Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Diseño de Aplicaciones 2

Obligatorio 1

3-TDD y Clean Code

Santiago Castro - 168347

https://github.com/ORT-DA2/168347

Grupo M6B

Tabla de contenido

1.	TDD	1
2.	Clean Code	8

1. TDD

En una primera instancia se implementaron las funcionalidades que no requerían hacerlas con TDD como pueden ser el manejo de usuarios y sesiones, para esto, se crearon los controladores de la WebApi UsersController y SessionsController, luego se crearon las interfaces de la lógica de negocio IUserLogic y IUserSessions con sus respectivas clases que las implementan. Adicionalmente se creó el paquete Repository el cual es el responsable de la persistencia de estos usuarios y sesiones.

Luego se identificaron otras funcionalidades que no requerían hacerlas con TDD como la de consultar el estado de una solicitud, para implementarla se tuvo que introducir ciertas clases del dominio como Area, Topic, TopicType, AdditionalField, Applicant y Requerst con sus respectivas lógicas y manejadores del repositorio. Aquí se introdujo la Inyección de dependencias.

En este punto se tuvo una API capaz de manejar usuarios y sesiones, además de consultar solicitudes agregadas a mano en la base de datos. Luego se decidió comenzar a implementar la primera funcionalidad con TDD que es la de ingreso de solicitudes por parte de usuarios.

Funcionalidad de creación de solicitudes:

Se creó el paquete de testing Tarea. Business Logic. Tests con la clase Request Tests la cual contiene el primer test de creación de solicitudes:

Etapa Red: Este test debe probar el método Create de la clase RequestLogic el cual dada una solicitud pasada por parámetro la debe insertar en la base de datos, para eso se debe utilizar los métodos Add y Save de una clase que implemente IRepository. Algunas clases del repositorio ya están implementadas ya que son necesarias para la funcionalidad de mostrar el estado de una solicitud. Esto facilitó al momento de escribir el test.

https://github.com/ORT-DA2/168347/commit/b2d26abf1f6ca06ca5b1e8f60a09c4ed4d91da7a

Etapa Green: En esta etapa se implementó el método Create de la clase RequestLogic el cual ya estaba creado, pero retornaba null ya que la clase implementa la interfaz ILogic. Se llama a los métodos Add y Save del repositorio.

https://github.com/ORT-DA2/168347/commit/1221c596c416fbd85b455831b72876a1743bd37b

Etapa Refactor: no aplicada.

Etapa Red: Se creó un nuevo test el cual prueba que si al momento de crear una solicitud con solicitante vacío debe retornar una excepción.

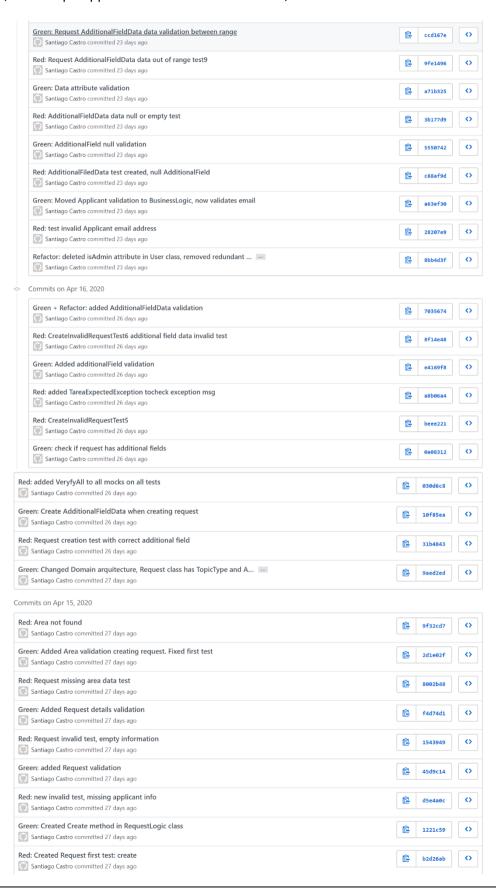
https://github.com/ORT-DA2/168347/commit/d5e4a0c9ea28171d50c1a5f9aa779c89f2c0d325

Etapa Green: Se creó un método Validate el cual verifica si los datos del solicitante están presentes en la solicitud a crear.

https://github.com/ORT-DA2/168347/commit/45d9c1410b976379dd86c2ec5c7ae2c8c68f4d90

Etapa Refactor: no aplicada.

De esta forma se fueron creando pruebas y luego modificando la lógica para que las pruebas sean satisfactorias. Cada ciclo se TDD se probó algo en concreto como por ejemplo si el campo detalles tiene largo mayor a 2000, si el campo Applicant.email es un email válido, etc



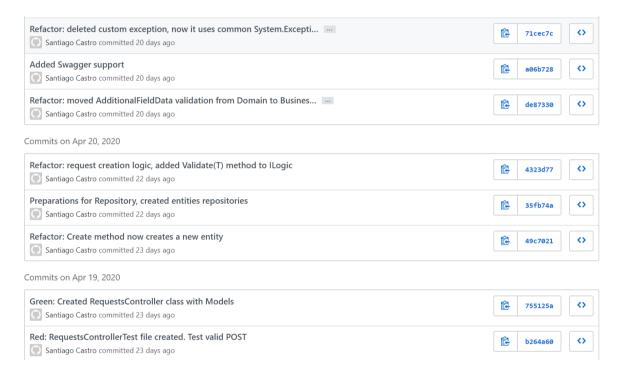
Luego se pasó a implementar la parte del Backend para esta funcionalidad, para esto se creó el paquete de prueba Tarea. WebApi. Tests con la clase Requests Controller Test y su primer test.

Etapa Red: Se creó el test CreateVaildRequestTest el cual prueba el funcionamiento del método Post de la clase RequestController pasando una instancia de RequestModel. El test es correcto si se ejecuta el método Create de RequestLogic y se retorna el status 200.

https://github.com/ORT-DA2/168347/commit/b264a60a063f4d631cd83da78416bdd4226db261

Etapa Green: Se creó el método Post de la clase RequestController el cual llama a la lógica y devuelve el status 200.

https://github.com/ORT-DA2/168347/commit/755125afb7f97fcca625eef4eb8858a0f47dde9d



Funcionalidad creación de tipos:

Etapa Red:

Se creó la clase TopicTypeTest con un test que prueba la creación de tipos. Para esto se debe llamar al repositorio para agregar el tipo a la base de datos y luego salvar el contexto. Además, se agregó la prueba en caso de que el argumento Topic es null, debe retornar una excepción.

https://github.com/ORT-DA2/168347/commit/f81f443893ac9943907b1ae2500d1942b807c1ec

Etapa Green: Se creó el método Create de la clase TopicTypeLogic el cual valida el parámetro de entrada y luego llama al repositorio para agregar el tipo a la base de datos.

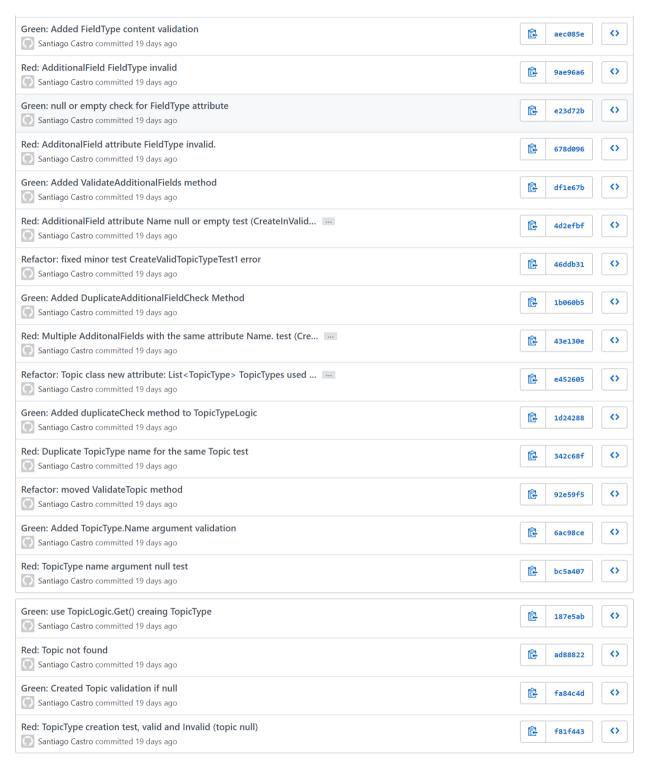
https://github.com/ORT-DA2/168347/commit/fa84c4dc86522d10c783e5daf6540538ec48064b

Etapa Red: Se prueba si el parámetro tema del tipo es null, debe retornar una excepción.

https://github.com/ORT-DA2/168347/commit/ad88822b0225ddb89e335dedfc9b2ceee340e110

Etapa Green: Implementada la validación al momento de crear tipos.

https://github.com/ORT-DA2/168347/commit/187e5ab4028ef70f45bee8ae1e3c2b2b18521cf0



En cada etapa de TDD se fueron creando pruebas de casos en que falla la creación de tipos como por ejemplo nombre duplicado para un tema, nombre nulo o vacío, nombres de campos adicionales duplicados, etc. Adicionalmente se fue haciendo refactor cuando se vio necesario mejorar la calidad de código extrayendo métodos o mejorando la lógica.

Etapa Red: Se creó la clase TopicTypesControllerTests con el primer caso de prueba de creación de tipos el cual prueba el método Post de la clase TopicTypesController. Este método debe llamar a la lógica y devolver el código status 201.

https://github.com/ORT-DA2/168347/commit/6aa33956bbfe2b0141eb8f62d31b0fb03589d71a

Etapa Green: Fue necesario crear las clases TopicTypeModel, AdditionalFieldModel, TopicModel y TopicTypeController. Luego se implementó el método Post el cual crea una instancia de TopicType y llama al método Create de TopicTypeLogic, finalmente devuelve el status 201.

https://github.com/ORT-DA2/168347/commit/7c0141867298730c72efba3469ccdbd8aad91b13

Funcionalidad Listar solicitudes existentes

Etapa Red: Se creó un nuevo test en RequestControllerTest donde llama al método Get de RequestController el cual debe devolver la lista de solicitudes del sistema, para eso se debe ejecutar el método GetAll de la clase RequestLogic.

https://github.com/ORT-DA2/168347/commit/538d706482bbdbbba6b6645dc1fbfc6943da12ed

Etapa Green: Se implementó el método Get de clase RequestsController el cual obtiene la lista de solicitudes del sistema por medio del método RequestLogic.GetAll() y devuelve la lista de solicitudes junto al código status 200.

https://github.com/ORT-DA2/168347/commit/6130d198db3a0b40b16285b233c5cbd87f7dc540

Funcionalidad Editar status de una solicitud

Etapa Red: Se creó el test UpdateRequestStatusTest1 en la clase RequestsControllerTest, el cual prueba el método Put de la clase RequestsController.

https://github.com/ORT-DA2/168347/commit/b423fb28b4c1d2c20d30f167b08bbba8c91b39e4

Etapa Green: Implementada la lógica que permita cambiar el estado de una solicitud sin a ver ninguna validación, además se implementó el método Put de la clase RequestsController el cual llama a la lógica mencionada.

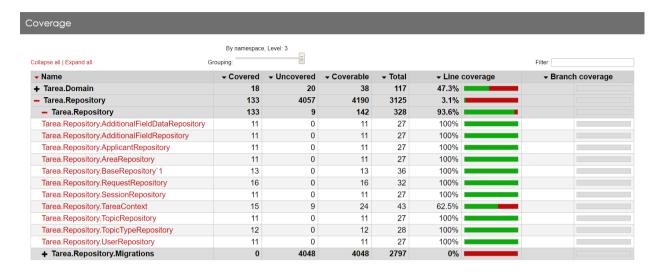
https://github.com/ORT-DA2/168347/commit/ac3181d5977ecd1d733395d9440e6d5c64e5d1dd

Etapa Refactor: El estado de una solicitud ahora es una string en vez de un enumerado. Se pasó una parte de la lógica de validación de estado de solicitudes del Backend hacia BusinessLogic.

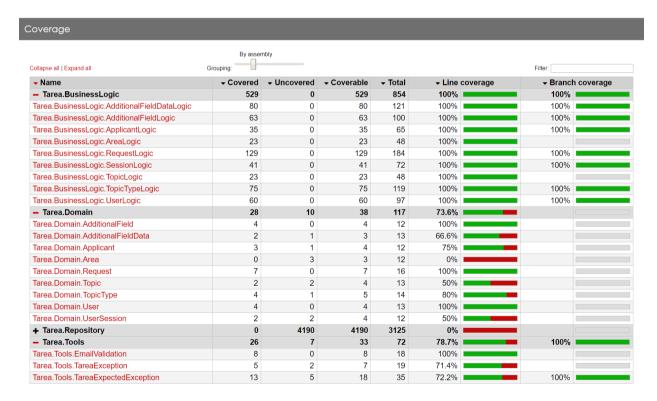
https://github.com/ORT-DA2/168347/commit/13bf758b81c9863bac38cd49001d8ef651ec04c4

Resultado de la ejecución de las pruebas y cobertura

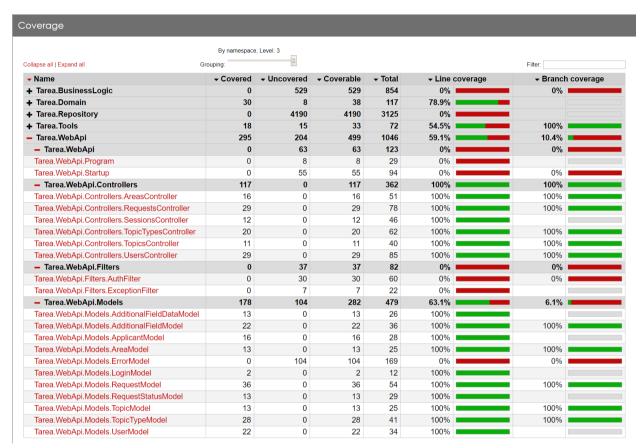
Tarea.Repository.Tests – Se hicieron pruebas unitarias para todas las clases del paquete, La única cobertura que no fue realizada fue la ejecución de los métodos set; de los DbSet en la clase TareaContext. Sin contar esto se alcanzó un 100% de cobertura



Tarea.BusinessLogic.Tests – Se hicieron pruebas unitarias para todas las clases de la lógica alcanzando un 100% de cobertura.



Tarea.WebApi.Tests – Se hicieron pruebas unitarias para todas las clases del paquete Controllers alcanzando un 100% de cobertura. Debido a la naturaleza de los filtros, no es posible probarlos de forma fácil, se consultó con el docente y se decidió no hacer pruebas unitarias, el funcionamiento de estos filtros se puede probar a través de pruebas de integración detalladas en el siguiente capítulo. De forma similar, no se hicieron pruebas unitarias a la clase ErrorModel del paquete Tarea.WebApi.Model por ser una clase que depende mucho de la entrada y sería tedioso probarla, lo ideal es probar su funcionamiento con pruebas de integración.



2. Clean Code

Este trabajo fue realizado teniendo en cuenta los criterios de Clean Code para que el código sea fácil de leer, entender y modificar. Para esto se aplicaron los siguientes puntos:

Nombres significativos (métodos y variables):

- Revelan la intensión: el nombre de los métodos explica qué hace. Nombre de variables explica para qué va a ser usada esa variable.
- Pronunciables
- No incluye su tipo: variables no incluyen su tipo (nombreString)
- Convenciones: Interfaces comienzan con la letra I (ILogic, IRequestLogic, IUserLogic)

```
public Request Create(Request request)

{
    GenerateRequestData(request);
    Validate(request);
    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);

    Validate(request);
```

Funciones:

- Chicas
- Pocos bloques
- Hacen una sola cosa y la hacen bien
- La menor cantidad de parámetros posible
- Excepciones en vez de códigos de error

Comentarios:

• No se usaron, el código se debe explicar por si solo

Estilo:

- Si una función llama a otra entonces tienen que estar cerca
- Cada nivel sigue su sangría

```
ate void GenerateRequestData(Request entity)
   ValidateTopicType(entity.TopicType);
   entity.Id = Guid.NewGuid();
   entity.TopicType = TopicTypeLogic.Get(entity.TopicType.Id);
  blic void Validate(Request request)
   if (String.IsNullOrEmpty(request.Details))
    if (request.Details.Length > 2000)
       throw new TareaException("ERR REQUEST DETAILS LONG");
   ApplciantLogic.Validate(request.Applicant);
   ValidateTopicType(request.TopicType);
ValidateAdditionalFields(request.AdditionalFields, request.TopicType);
  vate List<AdditionalFieldData> GenerateFields(List<AdditionalFieldData> additionalFields)
       additionalFields.ForEach(af => fields.Add(AdditionalFieldDataLogic.Create(af)));
   return fields;
2 references
private void ValidateTopicType(TopicType topicType)
   if (topicType == null)
        throw new TareaException("ERR_REQUEST_TOPICTYPE_NULL_NOTFOUND");
```

Objetos y estructuras de datos:

- Ocultar implementación de métodos a través de interfaces: ILogic, IRequestLogic, etc.
- Uso de polimorfismo para definir distinto comportamiento de una función para distintos tipos de objetos que implementen algo en común: Clase abstracta BaseRepository.

Manejo de errores:

• Independiente de la lógica: Uso de ExceptionFilter el cual su funcionamiento ya fue explicado.

Clases:

• Chicas y con pocas responsabilidades.