

Sviluppo di applicazioni mobili

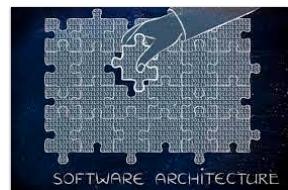
Architettura di una app

L'architettura su cui si basa l'app è fondamentale per garantire che l'app sia **robusta, testabile e manutenibile**

Android fornisce un insieme di librerie e componenti per creare app secondo le migliori pratiche

Cosa è un'architettura sw?

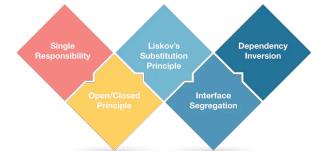
L'architettura software di un sistema definisce come il sistema è suddiviso in **componenti** e come questi **interagiscono** fra loro attraverso **interfacce**



Principi di base di progettazione

I principi di progettazione SOLID abilitano alla creazione di software più **manutenibile, comprensibile, riutilizzabile e testabile**

S.O.L.I.D.



Single responsibility

- Un modulo sw dovrebbe avere una sola responsabilità
 - Ogni classe dovrebbe avere una e una sola ragione per cambiare, ovvero, una classe deve essere responsabile di un unico aspetto o funzionalità del sistema
- Garantisce:
 - **Testing facilitato**
 - Un componente con una sola responsabilità richiederà molto meno casi di test
 - **Loose coupling**
 - Meno funzionalità in una singolo componente avranno meno dipendenze
 - **Modificabilità semplificata**
 - Componenti più semplici

Esempio

- Supponiamo di avere una classe `UserProfileManager` che gestisce un profilo utente.
- La classe contiene metodi sia per aggiornare i dati dell'utente che per visualizzarli

```
class UserDataManager {
    public void updateUserData(String name, String email) {
        // Logica per aggiornare i dati dell'utente nel database
        System.out.println("Aggiornamento dati utente nel database.");
    }

    public String getUserData() {
        // Logica per recuperare i dati dell'utente
        return "Nome: Mario Rossi, Email: mario.rossi@example.com";
    }
}

class UserDataView {
    public void displayUserData(String userData) {
        // Logica per visualizzare i dati dell'utente
        System.out.println("Visualizzazione dati: " + userData);
    }
}
```

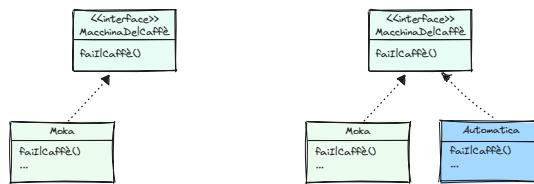
```
class UserProfileManager {
    public void updateUserData(String name, String email) {
        // Logica per aggiornare i dati dell'utente nel database
        System.out.println("Aggiornamento dati utente nel database.");
    }

    public void displayUserData() {
        // Logica per visualizzare i dati dell'utente
        System.out.println("Visualizzazione dati utente...");
```

- Per rispettare il principio di Single Responsibility, si separano le due responsabilità in due classi distinte:
 - una per gestire i dati e
 - un'altra per occuparsi della visualizzazione

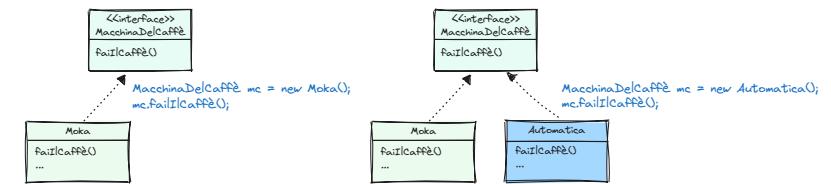
Open-Closed

- I componenti dovrebbero essere **aperti** all'estensione, ma **chiusi** alla modifica
- Garantisce:
 - Non modificabilità del codice:**
 - il rischio di introdurre bug è mitigato



Liskov Substitution

- A parità di **contratto**, un componente dovrebbe poter essere sostituito senza compromettere il sistema
- Garantisce:
 - Supporto all'evoluzione** del sw
 - Supporto allo sviluppo incrementale** (sub)



Interface Segregation

- Interfacce troppo ampie dovrebbero essere divise in interfacce più piccole

```
public interface BearKeeper {  
    void washTheBear();  
    void feedTheBear();  
    void petTheBear();  
}  
  
public interface BearCleaner {  
    void washTheBear();  
}  
  
public interface BearFeeder {  
    void feedTheBear();  
}  
  
public interface BearPetter {  
    void petTheBear();  
}  
  
public class BearCarer implements BearCleaner, BearFeeder {  
    public void washTheBear() {  
        //I think we missed a spot...  
    }  
  
    public void feedTheBear() {  
        //Tuna Tuesdays...  
    }  
}
```

- Garantisce
 - **Separation of concerns**
 - **Loose coupling**

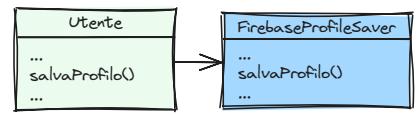
Dependency Inversion

- Il principio si riferisce al disaccoppiamento fra i componenti
 - I componenti dovrebbero dipendere da astrazioni, non da componenti concrete

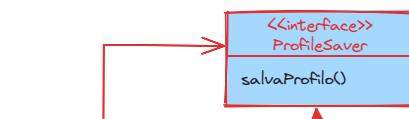
- **Garantisce**

- Una chiara e corretta definizione dei confini
- Evoluzione
- Manutenzione
-

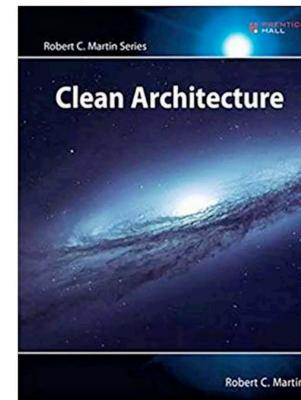
Ma cosa significa?



```
public class Utente {
    ...
    private FirebaseProfileSaver saver;
    ...
    public void salvaProfilo() {
        saver.salvaProfilo();
    }
    ...
}
```



```
public class Utente {
    ...
    private ProfileSaver saver;
    ...
    public void salvaProfilo() {
        saver.salvaProfilo();
    }
    ...
}
```



Robert C. Martin
Aka Uncle Bob

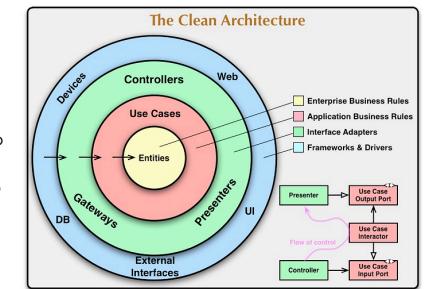
*La Clean Architecture, è un'architettura software progettata per rendere il sistema indipendente da dettagli **implementativi**, altamente **modulare**, e facilmente **testabile***

Si basa su una struttura a cerchi concentrici, dove ogni livello ha dipendenze verso l'interno, rispettando il principio di **Dependency Inversion**

The Clean Architecture

Un insieme di linee guida per progettare l'architettura di un software

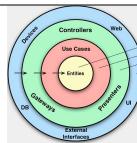
- Definisce come partizione in livelli il software definendo in maniera chiara i confini fra questi
- Al centro: il codice di alto livello (cioè la logica pura)
- All'esterno: il codice di basso livello
- È governato dalla **Dependency Rule**
 - il codice di basso livello (più esterno) può dipendere da quello di livello superiore (più interno), ma mai il contrario
 - in altre parole, che nessun elemento di un cerchio interno può sapere nulla di qualcosa di un cerchio esterno, cioè il **cerchio interno non dovrebbe dipendere** da quello esterno



Livello Entity

- Questo è il livello centrale dell'architettura e contiene le entità principali dell'applicazione
- Le entità rappresentano gli oggetti principali del dominio dell'applicazione e contengono la logica di business più generale e indipendente dal contesto specifico
- Non contengono dettagli di implementazione come database o interfaccia utente

Esempi di entità: "User," "Product," o "Order", calculateDiscount()

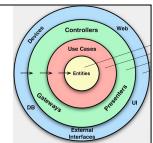


Livello Use Cases

- Questo livello contiene i casi d'uso che rappresentano i comportamenti specifici dell'applicazione
- I casi d'uso coordinano le operazioni tra le entità e gestiscono la logica di business
- Si tratta di logica di business perché riguarda **cosa** viene fatto, non **come** viene fatto

Esempio: "AddToCartUseCase" gestisce la logica per l'aggiunta di un prodotto al carrello:

1. Verifica prodotto disponibile
2. Salva l'elemento nel carrello
3. Aggiorna la UI



Livello delle Interfacce degli Adattatori

Interface Adapters



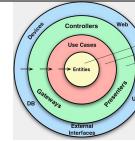
- Questo livello si occupa di **adattare** l'applicazione (e quindi i dati) a elementi esterni, come il database, l'interfaccia utente e i servizi web
- Include componenti come i **Repository**, i **Presenter** e i **Controller**

Esempi:

- "UserRepository" si interfaccia con un database per il recupero e la memorizzazione degli utenti
- "ProductPresenter" gestisce la presentazione dei dati relativi ai prodotti nella UI

Livello dei Framework e delle Librerie

Frameworks and Libraries



- Questo è il livello esterno che contiene le tecnologie di base come il framework di sviluppo, le librerie di terze parti e i dettagli di implementazione specifici della piattaforma
- In questo livello, si trova tutto ciò che è legato a una tecnologia particolare, come il framework Android o una libreria di persistenza specifica

Esempi:

- L'interfaccia utente realizzata con Jetpack Compose o Android XML.
- L'uso di Firebase per la gestione dell'autenticazione o il framework Spring per la gestione dei servizi web

Si desidera aggiungere un nuovo utente all'applicazione

Il livello **Entità** ha, fra le altre, l'entità **User**.

Il caso d'uso "AddUserUseCase" nel livello dei Casi d'Uso coordina questa azione. Si interfaccia con il "UserRepository" nel livello delle Interfacce degli Adattatori per memorizzare il nuovo utente.

"UserRepository" quindi utilizza **Room** per effettuare l'operazione di memorizzazione

La chiave di questa architettura è la separazione chiara delle responsabilità e la modularità, che semplificano la manutenzione, il testing e la scalabilità del codice

Clean Architecture e Android

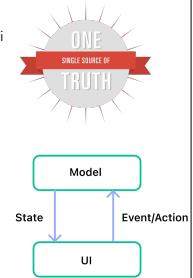
- La Clean Architecture tende ad essere un punto di riferimento per **sistemi complessi**
- Le app Android non ricadono in questa categoria
- Per questo, sebbene riconoscendo la validità dei principi, le raccomandazioni sull'architettura delle app, note come **Modern App Architecture**, sono una versione un po' più flessibile della clean architecture

Architettura Moderna delle app Android

★ **Note:** The recommendations and best practices present in this page can be applied to a broad spectrum of apps to allow them to scale, improve quality and robustness, and make them easier to test. However, you should treat them as guidelines and adapt them to your requirements as needed.

Principi alla base da seguire

- **Separation of concerns**
 - Separazione delle responsabilità
- **Drive UI from data model**
 - L'interfaccia utente dovrebbe essere derivata dallo stato dei dati, non costruita manualmente
 - La UI deve aggiornarsi automaticamente quando il modello di dati cambia, mantenendo una chiara separazione tra presentazione e dati sottostanti
- **Single source of truth (SSOT)**
 - Si dovrebbe mantenere una singola fonte di dati **autorevole** e **affidabile** per un insieme specifico di informazioni
- **Unidirectional Data Flow**
 - Il flusso dei dati deve procedere in una sola direzione:
 - lo stato scorre dai dati verso la UI,
 - gli eventi o azioni dell'utente fluiscano nella direzione opposta, verso la logica di business che aggiorna lo stato
 - Questo approccio riduce gli effetti collaterali e semplifica il debugging



Tre livelli

- Google semplifica la sua architettura in tre livelli

- **UI Layer**

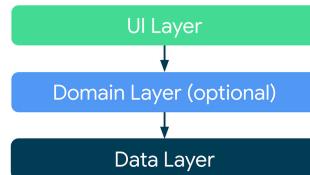
- gestisce l'input e l'output degli utenti e l'aggiornamento della visualizzazione

- **Data Layer**

- contiene la logica di business dell'applicazione ed espone i dati dell'applicazione

- (opzionale) **Domain Layer**

- per semplificare e riutilizzare le interazioni tra i layer UI e Data

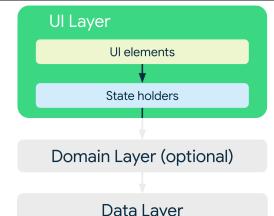


★ Nota: le frecce nei diagrammi di questa guida rappresentano le dipendenze tra le classi. Ad esempio, il livello dominio dipende dalle classi del livello dati.

UI Layer

Overview

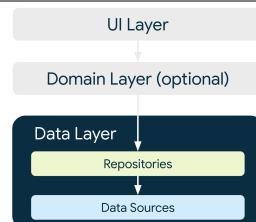
- Il ruolo del livello UI (o livello di presentazione) è
 - la **visualizzazione** dei dati dell'applicazione sullo schermo
 - l'**aggiornamento** dei dati quando cambiano
 - a causa dell'interazione dell'utente (come la pressione di un pulsante) o dell'input esterno (come una risposta di rete)
- Il livello UI è costituito da due elementi:
 - **UI elements**
 - consentono di visualizzare i dati sullo schermo
 - Per creare questi elementi, View o Jetpack Compose
 - **State Holder** (come le classi `ViewModel`)
 - contengono i **dati**
 - li espongono all'interfaccia utente
 - gestiscono la logica di visualizzazione



Data Layer

Overview

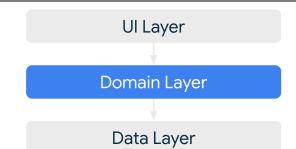
- Definisce le regole (business logic) che determinano il modo in cui l'app **crea, archivia e modifica** i dati
- Il livello dati è costituito da due elementi:
 - Repository**
 - contengono uno o più data source
 - un repository per tipo di dato**
 - espongono i dati al resto della app
 - centralizzano la modifica dei dati
 - risolvono i conflitti quando esistono più data source
 - nascondono (astrazione) la data source
 - Data Source**
 - gestisce una sola fonte di dati
 - file, rete, database locale



Domain Layer

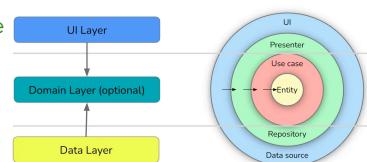
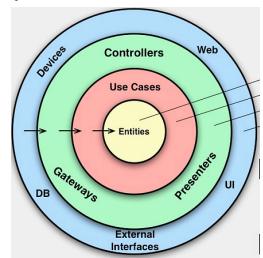
Overview

- Incapsula la logica di business complessa oppure quella semplice, ma usata da più State Holder (e.g., ViewModel)
- Le classi in questo livello sono generalmente chiamate 'use case' o 'interactor'
- Ogni use case dovrebbe avere la responsabilità su una singola funzionalità
 - Ad esempio, la app potrebbe avere una classe `GetTimeZoneUseCase` se diversi ViewModel dipendono dal fuso orario per visualizzare il messaggio corretto sullo schermo



Confronto tra l'Architettura Moderna delle App di Google e Clean Architecture

- Il **UI Layer** di Android corrisponde alla **UI** al **Presenter** di Clean Architecture
- Il **Domain Layer** corrisponde a **Use case** e a **Entity** di Clean Architecture
- Il **Data Layer** corrisponde ai **Repository** e al **Data Source** di Clean Architecture



Gestione delle dipendenze fra componenti

- Le classi dipendono da altre classi per il loro funzionamento
- Design pattern per gestire le dipendenze di una classe specifica:
 - **Dependency Injection**
 - È una tecnica per realizzare il principio di Dependency Inversion
 - È il meccanismo con cui le dipendenze vengono fornite (iniettate) dall'esterno, invece di essere create all'interno di una classe
 - **Service locator**
 - È un pattern alternativo alla Dependency Injection
 - Invece di ricevere la dipendenza dall'esterno, la classe la chiede a un registro globale (il locator), che sa come crearla o restituirla

Dependency injection

Problema

- Le classi hanno bisogno di riferimenti ad altre classi



La classe Car ha bisogno di un riferimento al suo Engine

- La classe costruisce la dipendenza di cui ha bisogno

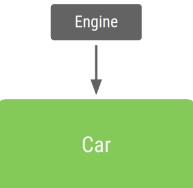
Car and Engine are tightly coupled - an instance of Car uses one type of Engine , and no subclasses or alternative implementations can easily be used. If the Car were to construct its own Engine , you would have to create two types of Car instead of just reusing the same Car for engines of type Gas and Electric .

```
class Car {  
    private Engine engine = new Engine();  
  
    public void start() {  
        engine.start();  
    }  
}  
  
class MyApp {  
    public static void main(String[] args) {  
        Car car = new Car();  
        car.start();  
    }  
}
```

Dependency injection

Soluzione manuale

- Le classi ricevono le istanze da cui dipendono dall'esterno
- Esistono due modi:
 - constructor injection*: attraverso il costruttore
 - field injection*: attraverso metodi setter



Reusability of Car . You can pass in different implementations of Engine to Car . For example, you might define a new subclass of Engine called ElectricEngine that you want Car to use. If you use DI, all you need to do is pass in an instance of the updated ElectricEngine subclass, and Car still works without any further changes.

```
class Car {  
    private final Engine engine;  
  
    public Car(Engine engine) {  
        this.engine = engine;  
    }  
  
    public void start() {  
        engine.start();  
    }  
}  
  
class MyApp {  
    public static void main(String[] args) {  
        Engine engine = new Engine();  
        Car car = new Car(engine);  
        car.start();  
    }  
}
```

Service Locator

- Si crea una classe nota come service locator che crea e memorizza le dipendenze e quindi fornisce tali dipendenze su richiesta
- Con il **service locator**, le classi hanno il controllo e chiedono che gli oggetti vengano iniettati
- Con il **dependency injection** le classi sono passive e qualche altra entità inietta proattivamente gli oggetti richiesti

```
class ServiceLocator {  
    private static ServiceLocator instance = null;  
  
    private ServiceLocator() {}  
  
    public static ServiceLocator getInstance() {  
        if (instance == null) {  
            synchronized(ServiceLocator.class) {  
                instance = new ServiceLocator();  
            }  
        }  
        return instance;  
    }  
  
    public Engine getEngine() {  
        return new Engine();  
    }  
  
    class Car {  
        private Engine engine = ServiceLocator.getInstance().getEngine();  
  
        public void start() {  
            engine.start();  
        }  
    }  
  
    class MyApp {  
        public static void main(String[] args) {  
            Car car = new Car();  
            car.start();  
        }  
    }  
}
```

General best practices

- <https://developer.android.com/topic/architecture#best-practices>

Programming is a creative field, and building Android apps isn't an exception. There are many ways to solve a problem; you might communicate data between multiple activities or fragments, retrieve remote data and persist it locally for offline mode, or handle any number of other common scenarios that nontrivial apps encounter.

Although the following recommendations aren't mandatory, in most cases following them makes your code base more robust, testable, and maintainable in the long run:

Don't store data in app components.

Avoid designating your app's entry points—such as activities, services, and broadcast receivers—as sources of data. Instead, they should only coordinate with other components to retrieve the subset of data that is relevant to that entry point. Each app component is rather short-lived, depending on the user's interaction with their device and the overall current health of the system.

Reduce dependencies on Android classes.

Your app components should be the only classes that rely on Android framework SDK APIs such as `Context`, or `Toast`. Abstracting other classes in your app away from them helps with testability and reduces coupling  within your app.

Create well-defined boundaries of responsibility between various modules in your app.

For example, don't spread the code that loads data from the network across multiple classes or packages in your code base. Similarly, don't define multiple unrelated responsibilities—such as data caching and data binding—in the same class. Following the recommended app architecture will help you with this.

Expose as little as possible from each module.

For example, don't be tempted to create a shortcut that exposes an internal implementation detail from a module. You

UI Layer

Introduzione

Sintesi

- Il ruolo dell'interfaccia utente (UI) è
 - quello di **mostrare** i dati dell'applicazione sullo schermo e
 - servire come principale **punto di interazione** dell'utente

Ogniqualvolta i dati cambiano a causa
dell'interazione dell'utente o di input esterni
l'UI dovrebbe aggiornarsi per riflettere tali modifiche



- I dati dell'applicazione che provengono dal livello dati sono di solito in un formato diverso rispetto alle informazioni necessarie per la visualizzazione
- Il livello UI è la **pipeline** che converte i cambiamenti dei dati dell'applicazione in una forma che l'UI può presentare e quindi li visualizza

Esempio

Una app recupera delle notizie che l'utente può leggere.

L'app ha una schermata con la lista delle notizie disponibili.

L'app consente agli utenti registrati di segnalare gli articoli che sono davvero interessanti.

Dato che in qualsiasi momento potrebbero esserci molti articoli, il lettore dovrebbe essere in grado di sfogliare gli articoli per categoria.

- In sintesi, l'app consente agli utenti di fare quanto segue:

- Visualizzare gli articoli disponibili da leggere
- Sfogliare gli articoli per categoria
- Accedere e contrassegnare come preferiti determinati articoli
- Accedere a alcune funzionalità premium se idonei



Stato UI

Definizione

- Lo **stato della UI** è l'informazione che l'applicazione stabilisce che l'utente dovrebbe vedere
- Gli **elementi UI** sono un mezzo per mostrare lo stato



- La UI mostra un elenco di articoli e alcuni metadati per ogni articolo
- Queste informazioni che l'applicazione presenta all'utente costituiscono lo stato della UI

```
data class NewsItemUiState(  
    val title: String,  
    val body: String,  
    val bookmarked: Boolean = false,  
    ...  
)
```

Stato UI

Evoluzione dello stato: Gestione attraverso UDF

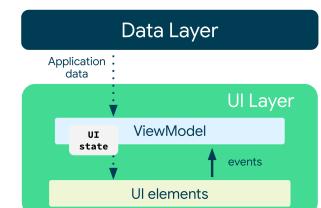
- Lo **stato della UI** può evolvere nel tempo
- L'aggiornamento della UI dovrebbe essere in incarico ad un **mediatore** che gestisce la **logica** e le **trasformazioni** dei dati
 - La UI può diventare troppo complessa se assume troppe responsabilità

La UI deve focalizzarsi principalmente sulla visualizzazione dello stato attuale

Stato UI

Gestione dell'evoluzione attraverso UDF: **State Holder**

- **State Holders**: classi responsabili del mantenimento dello stato della UI e della logica necessaria al suo aggiornamento
- Le classi **ViewModel** sono tipici state holder

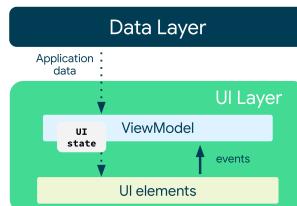


💡 Key Point: The **ViewModel** type is the recommended implementation for the management of screen-level UI state with access to the data layer. Furthermore, it survives configuration changes automatically. **ViewModel** classes define the logic to be applied to events in the app and produce updated state as a result.

Stato UI

Gestione dell'evoluzione attraverso UDF: State Holder

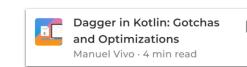
- Lo schema in cui lo stato scorre verso il basso e gli eventi verso l'alto è chiamato flusso di dati unidirezionale (UDF)
- Funzionamento:
 - Il ViewModel contiene ed espone lo stato che deve essere visualizzato dagli UI elementi
 - Gli UI elementi notificano al ViewModel gli eventi dell'utente
 - Il ViewModel gestisce le azioni dell'utente e aggiorna lo stato
 - Lo stato aggiornato viene restituito agli elementi UI per il rendering
 - La procedura si ripete per ogni evento che causa una mutazione dello stato



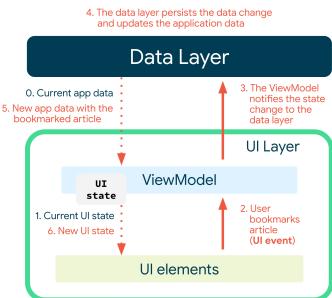
Stato UI

Gestione dell'evoluzione attraverso UDF: State Holder - Esempio

- Nell'esempio, viene mostrato un elenco di articoli, ciascuno con titolo, descrizione, fonte, nome dell'autore, tempo di lettura e se è stato aggiunto ai preferiti
- La UI di ogni articolo ha il seguente aspetto:



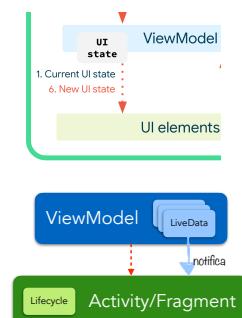
- Un utente che richiede l'aggiunta ai preferiti di un articolo è un esempio di **evento** che può causare **mutazioni di stato**
- Il ViewModel ha il compito di definire tutta la logica necessaria
 - per compilare tutti i campi nello stato dell'interfaccia utente
 - ed elaborare gli eventi necessari per la visualizzazione completa dell'interfaccia utente



Stato UI

Esposizione dello stato: LiveData (o StateFlow)

- Lo stato è mantenuto dagli State Holder (ViewModel)
- Poiché lo stato evolve nel tempo, il ViewModel lo espone tramite una struttura dati osservabile, come **LiveData** o **StateFlow**
- La UI (Activity o Fragment) si registra come osservatore:
 - ogni volta che lo stato cambia, la UI reagisce automaticamente e si aggiorna, senza dover interrogare manualmente il ViewModel
- Sintesi:
 - Il ViewModel è il produttore di stato
 - La UI è l'osservatore
 - LiveData / StateFlow sono il canale reattivo che collega i due



Eventi UI

- Gli eventi UI sono azioni che devono essere gestite nel livello UI, sia dall'UI che dal ViewModel
- Il tipo più comune di eventi è quello degli eventi utente
 - L'**utente produce** eventi interagendo con l'applicazione, ad esempio toccando lo schermo o generando gesti
 - La **UI consuma** questi eventi utilizzando callback come gli ascoltatori onClick()
- Due tipi di eventi:
 - Business logic:** cosa occorre fare a fronte di un cambiamento di stato
 - Il ViewModel è responsabile della loro gestione
 - Ad esempio l'aggiornamento dello stato di una news - bookmarked
 - UI behavior logic:** come mostrare il cambiamento di stato
 - La UI/gli UI elementi sono responsabili della gestione
 - Ad esempio, il click implica la navigazione verso un'altra schermata

A Key terms:

- UI:** View-based or Compose code that handles the user interface.
- UI events:** Actions that should be handled in the UI layer.
- User events:** Events that the user produces when interacting with the app.



```
class LatestNewsActivity : AppCompatActivity() {

    private lateinit var binding: ActivityLatestNewsBinding
    private val viewModel: LatestNewsViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        /* ... */

        // The expand details event is processed by the UI that
        // modifies a View's internal state.
        binding.expandButton.setOnClickListener {
            binding.expandedSection.visibility = View.VISIBLE
        }

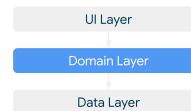
        // The refresh event is processed by the ViewModel that is in charge
        // of the business logic.
        binding.refreshButton.setOnClickListener {
            viewModel.refreshNews()
        }
    }
}
```

Domain Layer

Introduzione

Sintesi

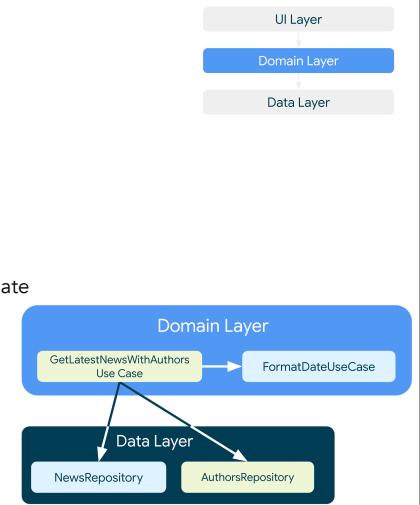
- Layer opzionale, responsabile
 - dell'incapsulamento della logica di business **complessa** o
 - della logica di business semplice che viene **riutilizzata** da più ViewModel
- Opzionale, perché non tutte le applicazioni hanno questi requisiti
- Offre i seguenti vantaggi:
 - Evita la duplicazione del codice
 - Migliora la leggibilità delle classi che utilizzano le classi del livello di dominio
 - Migliora la testabilità dell'applicazione
 - Evita classi di grandi dimensioni, consentendo di suddividere le responsabilità



Introduzione

Esempio

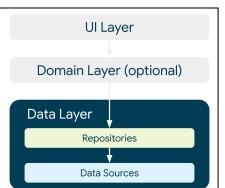
- Naming convention *FaiQualcosaUseCase*
- Domani Layer contiene una classe che recupera i dati da due repository
 - Business logic complessa
- Domani Layer che definisce una classe per convertire le date
 - Business logic utilizzata da più ViewModel
- Dipendenze



Data Layer

Introduzione

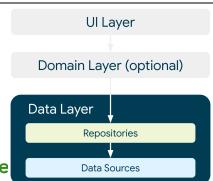
- Il **Data Layer** contiene i **dati** dell'applicazione e la **logica di business**
 - La business logic è ciò che dà valore all'applicazione: è costituita da regole che determinano come i dati dell'applicazione devono essere *creati, memorizzati e modificati*
- Questa *separation of concerns* consente
 - di utilizzare il data layer su più schermate
 - di condividere le informazioni tra le diverse parti dell'applicazione e
 - di riprodurre la logica di business al di fuori dell'interfaccia utente per i test unitari



Introduzione

Architettura

- Il data layer è costituito da **repository**, i quali contengono uno o più **data source**
- È buona pratica definire un repository per ciascun tipo di dato trattato (**SSOT**)
- Ad esempio, si crea una classe `MoviesRepository` class per la gestione di dati di tipo film e una classe `PaymentsRepository` class per la gestione di dati relativi ai pagamenti
- Le classi **Repository** hanno la responsabilità di:
 - Esporre i dati al resto dell'applicazione
 - Centralizzare le modifiche ai dati
 - Risolvere i conflitti tra più fonti di dati
 - Astrarre le fonti di dati dal resto dell'applicazione
- Le classi **Data Source** hanno la responsabilità di lavorare con una sola fonte dati
 - file, risorsa di rete, database locale

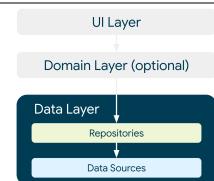


Introduzione

Architettura

Gli altri livelli della gerarchia **non** dovrebbero mai accedere direttamente ai data source

- I punti di ingresso al Data Layer sono sempre le classi **Repository**
 - Le classi che mantengono lo stato (`ViewModel`) o le classi degli Use Case non devono mai avere un Data Source come dipendenza diretta
- L'uso di classi repository come punti di ingresso consente ai diversi livelli dell'architettura di **scalare** in modo indipendente



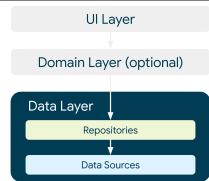
Introduzione

Architettura: Acquisizione Data Source e APIs

- Seguendo le migliori pratiche di dependency injection, il repository prende i Data Source come dipendenze nel suo costruttore

```
1 public class MoviesRepository {  
2     private MoviesRemoteDS moviesRemoteDS; //remote  
3     private MoviesLocalDS moviesLocalDS; //local  
4  
5     public MoviesRepository(MoviesRemoteDS moviesRemoteDS,  
6                             MoviesLocalDS moviesLocalDS) {  
7         this.moviesRemoteDS = moviesRemoteDS;  
8         this.moviesLocalDS = moviesLocalDS;  
9     }  
10    //...  
11 }
```

- Le classi Repository espongono generalmente funzioni
 - per eseguire chiamate di **creazione, lettura, aggiornamento e cancellazione** (CRUD) o
 - per ricevere notifiche sulle modifiche dei dati nel tempo



```
1 public class MoviesRepository {  
2     private MoviesRemoteDS moviesRemoteDS; //remote  
3     private MoviesLocalDS moviesLocalDS; //local  
4  
5     public MoviesRepository (MoviesRemoteDS moviesRemoteDS,  
6                             MoviesLocalDS moviesLocalDS) {  
7         this.moviesRemoteDS = moviesRemoteDS;  
8         this.moviesLocalDS = moviesLocalDS;  
9     }  
10    public void modifyMovie(Movie movie) { ...}  
11    //...  
12 }
```

Introduzione

Architettura: naming convention

In this guide, repository classes are named after the data that they're responsible for. The convention is as follows:
type of data + Repository.

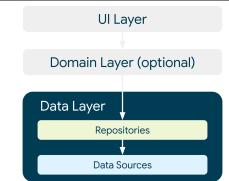
For example: `NewsRepository`, `MoviesRepository`, or `PaymentsRepository`.

Data source classes are named after the data they're responsible for and the source they use. The convention is as follows:

type of data + type of source + DataSource.

For the type of data, use `Remote` or `Local` to be more generic because implementations can change. For example: `NewsRemoteDataSource` or `NewsLocalDataSource`. To be more specific in case the source is important, use the type of the source. For example: `NewsNetworkDataSource` or `NewsDiskDataSource`.

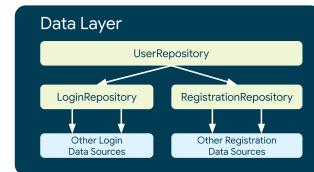
Don't name the data source based on an implementation detail—for example, `UserSharedPreferencesDataSource`—because repositories that use that data source shouldn't know how the data is saved. If you follow this rule, you can change the implementation of the data source (for example, migrating from `SharedPreferences` to `DataStore`) without affecting the layer that calls that source.



Introduzione

Architettura: molteplici livelli di Repository

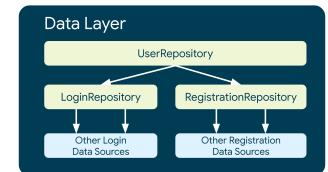
- In alcuni casi un repository potrebbe dover dipendere da altri repository
- Ciò può avvenire
 - perché i dati coinvolti sono un'aggregazione di più Data Source o
 - perché la responsabilità deve essere incapsulata in un'altra classe di repository
- **UserRepository**, un repository che gestisce i dati di autenticazione degli utenti, potrebbe dipendere da **LoginRepository** e **RegistrationRepository** per soddisfare i suoi requisiti



Introduzione

Architettura: Source of Truth

- È importante che ogni repository definisca un'**unica fonte di verità**
- La fonte di verità contiene sempre dati coerenti, corretti e aggiornati
- L'origine della verità può essere un Data Source, ad esempio il database, o anche una cache in memoria
- I repository combinano diverse Data Source e risolvono ogni potenziale conflitto tra esse per aggiornare la singola fonte di verità regolarmente o in seguito a un evento inserito dall'utente
- I diversi repository dell'applicazione possono avere diverse fonti di verità
 - Ad esempio, la classe **LoginRepository** potrebbe usare la sua cache come fonte di verità e la classe **PaymentsRepository** potrebbe usare il Data Source di rete
- Per fornire un supporto offline-first, un Data Source, come un database, è la fonte di verità consigliata



Offline-first app

Un'applicazione offline-first è un'applicazione in grado di eseguire tutte o un sottoinsieme critico delle sue funzionalità principali senza accedere a Internet

In altre parole, può eseguire una parte o tutta la sua logica di business offline

- Il Data Layer offre accesso ai dati che possono aver origine da fonti dati esterne al dispositivo
 - A tal fine, potrebbe essere necessario ricorrere alle risorse di rete per rimanere aggiornati
 - La disponibilità della rete non è sempre garantita
 - Larghezza di banda Internet limitata
 - Interruzioni transitorie della connessione, ad esempio in ascensore o in galleria
 - Accesso occasionale ai dati. Ad esempio, i tablet solo WiFi.
- Per garantire che l'applicazione funzioni correttamente offline, deve essere in grado di:
 - Presentare immediatamente agli **utenti** i **dati locali**, invece di attendere il completamento o il fallimento della prima chiamata di rete
 - Recuperare i dati solo in condizioni ottimali, ad esempio durante la ricarica o la connessione WiFi

Un'applicazione in grado di soddisfare i criteri sopra descritti viene spesso definita un'applicazione offline-first

L'origine dati locale è la fonte attendibile canonica dell'app
★ Nota: un repository con accesso di rete in un'app con priorità ai contenuti offline deve sempre avere un'origine dati locale.

