Universidad ORT Uruguay

Facultad de Ingeniería

https://github.com/IngSoft-DA2-2023-2/276280-243218

1er Obligatorio Diseño de Aplicaciones

Evidencia de Clean Code y de la aplicación de TDD

Gonzalo Loureiro – 243218

Santiago Alfonso – 276280

02 de May de 2024

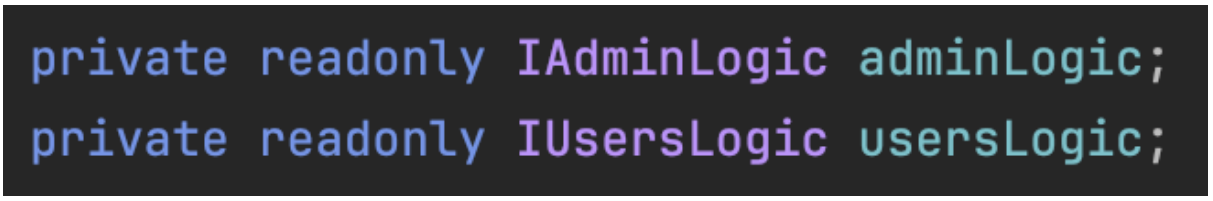
Índice

Índice **1**
Clean Code **2**
Testing **4**
Evidencia TDD 4
Cobertura 5

Clean Code

Clean Code se refiere a un estilo de programación que enfatiza la creación de código que no solo funciona correctamente, sino que también es fácil de leer, entender y modificar. Un código limpio es accesible para otros programadores, incluyendo futuros tú, facilitando la mantención y expansión del software. Es claro, conciso, y sigue principios y patrones de diseño reconocidos que mejoran la calidad del software y reducen la posibilidad de errores. Además, el código limpio generalmente acompaña una base de pruebas sólida que asegura su correcto funcionamiento a través del tiempo.

Nombres Significativos: Usar nombres que revelen intención, eviten desinformación, y proporcionen información suficiente para que el lector del código pueda entenderlo sin necesidad de comentarios adicionales.



Como se evidencia en la imagen, el nombre de tanto las clases como las variables revelan que es (componente lógico, parte de Business Logic) y a que recurso está asignado (Admin y User respectivamente)

Funciones: Las funciones deben ser pequeñas, hacer una sola cosa, y hacerla bien. Deben tener un número reducido de argumentos, no tener efectos secundarios, y sus nombres deben describir su propósito.

La intención a lo largo de todo el obligatorio fue mantener lo más contenido y reducido posible las tareas de una función, se logró tal cometido como se evidencia en los ejemplos de código presentados.

```
public Request CreateRequest(string description, Apartment department, Category
category)
{
   var newRequest = new Request()
   {
       Description = description,
       Department = department,
       Category = category,
       Status = RequestStatus.Open,
       AssignedToMaintenanceId = department.Building.Id,
       BuildingAssociated = department.Building
   };
   requestRepo.Add(newRequest);
   return newRequest;
}
```

Como se puede ver en el snippet presentado, la unica tarea que tiene esta funcion, es crear una solicitud de servicio.

```
public Invitation Create(Invitation invitationData)
{
   List<Invitation> invitations = invitationRepo.GetAll();
   bool repeatedInvitation = invitations.Exists(invitation =>
       invitation.RecipientEmail == invitationData.RecipientEmail &&
invitation.DeadLine > DateTime.Now
       && invitation.Status != RejectedStatus);
   if (repeatedInvitation)
   {
       throw new DuplicateEntryException();
   }
   Invitation newInvitation = new Invitation()
   {
       DeadLine = invitationData.DeadLine, RecipientEmail =
invitationData.RecipientEmail,
       Name = invitationData.Name, Status = PendingStatus
   };
   invitationRepo.Create(newInvitation);
   return newInvitation;
}
```

Comentarios: Los comentarios deben usarse sabiamente. Es mejor tener código que sea autoexplicativo que código lleno de comentarios. Los comentarios deben evitar ser descriptivos y no deben contener información redundante o engañosa.

Mantuvimos los comentarios a un minimo, unicamente si verdaderamente eran necesarios

Objetos y Estructuras de Datos: Aboga por exponer menos los detalles de implementación y más las abstracciones. Es decir, utilizar encapsulamiento para esconder los datos y exponer operaciones para manipular esos datos.

Manejo de Errores: Tratar el manejo de errores como una parte separada del programa, utilizando excepciones en lugar de códigos de retorno, y creando contextos de ejecución donde se sabe que no ocurrirán errores.

Hicimos uso de tanto excepciones personalizadas como de Filtros de Excepción para la WebAPI para asi tener mejor control de que errores presentar al usuario


```
public class DuplicateEntryException :
Exception
{
   public DuplicateEntryException() :
base("Please ensure not to input
duplicate unique values.") { }
}
```

Pruebas Unitarias: Las pruebas deben ser limpias, leerse bien, y ser fáciles de entender. Deben cubrir cada uno de los casos de uso esperados y excepcionales del código que prueban.

En el contexto de la tarea, las pruebas unitarias son parte integra de la misma ya que como sera mencionado mas adelante, se requiere un 90% de cobertura de codigo. Tomamos la decisión de separar las pruebas en tres grandes grupos,

DataAccessTest, ControllerTest y ServicesTest los que testean distintas parte del sistema.

Test Correcto

```
[TestMethod]
   public void GetAllAdmins()
   {
       List<Admin> expectedList = new List<Admin>() { new Admin() { Id = UserId } };
       adminRepo.Setup(repo => repo.GetAll()).Returns(expectedList);
       adminService = new AdminLogic(adminRepo.Object);

       List<Admin> returnedList = adminService.GetAll();
       adminRepo.VerifyAll();
       CollectionAssert.AreEqual(expectedList, returnedList);
   }
```

Test Falla

```
[TestMethod]
   [ExpectedException(typeof(EmptyOrNullException))]
   public void InValidAdminNameThrowsException()
   {
       Admin newAdmin = new Admin() { Email = ValidEmail, Id = UserId, Password =
ValidPassword, Name = EmptyString, LastName = ValidLastName };
       adminRepo.Setup(repo => repo.Add(It.IsAny<Admin>()));
       adminService = new AdminLogic(adminRepo.Object);
       Admin returnedAdmin = adminService.Create(newAdmin);
       adminRepo.VerifyAll();
   }
```

Testing

Evidencia TDD

La idea detrás de TDD, es primero hacer todos los test para las funciones a implementar en una clase, luego crear la clase y sus funciones para así al final hacer los cambios necesarios en los tests/funciones para que los tests sean exitosos.

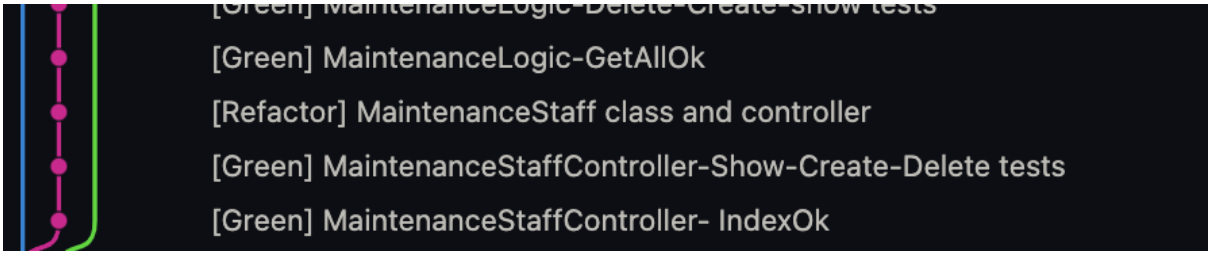
Usualmente y es una buena práctica hacerlo, se crean los tests, estos fallaran ya que la clase y sus funciones las cuales deben testear, no existen (RED phase). Aquí se debería hacer un primer commit para evidenciar el hecho de que se pasó por la etapa de RED.

La siguiente etapa es GREEN, se debe escribir la mínima cantidad de código necesaria para que los test funcionen, esto suele ser el implementar las funciones de la clase a ser testeada, aquí se debe hacer un segundo commit.

La tercera y a veces última etapa es REFACTOR, en esta etapa se debe modificar las funciones para que no solo hagan que los test pasen, sino que también cumplan con los estándares de Clean Code y a su vez con los requerimientos planteados en la letra de la tarea.

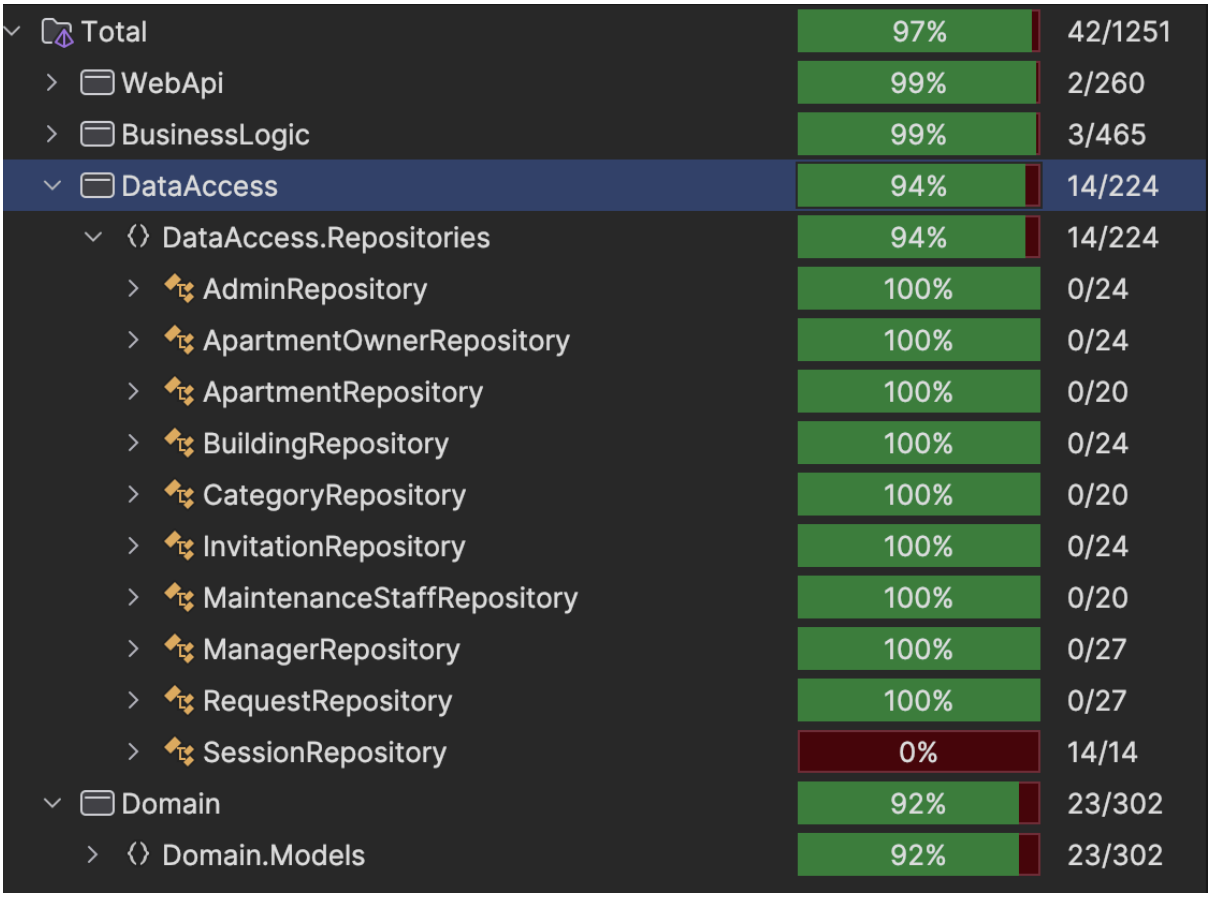
A continuación se presentan un par de ejemplos de TDD a lo largo del proyecto.

Se mencionó que no era tan necesario el commit en fase **red** por lo que veremos ejemplos mayoritariamente de fase **green** y **refactor,** de igual manera algunos commit en fase **red** si completamos



Cobertura

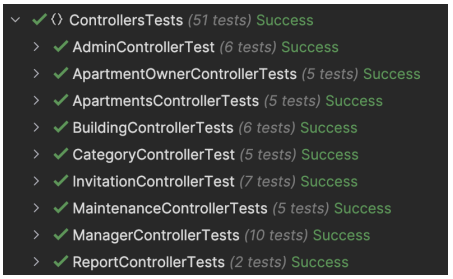
La cobertura de código es un parámetro para determinar qué funciones y líneas de código fueron sometidas a pruebas. Es muy útil para evaluar la calidad del conjunto de pruebas. Se pedía entre un 90 y 100 porciento de cobertura, en esta ocasión, logramos obtener un **97** porciento



La única sección que quedó sin testear bajo los principios de TDD fue el Session Repository dado que no teníamos del todo claro el camino a tomar para su implementación.

Podemos separar el testing en tres diferentes grupos de tests, DataAccess, Controller y Services.

Como bien detallan su nombre, los test dentro de DataAccess testean los repositorios para acceder a la base de datos. Dentro de estos test testamos que las funciones relacionadas a la creacion, actualizacion y borrado de datos de la base de datos, funcione como es esperado. Como se ve en la imagen, el 100% de los tests dentro de esta sección funcionan.



Por otro lado, los test dentro de Controller testean los controladores de la web api. Como se ve en la imagen, el 100% de los tests dentro de esta sección funcionan.

Y por ultimo, tenemos los tests de los servicios, estos son los encargados de verificar que cuando se instancia un nuevo recurso (i.e. Request) este se cree de forma correcta y todas las validaciones especificadas sean tenidas en cuenta.