Gourmet Catalog

Proyecto - Análisis y Diseño de Sistemas

2022

Mejorar el diseño y arquitectura de la aplicación dada, siguiendo los principios de Clean Code y SOLID. Además extender su funcionalidad y realizar el testeo de la aplicación.

Autores: Marten, Juan

Gandolfo, Matias

Índice

Análisis del estado original del código

La aplicación presentada inicialmente sigue una arquitectura monolítica, está organizada de manera tal que un clase mantiene toda la lógica necesaria para el funcionamiento de la aplicación (MainWindow), una clase que implementa el funcionamiento de la base de datos (DataBase) y una clase que representa a los objetos resultados de realizar una búsqueda (SearchResult).

Análisis de MainWindow

Lo primero a observar es que la clase es considerablemente larga, además de que la mayor parte de la misma está contenida en un único método que actúa como método main, y mantiene funcionando a la aplicación, configurando la vista, actualizando sus valores, realizando las búsquedas, parseando los resultados de búsqueda todo esto de manera poco legible debido a la ausencia de Clean Code y SOLID, en otras palabras, gracias a la ausencia de nombres descriptivos y la unificación de diferentes funcionalidades intentar comprender lo que se está haciendo en ese método es bastante complicado, por lo menos a simple vista.

Análisis de DataBase

Con respecto a esta clase, podemos ver que tiene las funcionalidades bien definidas, es decir, cada método representa alguna función importante de la clase. El principal detalle observado en un principio es que hay varias secciones de código repetidas, métodos que no se usan y comentarios por todos que buscan explicar clase, estos mismos deberían ser removidos aplicando Clean Code.

Mejoras de diseño y refactoring

MVP

Sabemos que la estructura del modelo MVP consiste de tres componentes:

- Modelo
- Presentador
- Vista

Antes del comienzo del desarrollo de la solución, hay que destacar que los tres componentes del MVP fueron divididos en dos partes, para un mejor manejo y lograr un código aún más limpio. Las partes en las que se dividió están basadas en las funcionalidades provistas por la aplicación, por lo que existe una parte enfocada a los artículos almacenados localmente y por otro lado, una enfocada a la búsqueda de artículos en la web.

Modelo para la búsqueda de artículos en internet

Esta parte además de contener la clase que implementa al modelo en sí, contiene las clases que modelan la búsqueda y sus resultados, como SearchLogic, SearchResult y ResponseParser. Por lo que, cuando se requiera realizar una búsqueda, seleccionar resultados de internet o bien guardar algo de internet en nuestra base de datos local, esta parte del modelo será la encargada de llevar a cabo la lógica de esas acciones y luego avisar mediante listeners, la finalización de las tareas.

Modelo para el manejo de artículos localmente guardados

La parte restante del componente modelo, es la que corresponde a la lógica del manejo para los artículos guardados localmente. Esta parte se encargará de las tareas de actualizar cambios de un artículo guardado, eliminar artículos de la base de datos local, seleccionar un artículo y mostrarlo. Para ello se utilizaran dos clases, la que modelaría a la base de datos y la clase que corresponde a la implementación de la lógica del modelo, que llevará a cabo las acciones requeridas y avisará mediante listeners cuando finalice.

Presentador para la búsqueda de artículos en internet

Este presentador se encarga de activar los eventos que el usuario provoca en la aplicación, más precisamente, los eventos relacionados a la búsqueda de artículos en la web. Una vez activados los eventos, su modelo correspondiente realizará sus actividades y el presentador estará atento (mediante listeners) a cuando finalicen para poder actualizar la vista y mostrarle el resultado al usuario.

Con respecto a la jerarquía entre presentadores, se consideró que son equivalentes, ninguno tiene una importancia mayor al otro que promueva la necesidad de imponer uno sobre el otro. Pero sí cabe aclarar, que el presentador que se está explicando ahora conoce al modelo de los artículos locales, debido a que cuando se desea guardar un artículo buscado en internet, le debe avisar a dicho modelo de esto, ya que es el que tiene acceso a la base de datos.

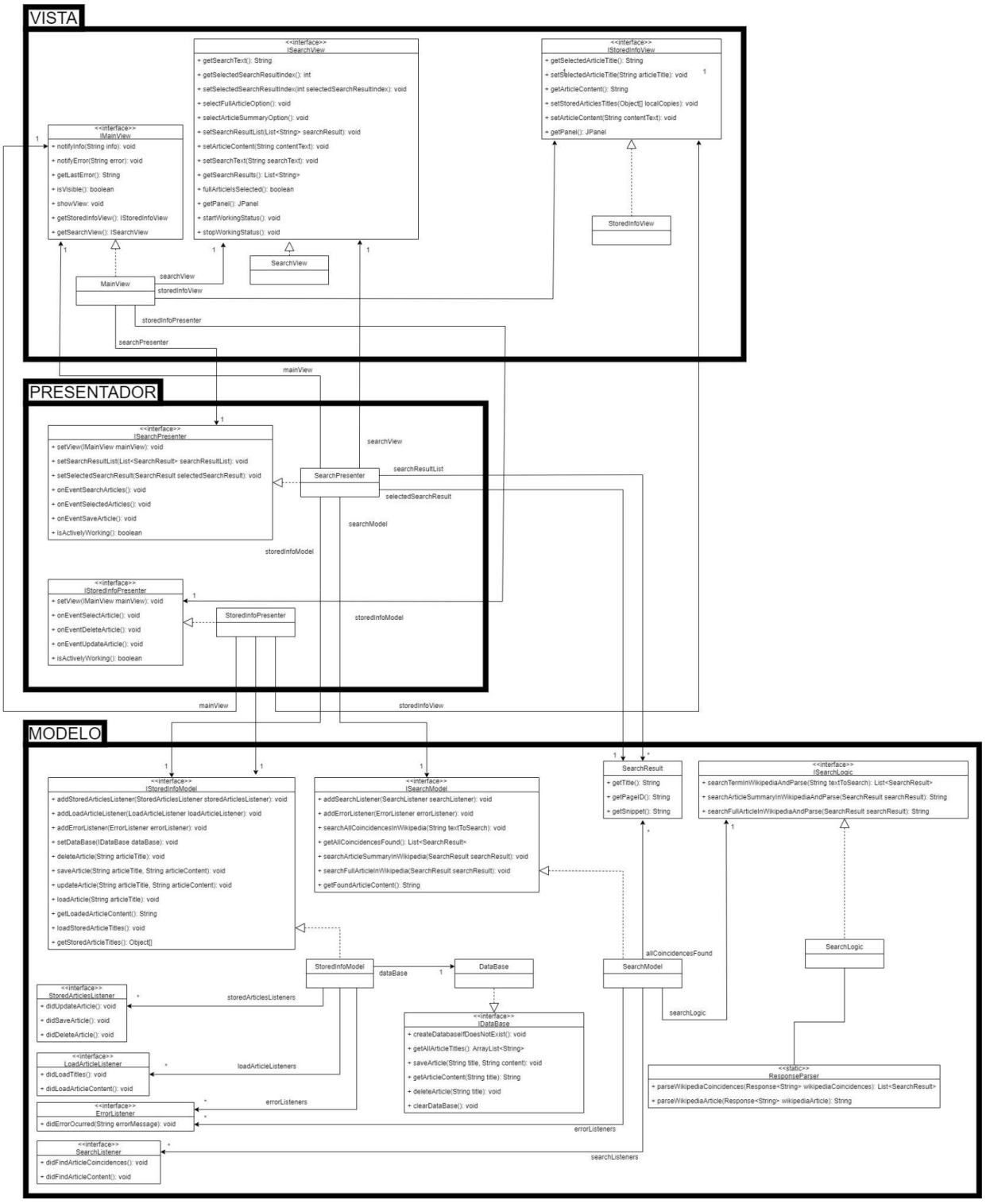
Presentador para el manejo de artículos localmente guardados

Este presentador se encargará de guardar efectivamente los artículos buscados, además de actualizar, mostrar y borrar los que ya están en la base de datos. La vista invocará a estos métodos característicos de este presentador, para que este último le comunique lo que debe hacer al modelo. El presentador estará escuchando al modelo con sus listener para saber qué ocurrió con la ejecución de esa acción, y mostrarle la información al usuario.

Vista

A pesar de que la misma está dividida en dos partes de igual manera que los componentes anteriores, no resulta necesario el desarrollo de ambas, ya que su única tarea es estar escuchando que sucede en los botones, detectar la interacción del usuario y enviar mensajes al presentador. No presentan ningún tipo de lógica adicional que sea considerable para el desarrollo aquí.

MVP UML



Se adjunta un archivo UML.jpg que contiene la imagen con una mejor resolución en caso de ser necesario revisar.

Clean Code

Respecto a lo que es Clean Code, se realizó trabajo sobre todos los aspectos posibles.

El primer foco a resolver fueron los nombres de las variables, ya que eran muy poco significativos y resultaba muy difícil comprender el funcionamiento del código sin poder entender qué función cumplía cada variable.

Luego de asignar nombres significativos a las variables se prosiguió agrupando el código en funciones, principalmente en la clase main que resolvía toda la lógica de la aplicación en un único método. Para eso se identificaron las funcionalidades, luego se dividieron en varios métodos más pequeños y con nombres descriptivos.

Habiendo llevado a cabo esa primera etapa de Clean Code, ya se puede ver un panorama general de que clases van a tener que tener que codigo, es decir, ya era momento de aplicar Single Responsibility, como dijimos, la clase principal realiza muchas tareas distintas.

Lo último que se realizó con esta clase fue la eliminación de los comentarios, una vez que se estuvo de acuerdo con todo el resto del código.

Con respecto a la clase que realiza el manejo de la base de datos, se eliminaron comentarios, se cambiaron los nombres de algunas variables y se crearon nuevos métodos para evitar código repetido.

La clase SearchResult tenía sus atributos públicos, lo cual no es muy buena práctica, por lo que se hicieron privados y se agregaron getters para los mismos.

SOLID

En un principio, se destacó el problema de Single Responsability de la clase main, para resolverlo, se crearon las siguientes clases:

- SearchLogic, su tarea es la realización de búsquedas en wikipedia.
- ResponseParser, encargada de parsear los resultados de búsqueda obtenidos de wikipedia.
- Modelo, es la clase que controla el estado de la aplicación, tiene acceso a la búsqueda y almacenamiento de artículos.
- Presentador, es la clase que le da a conocer al modelo lo que ocurre en la vista e inversamente, actualiza la vista con la información que le brinda el modelo.
- Vista, se encarga de la creación de los listener de los componentes gráficos, que comunicaran al presentador lo ocurrido en la interfaz de usuario.

Una vez que se llegó a este punto, se observó que las clases seguían teniendo un problema, eran considerablemente largas y si observabas bien estaba la posibilidad de separar la parte de la aplicación que maneja los artículos guardados localmente, con la parte que realiza las búsquedas en la web. Esto llevó a que se volviera a aplicar Single Responsibility Principle sobre los tres componentes del MVP, dando lugar a dos presentadores, modelos y vistas.

Además la clase SearchResult era una componente gráfica que a la vez representaba información de los resultados de búsqueda, por lo que se separó de la parte gráfica para que solo represente la información de los resultados de búsqueda.

En particular con la vista, se requirió crear una tercera vista contenedora de las otras dos vistas con las que interactúan los usuarios. Esta no realiza ninguna tarea en particular más allá de la de recibir dos componentes visuales, juntarlos y mostrarlos por pantalla.

No se consideró adecuado darle esa responsabilidad a ninguna de las vistas que surgieron por el principio aplicado, ya que tienen un nivel de responsabilidad equivalente por lo que sería poco intuitivo realizar estas tareas en alguna de ellas.

Otro principio que no se está cumpliendo por la clase main original, es el Open Closed Principle. Esto se debe a que cualquier modificación en la funcionalidad del programa, afectará a la única clase existente, no es posible extender la funcionalidad sin tener que modificar el código que ya estaba escrito.

El código escrito originalmente, deberá ser modificado sea cual sea el cambio que se quiera generar. Esto fue solucionado con la definición de interfaces y la inyección de dependencias, lo que permite por ejemplo cambiar o extender el módulo de búsqueda o de base de datos sin tener que modificar las clases existentes.

Interface Segregation Principle en la clase main, ya que tiene todos los métodos necesarios para la ejecución del programa, y no hace ningún tipo de separación, lo que provoca tener que recompilar y redeployar el programa entero.

Actualmente cualquier tipo de interacción pasa por la misma clase, lo que implica que cualquier modificación de estas interacciones hace que el resto del código quede inutilizable, y por lo tanto el programa también, hasta que se compile la nueva funcionalidad.

Notificación de acciones exitosas y fallidas

Para notificar al usuario de las acciones exitosas y fallidas se agregaron a la clase MainView los métodos notifyInfo y notifyError, los cuáles muestran información al usuario con un popup.

Respecto a las acciones exitosas, cuando un modelo finaliza la acción notifica al presentador mediante sus listeners, el cuál notifica a la vista con el método notifyInfo.

Y respecto a las acciones fallidas, los módulos de búsqueda y de base de datos lanzan excepciones en sus métodos cuando se produce algún error, dichas excepciones serán atrapadas por el correspondiente modelo, el cuál notifica al presentador mediante un listener de errores. Luego el presentador le informa a la vista del error usando el método notifyError indicando si el error fue en la base de datos o en la búsqueda.

Extensión de la funcionalidad de búsqueda

Para el desarrollo de este requerimiento se debió agregar una API de búsqueda, que obtenga la totalidad del contenido de la página web, para esto se agregó una nueva interfaz de API siguiendo la ayuda brindada en el comentario de la otra API. Una vez fue creada la interfaz y se la dio a conocer a SearchLogic, se debió implementar dos métodos en la clase SearchLogic, uno para buscar la página completa y otro solamente el primer extracto.

De la misma manera se crearon dos métodos en el modelo, uno para cada caso.

Es el presentador quien determina a cuál de los dos métodos se debe llamar, ya que tiene acceso a la vista y puede consultar el estado de los componentes, por lo que si el usuario selecciona una opción, el presentador puede consultarle a la vista cual es y luego determinar a qué método se debe llamar.

Para la implementación gráfica que permite al usuario seleccionar una opción, se utilizó la recomendada (JRadioButton). Por defecto estará seleccionada la opción de obtener el primer extracto, la misma que estaba por defecto en la aplicación original.

Testing unitario

El objetivo de estos tests, fue probar como unidad las clases presentadores. Para ello, utilizando mocks de las vistas y los modelos (usando el framework Mockito), se testeo que al activar un evento del presentador, se envíe el mensaje correcto al modelo.

Los casos de test unitario para el presentador de la búsqueda en la web son:

- testSearch: verifica que si escribimos algo para buscar y apretamos la tecla enter, la información escrita llegue correctamente al modelo para que realice su búsqueda.
- testSelectFullArticle: verifica que, luego de realizar una búsqueda del artículo completo, al elegir una de las opciones, el artículo seleccionado llegue correctamente al modelo.
- testSelectArticleSummary: mismo que el anterior, pero con la búsqueda del primer extracto.
- testSaveArticle: verifica si para un artículo que fue seleccionado y luego se desea guardar, este articulo llegue correctamente al modelo para guardarse.
- testSaveNullArticle: verifica que si se desea guardar un artículo, pero todavía no se seleccionó ninguno, no llegue ningún mensaje al modelo para guardarlo, sino que el presentador se detenga.

Los casos de test unitario para el presentador que maneja los artículos localmente guardados son:

- testSelectedArticle: verifica que si se selecciona un artículo guardado que se desea cargar en la vista, el presentador envíe el mensaje al modelo con el título correspondiente al artículo seleccionado.
- testDeleteArticle: verifica que si hay un artículo seleccionado cargado en la vista y se desea eliminar, se le envíe al modelo el mensaje de eliminar artículo con el título correspondiente al mismo.
- testUpdateArticle: verifica que si hay un artículo seleccionado cargado en la vista y se desea actualizar su contenido después de modificarlo, se envíe el mensaje correspondiente de actualización al modelo con el título y el nuevo contenido del mismo.

Testing de integración

Para el testing de integración se probaron las funcionalidades principales de la aplicación y los casos de error. Para ello se utilizaron mocks del módulo de búsqueda y del módulo de base de datos, el de la base de datos solo para forzar los casos de error mediante excepciones.

Los casos del test de integración son:

- testSearchTerm: verifica que al realizar la operación de búsqueda de un término se muestren los resultados de búsqueda esperados.
- testSearchFullArticle: verifica que al realizar una búsqueda de un término y elegir un resultado de búsqueda habiendo seleccionado la opción de artículo completo se muestre el artículo esperado.
- testSearchArticleSummary: verifica que al realizar una búsqueda de un término y elegir un resultado de búsqueda habiendo seleccionado la opción de resumen del artículo se muestre el artículo esperado.
- testSaveArticle: verifica que luego de buscar un artículo y seleccionar la opción de guardar localmente el artículo se guarde en la base de datos.
- testSaveArticleWithoutSelecting: verifica que al seleccionar la opción de guardar localmente sin haber buscado un artículo se muestre un error.
- testEmptySearch: verifica que al buscar un término sin ingresar nada en el campo de búsqueda se muestre un error.
- testSearchErrorWhenSearchingTerm: verifica que en caso de producirse un error en el módulo de búsqueda mientras se ejecuta la búsqueda de un término se muestre un error.
- testSearchErrorWhenSearchingFullArticle: verifica que en caso de producirse un error en el módulo de búsqueda mientras se ejecuta la búsqueda de un artículo completo se muestre un error.
- testSearchErrorWhenSearchingArticleSummary: verifica que en caso de producirse un error en el módulo de búsqueda mientras se ejecuta la búsqueda de un resumen de artículo se muestre un error.
- testSelectStoredArticle: verifica que al seleccionar un artículo guardado en la base de datos se muestre su contenido.
- testDeleteArticle: verifica que al seleccionar un artículo guardado y elegir la opción de eliminar ese artículo sea eliminado de la base de datos.
- testUpdateArticle: verifica que al seleccionar un artículo guardado, modificarlo y elegir la opción de actualizar, el contenido del artículo en la base de datos sea actualizado.
- testDeleteArticleWithoutSelecting: verifica que al seleccionar la opción de eliminar sin haber seleccionado un artículo se muestre un error.
- testUpdateArticleWithoutSelecting: verifica que al seleccionar la opción de actualizar sin haber seleccionado un artículo se muestre un error.
- testDatabaseErrorWhenSelectingArticle: verifica que en caso de producirse un error en el módulo de base de datos mientras se busca un artículo seleccionado se muestre un error.
- testDatabaseErrorWhenDeletingArticle: verifica que en caso de producirse un error en el módulo de base de datos mientras se elimina un artículo seleccionado se muestre un error.
- testDatabaseErrorWhenUpdatingArticle: verifica que en caso de producirse un error en el módulo de base de datos mientras se actualiza un artículo seleccionado se muestre un error.

Consideraciones

- Durante el desarrollo del proyecto nos encontramos con un error en los hilos de Java al intentar desactivar el estado ‘working status’ (el cuál deshabilita las componentes gráficas mientras se realiza una búsqueda). Debido a que el error se originaba en una de las clases de hilos de Java y no en el código fuente del proyecto no pudimos encontrar el motivo del error para resolverlo, por lo que se decidió quitar el estado ‘working status’ del funcionamiento de la aplicación en favor de mantener el uso de hilos para las acciones asincrónicas y evitar errores. Se adjunta el stacktrace completo del error en el .zip del proyecto.
- El método main de la aplicación se encuentra en la clase Main dentro del paquete presenter.
- Para el testing de integración no se consideran casos que no sean posibles dada la mecánica de la aplicación, como por ejemplo buscar en la base de datos un artículo que no exista, ya que sólo se le permite al usuario seleccionar artículos que se encuentren en la base de datos. Tampoco se considera el caso de que la base de datos no exista porque consideramos que excede el alcance del test, aunque debería estar cubierto por los casos en que el módulo de base de datos produce una excepción.