Desarrollo de Software (2021-2022)
Grado en Ingeniería Informática
Universidad de Granada

Práctica 2. Car Configurator

Sergio Quijano Rey
Fernando Valdés Navarro
Ignacio Carvajal Herrera
Carlos Corts Valdivia

21 de abril de 2022

Índice

1. Introducción

Nuestra aplicación consiste en un personalizador de coches, en el cual puedes elegir diferentes modelos y configurar distintas partes con varias opciones. Una vez finalizado el proceso de configuración, el usuario podrá o bien guardar la configuración para más adelante, o bien añadirlo al carrito y comenzar el pago.

En esta etapa de la aplicación, estamos utilizando una galería de imágenes reducida y estática, pero en una versión más completa de la app se podría pensar en acceder a una Base de Datos en la que la marca de coches sube fotos de los distintos modelos configurados con todo tipo de opciones, de forma que según el usuario va configurando el coche, podría ir observando cuál sería el resultado.

2. Requisitos funcionales

- **RF 1. Gestión de usuarios:** este requisito queda pospuesto para futuras versiones, ya que sería necesario interactuar con una BD.
  - **RF 1.1.** Darse de alta como usuario.
  - **RF 1.2.** Darse de baja como usuario.
  - **RF 1.3.** Identificarse como usuario.
  - **RF 1.4.** Consultar tus datos de usuario.
  - **RF 1.5.** Modificar tus datos de usuario.
- **RF 2. Gestión de configuraciones:** este requisito se aplica tanto a modelos preconfigurados como a los que el usuario configura desde 0.
  - **RF 2.1.** Comenzar nueva configuración desde 0.
  - **RF 2.2.** Borrar una configuración guardada.
  - **RF 2.3.** Modificar configuración ya existente (preconfigurada o guardada).
  - **RF 2.4.** Previsualizar una configuración ya existente (opción pospuesta para futuros accesos a BD con una galería más grande y variada de fotos).
- **RF 3. Gestión del catálogo:** este requisito queda pospuesto para futuras versiones, ya que sería necesario interactuar con una BD. Además, iría más destinado a un usuario con privilegios de administrador.
  - **RF 3.1.** Añadir nueva parte configurable.
  - **RF 3.2.** Eliminar parte configurable.
  - **RF 3.3.** Modificar parte configurable (nombre, descripción, etc.).
  - **RF 3.4.** Añadir nueva opción de una parte configurable.
  - **RF 3.5.** Eliminar una opción de una parte configurable.

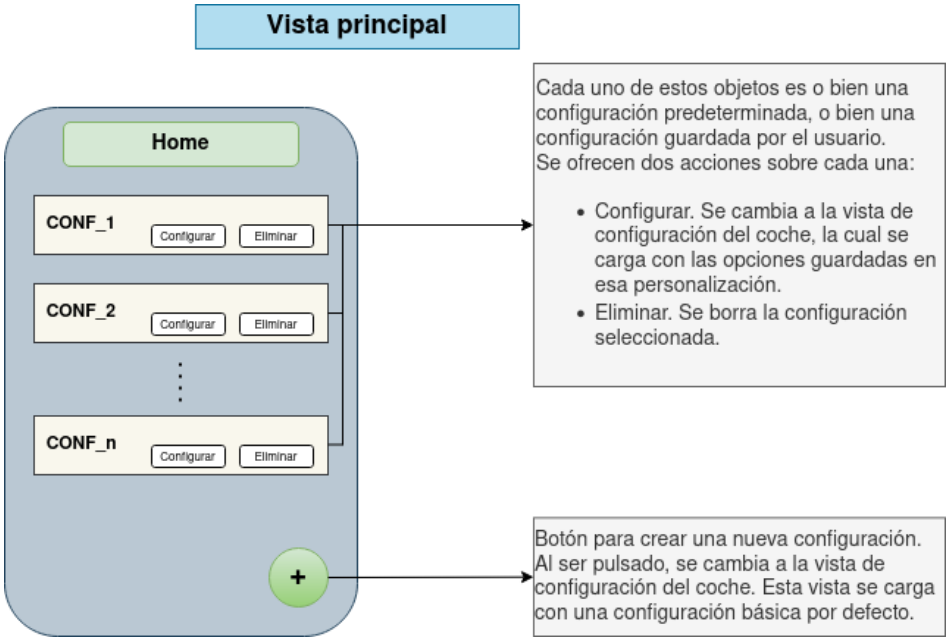
- **RF 3.6.** Modificar una opción de una parte configurable (nombre, descripción, fotos, etc.).

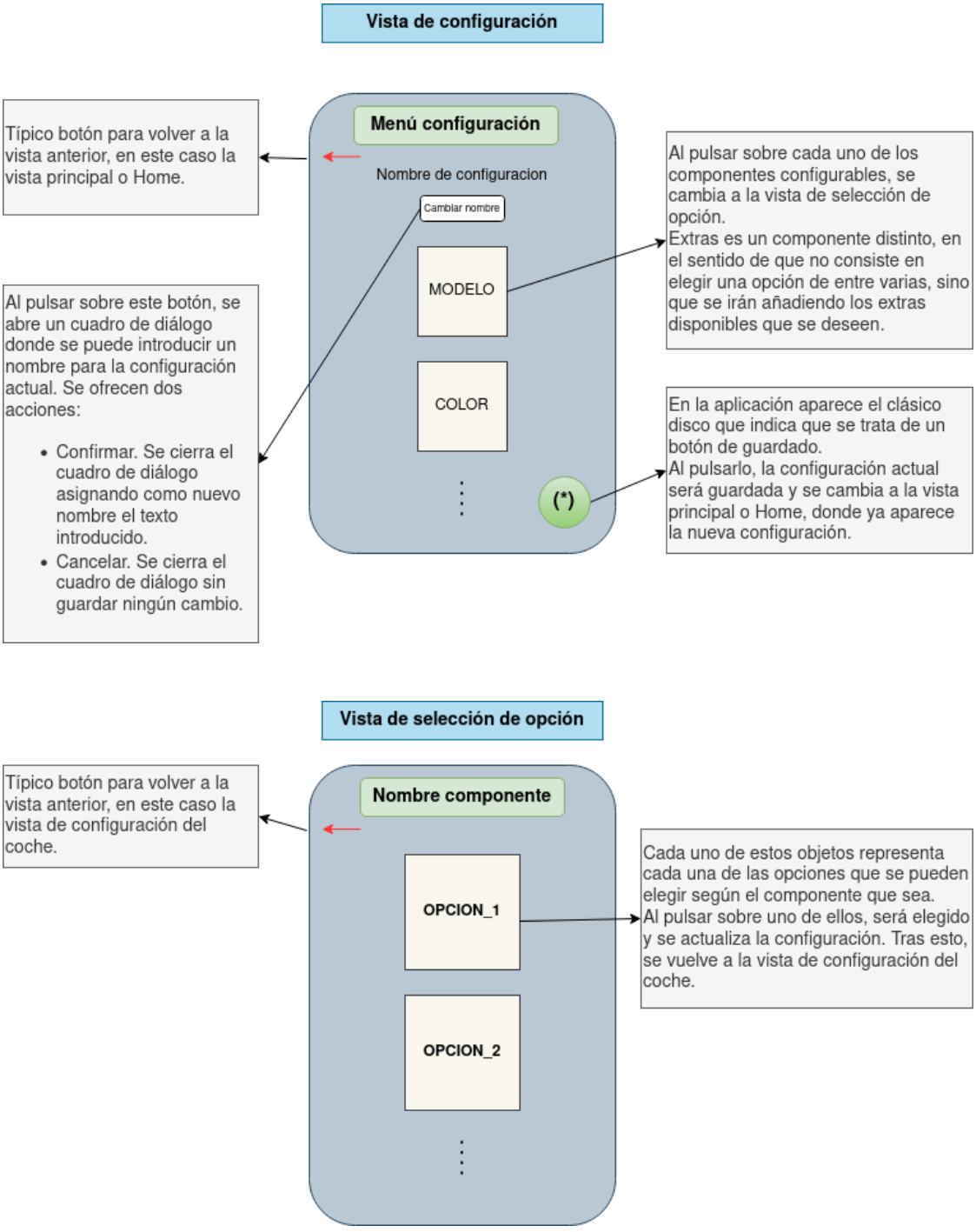
- **RF 4. Gestión de compra:** este requisito realmente no es responsabilidad de la aplicación, sino que sería administrado por una tercera parte.
  - **RF 4.1.** Realizar pedido (revisar carrito y proceder al pago).
  - **RF 4.2.** Consultar pedidos realizados (para futuras versiones en las que haya gestión de usuarios).
  - **RF 4.3.** Cancelar o modificar pedido (no es responsabilidad de la aplicación, sino de una entidad externa).
  - **RF 4.4.** Realizar pago (no es responsabilidad de la aplicación, sino de una entidad externa).

3. Requisitos no funcionales

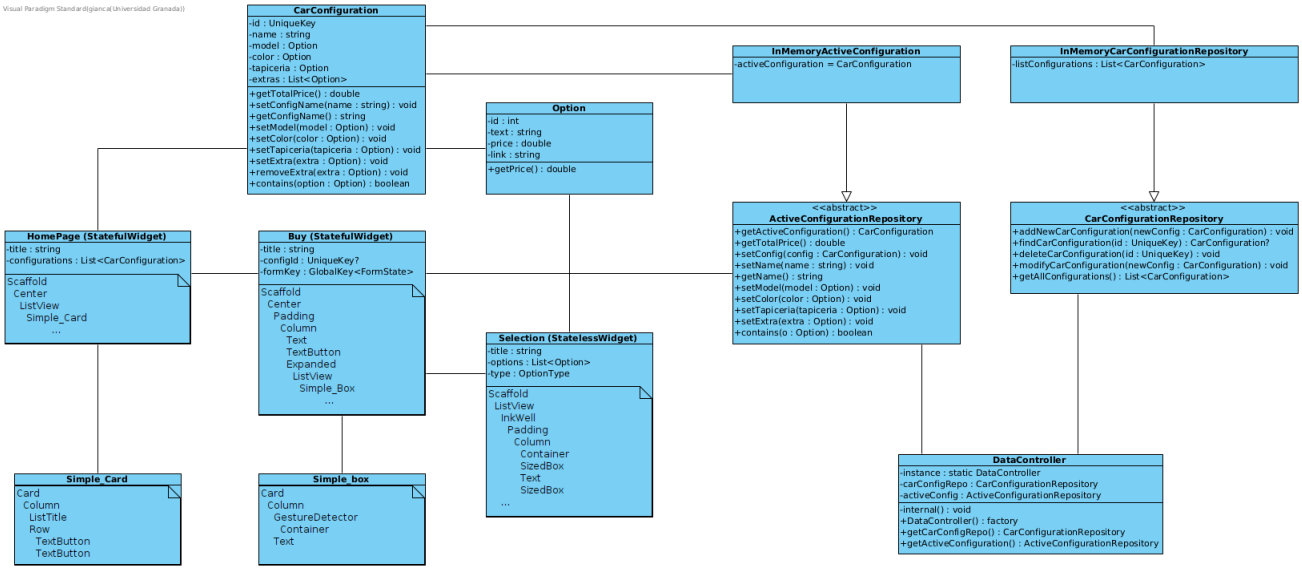
- **RNF 1. Seguro:** deben mantenerse protegidos los datos de los usuarios.
- **RNF 2. Evolutivo:** permitiendo añadir/eliminar/modificar partes configurables, opciones para una parte, métodos de pago, etc.
- **RNF 3. GUI amigable:** fácil de usar, intuitiva, destinada a un usuario medio.

4. Diagramas sobre funcionamiento de la aplicación




Vista de configuración
Típico botón para volver a la vista anterior, en este caso la vista principal o Home.
Menú configuración
Nombre de configuracion
Cambiar nombre
MODELO
COLOR
(*)
Al pulsar sobre cada uno de los componentes configurables, se cambia a la vista de selección de opción.
Extras es un componente distinto, en el sentido de que no consiste en elegir una opción de entre varias, sino que se irán añadiendo los extras disponibles que se deseen.
Al pulsar sobre este botón, se abre un cuadro de diálogo donde se puede introducir un nombre para la configuración actual. Se ofrecen dos acciones:
Confirmar. Se cierra el cuadro de diálogo asignando como nuevo nombre el texto introducido.
Cancelar. Se cierra el cuadro de diálogo sin guardar ningún cambio.
En la aplicación aparece el clásico disco que indica que se trata de un botón de guardado.
Al pulsarlo, la configuración actual será guardada y se cambia a la vista principal o Home, donde ya aparece la nueva configuración.
Vista de selección de opción
Típico botón para volver a la vista anterior, en este caso la vista de configuración del coche.
Nombre componente
OPCION_1
OPCION_2
Cada uno de estos objetos representa cada una de las opciones que se pueden elegir según el componente que sea.
Al pulsar sobre uno de ellos, será elegido y se actualiza la configuración. Tras esto, se vuelve a la vista de configuración del coche.

5. Diagrama de clases



6. Patrones

6.1. Patrón repositorio

En este patrón buscamos modelar cómo vamos a manejar la persistencia de datos. Es una parte fundamental por los siguientes motivos:

- En la aplicación, vamos a acceder constantemente a esta parte de persistencia de datos. Por tanto, para mantener un código limpio es fundamental que el acceso a la persistencia de datos sea cómoda y sencilla
- Es una parte que va a ser modificada. Ahora mismo solo modelamos esta persistencia de datos usando objetos en memoria. Pero en las siguientes prácticas, desarrollaremos una *API Rest* para implementar esta persistencia de datos con una base de datos
- A la hora de escribir *tests* en nuestra aplicación (cosa que no hemos hecho), es importante poder *mockear* la persistencia de datos

Esto motiva el uso de patrón repositorio. Declaramos una interfaz que indica cómo se realizan las operaciones *CRUD* de distintos conceptos de nuestra capa de negocio. Con esto, el acceso a estos datos es uniformizado y simplificado en toda la aplicación. Podemos tener distintas implementaciones para estas operaciones (ahora mismo solo tenemos objetos en memoria, pero con ese patrón es muy fácil implementar el acceso a una base de datos, a una *API REST*, ...)

Además, a la hora de implementar un *mock*, solo tenemos que realizar una implementación de la interfaz. De hecho, los objetos que tenemos almacenados en memoria para nuestra aplicación sirven como un futuro *mock*.

Mostramos el código que implementa esta patrón en las siguientes figuras:

```dart
/// Clase abstracta que usaremos como interfaz para definir la funcionalidad que debe implementar
/// un repositorio de configuraciones de coches
/// Notar que estamos usando el patron repositorio
abstract class CarConfigurationRepository{

  /// Añadimos una nueva configuracion al repositorio
  void addNewCarConfiguration(CarConfiguration newConfig);

  /// Buscamos una configuracion en el sistema usando su identificador
  CarConfiguration? findCarConfiguration(UniqueKey id);

  /// Borramos una configuracion del repositorio
  void deleteCarConfiguration(UniqueKey id);

  /// Modificamos una configuracion en el repositorio
  ///
  /// Notar que no pasamos el identificador de la configuracion que queremos modificar. Esto es
  /// porque CarConfiguration tiene ya dicha identificacion como atributo
  void modifyCarConfigurations(CarConfiguration newConfig);

  /// Devuelve una lista con todas las configuraciones almacenadas en el sistema
  List<CarConfiguration> getAllConfigurations();
}



/// Implementamos el repositorio de configuraciones
/// Guardamos y administramos los datos en memoria. No estamos conectandonos a una base de datos
/// ni usando el sistema de ficheros para persistir los datos. Esto se hara mas adelante cuando
/// consumamos una API REST que exponga la persistencia de datos
class InMemoryCarConfigurationRepository implements CarConfigurationRepository{

  /// Lista que usamos como 'base de datos'
  List<CarConfiguration> listConfigurations = [];
```

```dart
abstract class ActiveConfigurationRepository {
  getTotalPrice();
  CarConfiguration getActiveConfiguration();
  void setConfig (CarConfiguration config);
  String getName ();
  void setName (String name);
  void setModel (Option model);
  void setColor (Option color);
  void setTapiceria (Option tapiceria);
  void setExtra (Option extra);
  bool contains(Option o);
}

class InMemoryActiveConfiguration implements ActiveConfigurationRepository {

  CarConfiguration activeConfig = CarConfiguration.origin("Nueva Configuracion");
```

6.2. Patrón *singleton*

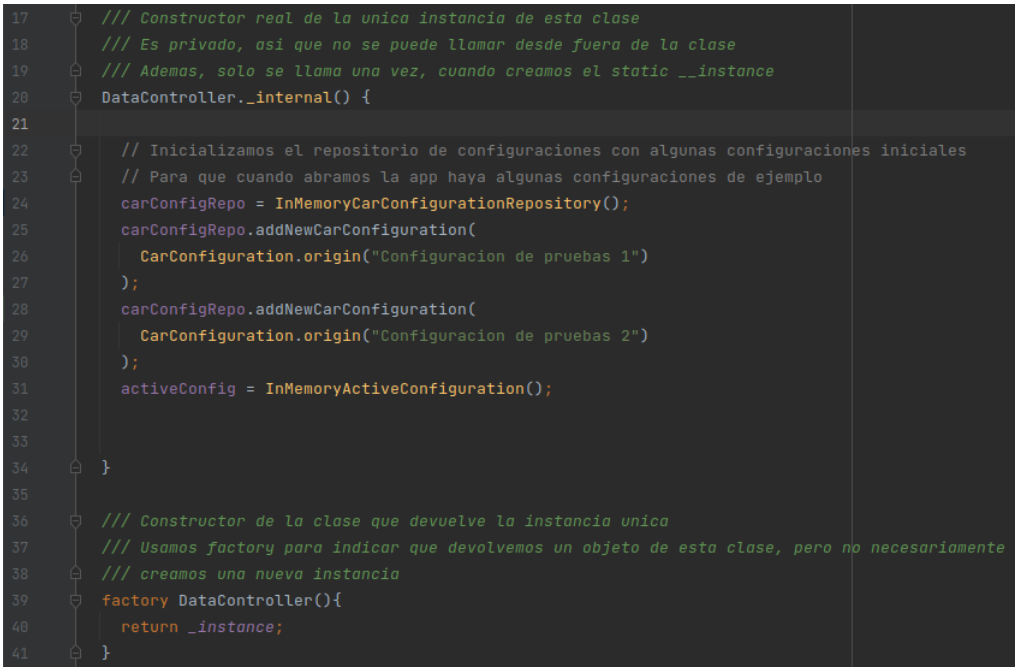
Con el patrón anterior hemos modelado el acceso a los datos. Sin embargo, hasta ahora, esto puede ser un poco molesto. La opción directa para usar estas implementaciones en distinta partes de la aplicación, es inicializarlas en la función principal e ir pasando estas implementaciones por parámetro, por toda la aplicación.

Esto es un problema en el caso que tengamos muchos repositorios distintos que ir pasando por toda la aplicación. Por tanto, pensamos que es adecuado el uso del patrón *Singleton*. En dicho *singleton*, que en nuestra código es la clase *DataController*, se recogen todas las interfaces tipo repositorio, se inicializan con cierta implementación deseada y se ponen a disposición con *getters*.

De esta forma, si una parte de la aplicación necesita acceder a algún repositorio, puede crear un nuevo objeto de la clase (para lo que solo necesitamos un *import '...'* en el fichero) y acceder al repositorio adecuado.

Mostramos el código que implementa este patrón en las siguientes figuras:

```
5   /// Clase que toma el control de toda la persistencia de datos
6   /// Es un singleton que da acceso a los distintos repositorios del sistema
7   /// Sigo la implementacion dada como ejemplo en la documentacion oficial:
8   ///   - https://flutterbyexample.com/lesson/singletons
9   class DataController{
10
11    // Instancia que usamos como unico objeto de la clase creado
12    static final DataController _instance = DataController._internal();
```



```dart
/// Constructor real de la unica instancia de esta clase
/// Es privado, asi que no se puede llamar desde fuera de la clase
/// Ademas, solo se llama una vez, cuando creamos el static __instance
DataController._internal() {

  // Inicializamos el repositorio de configuraciones con algunas configuraciones iniciales
  // Para que cuando abramos la app haya algunas configuraciones de ejemplo
  carConfigRepo = InMemoryCarConfigurationRepository();
  carConfigRepo.addNewCarConfiguration(
    CarConfiguration.origin("Configuracion de pruebas 1")
  );
  carConfigRepo.addNewCarConfiguration(
    CarConfiguration.origin("Configuracion de pruebas 2")
  );
  activeConfig = InMemoryActiveConfiguration();


}

/// Constructor de la clase que devuelve la instancia unica
/// Usamos factory para indicar que devolvemos un objeto de esta clase, pero no necesariamente
/// creamos una nueva instancia
factory DataController(){
  return _instance;
}
```

En estas figuras se puede apreciar que hemos implementado manualmente el patrón. Dart no trae por defecto el patrón, pero es muy sencillo de implementar, haciendo que el constructor devuelva un atributo estático.

6.3. Otros patrones

Impuesto por *Flutter*, estamos usando constantemente el patrón *composite*. Estamos componiendo *widgets* dentro de otros *widgets* para crear las vistas.

En segundo lugar, buscamos reusar todo el código posible para la creación de las vista. Esto lo hacemos en el módulo de componentes, donde implementamos una serie de funciones para crear componentes que usamos a lo largo de toda la aplicación. Con esto evitamos repetir el mismo código varias veces y, más importante, mantenemos una interfaz visual uniforme. Si queremos cambiar el *look & feel* de la aplicación, podemos cambiar estas funciones y automáticamente cambiará casi toda la interfaz gráfica de la aplicación.

En parte estamos implementando el patrón *Modelo - Vista - Controlador*, aunque sin el controlador. Tenemos un modelo para la configuración del coche, que interactúa con la parte de persistencia de datos y con las vistas de la aplicación.

7. Uso de *Github*

Como sistema de control de versiones hemos usado *Github*. Para organizar el trabajo, hemos usado los siguientes procedimientos:

- Uso de *issues* para marcar las tareas a completar. También hemos usado *issues* para discutir sobre distintos patrones a usar
- Uso de ramas para desarrollar código en paralelo. Con *pull requests* revisamos y añadimos el código a la rama principal del repositorio

El repositorio se puede encontrar en https://github.com/fervalnav/CarConfigurator.