Taller 5 DPOO Patrones

Estudiante: Camilo Daza **Código:** 201416461

Parte 1: Información general del proyecto

Proyecto: Flutter - https://github.com/flutter/flutter

Flutter es un proyecto, creado por Google, de código abierto que se compone de una serie de herramientas de desarrollo de software. Este se utiliza para crear aplicaciones de forma nativa para móvil, web y escritorio a partir de un código base único. Es principalmente conocido por sus aplicaciones de gran rendimiento y con una experiencia de usuario muy similar a la de las aplicaciones nativas. Por último, Flutter es compatible con el desarrollo de aplicaciones en seis plataformas: iOS, Android, web, Windows, MacOS y Linux.

En cuanto a su estructura general de diseño, este proyecto sigue una estructura basada en widgets personalizables, dónde estos corresponden a una unidad básica o elemento visual simple, que luego se juntan para construir un interfaz de usuario. Este proyecto está escrito en el lenguaje de programación Dart, un lenguaje creado por Google y que está optimizado específicamente para el desarrollo de interfaces de usuario.

Por último, Flutter presente dos dificultades principales para el avance del proyecto. Por un lado, al ser un lenguaje de programación nuevo y una arquitectura basada en widgets nueva, se necesita cierto tiempo para aprenderlo a usar correctamente. Por otro lado, Flutter es relativamente nuevo y su ecosistema se encuentra aún en crecimiento, por lo que sus librerías y paquetes puede que no sean tan completas como la de otros proyectos o herramientas.

Parte 2: Información y estructura del fragmento del proyecto donde aparece el patrón

```dart
class CachingIterable<E> extends IterableBase<E> {
  /// Creates a [CachingIterable] using the given [Iterator] as the source of
  /// data. The iterator must not throw exceptions.
  ///
  /// Since the argument is an [Iterator], not an [Iterable], it is
  /// guaranteed that the underlying data set will only be walked
  /// once. If you have an [Iterable], you can pass its [iterator]
  /// field as the argument to this constructor.
  ///
  /// You can this with an existing `sync*` function as follows:
  ///
  /// ```dart
  /// Iterable<int> range(int start, int end) sync* {
  ///   for (int index = start; index <= end; index += 1) {
  ///     yield index;
  ///   }
  /// }
  ///
  /// Iterable<int> i = CachingIterable<int>(range(1, 5).iterator);
  /// print(i.length); // walks the list
  /// print(i.length); // efficient
  /// ```
  ///
  /// Beware that this will eagerly evaluate the `range` iterable, and because
  /// of that it would be better to just implement `range` as something that
  /// returns a `List` to begin with if possible.
  CachingIterable(this._prefillIterator);
```

flutter / packages / flutter / lib / src / foundation / **basic_types.dart**

Code | Blame | 250 lines (217 loc) · 7.65 KB

```dart
    return CachingIterable<T>(super.map<T>(toElement).iterator);
  }

  @override
  Iterable<E> where(bool Function(E element) test) {
    return CachingIterable<E>(super.where(test).iterator);
  }

  @override
  Iterable<T> expand<T>(Iterable<T> Function(E element) toElements) {
    return CachingIterable<T>(super.expand<T>(toElements).iterator);
  }

  @override
  Iterable<E> take(int count) {
    return CachingIterable<E>(super.take(count).iterator);
  }

  @override
  Iterable<E> takeWhile(bool Function(E value) test) {
    return CachingIterable<E>(super.takeWhile(test).iterator);
  }

  @override
  Iterable<E> skip(int count) {
    return CachingIterable<E>(super.skip(count).iterator);
  }

  @override
  Iterable<E> skipWhile(bool Function(E value) test) {
```

Flutter es un proyecto enorme, en el que distintos tipos de patrones de diseño son usados a lo largo de todo el código, en las anteriores imágenes se muestran dos

implementaciones del patrón Iterator de los patrones de diseño GoF del tipo behavioral patterns. En esta sección se presentaron estas dos partes del código, pero es un tipo de patrón de suma importancia y de un gran uso a lo largo de todo el proyecto.

Parte 3: Información general sobre el patrón

El patrón iterator es un patrón de diseño de comportamiento, el cual es ampliamente usado en una gran cantidad de proyectos. Este patrón, se encarga de proveer acceso a los distintos elementos de una agrupación de objetos ordenadores secuencialmente. Lo anterior, lo logra hacer sin la necesidad de conocer o acceder directamente a la estructura interna de la agrupación que está recorriendo.

Dentro de las características principales que se cumplen con este patrón se presentan, el encapsulamiento, la flexibilidad y el acceso uniforme. En primer lugar, los detalles de cómo se almacenan los elementos en la agrupación no se exponen al código del usuario, solo permitiendo atravesar la agrupación por medio de iterator pero sin exponer mayor información de la estructura. En segundo lugar, el patrón iterator añade gran flexibilidad a las distintas implementaciones de los proyectos. Lo anterior, dado que los programadores no deben incorporar la implementación concreta de la agrupación, sino solo en la implementación del iterador, por lo que cambios internos en la estructura no afectarían la iteración sobre la agrupación. En tercer lugar, este patrón se caracteriza por permitir una manera transversal y uniforme de acceder a múltiples tipos de agrupaciones, usando una interfaz común. En conclusión, el patrón iterator permite recorrer de manera sencilla agrupaciones, sin necesidad de conocer la estructura del objeto recorrido, permitiendo cambios internos de la estructura y bajo un modelo uniforme y sencillo.

Parte 4: Información del patrón aplicado al proyecto

Este patrón es de suma importancia para Flutter, puesto que al ser un proyecto enfocado en widgets y tener tan gran tamaño, va a necesitar distintos tipos de estructuras para almacenar la información de los aplicativos. De igual forma, se puede usar el iterador para generar optimizaciones y eficiencias en algunas zonas del código, lo anterior, para generar un comportamiento más ligero de la aplicación a crear.

Por ejemplo, en la parte 2 se pegó el código en el que se da uso del patrón iterator para dos funcionalidades distintas. Por un lado, se implementa el patrón sobre un código en la clase Utils, enfocada directamente a métodos que facilitan la ejecución de otras tareas, por medio del llamado a este. Por ejemplo, en el código mostrado, el método ZipStrict permite comparar dos listas iterables y entender si estas son de igual tamaño. Lo anterior, facilita la implementación de múltiples métodos a lo largo de todo el proyecto, siendo este método muy flexible de ser llamado, dado que no necesita entender la estructura de las listas para realizar la comparación. Por otro lado, en el segundo código, se implementa con la intención de implementar la funcionalidad de almacenamiento de caché. En donde la clase CachingIterable es una clase que encapsula el funcionamiento de almacenamiento

de caché. De esta manera, y usando la función expand, se transforman los elementos en una colección iterable para luego poder recorrer a través de ellos. La intención en este caso es poder almacenar los elementos y poder acceder y recorrerlos fácilmente, para evitar recalcular los elementos y de manera más eficiente poder acceder a ellos en un futuro.

Como las dos anteriores, a lo largo de Flutter se presentan una gran cantidad de usos distintos del patrón iterator. Esto, permitiendo acceder a distintas agrupaciones de datos, con el fin de generar mayor flexibilidad, e incluso mejorar la eficiencia de algunas funcionalidades.

Parte 5: ¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto? ¿Qué ventajas tiene?

La utilización del patrón Iterator en estos puntos del proyecto proporciona varias ventajas:

- **Encapsulamiento de las agrupaciones:** Permitiendo que se logre realizar el recorrido buscado de las estructuras, sin necesidad de depender de la estructura de la agrupación a recorrer.
- **Flexibilidad:** El código generado se convierte en uno mucho más flexible, en donde se puede alterar en gran medida la agrupación o lista a recorrer, y el funcionamiento del iterador será el mismo.
- **Facilidad en mantenimiento del código:** Dado que aunque se den cambios en las estructuras, no habrá dependencia de esas partes del código, e incluso, permite realizar mejoras o cambios a los iteradores sin afectar las estructuras.

Parte 6: ¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?

Aunque el patrón iterator tiene múltiples ventajas, cómo se ha expuesto anteriormente, también presenta posibles desventajas a considerar si se va a implementar. Por un lado, en casos en los que la estructura a recorrer es de gran tamaño, el uso del iterador puede representar exigencias muy grandes en uso de memoria, dado que este va almacenando los objetos iterados en memoria. Por otro lado, manejar iteradores puede agregar una complejidad adicional al código que en algunos casos no se necesitaba, en los la estructura a recorrer es conocida e inmutable, y el uso del iterador solo complejiza la implementación. De esta manera, en algunos casos de estructuras de gran tamaño o estructuras conocidas y que no cambian, se pueden tener ineficiencias y complejizar el código innecesariamente.

Parte 7: ¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?

Para lograr una implementación similar, se pudieron haber usado recorridos for para cada uno de los métodos. Con lo anterior se podría lograr una solución similar, pero sería necesario que el código en cuestión conociera más información de las estructuras a

recorrer. De esta manera, dado que se pierde modularidad y se necesita trasladar detalles de la implementación, esta solución dificultaría en gran medida el desarrollo de un proyecto de gran tamaño como lo es Flutter y generaría distintas ineficiencias en su implementación.