

Universidad ORT Uruguay
Facultad de Ingeniería
Ing. Bernard Wand-Polak

Segundo Obligatorio

Diseño de aplicaciones 2

2016

Docente: Nicolás Fornaro

Grupo: N6A



Nombre: Fernando Artola
Nº de estudiante: 181331
Email: fernando.artola@hotmail.com



Nombre: Juan Bautista Heber
Nº de estudiante: 180829
Email: juanbautistaheber@gmail.com

Índice

Descripción general.....	3
Problemas conocidos	3
Instructivo de instalación	4
Correcciones de la primera entrega.....	5
Análisis de los Requerimientos	6
Requerimientos funcionales	6
Requerimientos no funcionales	7
Justificación del Back-end	7
Justificación y calidad del front-end.....	9
Como fue solicitado en la letra el front-end fue desarrollado utilizando la librería.....	9
Diagrama de componentes	11
Diagrama de entrega.....	12
Diagramas de paquetes.....	13
Diagramas de clases	14
Entities.....	14
Logic	15
Diagramas de interacción.....	16
Informe de las métricas.....	17
Dependency Graph.....	19
Dependency Matrix.....	20
Treemap Metric View.....	21
Abstractness vs. Instability.....	22
Descripción de los endpoints del api	23
AdminController.....	23
GameSettingsController	23
InvitationCodeController.....	24
PlayerController	24
StockController.....	24
StockHistoryController	25
StockNewsController.....	25
TransactionController	25
UserController	26

Modelo de tablas de estructura de la base de datos	27
Evidencia de Clean Code	28
Nombres significativos	28
Funciones	29
Comentarios	29
Formato.....	30
Objetos y estructura de datos.....	31
Manejar errores	32
Pruebas unitarias.....	32
Clases.....	32
Cobertura de las pruebas	33

Descripción general

Tal como se fue encargado, se entregó un sistema el cual es capaz de realizar todas las funcionalidades del juego a través de una página web.

La aplicación es capaz de guardar todos los datos en la base de datos. Para esto utilizamos Entity Framework de la manera code first.

El sistema se separa en cuatro partes principales, el modelo del sistema, la base de datos, el front end de la página, y la capa REST.

Antes de comenzar a realizar el sistema, nos fijamos los objetivos de que el sistema sea mantenible, flexible y de calidad. Para cumplir con estas características nosotros seguimos las pautas del libro "Clean Code" del capítulo 2 al 10 y el 12. De esta manera, cuando nosotros queramos realizar modificaciones, podrán realizarse de manera fácil y rápida ya que el código se entenderá inmediatamente.

Utilizamos inyección de dependencias en todo el proyecto.

La inyección de dependencias es una herramienta comúnmente utilizada en varios patrones de diseño orientado a objetos, consiste en inyectar comportamientos a componentes.

Esto no es más que extraer responsabilidades a un componente para delegarlas en otro, estableciendo un mecanismo a través del cual el nuevo componente pueda ser cambiado en tiempo de ejecución.

Problemas conocidos

El mismo día de la entrega nos dimos cuenta que implementamos la funcionalidad de modificar y borrar stocks en la Windows form application y la debíamos implementar en la web, no nos dio el tiempo para cambiarlo pero por lo menos anda a la perfección en el Windows form.

La ventana que aparece en la página 6 de la primera letra del obligatorio nos pareció innecesaria. Nos pareció que alcanzaba con la vista de buscar stocks agregándole la muestra de noticias relacionadas.

Instructivo de instalación

1. Abrir el proyecto Stockapp del Visual Studio
2. Cambiar el connection string ubicado en el web.config dentro del paquete Stockapp.Portal
3. Abrir la consola del administrador de paquetes nuget y seleccionar el paquete Stockapp.Data.Access
4. Escribir los comandos enable-migrations, add migration 1, corremos la aplicación e intentamos iniciar usuario con cualquier dato, en este momento se crearan la base de datos y los datos de prueba
5. Ir al sql server managment, seguridad, inicio de sesión, botón derecho en NT AUTHORITY\SYSTEM, asignación de usuario, tiqueamos la opción asignar de Context y finalmente tiqueamos la opción db_owner
6. Abrir el IIS Mangaer
7. Botón derecho en sitios, agregar sitio web
8. Nombre del sitio: Stockapp
Ruta de acceso fija: Ruta donde tenemos el proyecto, carpeta Stockapp.Portal.
Por ejemplo:
C:\Users\ferna\Desktop\Facultad\Diseño 2\Stockapp\Stockapp\Stockapp.Portal
Puerto: Cualquiera por ejemplo 14
Nombre de host: localhost
9. Abrimos Grupo de aplicaiones, botón derecho en Stockapp, y cambiamos Identidad por LocalSystem
10. Verificamos que haya quedado publicado correctamente ingresando en el navegador con la siguiente dirección <http://localhost:PuertoQueElegimos> (ej: http://localhost:14)

Al realizar estos pasos crearemos la base de datos y se cargaran los datos de prueba automáticamente.

Igualmente en el cd y en el bitbucket incluimos una carpeta con el .bak y .sql de la base de datos con datos de prueba y otros sin datos de prueba.

Correcciones de la primera entrega

Se corrigieron los siguientes puntos de la primera entrega:

- Funcionalidad de realizar transacciones y refrescar el portfolio
- Manejo de los stocks en un portfolio mediante la clase actions
- Se incluyó diagrama de componentes para acompañar la justificación del backend y se lo puso junto con el diagrama de entrega
- Se incluyó la justificación del uso de inversión de dependencias (En el título Descripción general página 2)
- Se corrigieron los diagramas con los nuevos cambios realizados para esta segunda entrega, se incluyó multiplicidad en los diagramas
- Se emprolijó el diagrama de clases sacando las 2 interfaces e indicando mediante palabras que todas las entidades implementan a estas 2
- Se cambiaron los diagramas de secuencia y se explicó cómo se conectan los Controllers con la lógica
- Se incluyó listado de los endpoints de la api
- Se arreglaron métodos que comenzaban su nombre con minúscula
- Se arreglaron los nombres de las clases de excepciones de las entidades de plural a singular
- Se arreglaron los Tests que no pasaban, ahora pasan todos

Análisis de los Requerimientos

En esta sección explicaremos el sistema que se nos fue encargado a realizar, explicando también cada uno de los requerimientos funcionales y no funcionales.

Requerimientos funcionales

Título	Prioridad	Descripción
Registrar un nuevo usuario	Alta	Un usuario será capaz de registrarse como un nuevo usuario ingresando su nombre completo, e-mail y una contraseña.
Iniciar Sesión	Alta	Un usuario ya registrado podrá ser capaz de iniciar sesión ingresando su e-mail y su contraseña.
Acceso al portfolio	Alta	Un usuario que ya esté ingresado y sea un jugador, podrá tener acceso su portfolio.
Realizar transacción	Alta	Un usuario que ya esté ingresado y sea un jugador, podrá realizar transacciones siempre y cuando sea posible.
Generación de invitaciones	Alta	Un usuario que ya esté ingresado y sea un administrador, podrá ser capaz de generar códigos de invitación, los que luego enviará a posibles futuros usuarios. Para ello indicará cuantos y el sistema presentará una lista de códigos generadas al azar, todos únicos, alfanuméricos y siempre de largo 8. Estos códigos deben quedar almacenados para luego ser validados cuando se registran los usuarios.
Modificación de las condiciones de juego	Alto	Un usuario que ya esté ingresado y sea un administrador, podrá ser capaz de cambiar variables que alteran el comportamiento del juego. 1) Modificar la cantidad inicial de dinero para los usuarios 2) Modificar la cantidad máxima de transacciones permitidas por usuario/día 3) Cambiar el algoritmo en uso para recomendación de acciones
Grafica de evolución	Media	Un usuario que ya esté ingresado y sea un jugador, podrá ser capaz de visualizar su gráfica de evolución.
Ver recomendación	Media	Un usuario que ya esté ingresado y sea un jugador, podrá ver las recomendaciones del sistema.
Buscar un stock	Media	Un usuario que ya esté ingresado y sea un jugador, podrá buscar un stock por nombre y contenido de la descripción.
Ingreso de noticias	Media	Un usuario que ya esté ingresado y sea un administrador, podrá ser capaz de ingresar una noticia indicando una fecha, un título, un contenido y una lista de stocks a los que la noticia está asociada.
Eliminar noticias	Media	Un usuario que ya esté ingresado y sea un administrador, podrá ser capaz de eliminar una noticia
Definir un stock	Media	Un usuario que ya esté ingresado y sea un administrador, podrá ser capaz de definir un stock indicando el código del mismo, un nombre, una descripción y la cantidad de acciones que compondrán el pool

Definir un nuevo precio	Media	Un usuario que ya esté ingresado y sea un administrador, podrá ser capaz de definir un nuevo precio de la acción de un stock.
Historial de transacciones	Media	Un usuario que ya esté ingresado y sea un jugador, podrá consultar en cualquier momento su historial de transacciones. Para obtener el mismo siempre será necesario especificar un rango de fechas y opcionalmente se podrá filtrar por stock
Ver el pool de acciones disponibles	Media	Un usuario que ya esté ingresado y sea un administrador, podrá ser capaz de consultar un listado que presente el tamaño del pool disponible para cada stock y el valor actual. Este listado debe permitir la navegación para editar los datos del stock.

Requerimientos no funcionales

Título	Prioridad	Descripción
Interfaz intuitiva	Media	La interfaz debe ser intuitiva, para esto se seguirán las guías de Material Design
Debe hostales en el IIS de Microsoft	Alta	La aplicación debe estar hosteada en e IIS de Microsoft

Justificación del Back-end

Antes de comenzar a realizar el sistema, tuvimos que tomar varias decisiones con respecto al diseño que llevaríamos a cabo. Una de estas decisiones era como separar el sistema para dividir las funcionalidad. Es por eso que concluimos en dividir el sistema en ocho proyectos distintos. Estos son: Stockapp.Data, Stockapp.Data.Access, Stockapp.Data.Repository, Stockapp.Logic, Stockapp.Portal, Stockapp.Resolver, Stockapp.Test, Stockapp.DesktopApp.

En el proyecto Stockapp.Data se encuentran todas las clases de las entidades involucradas. En estas clases se encuentran únicamente sus atributos, sus constructores y métodos de getter y setter. Como queríamos que las entidades solo se conozcan a sí mismas creamos todos sus atributos privados y sus métodos públicos para que puedan ser accedidos desde otras clases. Intentamos que estas clases queden lo más chicas posibles por la misma razón previamente dicha, que se conozcan únicamente a sí mismas. Si algunos métodos que están actualmente en la lógica, lo hubiésemos puesto en las entidades, iban a comenzar a tener objetos de otras clases y llamar a métodos de otras clases. Haciéndolas bien chicas es posible obtener una alta cohesión y un bajo acoplamiento.

El proyecto Stockapp.Logic tiene nueve clases. En estas nueve clases se encuentran los métodos relacionados con las entidades del nombre de la clase. Realizamos esta distinción

entre las lógicas y las entidades en si para que las entidades no se relacionen con otras entidades y de esta forma mantener una alta cohesión y bajo acoplamiento. En el proyecto lógica, con ayuda de las validaciones, se fija si los datos ingresados son correctos. En caso de que no sean correctos, tira una excepción acorde a la validación que no se cumplió. En caso de que si sean correctos los datos ingresados, las lógicas llaman a los repositorios para completar lo pedido, ya sea obtener una entidad, agregar una entidad entre otras posibles funciones.

El proyecto Stockapp.Test contiene una clase por cada clase lógica, la cual prueba todos los métodos de la clase lógica asociada. Para esto instalamos xUnit y el runner de xUnit. En este proyecto también se encuentran otras dos clases como las pruebas del repositoryTest para cada una de las entidades.

El proyecto Stockapp.Data.Repository contiene las clases DependencyResolver, GenericRepository, IRepository, IUnitOfWork, UnitOfWork. Estas clases contienen los métodos que interactuar directamente con la base de datos.

El proyecto Stockapp.Data.Access contiene las clases inicializador y context. La clase inicializadora agrega datos a la base de datos y la clase context es la que posee todo el contexto del programa.

En el proyecto Stockapp.Portal se encuentra todo lo relacionado con la Web api, este es un proyecto webApi2. De este proyecto solo utilizamos los controladores y los archivos de configuración. Cada controlador tiene los métodos necesarios para brindar funcionalidad a la API. Contamos con tres controladores: UserController, StockController, TransactionController.

En el paquete Stockapp.Resolver se encuentran las clases ComponentLoader, IComponent, IRegisterComponent. Resuelven problemas de las clases que están en el paquete Stockapp.Data.Repository.

El paquete Stock.DesktopApp es de tipo Windows form y se encarga de realizar las funcionalidades del administrador solicitadas en la letra, esto lo hace la clase DesktopApp.cs. Se controlaron las correspondientes validaciones y se usó una interfaz amigable y fácil de manipular para el usuario. En todo momento se le informa mediante mensajes de confirmación, información y alertas al usuario para mantenerlo informado del curso que está tendiendo en la aplicación.

Realizamos esta división para separar de forma eficaz todo lo que no tendría que conocerse de

otras clases. Por ejemplo la interfaz nunca debería conocer la estructura del modelo, debería funcionar de la misma manera sin importar como fue implementado. Esta división nos da mayor cohesión y menor acoplamiento.

Justificación y calidad del front-end

Como fue solicitado en la letra el front-end fue desarrollado utilizando la librería Angular.js. Para el estilo utilizamos un template gratuito de Bootstrap llamado sb-admin-2, <http://blackrockdigital.github.io/startbootstrap-sb-admin-2/>. Este template viene con archivos css y javascript, nosotros utilizamos los css e incluimos los javascript para que no genere errores, pero no utilizamos ninguna de las funcionalidades.

La estructura del proyecto web se divide en la página de inicio(Index.html) y los Web.config. Estas carpetas son: Fonts, views, scripts, content. La carpeta Font contiene los iconos que fueron utilizados en la página, estos iconos fueron descargados de una librería gratuita llamada FontAwesome.

La carpeta views contiene las distintas vistas que se cargan en la página index.html con el componente “ng-view”. Además cada vista tiene su propio controlador, esto hace que la página quede modularizada y cada controlador funciona de manera independiente del resto. Esto permite poder acceder a los distintos controladores para otras vistas.

- **login.html:** Esta vista muestra una pequeña tarjeta con los campos mail y contraseña para lograr al usuario. También contiene un botón registrar que nos lleva a la vista de registro (register.html).
- **register.html:** Esta vista permite a una persona poder registrarse como usuario e ingresar otra información de interés.
- **portfolio.html:** Esta vista es la página de inicio de todos los usuarios jugadores. Esta muestra sus acciones y su dinero.
- **adminHome.html:** Esta vista únicamente tiene el propósito de ser la página de inicio de los usuarios administradores.
- **allTransactions.html:** Esta vista muestra todas las transacciones
- **edit.html:** Esta vista cambia el perfil del usuario y el player o el admin
- **generateInvitationCodes.html:** Esta vista genera invitation codes
- **home.html:** Esta vista es la página por defecto
- **login.html:** Esta vista realiza el inicio de sesión del player o el admin
- **newPrice.html:** Esta vista permite definir un nuevo precio de un stock
- **newTransaction.html:** Esta vista permite realizar transacciones, éstas pueden ser de tipo Buy o Sell
- **registerNews.html:** Esta vista permite registrar noticias sobre un stock
- **transactionHistory.html:** Esta vista permite ver transacciones filtradas por fecha y/o stock
- **viewStockPool.html:** Esta vista muestra todas las acciones que tiene el player o el admin
- **searchStock.html:** Esta vista busca los stocks por nombre y/o descripción y muestra una gráfica con el avance y la recomendación.

La carpeta scripts contiene todos los .js de la aplicación web, esta carpeta se divide en dos subcarpetas, las cuales son app y lib. La carpeta lib es la que contiene todos los plugins angular.js. La carpeta app se divide en tres subcarpetas y un archivo .js suelto llamado

application.js, este archivo es el que da inicio a la aplicación angular por medio del ng-app. Las tres subcarpetas a las que hacíamos referencia son: controllers, services y validations.

Controllers: La carpeta controllers contiene los controladores de cada una de las vistas y se encargan de manejar el código y las variables de manera local a la vista.

Services: La carpeta services contiene los servicios que se van a utilizar a lo largo de toda la aplicación. El objetivo de estos servicios es tener un código que se pueda compartir de controlador en controlador sin necesidad de repetir código. Es una manera de optimizar y mantener orden. La mayoría de los servicios que creamos se encargan de interactuar con la web.api. Solo hay un servicio de los que creamos que no interactúa con la web.api y su función principal es almacenar alguna variable global como el usuario que se logea y una bandera para saber si hay alguien logeado y de esta manera habilitar o no el menú.

Content: Esta carpeta contiene los estilos css de la aplicación. Para este proyecto bastó con utilizar el estilo por defecto del template descargado.

GlobalService.js Almacena el usuario logeado y una bandera para saber si hay alguien logeado.

El servicio contiene los métodos get y set correspondientes para estas variables.

UserService.js Se encarga de comunicarse con el controlador de los usuarios en la web api.

Diagrama de componentes

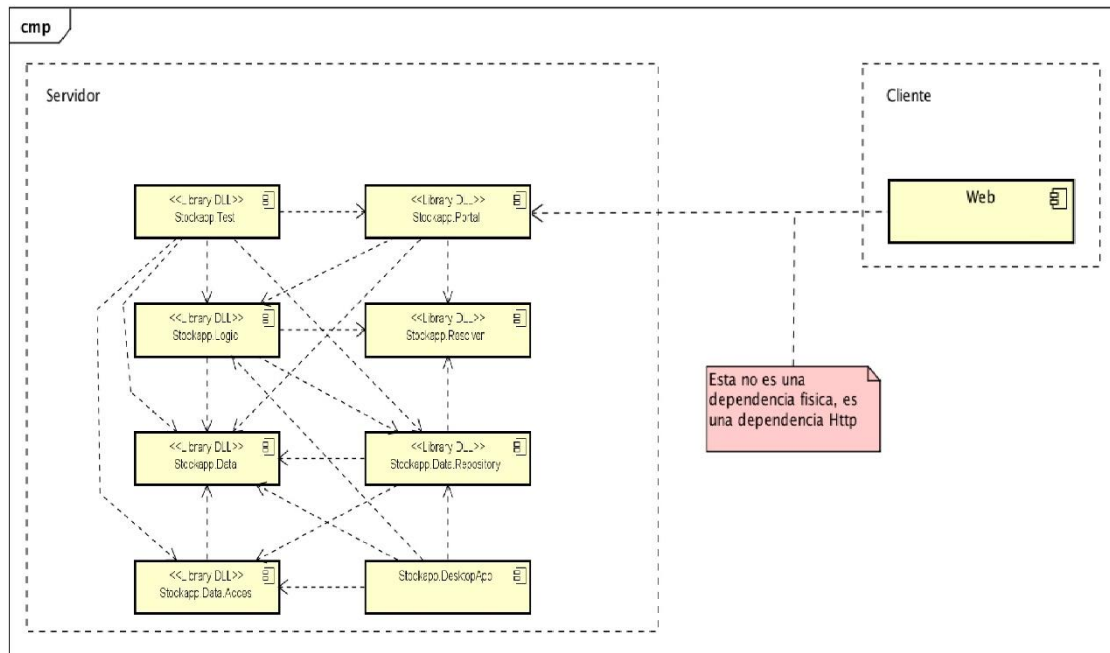
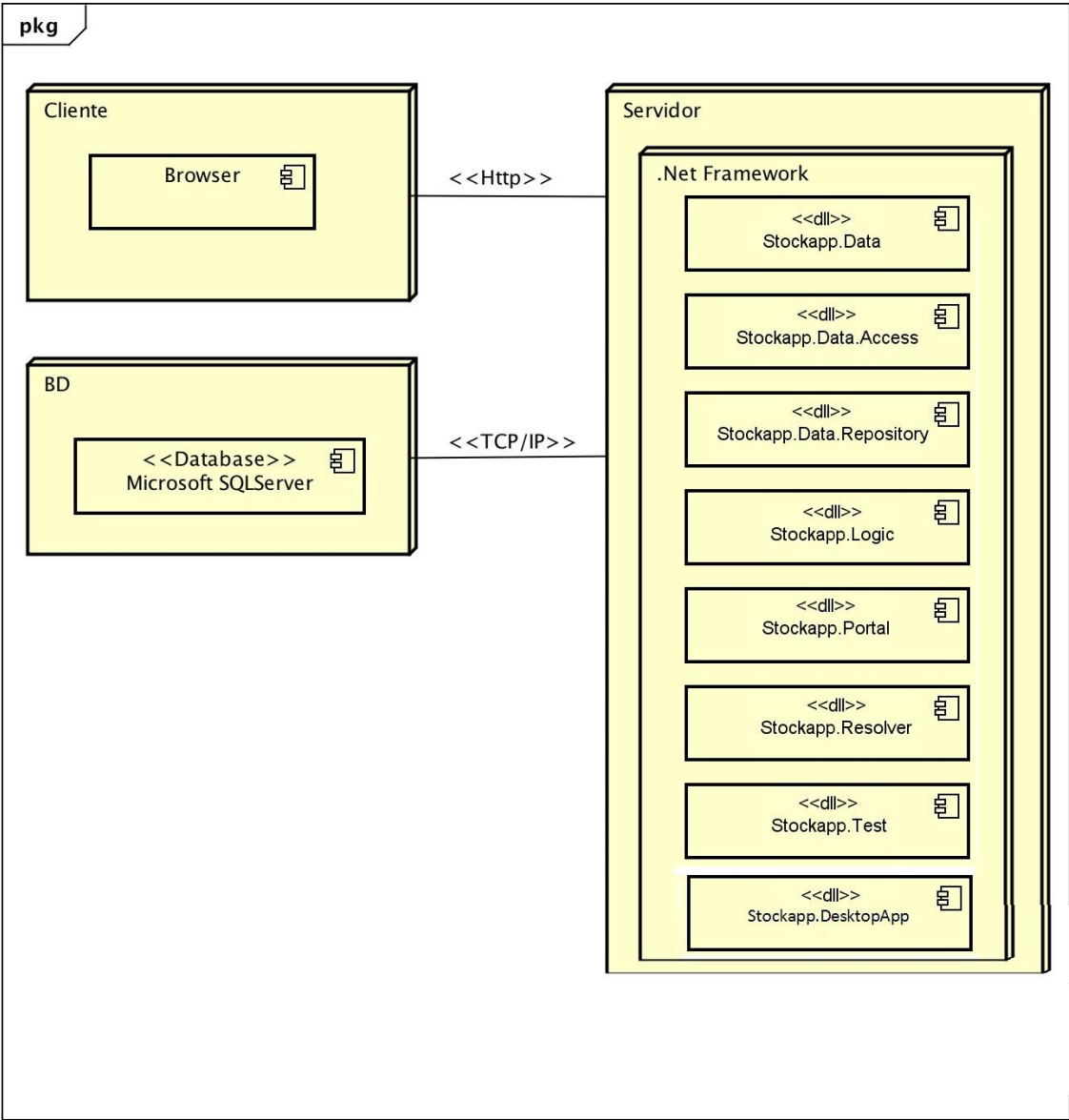
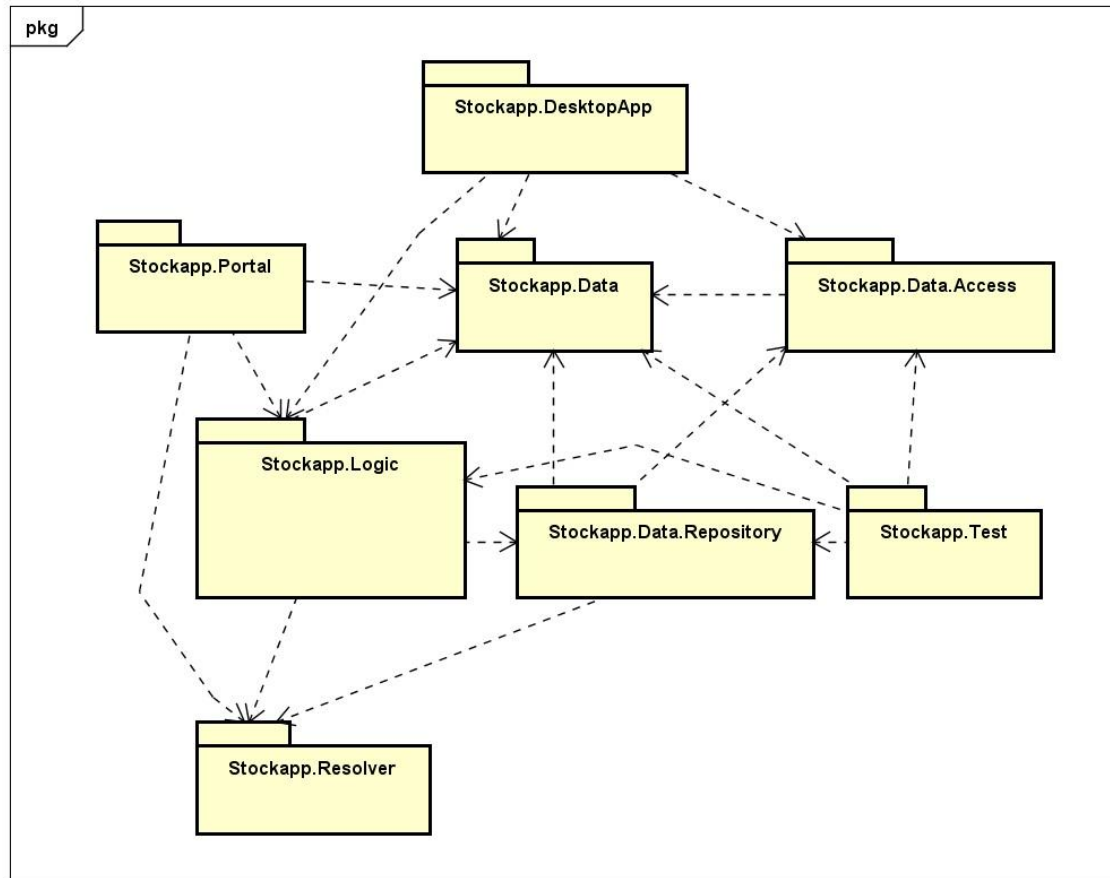


Diagrama de entrega

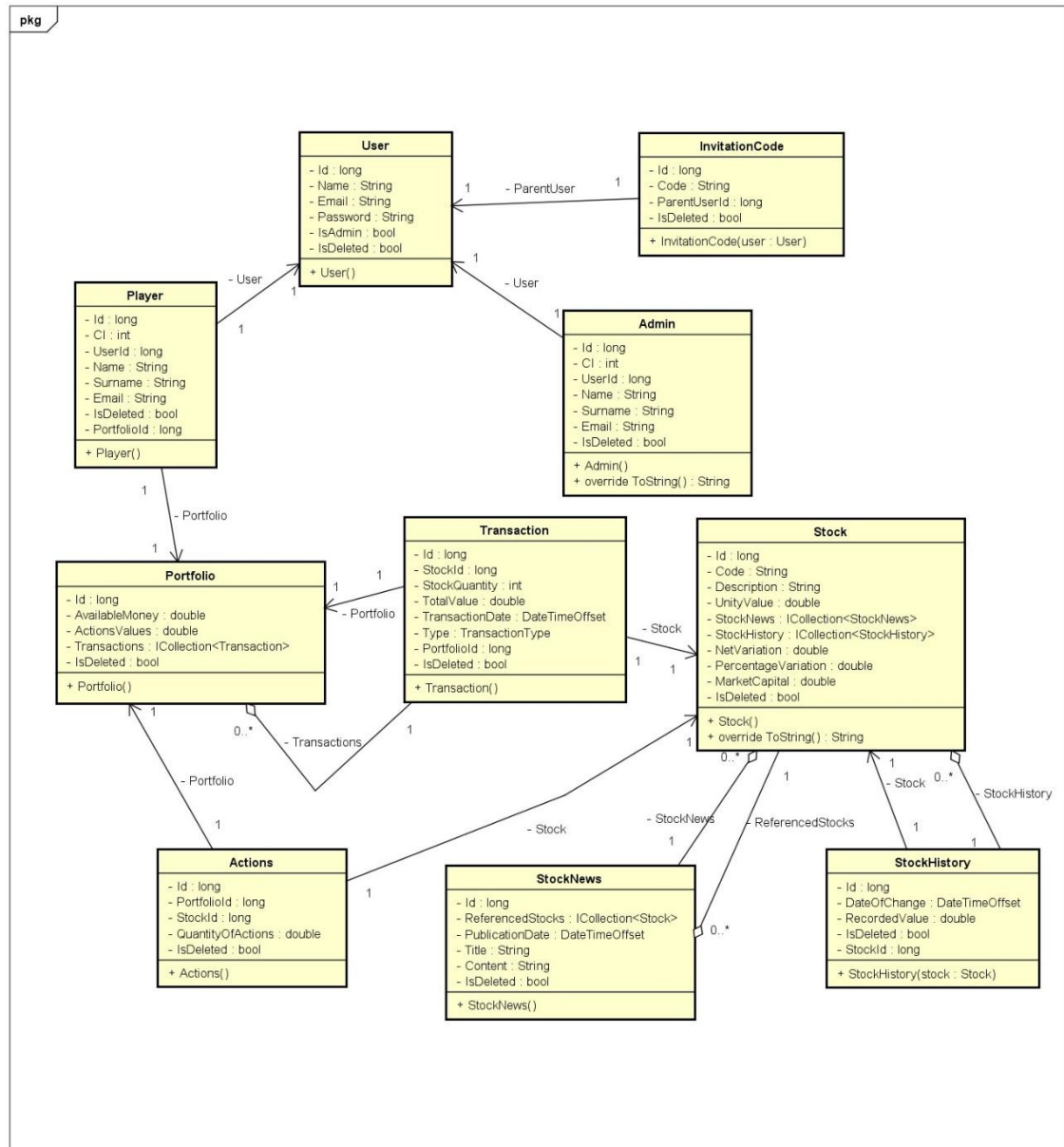


Diagramas de paquetes



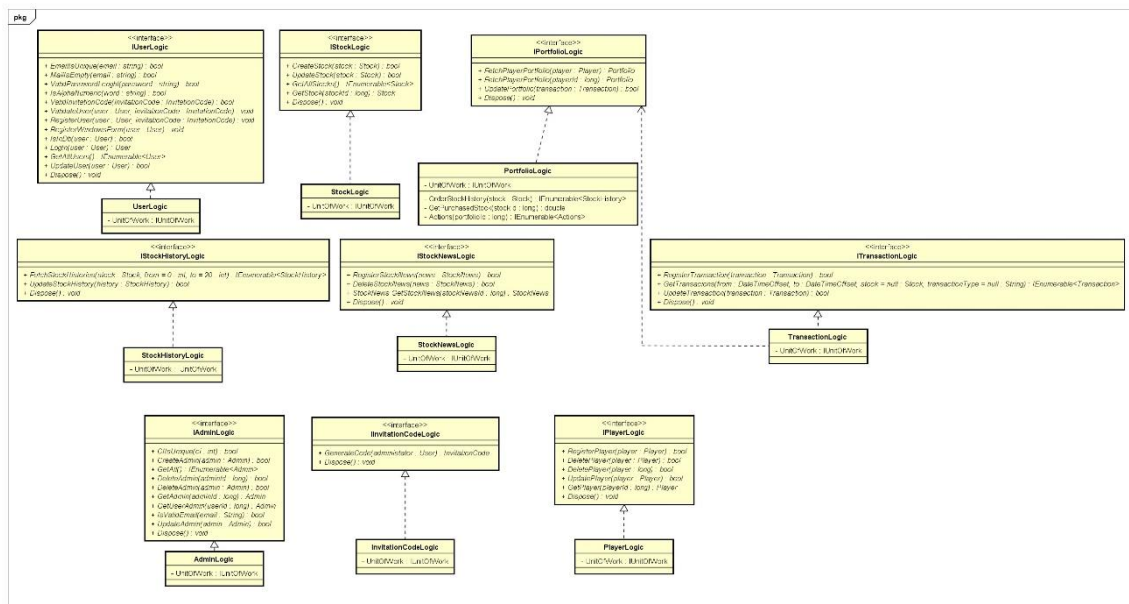
Diagramas de clases

Entities



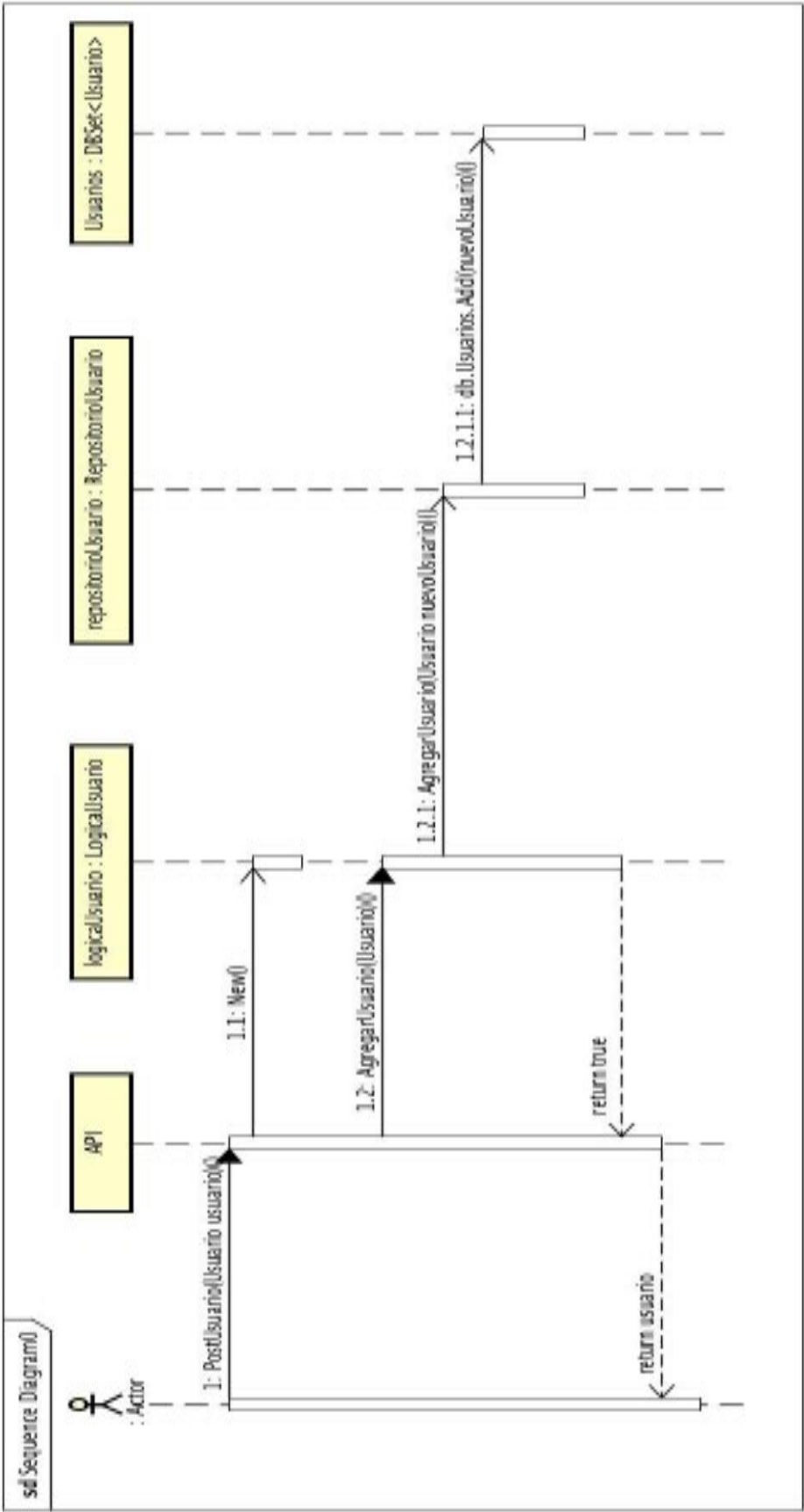
Todas las entidades implementan a las interfaces **ISoftDelete** e **Identificable**. La interfaz **ISoftDelete** se encarga de realizar un borrado lógico en la base de datos mientras que la interfaz **Identificable** se encarga de toda clase tenga un id.

Logic



Diagramas de interacción

User controller: Post



Los Controllers por ejemplo los de user, llaman a UserLogic donde esta implementa IUserLogic, se solicita en el controller el método que queremos llamar de la interfaz, acordémonos que este método está implementado en la lógica. Se controla que los datos lleguen bien y se devuelve lo solicitado. Cada controller tiene su ruta por ejemplo la ruta de Get de user es: "api/user" indicando que es un controller de tipo Get.

Informe de las métricas

Para el análisis de las métricas utilizamos la herramienta NDepend. Analizamos el proyecto Stockapp.Portal ya que es donde se encuentra implementada nuestra interfaz.

Como podemos ver en resumen de reglas violadas hay muchas cosas para mejorar. Esta herramienta es muy útil ya que no se pierde tiempo analizando el código uno mismo para que mejore sino que ésta lo analiza por sí misma y te sugiere mejoras según la calidad de tu código.

Por ejemplo algunos de los tantos aspectos que tenemos que mejorar son:

- Tenemos métodos con muchos parámetros
- Tenemos métodos muy complejos
- Tenemos tipos y métodos que al parecer no se usan

A continuación se muestran los resultados de la ejecución del NDepend:

Application Metrics

Note: Further Application Statistics are available.

Lines of Code

3 274

654 (NotMyCode)

Method Complexity

14 Max

1.7 Average

Types

106

8 Assemblies
23 Namespaces
533 Methods
115 Fields
118 Source Files

Code Coverage by Tests

N/A because no coverage data specified

Comment

24.67%

1 072 Lines of Comment

Third-Party Usage

17 Assemblies used
52 Namespaces used
208 Types used
247 Methods used
10 Fields used

Rules summary


97 42 8


This section lists all Rules violated, and Rules or Queries with Error


■ Number of Rules or Queries with Error (syntax error, exception thrown, time-out): 0

■ Number of Rules violated: 50

Summary of Rules violated

 Rules can be checked live at development time, from within Visual Studio. [Online documentation.](#)

 NDepend rules report too many flaws on existing code base? Use the option [Recent Violations Only!](#)

 Some Critical Rules are violated. Critical Rules can be used to break the build process if violated. [Online documentation.](#)

Display 25 records



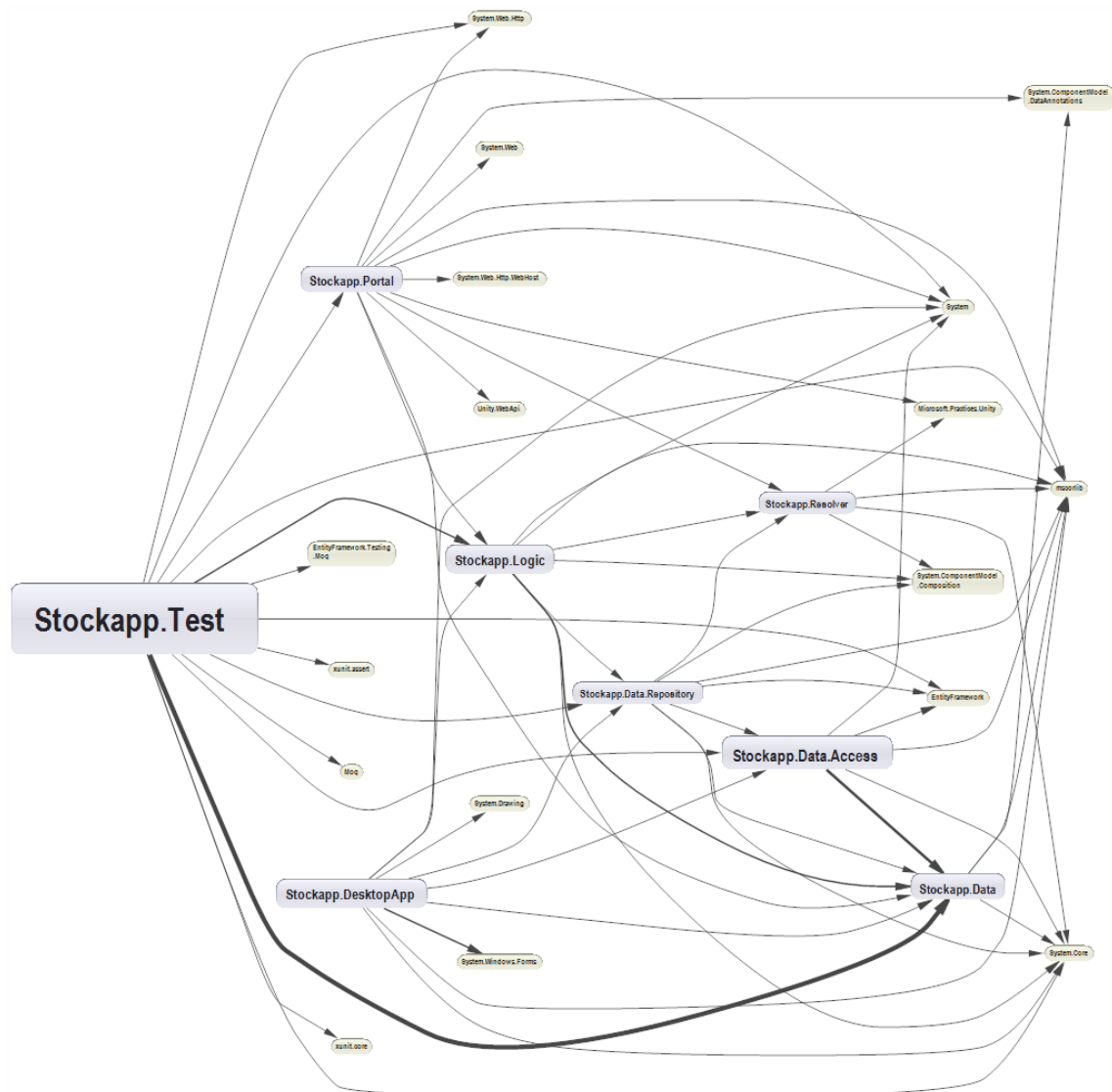
Name	# Matches	Elements	Group
 Types too big - critical	1	types	Project Rules \ Code Quality
 Methods too complex - critical	4	methods	Project Rules \ Code Quality
 Methods with too many parameters - critical	1	methods	Project Rules \ Code Quality
 Quick summary of methods to refactor	28	methods	Project Rules \ Code Quality
 Methods too big	2	methods	Project Rules \ Code Quality
 Methods too complex	9	methods	Project Rules \ Code Quality
 Methods potentially poorly commented	12	methods	Project Rules \ Code Quality
 Methods with too many parameters	5	methods	Project Rules \ Code Quality
 Types with too many methods	1	types	Project Rules \ Code Quality
 Types with too many fields	1	types	Project Rules \ Code Quality
 Types with poor cohesion	2	types	Project Rules \ Code Quality
 Class with no descendant should be sealed if possible	80	types	Project Rules \ Object Oriented Design
 A stateless class or structure might be turned into a static type	29	types	Project Rules \ Object Oriented Design
 Non-static classes should be instantiated or turned to static	39	types	Project Rules \ Object Oriented Design
 Methods should be declared static if possible	92	methods	Project Rules \ Object Oriented Design
 Constructor should not call a virtual method	3	methods	Project Rules \ Object Oriented Design
 Types with disposable instance fields must be disposable	12	types	Project Rules \ Design
 Classes that are candidate to be turned into structures	2	types	Project Rules \ Design
 Avoid namespaces with few types	12	namespaces	Project Rules \ Design
 Instances size shouldn't be too big	3	types	Project Rules \ Design
 Boxing/unboxing should be avoided	88	methods	Project Rules \ Design
 Avoid namespaces mutually dependent	1	namespaces	Project Rules \ Architecture and Layering
 Avoid namespaces dependency cycles	1	namespaces	Project Rules \ Architecture and Layering
 UI layer shouldn't use directly DAL layer	1	types	Project Rules \ Architecture and Layering
 Assemblies with poor cohesion (RelationalCohesion)	1	assemblies	Project Rules \ Architecture and Layering

Showing 1 to 25 of 50 entries

1 2

Name	# Matches	Elements	Group
 Potentially dead Types	1	types	Project Rules \ Dead Code
 Potentially dead Methods	1	methods	Project Rules \ Dead Code
 Methods that could have a lower visibility	237	methods	Project Rules \ Visibility
 Types that could have a lower visibility	44	types	Project Rules \ Visibility
 Fields that could have a lower visibility	3	fields	Project Rules \ Visibility
 Types that could be declared as private, nested in a parent type	1	types	Project Rules \ Visibility
 Avoid publicly visible constant fields	2	fields	Project Rules \ Visibility
 Fields should be declared as private	3	fields	Project Rules \ Visibility
 Avoid public methods not publicly visible	1	methods	Project Rules \ Visibility
 Fields should be marked as ReadOnly when possible	2	fields	Project Rules \ Immutability
 Property Getters should be immutable	10	methods	Project Rules \ Immutability
 Instance fields should be prefixed with a "m_"	115	fields	Project Rules \ Naming Conventions
 Methods name should begin with an Upper character	20	methods	Project Rules \ Naming Conventions
 Avoid methods with name too long	15	methods	Project Rules \ Naming Conventions
 Avoid having different types with same name	1	types	Project Rules \ Naming Conventions
 Avoid prefixing type name with parent namespace name	1	types	Project Rules \ Naming Conventions
 Avoid naming types and namespaces with the same identifier	2	types	Project Rules \ Naming Conventions
 Don't call your method Dispose	9	methods	Project Rules \ Naming Conventions
 Avoid defining multiple types in a source file	4	types	Project Rules \ Source Files Organization
 Types with source files stored in the same directory, should be declared in the same namespace	3	namespaces	Project Rules \ Source Files Organization
 Types declared in the same namespace, should have their source files stored in the same directory	2	namespaces	Project Rules \ Source Files Organization
 Mark assemblies with CLSCompliant	8	assemblies	Project Rules \ .NET Framework Usage \ System
 Do not raise too general exception types	1	methods	Project Rules \ .NET Framework Usage \ System
 Collection properties should be read only	4	methods	Project Rules \ .NET Framework Usage \ System.Collection
 Float and Date Parsing must be culture aware	1	methods	Project Rules \ .NET Framework Usage \ System.Globalization

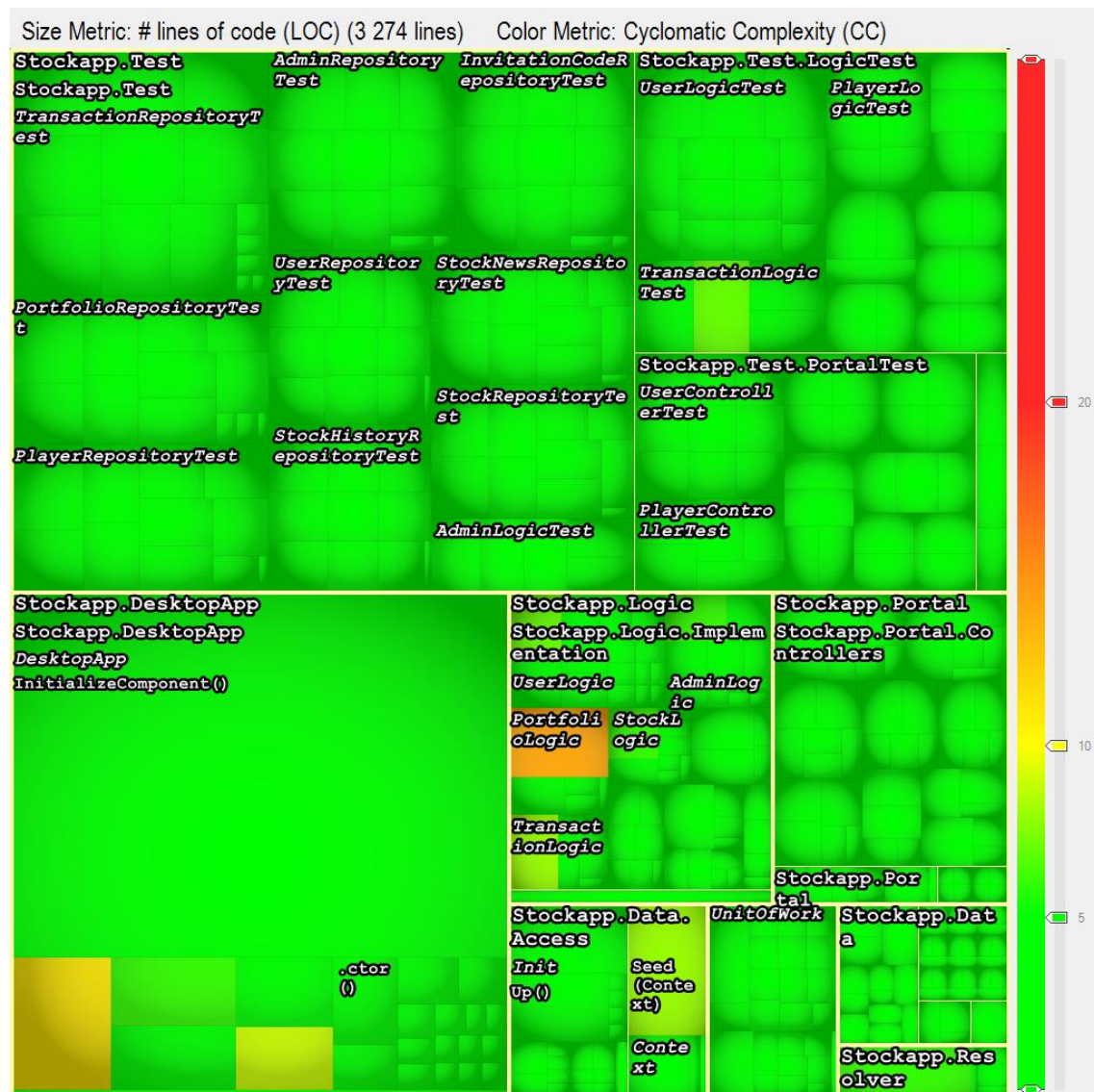
Dependency Graph



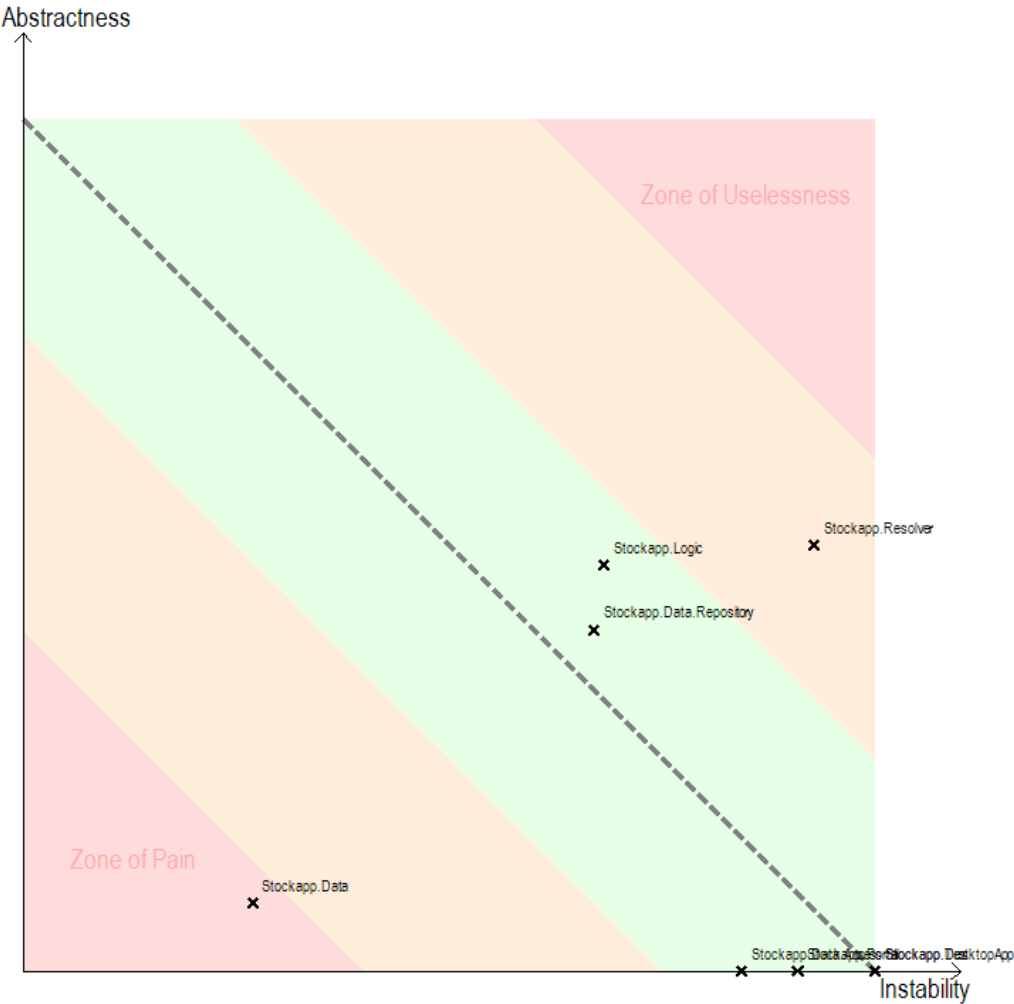
Dependency Matrix

		0	1	2	3	4	5	6	7
+ Stockapp.Test	0			8	18	20		10	28
+ Stockapp.DesktopApp	1				1	1		1	1
+ Stockapp.Portal	2	9			9		1		10
+ Stockapp.Logic	3	20	8	10		11	1		20
+ Stockapp.Data.Repository	4	3	2		3		1	2	3
+ Stockapp.Resolver	5			1	2	2			
+ Stockapp.Data.Access	6	1	1			2			3
+ Stockapp.Data	7	15	5	13	15	13		13	
+ mscorlib	8	24	25	22	27	27	20	32	18
+ EntityFramework	9	2				9		15	
+ System.ComponentModel.DataAnnotations	10			1					5
+ System.Core	11	12	1		10	11	5	9	1
+ System	12	1	5	1	2			1	
+ System.Web.Http	13	6		24					
+ Microsoft.Practices.Unity	14			5			5		
+ System.ComponentModel.Composition	15				1	1	9		
+ System.Web.Http.WebHost	16			1					
+ Unity.WebApi	17			1					
+ System.Web	18			1					
+ System.Windows.Forms	19		16						
+ System.Drawing	20		6						
+ xunit.core	21	3							
+ Moq	22	10							
+ xunit.assert	23	1							
+ EntityFramework.Testing.Moq	24	1							

Treemap Metric View



Abstractness vs. Instability



Descripción de los endpoints del api

Se intentó aplicar para la mayoría los CRUD para la mayoría de los controles de las entidades.

Para obtener los elementos utilizamos GET. Utilizamos dos tipos de GET, el que no recibe ningún valor es para obtener todos los datos, el que recibe un id es para obtener el objeto deseado.

Para crear usamos los POST, que son utilizados para enviar datos, que son creados en el web api, para luego ser insertados en la base de datos.

Para actualizar utilizamos los PUT, es necesario enviar el id y el objeto actualizado.

Para borrar utilizamos los DELETE, es necesario enviar el id. Aclaremos que consiste de un borrado lógico esto implica que quedará marcado como borrado en la base de datos pero seguirá en la misma.

Todos los Controllers devuelven un `IHttpActionResult`.

Al llamar a cada controller se debe especificar el tipo de este (GET, POST, PUT o DELETE).

En todos los Controllers si hay algún error en el modelo se devuelve `BadRequest(ModelState)`.

Si no se encuentra lo solicitado por el Controller entonces devuelve `NotFound()`.

En otro caso devuelve lo solicitado.

Todas las clases Controllers tienen un controller llamado `Dispose(bool disposing)` que lo que hace es desechar un controller cualquiera.

AdminController

Nombre	Descripción
<code>Get(long userId)</code>	Obtiene un administrador de la base de datos a partir del id de un usuario. La ruta para llamar a este controller es: <code>"api/admin/{userId:long}"</code> donde debemos reemplazar <code>{userId:long}</code> por el id del usuario a ingresar. Este controller nos devuelve un administrador.
<code>PutAdmin(Admin admin)</code>	Actualiza la información del administrador si la nueva información es correcta. En este caso este PUT no es necesario conocer el id del administrador a modificar ya que el mismo estará con la sesión iniciada por lo tanto ya tenemos el id. La ruta para llamar a este controller es: <code>"api/user/"</code> .
<code>DeleteAdmin(long id)</code>	Borra el administrador de la base de datos si encuentra al mismo en la base. La ruta para llamar a este controller es: <code>"api/user/{id:long}"</code> donde debemos reemplazar <code>{id:long}</code> por el id del admin a borrar.

GameSettingsController

Nombre	Descripción
<code>PutGameSettings(long id, GameSettings settings)</code>	Actualiza la información de los game settings, primero busca los game settings solicitados en la base por id y luego si son válidos los nuevos datos de los game settings los modifica. La ruta para llamar a este controller es: <code>"api/gamesettings/{id:long}"</code> donde debemos reemplazar <code>{id:long}</code> por el id del game settings a modificar.
<code>GetGameSettings()</code>	Obtiene los game settings de la base de datos. La ruta para llamar a este controller es: <code>"api/gamesettings/"</code> .

InvitationCodeController

Nombre	Descripción
PostInvitationCode(User user)	Crea un nuevo código de invitación y lo guarda en la base de datos. Esto lo realiza siempre y cuando el usuario sea de tipo administrador. La ruta para llamar a este controller es: " api/invitationcode ".

PlayerController

Nombre	Descripción
Get(long userId)	Obtiene un usuario de la base de datos a partir del id del mismo. La ruta para llamar a este controller es: " api/player/{userId:long} " donde debemos reemplazar {userId:int} por el id del usuario a ingresar. Este controller nos devuelve un player.
PutPlayer(Player player)	Actualiza la información del player si la nueva información es correcta. En este caso este PUT no es necesario conocer el id del player a modificar ya que el mismo estará con la sesión iniciada por lo tanto ya tenemos el id. La ruta para llamar a este controller es: " api/player/ ".
PostPlayer(Player newPlayer)	Crea un nuevo player y lo guarda en la base de datos. Esto lo realiza siempre y cuando los campos sean correctos y no haya otro player con el mismo email. La ruta para llamar a este controller es: " api/player/ ".
DeletePlayer(long id)	Borra el player de la base de datos si encuentra al mismo en base. La ruta para llamar a este controller es: " api/user/{id:long} " donde debemos reemplazar {id:long} por el id del player a borrar.

StockController

Nombre	Descripción
Get(long stockId)	Obtiene un stock de la base de datos a partir del id del mismo. La ruta para llamar a este controller es: " api/stock/{stockId:long} " donde debemos reemplazar {stockId:long} por el id del stock a ingresar. Este controller nos devuelve un stock.
GetAll()	Obtiene todos los stocks de la base de datos. La ruta para llamar a este controller es: " api/stock/ "
GetFilterStocks(string name = "", string description = "")	Obtiene los stocks filtrados por nombre y/o descripción. La ruta para llamar a este controller es: " api/stock/getfilterstocks/{name?}/{description?} " donde debemos reemplazar {name?} y/o {description} por el nombre y/o descripción que queremos filtrar.
PutStock(long id, Stock stock)	Actualiza la información del stock, primero busca el stock solicitado en la base por id y luego si son válidos los nuevos datos del stock los modifica. La ruta para llamar a este controller es: " api/stock/{id:long} " donde debemos reemplazar {id:long} por el id del stock a modificar.
PostStock(Stock stock)	Crea un nuevo stock y lo guarda en la base de datos. Esto lo realiza siempre y cuando los campos sean correctos y no haya otro stock con el mismo código. La ruta para llamar a este controller es: " api/stock/ ".

StockHistoryController

Nombre	Descripción
Get(Stock stock, int from = 0, int to = 20)	Obtiene todo el historial de un stock a partir del id y filtrado por fechas de la base de datos. La ruta para llamar a este controller es: " api/stockhistory/{from:int}/{to:int} " donde debemos reemplazar {from:int} y {to:int} por las fechas que querramos filtrar.
PutStockHistory(long id, StockHistory stockHistory)	Actualiza la información de un stockHistory, primero busca el stockHistory solicitado en la base por id y luego si son válidos los nuevos datos del stock los modifica. La ruta para llamar a este controller es: " api/stockhistory/{id:long} " donde debemos reemplazar {id:long} por el id del stockHistory a modificar.

StockNewsController

Nombre	Descripción
Get(long stockId)	Obtiene las noticias de un stock de la base de datos a partir del id del stock. La ruta para llamar a este controller es: " api/stocknews/{stockId:long} " donde debemos reemplazar {stockId:long} por el id del stock a ingresar.
PostStockNews(StockNews stockNews)	Crea una nueva stockNews y la guarda en la base de datos. Esto lo realiza siempre y cuando los campos sean correctos. La ruta para llamar a este controller es: " api/stocknews/ ".
DeleteStockNews(StockNews stockNews)	Borra la stockNews de la base de datos si encuentra al mismo en base. La ruta para llamar a este controller es: " api/stocknews/ ".

TransactionController

Nombre	Descripción
Get(DateTimeOffset from, DateTimeOffset to, long stockId = null, string transactionType = null)	Obtiene las transacciones de un stock a partir del id de la base de datos, filtrado por fechas y tipo de stock. La ruta para llamar a este controller es: " api/transaction/{from:datetime}/{to:datetime}/{transactionType:alpha?}/{stockId:long?} " donde debemos reemplazar {from:int} y {to:int} por las fechas que querramos filtrar, {transactionType:alpha} por el tipo de la transacción que queremos filtrar estas pueden ser Buy o Sell y finalmente {stockId:long} por el id del stock.
PutTransaction(long id, Transaction transaction)	Actualiza la información de una transaction, primero busca la transaction solicitado en la base por id y luego si son válidos los nuevos datos del stock los modifica. La ruta para llamar a este controller es: " api/transaction/{id:long} " donde debemos reemplazar {id:long} por el id de la transaction a modificar.
PostTransaction(Transaction transaction)	Crea una nueva transaction y la guarda en la base de datos. Esto lo realiza siempre y cuando los campos sean correctos. La ruta para llamar a este controller es: " api/transaction/ ".

UserController

Nombre	Descripción
Get()	Obtiene todos los usuarios de la base de datos. La ruta para llamar a este controller es: "api/user/"
PutUser(long id, User user)	Actualiza la información de un usuario, primero busca el usuario solicitado en la base por id y luego si son válidos los nuevos datos del stock los modifica. La ruta para llamar a este controller es: "api/user/{id:long}" donde debemos reemplazar {id:long} por el id del usuario a modificar.
PostUser(RegisterUserDTO newUser)	Crea un nuevo usuario y la guarda en la base de datos. Esto lo realiza siempre y cuando los campos sean correctos. La ruta para llamar a este controller es: "api/user/" .
DeleteUser(long id)	Borra el usuario de la base de datos si encuentra al mismo en base. La ruta para llamar a este controller es: "api/user/" .
SignIn(string email, string password)	Inicia sesión al usuario por email y contraseña, verifica que este esté en la base de datos y que su email y contraseña sean correctos. La ruta para llamar a este controller es: "api/user/signin/{email}/{password}/" .

Modelo de tablas de estructura de la base de datos



Evidencia de 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 del libro 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 de código sucio. Robert C. Martin, alias Uncle Bob, escribo 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 para demostrar que se cumplen las reglas de clean code.

Nombres significativos

Esta pauta hace referencia a la selección de un buen nombre para nuestras variables, métodos, clases, paquetes y todo lo nombraste en un proyecto.

Para nosotros esta es una de las pautas más importantes, ya que si un nombre puede llegar a explicar perfectamente para que va a ser usado, no se necesita tiempo pensando y razonando para que se creó y que se hace con esa variable, método o otro.

Por ejemplo, en las siguientes firmas, con solo leer el nombre del método se puede intuir que hace:

```
public void RegisterUser(User user, InvitationCode invitationCode);  
public bool ValidPasswordLenght(string password);
```

Funciones

Según estas pautas, las funciones deberían hacer una única cosa, se pequeños, estar bien indentados y tener la menor cantidad de argumentos posibles.

En el siguiente método podemos observar un método corto, con un solo argumento, el cual es imposible de sacar y hace una sola cosa:

```
public bool MailIsEmpty(string email)
{
    return (email == string.Empty) ? true : false;
}
```

Comentarios

Los comentarios en el código, según Robert Martin, son un mal necesario. Si nuestro código fuese perfecto no necesitaríamos comentarios. Pero es difícil llegar a la perfección, por eso Robert Martin especifica cuando el código es necesario y cuando es mejor mejorar el código.

```
/// <summary>
/// User email. Used for registration
/// </summary>
[Required]
[RegularExpression(@"^([\w\.\-]+)@([\w\-]+)(\.(\w){2,3})+$")]
public string Email { get; set; }
```

Formato

Con formato se refiere a como 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:

```
5 namespace Stockapp.Data
6 {
7     78 references | Juan Bautista Heber, 3 days ago | 3 authors, 3 changes
8     public class InvitationCode : ISoftDelete, Identifiable
9     {
10         /// <summary>
11         /// Database generated Id
12         /// </summary>
13         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
14         268 references | 0/36 passing | Juan Bautista Heber, 3 days ago | 3 authors, 3 changes
15         public long Id { get; set; }
16
17         /// <summary>
18         /// Generated Code
19         /// </summary>
20         31 references | 0/6 passing | Juan Bautista Heber, 42 days ago | 2 authors, 2 changes
21         public string Code { get; set; }
22
23         /// <summary>
24         /// Reference to that created the code
25         /// </summary>
26         5 references | Juan Bautista Heber, 3 days ago | 3 authors, 3 changes
27         public long ParentUserId { get; set; }
28
29         /// <summary>
30         /// User that created the code
31         /// </summary>
32         18 references | 0/4 passing | Juan Bautista Heber, 42 days ago | 2 authors, 2 changes
33         public virtual User ParentUser { get; set; }
34
35         /// <summary>
36         /// Soft delete
37         /// </summary>
38         272 references | 0/38 passing | Juan Bautista Heber, 42 days ago | 2 authors, 2 changes
39         public bool IsDeleted { get; set; }
40
41         20 references | 0/6 passing | Juan Bautista Heber, 42 days ago | 1 author, 1 change
42         public InvitationCode()
43         {
44             IsDeleted = false;
45         }
46     }
47 }
```

Objetos y estructura de datos

En este capítulo cuenta de las pautas que debe seguir un objeto. Él habla de la visibilidad de métodos y de atributos de clases y la ley de Demeter (no hablar con extraños) entre otras cosas.

En el siguiente código vemos como un método de `UserRepositoryTest` fue hecho `private` para que no pueda ser llamado desde otras clases:

```
private List<User> GetUserList()
{
    return new List<User>
    {
        new User()
        {
            Name = "jbheber",
            Password = "Jb.12345",
            Email = "juanbheber@outlook.com",
            IsAdmin = false,
            IsDeleted = false,
            Id = 1
        },
        new User()
        {
            Name = "fartolaa",
            Password = "Art.12345",
            Email = "artolaa@outlook.com",
            IsAdmin = false,
            IsDeleted = false,
            Id = 2
        },
        new User()
        {
            Name = "jheber",
            Password = "Jh.1234554",
            Email = "juanbautistaheber@gmail.com",
            IsAdmin = true,
            IsDeleted = false,
            Id = 3
        },
        new User()
        {
            Name = "arto",
            Password = "Artoo.1234554",
            Email = "arto@gmail.com",
            IsAdmin = true,
            IsDeleted = true,
            Id = 4
        },
        new User()
        {
            Name = "maca",
            Password = "Maluso.1234554",
            Email = "macaluso@gmail.com",
            IsAdmin = false,
            IsDeleted = false,
            Id = 5
        }
    };
}
```


Manejar errores

Este capítulo dice que todo proyecto necesita manejar los errores, pero no dificultar la lectura del código por tener muchos controles de error.

Nosotros para eso creamos una clase de excepción para cada entidad a continuación se muestran las excepciones de usuario:

```
1  using System;
2
3  namespace Stockapp.Data.Exceptions
4  {
5      [SerializableAttribute]
6      public class UserException : Exception
7      {
8          public UserException()
9              : base()
10         {
11         }
12
13         public UserException(String message)
14             : base(message)
15         {
16         }
17
18         public UserException(String message, Exception innerException)
19             : base(message, innerException)
20         {
21         }
22     }
23 }
24
```

Pruebas unitarias

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

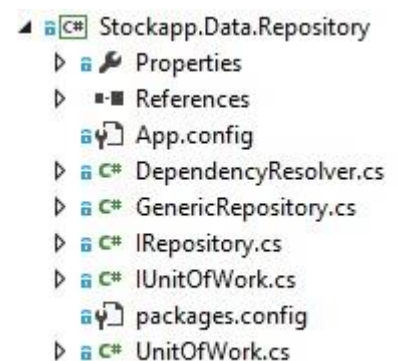
1. No escribirás código de producción hasta haber escrito una prueba fallida.
2. No escribirás más sobre una prueba unitaria que lo suficiente para que falle.
3. No escribirás más código de producción que el necesario para que pase la prueba.

La evidencia de que cumplimos con estas leyes se puede ver en los commits realizados en bitbucket.

Clases

Las clases deberían ser chicas 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*. 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 Stockapp.Data.Repository, 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.



Cobertura de las pruebas

Pruebas: Superada (200)

✓ Stockapp.Test.AdminLogicTest.CreateAdminTest	39 ms
✓ Stockapp.Test.AdminLogicTest.DeleteAdminByIdTest	1 s
✓ Stockapp.Test.AdminLogicTest.DeleteAdminTest	11 ms
✓ Stockapp.Test.AdminLogicTest.GetAdminTest	13 ms
✓ Stockapp.Test.AdminLogicTest.UpdateAdminTest	19 ms
✓ Stockapp.Test.AdminRepositoryTest.DeleteAdminByIdTest(index: 0)	122 ms
✓ Stockapp.Test.AdminRepositoryTest.DeleteAdminByIdTest(index: 1)	5 s
✓ Stockapp.Test.AdminRepositoryTest.DeleteAdminTest(index: 0)	101 ms
✓ Stockapp.Test.AdminRepositoryTest.DeleteAdminTest(index: 1)	96 ms
✓ Stockapp.Test.AdminRepositoryTest.GetAllAdminTest	90 ms
✓ Stockapp.Test.AdminRepositoryTest.GetByIdTest(index: 1)	52 ms
✓ Stockapp.Test.AdminRepositoryTest.GetByIdTest(index: 3)	2 s
✓ Stockapp.Test.AdminRepositoryTest.GetFilterAdminTest	74 ms
✓ Stockapp.Test.AdminRepositoryTest.GetNonDeletedAdminTest	76 ms
✓ Stockapp.Test.AdminRepositoryTest.InsertAdminTest	87 ms
✓ Stockapp.Test.AdminRepositoryTest.InsertSingleAdminTest	95 ms
✓ Stockapp.Test.AdminRepositoryTest.UpdateAdminTest(index: 0)	95 ms
✓ Stockapp.Test.AdminRepositoryTest.UpdateAdminTest(index: 1)	76 ms
✓ Stockapp.Test.InvitationCodeRepositoryTest.DeleteInvitationCodeByIdTest(index: 0)	80 ms
✓ Stockapp.Test.InvitationCodeRepositoryTest.DeleteInvitationCodeByIdTest(index: 1)	93 ms
✓ Stockapp.Test.InvitationCodeRepositoryTest.DeleteInvitationCodeTest(index: 0)	95 ms
✓ Stockapp.Test.InvitationCodeRepositoryTest.DeleteInvitationCodeTest(index: 1)	86 ms
✓ Stockapp.Test.InvitationCodeRepositoryTest.GetAllInvitationCodeTest	65 ms

fema_DESKTOP-TVSUMS0 2016-05-10 22_14				
Jerarquía	No cubiertos (bloques)	No cubiertos (% de bloques)	Cubiertos (bloques)	Cubiertos (% bloques)
▲ fema_DESKTOP-TVSUMS0 2016...	1859	19,82 %	7521	80,18 %
▲ stockapp.data.access.dll	1266	98,83 %	15	1,17 %
▸ Stockapp.Data.Access	728	97,98 %	15	2,02 %
▸ Stockapp.Data.Access.M...	538	100,00 %	0	0,00 %
▲ stockapp.data.dll	76	57,14 %	57	42,86 %
▸ Stockapp.Data	26	41,94 %	36	58,06 %
▸ Stockapp.Data.Exceptions	50	92,59 %	4	7,41 %
▸ Stockapp.Data.Extensions	0	0,00 %	17	100,00 %
▲ stockapp.data.repository.dll	15	12,82 %	102	87,18 %
▸ Stockapp.Data.Repository	15	12,82 %	102	87,18 %
▲ stockapp.logic.dll	158	34,13 %	305	65,87 %
▸ Stockapp.Logic	11	100,00 %	0	0,00 %
▸ Stockapp.Logic.Implem...	147	32,52 %	305	67,48 %
▲ stockapp.portal.dll	254	52,59 %	229	47,41 %
▸ Stockapp.Portal	35	100,00 %	0	0,00 %
▸ Stockapp.Portal.Controll...	219	49,32 %	225	50,68 %
▸ Stockapp.Portal.Models	0	0,00 %	4	100,00 %
▲ stockapp.test.dll	90	1,30 %	6813	98,70 %
▸ Stockapp.Test	2	0,04 %	4676	99,96 %
▸ Stockapp.Test.LogicTest	62	4,32 %	1374	95,68 %
▸ Stockapp.Test.PortalTest	26	3,30 %	763	96,70 %

Utilizando la herramienta de analizador de cobertura de todas las pruebas llegamos a que las pruebas cubren un **80,18 %** del código.

En conclusión tener más del 90 % del código probado nos da mucha confianza y seguridad de que las cosas anden como deberían andar. Esta seguridad también se transmite a confianza al realizar cambios, ya que si algo deja de andar nos enteraremos instantáneamente al ejecutar las pruebas. El programa se torna más mantenible y sostenible.