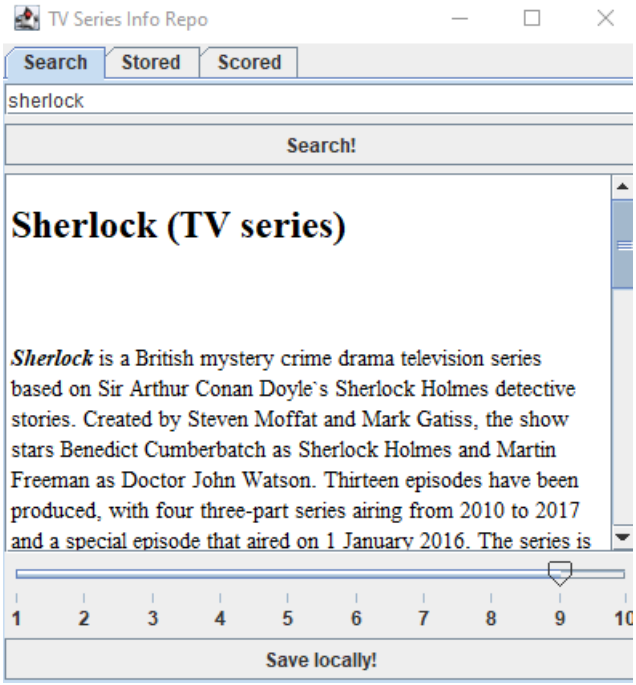
UNIVERSIDAD NACIONAL DEL SUR

Diseño y Desarrollo de Software

TV Series Info Repo



Garcia Romina

Departamento de Ciencias e Ingeniería de la Computación

2024

Índice

1. Introducción

En este proyecto, partiendo de una implementación de TV Series Info Repo se debía realizar las tareas del sprint actual, que consistían en rediseñar la arquitectura e implementar una nueva funcionalidad. Para esto se requería refactorizar todo el código utilizando el patrón de MVP siguiendo los principios SOLID y clean code. Además implementar una historia de usuario sobre la posibilidad de puntuar series, y otras funcionalidades opcionales.

2. Rediseño

La aplicación se rediseñó para que siguiera el modelo MVP, Modelo Vista Presentador. Con este fin, toda la funcionalidad de la clase MainWindow fue repartida en dos grandes partes, la parte de la vista correspondiente a mostrar en pantalla la interfáz gráfica, y el modelo encargado de realizar toda la lógica necesaria, ambas conectadas por el presentador. En un principió creé tres clases SearchView, SearchPresenter y SearchModel, en las cuales repartí toda la implementación y la adapté para que siguiera el patrón MVP, es decir la vista solo conoce al presentador y le avisa cuando recibe eventos del usuario, luego el presentador busca en la vista los datos que necesite y le pide al modelo que realice la tarea necesaria. Una vez que el modelo termina de computar, le avisa al presentador a través de listeners y este último muestra en la vista el resultado.

Luego, descubrí que el modelo poseía muchas responsabilidades, por lo que lo separé en varias clases. Por un lado SearchSeriesModel se encarga solo de encontrar los resultados de la búsqueda pedida, llamando a SearchSeriesAPI que se encarga de conectarse con la API de Wikipedia. SearchWikiPageModel por su parte, se encarga de recuperar el texto de una determinada página, llamándo a SearchWikiPageAPI que se conecta con la API. Por otro lado, todas las consultas que se necesiten hacer a la base de datos se harán con DataBaseModel, llamando a DataBase que realiza las consultas SQL.

De manera similar, la representación gráfica de la aplicación se representa con varias vistas. Primero, la clase TVSeriesSearcherWindow se encarga de crear la ventana y mantiene el TabbedPane donde cada una de las pestañas es manejada por una vista distinta: SearcherView, StoredView y ScoredView. Cada una de estas se encarga de mostrar la información correspondiente y avisar a los presentadores cuando el usuario produce algún evento.

Al existir varias vistas y modelos, parece natural que existan también varios presentadores, uno por cada vista. De esta forma cada vista le avisa a su propio presentador cuando suceden eventos, y cada uno atiende las consultas particulares. Sin embargo, a veces eventos que suceden en una vista afectan a otra, por lo que en estas situaciones los presentadores usan de intermediario al presentador de la vista general, ya que este es el que crea los demás presentadores y los asigna a cada vista, cumpliendo el rol de presentador general.

A su vez, existen clases utilizadas transversalmente a este patrón, ya que brindan herramientas para procesar texto o encapsular información. En particular la clase SearchResult dada, que heredaba de JMenuItem, fue reemplazada por la clase WikiPage que en vez de ser un JMenuItem, posee un atributo gráfico. WikiPage posee todas las características que puede tener una página, título, párrafo, puntaje, representación gráfica etc y métodos para acceder a ellos. De esta forma las diferentes clases pueden pasar información sobre las páginas encapsulando en objetos de esta clase.

Durante toda la refactorización se tomaron decisiones para respetar los conceptos de Clean Co-

de. Para esto se eliminaron todos los comentarios, se reemplazaron nombres de variables y métodos para que sean más descriptivos y se identó apropiadamente el código.

3. Principios SOLID

Todos los cambios se realizaron teniendo en cuenta los principios SOLID, en particular se ilustrará algunos ejemplos para cada principio:

Interface SegregationVarias interfaces específicas a los clientes son mejores que una sola interface de propósito general. Al tener varios modelos y varias interfaces para las API también estamos cumpliendo este principio, ya que no existe una interfaz que implementen tanto SearchSeriesAPI como SearchWikiPageAPI, sino que ambas tienen su propia interfaz. Dependence invertion: Módulos de alto nivel no deberían depender de módulos de bajo nivel.Ambos deberían depender de abstracciones Para seguir este principio, los modelos no acceden directamente a las API o la base de datos, los modelos tienen la responsabilidad de saber que datos quieren pero no como conseguirlos, por eso llaman a otras clases que se encargan de las llamadas de más bajo nivel De igual manera, los presentadores saben que se debe mostrar cierta información pero son las vistas las que se encargan de editar los componentes gráficos correspondientes También, el presentador sabe que se debe procesar el texto para que tenga el formato html pero le delega la propia modificación a una clase util

- **Single Responsability:** *una clase debería tener una, y solo una, razón para cambiar.*
  - Para respetar este principio es que se dividieron los modelos, presentadores y vistas como mencionadas en la sección anterior.
  - Tambien, SearchResult se separo en WikiPage y WikiPageMenuItem, separando la responsabilidad de mantener la información de una página de su representación gráfica.
- **Open-Closed:** *Los módulos de software deberían estar abiertos para extensión pero cerrados para modificación*
  - Las vistas implementan una interfaz, por lo que si en un futuro sprint se quiere cambiar completamente una vista, fácilmente se puede remplazar por otra que respete la misma interfaz, o extender con nuevas funcionalidades.
  - Todos los modelos utilizan una clase que implementa la comunicación con la API/base de datos, pero esta clase la reciben como una interfaz, pudiendose extender el modelo para usar otras APIs sin tener que modificarlo.
- **Liskov:** *las clases derivadas deben poder sustituirse por sus clases base*
  - No se implemento ninguna jerarquía de herencia, por lo que no aplica este principio.
- **Interface Segregation:** *Varias interfaces específicas a los clientes son mejores que una sola interface de propósito general.*
  - Al tener varios modelos y varias interfaces para las API también estamos cumpliendo este principio, ya que no existe una interfaz que implementen tanto SearchSeriesAPI como SearchWikiPageAPI, sino que ambas tienen su propia interfaz.
  - Que cada vista tenga su interfaz en vez de que exista una interfaz vista común a todas cumple con este principio.

- **Dependence invertion:** *Módulos de alto nivel no deberían depender de módulos de bajo nivel.Ambos deberían depender de abstracciones*

    - Para seguir este principio, los modelos no acceden directamente a las API o la base de datos, los modelos tienen la responsabilidad de saber que datos quieren pero no como conseguirlos, por eso llaman a otras clases que se encargan de las llamadas de más bajo nivel
    - De igual manera, los presentadores saben que se debe mostrar cierta información pero son las vistas las que se encargan de editar los componentes gráficos correspondientes
    - También, el presentador sabe que se debe procesar el texto para que tenga el formato html pero le delega la propia modificación a una clase util

4. Organización del Testing

Se requería realizar tanto testing de unidad para los modelos, como testing de integración.

Testing de Unidad:

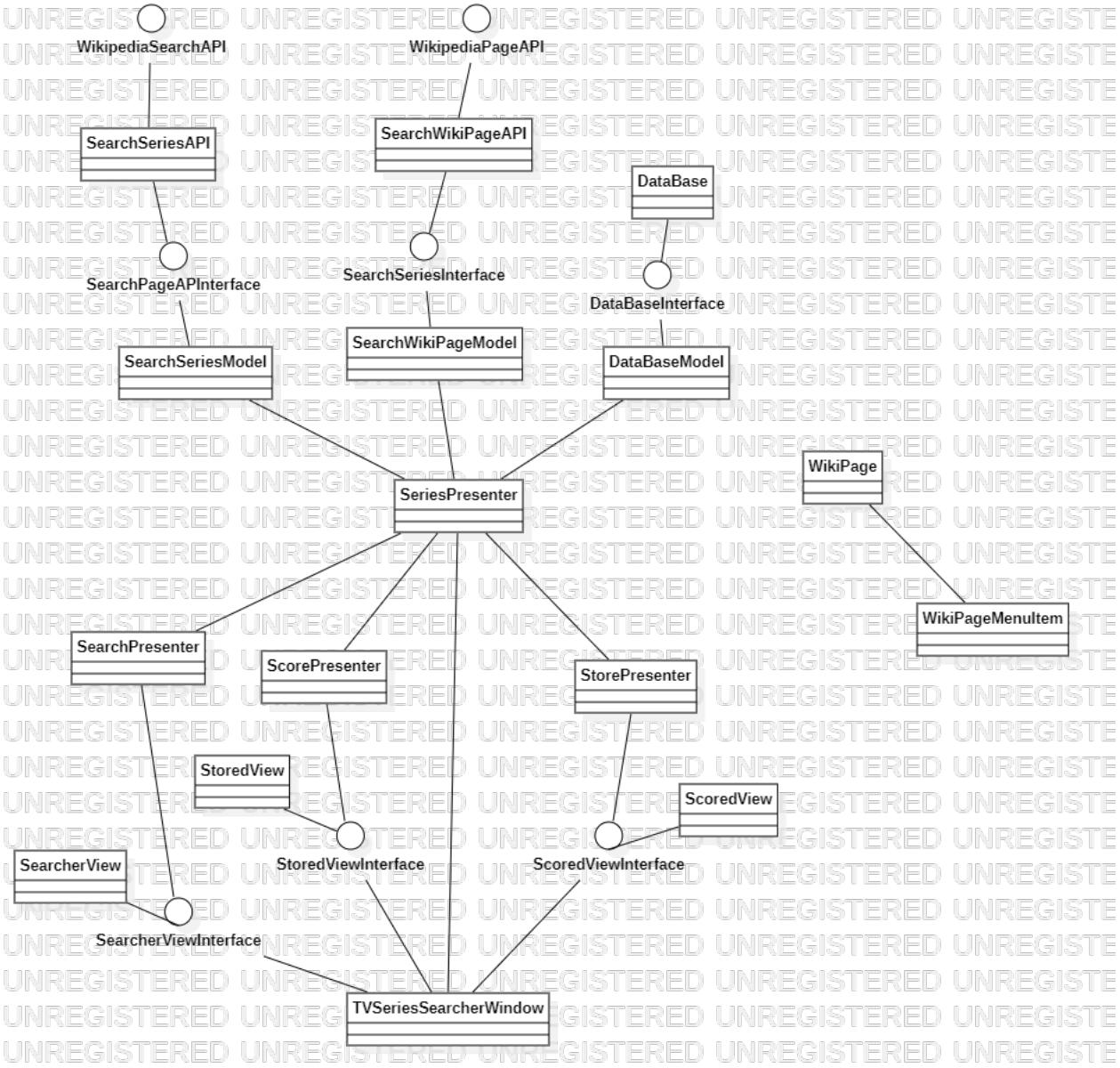
Para testear los modelos en solitario, se utilizaron stubs de SearchPageAPI, SearchWikiPageAPI y DataBase, para que los modelos puedan obtener los resultados necesarios pero sin acceder a las API o base de datos verdaderas. Estos stubs implementan las interfaces respectivas pero solamente devuelven datos de ejemplo. Luego la clase SearchModelTest crea un searchSeriesModel, WikiPageModel y DatabaseModel con los stubs y se testean todas las funcionalidades de los modelos.

Testing de integración:

Aquí también se utilizaron los mismos stubs para reemplazar a las API y base de datos, pero se crearon todas las vistas presentadores y modelos como la aplicación original y se testean varias posibles acciones del usuario.

Por ejemplo, para testear la búsqueda de una página. Se simula que se escribe “Breaking Bad” en el área de texto y se simula que se aprieta el botón de búsqueda, luego se simula que se selecciona el primer resultado, y se verifica si el texto que se muestra es el esperado. En este caso empezamos con un evento en la vista de búsqueda y corroboramos el cambio de esta vista, pero para que esto suceda debe funcionar todo el intercambio de mensajes entre los presentadores vistas y modelos.

5. Diagarama de Calses



6. Deciciones Adicionales

Para mostrar mensajes de error como de éxito, se utiliza manejo de excepciones como de listeners para que los presentadores reciban el error que pueda haber sucedido y le avisen al usuario.

Además en el caso de querer borrar una página guardada, antes de realizarlo se pregunta si realmente se quiere borrar.

También se implementó el opcional de mostrar el url a la página buscada, que al apretarse abre la página en el navegador.

7. Conclusión

Se refactorizó el código recibido siguiendo los patrones y principios vistos en la materia y se extendió para brindar las nuevas funcionalidades pedidas y mejorar la interacción con el usuario.