

Pertenece a: Vanegas, José Luis

Cédula: 3-743-157

**Investigación2**

**Ciclo de Vida de los Datos Android Architecture Components**

LiveData es quizás el componente más necesario y útil de Android Architecture Components. Se trata de una envoltura observable para nuestros datos con la ventaja de que está asociado a un ciclo de vida (de un fragment o activity).

**Ventajas**

Tu UI siempre mostrará los datos más actualizados. LiveData no deja de ser un Observable vitaminado. Si construimos nuestra UI para que recoja la información a mostrar observando objetos LiveData, cualquier cambio que se produzca en estos datos será transmitido a cualquier vista o componente que lo esté observando. Esto significa que siempre se mantendrán actualizados.

No más errores ni código defensivo para evitar activities/fragments inactivos. Como LiveData está asociado al ciclo de vida de la activity o fragment, no se producirán llamadas a sus observables si la vista no está activa.

Tus datos siempre actualizados. Si la vuelta de un servicio ocurre cuando nuestra activity está en segundo plano, ese valor se almacena en el objeto LiveData. Además, en cuanto esa activity vuelva a estar activa, lo notificará inmediatamente a los observadores relacionados.

Evita Memory Leaks. Los observadores están asociados a un objeto con ciclo de vida. En el momento que ese objeto es destruido, los observadores se liberan de memoria.

**Usos**

La clase LiveData<T> es un objeto abstracto que no podemos instanciar como tal. Para la creación de estos objetos usaremos la clase MutableLiveData<T> la cual ya implementa todo lo necesario. Declaramos ese objeto en nuestro ViewModel, Repositorio o incluso en un Presenter de esta forma:

private final MutableLiveData<User> mUserInfoLiveData = new MutableLiveData<>();

**Patrones de Diseño en Java**

Los patrones del diseño tratan los problemas del diseño que se repiten y que se presentan en situaciones particulares del diseño, con el fin de proponer soluciones a ellas. Por lo tanto, los patrones de diseño son soluciones exitosas a problemas comunes. Existen muchas formas de implementar patrones de diseño. Los detalles de las implementaciones son llamadas estrategias.

**Historia de los patrones de diseño**

Un patrón de diseño es una abstracción de una solución en un nivel alto. Los patrones solucionan problemas que existen en muchos niveles de abstracción. Hay patrones que abarcan las distintas etapas del desarrollo; desde el análisis hasta el diseño y desde la arquitectura hasta la implementación.

Muchos diseñadores y arquitectos de software han definido el término de patrón de diseño de varias formas que corresponden al ámbito a la cual se aplican los patrones. Luego, se dividió los patrones en diferentes categorías de acuerdo a su uso.

Los diseñadores de software extendieron la idea de patrones de diseño al proceso de desarrollo de software. Debido a las características que proporcionaron los lenguajes orientados a objetos (como herencia, abstracción y encapsulamiento) les permitieron relacionar entidades de los lenguajes de programación a entidades del mundo real fácilmente, los diseñadores empezaron a aplicar esas características para crear soluciones comunes y reutilizables para problemas frecuentes que exhibían patrones similares.

Fue por los años 1994, que apareció el libro "Design Patterns: Elements of Reusable Object Oriented Sofware" escrito por los ahora famosos Gang of Four (GoF, que en español es la pandilla de los cuatro) formada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Ellos recopilaron y documentaron 23 patrones de diseño aplicados usualmente por expertos diseñadores de software orientado a objetos. Desde luego que ellos no son los inventores ni los únicos involucrados, pero ese fue luego de la publicación de ese libro que empezó a difundirse con más fuerza la idea de patrones de diseño.

**Modelo vista controlador (MVC)**

Modelo Vista Controlador (MVC) es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

Se trata de un modelo muy maduro y que ha demostrado su validez a lo largo de los años en todo tipo de aplicaciones, y sobre multitud de lenguajes y plataformas de desarrollo.

* El **Modelo** que contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia.
* La **Vista**, o interfaz de usuario, que compone la información que se envía al cliente y los mecanismos interacción con éste.
* El **Controlador**, que actúa como intermediario entre el Modelo y la Vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

**Modelo–Vista–Presentador (MVP)**

Modelo–Vista–Presentador (MVP) es una derivación del patrón arquitectónico modelo–vista–controlador (MVC), y es utilizado mayoritariamente para construir interfaces de usuario.

En MVP el presentador asume la funcionalidad del "intermediario". En MVP, toda lógica de presentación es colocada al presentado

**Descripción de patrón**

MVP es un patrón arquitectónico de interfaz de usuario diseñada para facilitar pruebas de unidad automatizada y mejorar la separación de inquietudes en lógica de presentación:

* El *modelo* es una interfaz que define los datos que se mostrará o no actuado en la interfaz de usuario.
* El *presentador* actúa sobre el modelo y la vista. Recupera datos de los repositorios (el modelo), y los formatea para mostrarlos en la vista.
* La *vista* es una interfaz pasiva que exhibe datos (el modelo) y órdenes de usuario de las rutas (eventos) al presentador para actuar sobre los datos.

Normalmente, la vista de implementación instancia el objeto de presentador en concreto, proporcionando una referencia a él. El siguiente código C# demuestra una simple vista de un constructor, donde ConcreteDomainPresenter implementa la interfaz IDomainPresenter:

public class DomainView : IDomainView

{

private IDomainPresenter domainPresenter = null;

///<summary>Constructor</summary>

public DomainView()

{

domainPresenter = new ConcreteDomainPresenter(this);

}

}

**Modelo vista modelo de vista (MVVM)**

El patrón modelo–vista–modelo de vista (en inglés model–view–viewmodel, abreviado MVVM) es un patrón de arquitectura de software. Se caracteriza por tratar de desacoplar lo máximo posible la interfaz de usuario de la lógica de la aplicación.

## Elementos

El modelo

Representa la capa de datos y/o la lógica de negocio, también denominado como el objeto del dominio. El modelo contiene la información, pero nunca las acciones o servicios que la manipulan. En ningún caso tiene dependencia alguna con la vista.

La vista

La misión de la vista es representar la información a través de los elementos visuales que la componen. Las vistas en MVVM son activas, contienen comportamientos, eventos y enlaces a datos que, en cierta manera, necesitan tener conocimiento del modelo subyacente.

Modelo de vista

El modelo de vista es un actor intermediario entre el modelo y la vista, contiene toda la lógica de presentación y se comporta como una abstracción de la interfaz. La comunicación entre la vista y el viewmodel se realiza por medio los enlaces de datos.