las properties van en mayúsculas y son publicas

```
25 usages
public int Id { get; set; }

66 usages
public string Name { get; set; }
```

1.4.2. Nombres de los métodos

Los métodos llevan nombres que utilizan verbos y empiezan con mayúsculas. Los que retornan un booleano llevan la palabra Is o exist delante, lo que son de acceso a datos llevan la palabra Get delante y los que son para el registro de datos la palabra Set o Add.

```
112 usages
public List<Category> GetCategories()
{
    return repository.Categories.Get();
}
```


11+4 usages

```
public void SetPlaylist(Playlist playlist)
{
    repository.Playlists.Add(playlist);
}
```

```
public bool IsSameAudioName(string audioName)
{
    return Name.ToLower() == audioName.ToLower();
}
```

1.4.3. Nombres de las Clases

Las clases llevan nombres que no son verbos y empiezan con mayúsculas, siguiendo el estilo Camell case, al igual que los métodos estas llevan nombres pronunciables, descriptivos y buscables.

```
C# AdministratorService.cs
C# AudioService.cs
C# CategoryService.cs
C# GuidService.cs
C# IAdministratorService.cs
C# IAudioService.cs
C# ICategoryService.cs
C# IGuidService.cs
C# IPatientService.cs
C# IPlaylistService.cs
C# IProblematicService.cs
C# IPsychoanalystService.cs
C# PatientService.cs
C# PlaylistService.cs
C# ProblematicService.cs
C# PsychoanalystService.cs
```

1.5. Formato

El formato del código hace a la comunicación, los archivos con menos líneas de código tienden a ser más fáciles de entender. Por esto tanto los métodos, como las clases intentan ser reducidas.

Para facilitar la lectura del código se ponen juntas las cosas que están relacionadas entre sí.

1.5.1. Formato variables y properties

Los atributos y properties se definen al comienzo de la clase todos juntos

```
public int PsychologistId { get; set; }  
16 usages  
public string Address { get; set; }  
10 usages  
public bool WorksOnline { get; set; }  
  
private List<Problematic> problematics;  
28 usages  
public List<Problematic> Problematics{ get=>problematics; set => SetProblematics(value); }  
14 usages  
public List<Meeting> Meetings{ get; set; }  
11 usages  
public DateTime CreationDate { get; set;}
```

Usar espacios, por ejemplo en =, == y != se dejan espacio a la izquierda y a la derecha.

```
if (obj == null) return false;  
if (obj.GetType() != GetType()) return false;  
return Name == ((Category)obj).Name && Id == ((Category)obj).Id;
```

1.5.2. Formato métodos

Los métodos son reducidos no tiene mas de 30 líneas y se encargan de una sola cosa, no están divididas en secciones. Además estos métodos no reciben mas de 2 o a lo sumo 3 parámetros.

```
273 usages  
public List<Playlist> GetPlaylistByName(string playlistName)  
{  
    List<Playlist> playlists = new List<Playlist>();  
    foreach (var playlist in repository.Playlists.Get())  
    {  
        if(playlist.IsSamePlaylistName(playlistName))  
            playlists.Add(playlist);  
    }  
  
    if (playlists.Count == 0)  
        throw new NotFoundPlaylist();  
    return playlists;  
}
```

Las clases mapper son excepciones a este punto ya que en los métodos de estas


clases es donde se concentra toda la dificultad, es por esto que estos métodos son mas extensos, algunos superan las 30 lineas, y reciben hasta 4 parámetros.

```
private static List<PlaylistAudioDto> UpdatePlaylistAudios(PlaylistDto objToUpdate, Playlist updatedObject, ContextDB context,
    AutoMapper audioMapper)
{
    List<PlaylistAudioDto> diffListOldValuesAudio = objToUpdate.PlaylistAudiosDto.Where(x => x.AudioDto == audioMapper.DtoToDomain(x.AudioDto, context)).ToList();

    List<Audio> diffListNewValuesAudio = new List<Audio>();
    if (updatedObject.Audios != null)
    {
        foreach (var audio in updatedObject.Audios)
        {
            Audio audioToAdd = audioMapper.DtoToDomain(objToUpdate.Audios.First(x => x.AudioDtoID == audio.Id), context);
            PlaylistAudioDto playlistAudioDto = new PlaylistAudioDto()
            {
                AudioID = audio.Id, PlaylistID = objToUpdate.PlaylistDtoID,
                AudioDto = audioMapper.DomainToDto(audio, context), PlaylistDto = objToUpdate
            };
            bool contain = false;
            if (objToUpdate.PlaylistAudiosDto != null)
            {
                foreach (var playlistAudio in objToUpdate.PlaylistAudiosDto)
                {
                    if (playlistAudio.AudioID == playlistAudioDto.AudioID &&
                        playlistAudio.PlaylistID == playlistAudioDto.PlaylistID)
                    {
                        contain = true;
                    }
                }
            }
            if (!contain)
            {
                diffListNewValuesAudio.Add(audioToAdd);
            }
        }

        diffListOldValuesAudio.AddRange(collection: diffListNewValuesAudio.Select(x => new PlaylistAudioDto() {
            AudioDto = audioMapper.DomainToDto(x, context), AudioID = x.Id,
            PlaylistID = objToUpdate.PlaylistDtoID, PlaylistDto = objToUpdate}));
    }
}
```

Step-Down Rule, leer el código “from top to bottom”. Cada función debe ser seguida por las de siguiente nivel de abstracción. Los métodos que se llaman a otros se encuentran declarados debajo o encima de estos, para así facilitar la lectura.



```

private bool AlreadyExistThisAudio(Audio audio)
{
    try
    {
        repository.Audios.FindById(audio.Id);
        return true;
    }
    catch (KeyNotFoundException)
    {
        return false;
    }
}

8+7 usages
public Audio SetAudio(Audio audio)
{
    if (!AlreadyExistThisAudio(audio))
    {
        return repository.Audios.Add(audio);
    }

    throw new AlreadyExistThisAudio();
}

```

1.5.3. Formato clases

Las clases no tienen más de 150 o a lo sumo 200 líneas de código, a excepción de la clase `playlistMapper` que tiene casi 300 líneas.

En formato vertical, se deja un espacio entre método y método para facilitar la lectura.

1.6. Indentación

Procedimos a realizar la Indentación sugerida por los lineamientos de clean code, aunque se asegura que cada línea de código cumpla con este punto.

1.7. Manejo de errores

No se devuelve `null`, en su lugar se devuelve una excepción. Para los casos en que las funciones deberían retornar `null` o que ocurre un error se lanzan excepciones, que luego se controlan con bloques `try catch` y en la API con el `filter`.

```

public D Find(Predicate<D> condition)
{
    List<T> dtos = entity.ToList();
    foreach (var dto in dtos)
    {
        var dDto = mapper.DtoToDomain(dto, context);
        var condResult = condition(dDto);
        if (condResult)
            return dDto;
    }
    throw new KeyNotFoundException();
}

```

Una excepción a esto, es la validación que se hace sobre los objetos devueltos por `FirstOrDefault` porque son métodos ajenos a nosotros y no verificar esto podría causar errores. En caso de que sea null el objeto se devuelve una excepción de las nuestras

```

CategoryDto categoryDto = categoriesSet.FirstOrDefault(
    x :CategoryDto => x.CategoryDtoID == category.Id || x.Name==category.Name
);
if (categoryDto == null)
{
    throw new InvalidCategory();
}

```

Para el manejo de errores utilizamos la clase `ExceptionFilters` de manera de centralizar el manejo de errores de la aplicación y encapsularlos en una única clase que tiene con esta única responsabilidad.

```

public void OnException(ExceptionContext context)
{
    List<Type> errors401 = new List<Type>()
    {
        typeof(NotFoundAdminLoginError),
        typeof(AuthenticationException)
    };
    List<Type> errors404 = new List<Type>()
    {
        typeof(NotFoundId),
        typeof(NotFoundAudio),
        typeof(NotFoundPlaylist),
        typeof(NotFoundCategory),
        typeof(NotFoundAdministrator),
        typeof(NotFoundPsychologist),
        typeof(NotFoundProblematic),
        typeof(KeyNotFoundException)
    };
    List<Type> errors409 = new List<Type>()
    {
        typeof(AlreadyExistThisAudio),
        typeof(InvalidCategory),
        typeof(InvalidProblematic)
    };
    List<Type> errors422 = new List<Type>()
    {
        typeof(InvalidNameLength),
        typeof(InvalidDescriptionLength),
        typeof(InvalidDurationFormat),
        typeof(InvalidUrl),
    };
};

```

1.8. Importaciones

No se importan namespaces innecesarios, esto se revisa en el refactor del código borrando las que no se usan.

1.9. Ley de demeter

Se cumple con la ley de demeter que dice; que dado un objeto, este solo debería de conocer (y acceder) a los métodos públicos de sus colaboradores directos. Para esto se

crean las clases services para evitar las llamadas del tipo repository.Categories.Add, en contrario se hace la llamada categoryService.SetCategory y en el service se hace el add.

```
[HttpPut(template: "{id}")]
2 usages
public IActionResult UpdateAudio([FromRoute] int id, [FromBody] AudioDto audioUpdated)
{
    _audioService.UpdateAudioById(id, audioUpdated.CreateAudio());
    return Ok("Audio Updated");
}
```

Para esto tenemos en los controllers de la api las instancias a los services. Cumpliendo así con o que indica clean code que solo se debe invocar funciones de; si mímimo, variables locales,un argumento o una variable de instancia

1.10. Código duplicados

Se utiliza el patrón de repositorio genérico, para poder con una sola clase manejar los repositorios de las distintas entidades.

```
namespace MSP.BetterCalm.BusinessLogic
{
    29 usages 1 inheritor 8+3 exposing APIs
    public interface IRepository<T>
    {
        21 usages 1 implementation
        T Add(T objectToAdd);
        11 usages 1 implementation
        void Delete(T objectToDelete);
        29 usages 1 implementation
        T Find(Predicate<T> condition);
        43 usages 1 implementation
        List<T> Get();
        18 usages 1 implementation
        T Update(T OldObject, T UpdatedObject);
        52 usages 1 implementation
        T FindById(int id);
    }
}
```