Stella Steinfeld (229398)
Eugenia Matto (227056)
Santiago Vairo(244023)

Diseño de aplicaciones 2

Evidencia de Clean Code y TDD

https://github.com/IngSoft-DA2/Steinfeld-Vairo-Matto

Declaración de autoría

Nosotros, Stella Steinfeld, Eugenia Matto y Santiago Vairo, 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 realizamos el obligatorio de 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

Buenas prácticas de clean code

Como parte del acuerdo a la hora de entregar el proyecto siempre tuvimos en mente las buenas prácticas aprendidas en clase de clean code. Hicimos un gran hincapié en que los nombres fueran claros y cumplieran con los parámetros esperados, que los métodos cumplieran solo son una responsabilidad y no superaran los 2 parámetros esperados. El código es su propia documentación y teniendo en cuenta lo expuesto anteriormente, no hay ningún comentario en el mismo. Los comentarios en el código solo los usamos durante el proceso de desarrollo para dejarnos @TODO entre nosotros para saber qué es lo que falta implementar, en el caso de que se tomase alguna decisión especial simplemente nos los comunicamos por whatsapp o cuando nos reuniamos para trabajar juntos.

Como ejemplo todo esto, expondremos el análisis de la clase AccountService.cs en nuestro proyecto (una selección totalmente arbitraria)

Nombres Claros y Significativos

```
    public PaginatedResult<AccountModel> GetAll(string? role, string?
fullName, int pageNumber, int pageSize)
    {
    }
```

Esto muestra que como AccountService, GetAll, role, fullName, pageNumber, y pageSize describen claramente el propósito de las variables y métodos.

Funciones cortas y con una única responsabilidad

```
public PaginatedResult<AccountModel> GetAll(string? role, string? fullName,
int pageNumber, int pageSize)
{
    var query = _userRepository.GetAccounts();

    if (!string.IsNullOrEmpty(role))
    {
        if (Enum.TryParse(role, true, out Role roleEnum))
        {
            query = query.Where(u => u.Role.ToString() == role);
        }
        else
```

```
        {
            throw new ArgumentException("Not valid Role.");
        }
    }

    if (!string.IsNullOrEmpty(fullName))
    {
        query = query.Where(u => (u.Name + " " +
u.Surname).Contains(fullName, StringComparison.OrdinalIgnoreCase));
    }

    var totalAccounts = query.Count();

    var accounts = query.Skip((pageNumber - 1) * pageSize).Take(pageSize)
        .Select(u => new AccountModel
        {
            Name = u.Name,
            Surname = u.Surname,
            FullName = u.Name + " " + u.Surname,
            Role = u.Role.ToString(),
            CreatedAt = u.CreatedAt
        }).ToList();

    return new PaginatedResult<AccountModel>
    {
        Items = accounts,
        TotalCount = totalAccounts,
        PageNumber = pageNumber,
        PageSize = pageSize
    };
}
```

El método GetAll se enfoca en una responsabilidad clara: obtener, filtrar y paginar las cuentas. Aunque realiza varias operaciones, el enfoque está en esa única responsabilidad.

Como punto de mejora podríamos sacarle responsabilidades a este método y dividirlo en métodos mas pequeños para que el método sea más corto.

Código bien organizado y estructurado

```
public PaginatedResult<AccountModel> GetAll(string? role, string? fullName,
int pageNumber, int pageSize)
{
    var query = _userRepository.GetAccounts();

    if (!string.IsNullOrEmpty(role))
    {
        if (Enum.TryParse(role, true, out Role roleEnum))
        {
            query = query.Where(u => u.Role.ToString() == role);
        }
        else
        {
            throw new ArgumentException("Not valid Role.");
        }
    }

    if (!string.IsNullOrEmpty(fullName))
    {
        query = query.Where(u => (u.Name + " " +
u.Surname).Contains(fullName, StringComparison.OrdinalIgnoreCase));
    }

    var totalAccounts = query.Count();

    var accounts = query.Skip((pageNumber - 1) * pageSize).Take(pageSize)
        .Select(u => new AccountModel
        {
            Name = u.Name,
            Surname = u.Surname,
            FullName = u.Name + " " + u.Surname,
            Role = u.Role.ToString(),
            CreatedAt = u.CreatedAt
        }).ToList();

    return new PaginatedResult<AccountModel>
    {
        Items = accounts,
        TotalCount = totalAccounts,
        PageNumber = pageNumber,
        PageSize = pageSize
    };
}
```

El flujo del código sigue una estructura lógica: primero se filtra por rol, luego por nombre completo, y finalmente se página y proyecta los resultados.

Manejo adecuado de excepciones

```
if (Enum.TryParse(role, true, out Role roleEnum))
{
    query = query.Where(u => u.Role.ToString() == role);
}
else
{
    throw new ArgumentException("Not valid Role.");
}
```

El código maneja correctamente las excepciones, lanzando una ArgumentException cuando el rol no es válido.

Principios SOLID

```
public class AccountService : IAccountService
{
    private readonly IUserRepository _userRepository;

    public AccountService(IUserRepository userRepository)
    {
        _userRepository = userRepository;
    }

    public PaginatedResult<AccountModel> GetAll(string? role, string?
fullName, int pageNumber, int pageSize)
    {
        var query = _userRepository.GetAccounts();
       }
}
```

El servicio AccountService sigue el principio de responsabilidad única (SRP), ya que se enfoca en la lógica relacionada con la obtención de cuentas. Dividir responsabilidades dentro de la clase AccountService no solo cumple con el principio de responsabilidad única (SRP), sino que también mejora la capacidad de testeo unitario, ya que los métodos que filtran por rol o por nombre pueden ser probados de forma aislada

Aplicación de TDD

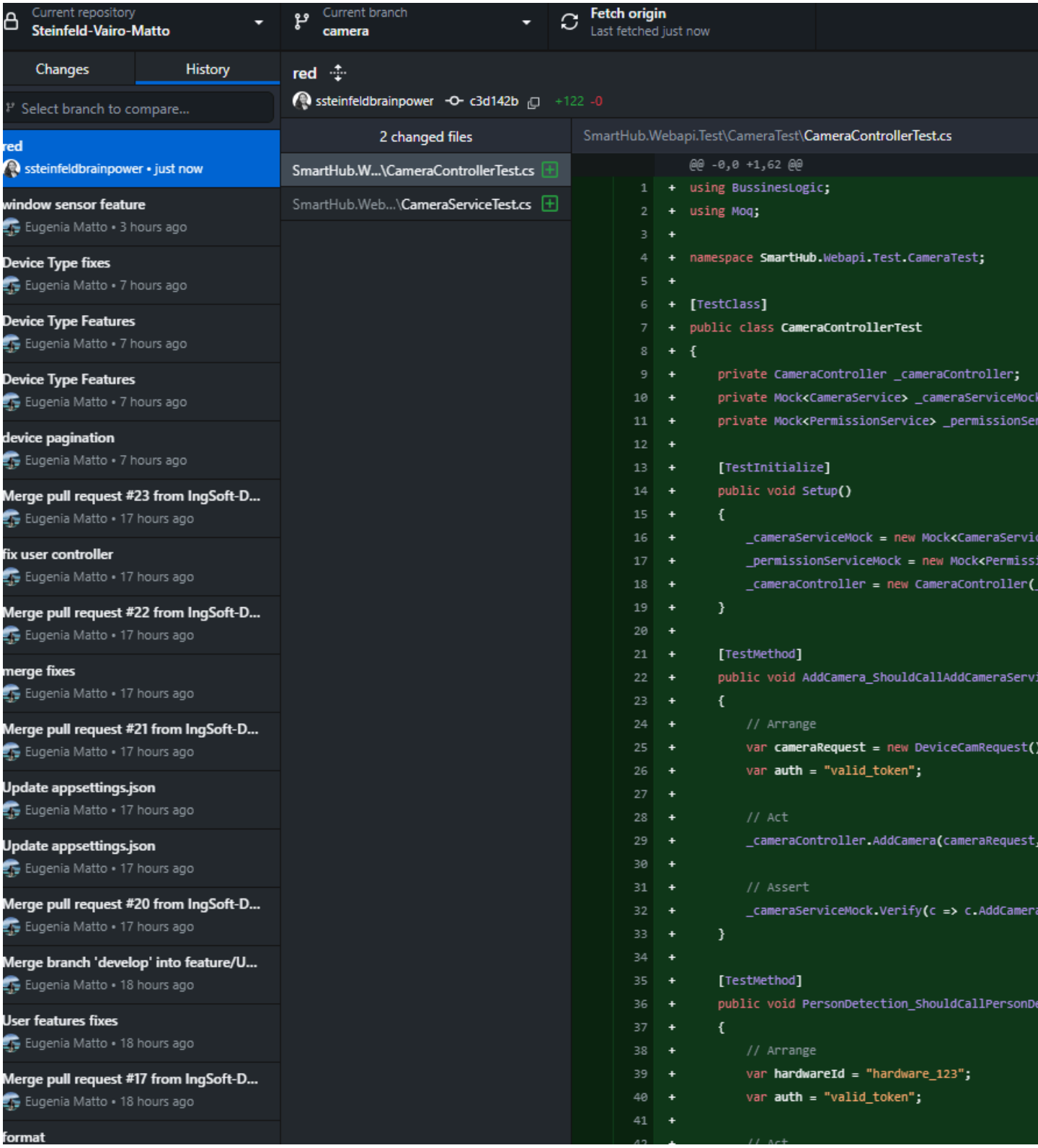
En esta entrega, debido a limitaciones de tiempo, no pudimos desarrollar la interfaz necesaria para los repositorios, lo que habría facilitado la aplicación de pruebas unitarias y, por lo tanto, no logramos aplicar completamente la metodología de desarrollo TDD (Test Driven Development) en la capa de acceso a datos (DataAccess). La implementación de una interfaz como IRepository es fundamental para desacoplar la lógica de negocio de la base de datos subyacente, lo que permite realizar simulaciones (mocks) en lugar de depender de una base de datos real durante las pruebas. Esta interfaz nos habría permitido construir pruebas más eficientes y confiables, garantizando que los repositorios fueran testeables de manera independiente del estado o configuración de una base de datos real.

Se intentó aplicar TDD durante todo el desarrollo del proyecto, comenzando siempre con los tests, luego escribiendo el código mínimo necesario para hacer que las pruebas pasaran, y finalmente refactorizando para mejorar la calidad del código. Este ciclo de RED, GREEN y REFACTOR fue repetido las veces que fue necesario para asegurar la calidad del desarrollo, y dejamos en el repositorio varios commits etiquetados con estas palabras clave para documentar de forma clara y explícita la aplicación de TDD. Más adelante, agregaremos en el documento un ejemplo sencillo con capturas de pantalla que ilustran este ciclo en la creación de la funcionalidad de cámaras.

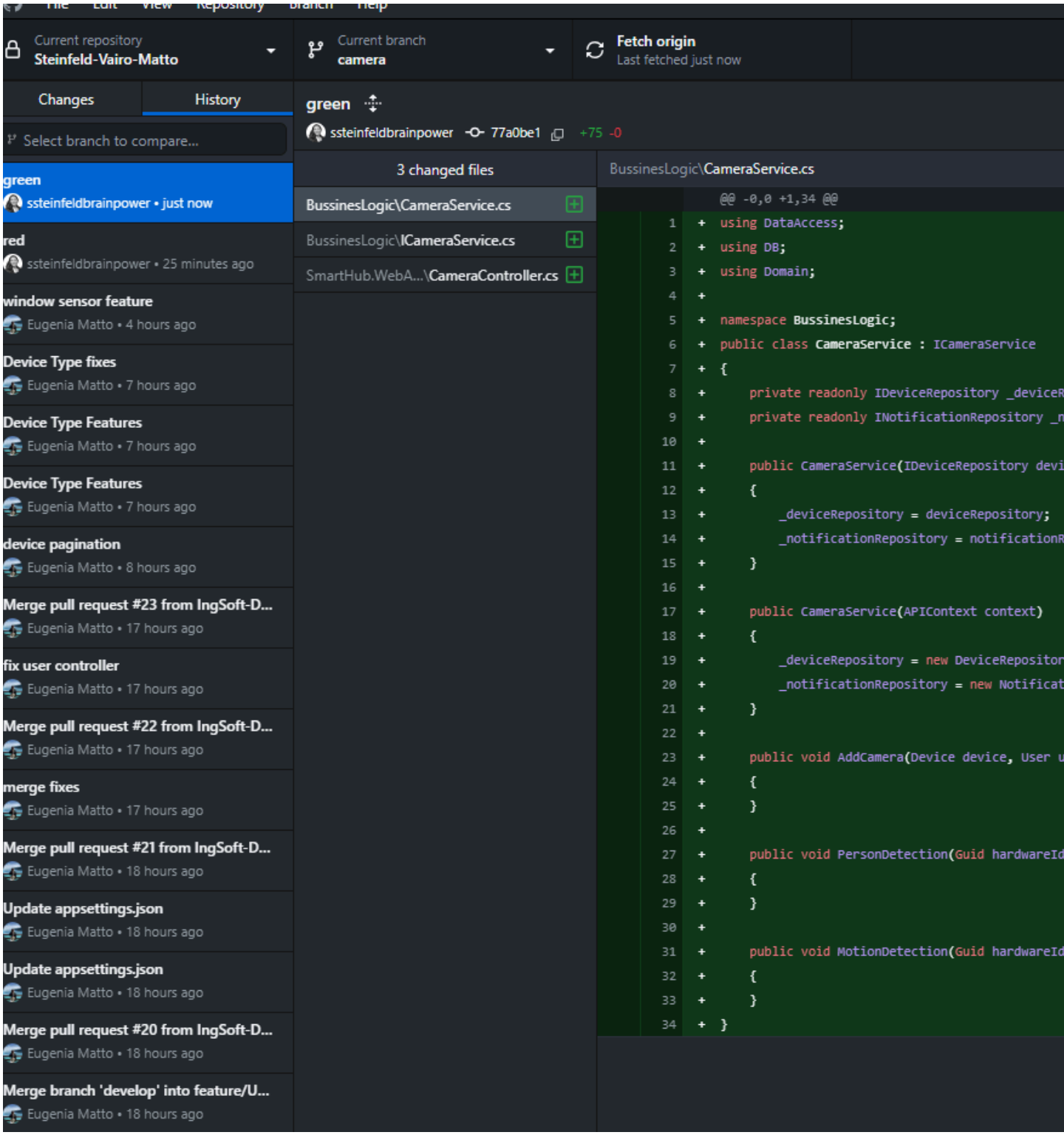
Sin embargo, hacia el final del proyecto, debido a las urgencias del caso, fue necesario agregar algunos tests para cumplir con los niveles de cobertura exigidos por la cátedra. Esto generó algunos inconvenientes, ya que la cobertura de código se vio afectada por varias clases de interfaz en la capa de BusinessLogic que no contienen código pero que, aun así, el analizador de cobertura de GitHub contabiliza como líneas no cubiertas.

Como resultado, la creación de la interfaz para los repositorios quedó como una deuda técnica que abordaremos en futuras entregas. En la próxima iteración, se priorizará la implementación de esta interfaz, lo que permitirá aplicar TDD de manera completa en la capa de acceso a datos. Al hacerlo, seremos capaces de crear pruebas unitarias que usen simulaciones en lugar de bases de datos reales, garantizando una arquitectura más limpia, mantenible y alineada con las mejores prácticas de desarrollo ágil.

RED en Camera



GREEN camera



REFACTOR camera