

android

COMPUTAZIONE PERVASIVA

Con lo sviluppo delle nuove tecnologie che ha portato alla nascita dei dispositivi di compilazione portabili è nata una nuova forma di computazione, quella **pervasiva**.

Essendo i dispositivo mobili e non più fissi si è dovuto cercare un nuovo modo di far comunicare i vari dispositivi per via delle connessioni precarie e in continuo cambiamento. Si cerca di progettare anche un software che sia in grado di utilizzare meno risorse possibili e che abbiano una potenza di calcolo minore per evitare di esaurire le risorse del dispositivo punto di tutto questo si occupa l'anno mobile computing. Questi dispositivi pervasivi sono sistemi distribuiti e quindi questi problemi dei sistemi distribuiti e mobile fanno parte del pervasive computing a cui si aggiungono altri problemi come la scalabilità invisibilità la comunicazione remota ecc.

Non tutti gli obiettivi del pervasive computer sono stati realizzati infatti creare uno **smartplace**, uno spazio in cui ci sono dispositivi di calcoli in modo da calcolare il mondo fisico e cyber fisico è ancora difficile da realizzare, ma non perché non possediamo le tecnologie adeguate ma perché dobbiamo migliorare i collegamenti.

È necessario rendere questi dispositivi **invisibili**, l'uomo non si deve accorgere che ci sono in modo che il mondo fisico e il cyber fisico si fondono in un solo spazio. Un altro problema è dovuto al numero sempre crescendo i dispositivi che si connettono tra loro quindi si cerca di realizzare la **scalabilità localizzata** cioè di mantenere le stesse qualità di servizi ma man mano che aumentano i nuovi dispositivi.

Sì cerca di mascherare le **condizioni di non uniformità** dovute diversi dispositivi presenti che possono impedire di collaborare fra loro cercando di creare un sistema per nascondere alle domande e questi problemi.

La computazione pervasiva ha modificato anche il modo in cui ci interfacciamo col software perché prima era necessario l'intervento umano per far svolgere qualcosa la macchina mentre oggi il computer è in grado di comunicare con tende anche in maniera asincrona.

Questo è stato reso possibile grazie al fatto che prendono informazioni nel mondo reale tramite i sensori sono in grado di elaborarli e notificare all'uomo qualcosa.

Oggi sia una architettura ed Edge cloudcentrica in cui tutti i dispositivi sono connessi tra loro tramite il cloud in cui le risorse informatiche sono regate attraverso la rete internet ospitate e gestite da server remoti.

Si provvede di avere un architettura ed internet centrica con cui internet il centro delle connessioni fra i dispositivi e i server. In questo modo l'elaborazione avviene in modo distribuita ai margini della rete contrariamente a come accade in quelle precedenti in cui le elaborazione avviene in economia i server.

Nell'architettura di calcolo distribuita ci sono tre segmenti che rappresentano muri diversi di elaborare e archiviare dati a seconda delle distanze dal punto di generazione

📍 **Edge Computing** Alla base di tutto troviamo l'**Edge**, ovvero il bordo della rete. I dati vengono elaborati **il più vicino possibile alla fonte che li genera**, spesso direttamente all'interno del dispositivo stesso. I dati vengono elaborati in tempo reale, senza dover prima inviare i dati a un server esterno. Questo approccio è ideale per situazioni in cui la velocità è fondamentale, come nelle auto a guida autonoma o nei dispositivi medici. Riduce la latenza, cioè il tempo che passa tra l'input e la risposta, e limita il traffico di rete.

▢ **Fog Computing** Salendo di un livello, troviamo il **Fog Computing**, un modello pensato per distribuire l'elaborazione più vicino alla rete, ma non necessariamente sul dispositivo.

Nel fog computing, i dati vengono inviati a **nodi** locali, un router o un server che li elaborano e, solo se necessario, li inoltrano al cloud. Questo modello è utile quando bisogna elaborare rapidamente grandi quantità di dati provenienti da più dispositivi, ma non si ha la necessità di farlo in tempo reale estremo.

▢ **Cloud Computing** Infine, abbiamo il **Cloud**, che rappresenta l'approccio centralizzato. I dati vengono inviati a server remoti, spesso situati in data center geograficamente lontani. Qui possono essere elaborati con grande potenza computazionale, salvati per lungo tempo o usati per addestrare algoritmi complessi di intelligenza artificiale.

Il cloud è perfetto per elaborazioni intensive. Tuttavia, presenta limiti in termini di latenza e dipendenza dalla connessione internet.

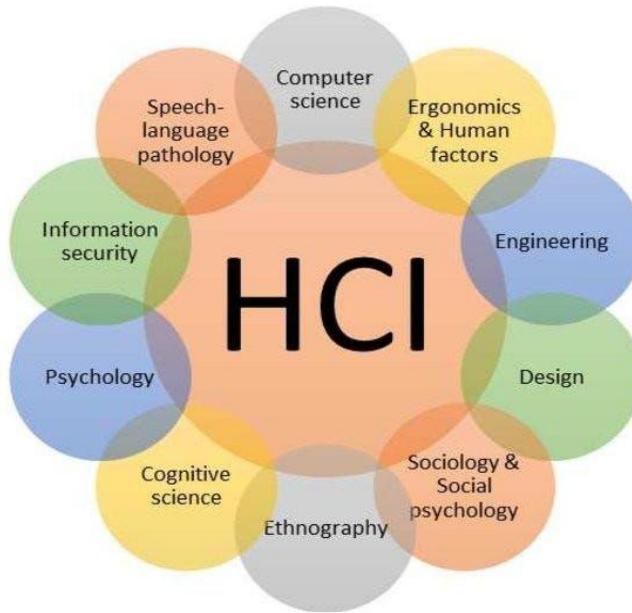
Edge, Fog e Cloud non si escludono a vicenda: **collaborano**. In un sistema moderno ed efficiente, i dati vengono prima processati localmente (edge), poi aggregati e raffinati in una rete vicina (fog), e infine archiviati o ulteriormente analizzati nel cloud.

HCI

La **Human computer interaction HCI** Eh lo studio dell'interazione uomo computer per la progettazione di sistemi informatici che siano usabili e affidabili. Il loro scopo è quello di semplificare l'attività umane e non di ostacolarle. Se hai il computer che le persone offrono diversi modi di interagire fra loro e l'interazione può essere

- **diretta** cioè tramite l'uso del dispositivo e di risposte da parte del calcolatore ;
- **indiretta** tramite sensori se analizzano il mondo esterno e danno dei feedback per svolgere azioni

La HCI è una materia multidisciplinare che coinvolge



Il principio base della disciplina è

usabilità cioè la facilità con il quale l'utente può interagire con la macchina per raggiungere obiettivi con efficacia, efficienza e soddisfazione per un certo contesto

#

accessibilità che indica se un sistema informatico può essere usato da tutti compreso chi ha disabilità

Interazione umana

L'uomo è colui che usa i calcolatori e devono essere progettati per assisterlo. Qui entrano in gioco le scienze cognitive che devono capire le capacità e le limitazioni per progettare in modo corretto un sistema informatico

Interazione processo con il quale l'utente fornisce la macchina in un input e la macchina lo processa restituendo un output o viceversa

Per la macchina si sono molte forme di IO, ma per l'essere umano sono principalmente i cinque sensi e più in particolare la vista l'udito e il tatto.

Interagiamo con la macchina principalmente con il tatto, le dita fungono da cursore. Solo recentemente siamo in grado di interagire anche con la voce per fornire un input.

Vista La vista è il principale mezzo con il quale otteniamo informazioni. La percezione visiva si può dividere in due fasi

1. ricezione fisica dello stimolo
2. elaborazione e un'interpretazione dello stimolo

Dobbiamo capire come funziona l'occhio e come la mente interpreta gli stimoli per progettare sistemi informatici. La prima cosa da capire è cosa l'utente vede punto la vista umana tende ad avere una visione centrale dove l'utente di attento e nota le cose; è una visione centrale che trascura le cose che sono fuori da quella centrale.

Capire questo ci permette di realizzare un design utile all'utente

- le info importanti devono essere al centro del campo
- le info meno importanti devono essere più all'esterno

Importante è capire anche come percepiamo i colori, dimensioni, e profondità per una buona progettazione delle interfaccia visive.

Bisogna definire il contesto in cui gli oggetti si trovano perché diamo sulla base di esso o un'interpretazione diversa dovuta al fatto che possiamo già conoscere il contesto e le info relative da conoscenze passate.

Definiamo correttamente il contesto permette all'utente di capire subito di cosa si tratta ed evita le illusioni ottiche che lo portano in disabilità quindi bisogna definire correttamente le dimensioni degli oggetti dove posizionarli e così via per poter sviluppare una buona interfaccia utente.



Udito Un altro canale di output per l'uomo può essere il canale uditivo grazie al quale siamo in grado di riconoscere i suoni in quanto il nostro sistema filtro i suoni che ascoltiamo per concentrarci sulle informazioni importanti il suono è spesso usato nelle interfacce grafiche ad esempio per dare informazioni sullo stato di qualcosa, focalizzare l'attenzione ecc.. quindi per notificare/rafforzare l'idea dell'utente.

Tatto Il tatto ci dà informazioni sull'ambiente circostante e specialmente nei sistemi informatici è la fonte di input ma anche di output. In questo modo siamo in grado di interagire con gli aggettivi fuori dagli smartphone in modo da svolgere determinati compiti. Anche i movimenti importanti perché comporta un elaborazione di informazioni in seguito a uno stimolo ricevuto toccando qualcosa. Ogni movimento richiede tempo che dipende dalle persone dall'età eccetera eccetera. In movimento si valuta con accuratezza cioè la precisione del movimento che dipende anche dalla velocità con il quale si reagisce.

Quanto si progetta un'interfaccia e bisogna considerare queste caratteristiche del movimento per progettare i bottoni che siano facilmente raggiungibili e premibili. (Pubblicità ingannevole)

Il tempo impiegato per colpire un bersaglio è una funzione della dimensione del bersaglio della distanza da percorrere

Legge di Fitts

$$T = a + b \cdot \log_2 \left(1 + \frac{Distanza}{Size} \right)$$

Dispositivi Il progettista dell'interfaccia deve essere a conoscenza delle proprietà del dispositivo e di tutti i fattori che influenzano il comportamento dell'interfaccia perché influenzano la natura dell'interazione.

Donald Norman da una definizione di interazione HC stabilimento che si tratta di un ciclo che si compone di due fasi

1. Esecuzione
2. Valutazione

|
V

L'utente stabilisce l'obiettivo, formula l'intenzione c'è l'obiettivo che deve realizzare, esegue le azioni sul dispositivo e percepisce lo stato del sistema e lo interpreta per capire il posto suggestivo per raggiungere l'obiettivo

È importante che l'utente capisca cosa fare per completare le loro azioni e le interfacce non devono essere di un tralcio anzi devono semplificare il raggiungimento dell'obiettivo.

DESIGN SBAGLIATO → lapsus cioè quando comprendiamo il sistema ma si fanno | errori di distrazione

V

errori quando si sbaglia perché non si è capito il sistema e quindi cosa deve fare per raggiungere l'obiettivo

Ergonomia L'ergonomia è lo studio delle caratteristiche fisiche dell'interazione quindi come progettare i controlli layout dei dispositivi ecc.

L'obiettivo è di migliorare come l'utente usa dispositivi per dare migliori prestazioni sull'uso di questi punto questo si riflette anche sul design, su come posizionare gli elementi sulla base dell'utilità del contesto ecc.

Quando progettiamo l'interfaccia dobbiamo tenere conto anche di come gli utenti usano i dispositivi infatti secondo studi la maggior parte delle persone usano il dispositivo con una mano sola e usano il pollice per toccare lo schermo, oppure con la seconda mano lo fissiamo oppure usano entrambe le mani quindi in base a come viene utilizzato progettiamo il design mettendo gli elementi importanti nelle parti dello schermo più facilmente raggiungibili in base alle modalità di uso mentre gli elementi meno utili o di uso frequente nelle parti più esterne un design corretto porta ad avere una pessima esperienza d'uso per l'utente finale

ANDROID E STRUTTURA APP

Android è un sistema operativo sviluppato da Google basato su Linux e progettato per dispositivi mobile è un sistema operativo open source . Quindi il suo codice sorgente è pubblico e può essere modificato da sviluppatori e produttori di hardware per creare la propria versione di Android e adattarlo a loro dispositivo, Il problema di questo è che ciò porta a una frammentazione eccessiva delle versioni e questo causa problemi quando dobbiamo sviluppare delle applicazioni, perché dobbiamo individuare il numero di dispositivi compatibili con la versione che dovremo sviluppare.

Android cerca di risolvere il problema di compatibilità creando degli appositi librerie che si possono utilizzare con andiamo a sviluppare il codice

Google gestisce il rilascio delle nuove versioni di Android sia per i propri dispositivi, la cui distribuzione avviene in maniera abbastanza rapida sia per gli altri dispositivi di altri produttori. Il sistema operativo di questi seguire. Deve seguire un **processo di qualifica** da parte di Google. Questo vuol dire che ogni qualvolta che Google rilascia un aggiornamento ogni produttore deve adattarlo al proprio hardware del dispositivo e questo richiede molto tempo perché modificano anche l'interfaccia. Una volta che hanno effettuato queste modifiche la versione deve essere approvata da Google per garantire che soddisfi standard di qualità compatibilità e sicurezza se si superano questi test al loro i produttori potranno rilasciare il loro aggiornamento per i loro dispositivi.

ARCHITETTURA

Il sistema Android è interamente sviluppato sopra al kernel Linux. Il motivo di questa scelta è dovuto al fatto che non sarebbero mai stati in grado di sviluppare un sistema completamente funzionante in tempi rapidi per poter entrare nel mercato. Linux era già un sistema operativo pienamente funzionante sotto ogni aspetto e che funzionava in modo molto efficiente e quindi l'idea di Google fu quello di estendere il sistema operativo Linux per adattarlo ai dispositivi mobile.

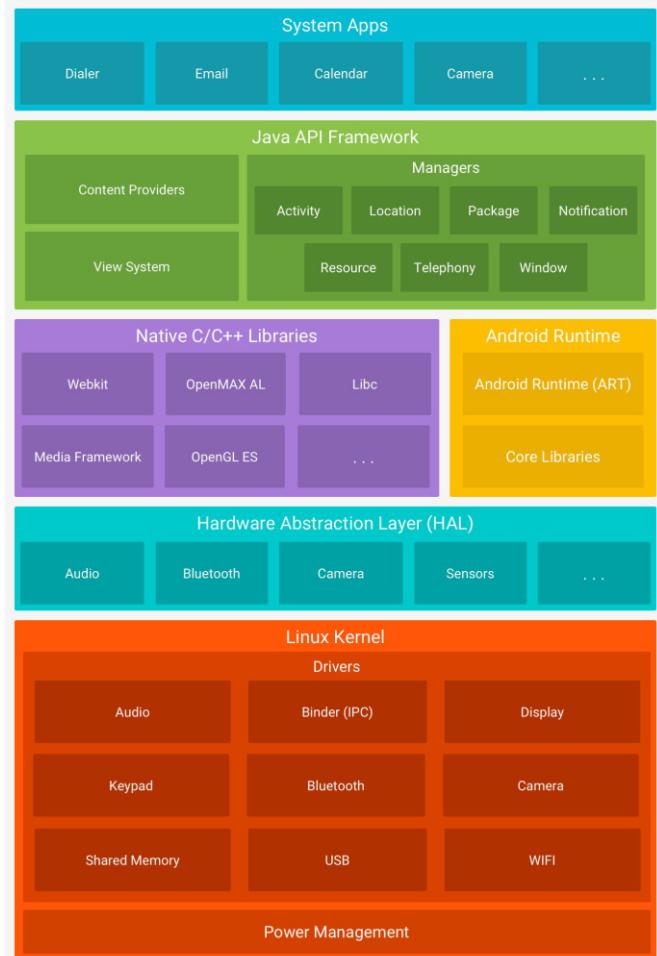
Linux Kernel

Questo livello fornisce i servizi di gestione dell'hardware. A questo livello, viene applicato uno schema di protezione per limitare l'accesso ai dati e alle risorse e consentirlo solo ai processi che posseggono l'adeguato livello di autorizzazione. Vi sono i moduli per la gestione della memoria, dei processi, del sistema di archiviazione e della comunicazione sulla rete. Vi sono i driver per la gestione dell'hardware in dotazione al dispositivo, ad esempio la memoria ausiliaria, la radio, la fotocamera. Oltre ai servizi offerti dal nucleo del sistema operativo, il

kernel di Android include alcuni componenti particolari, quali ad esempio il sistema di risparmio energetico, il sistema di gestione e condivisione della memoria, un meccanismo di comunicazione tra processi chiamato **binder**, che permette ai processi di condividere dati e servizi.

Hardware Abstraction Level

Sopra al kernel troviamo HAL che fornisce interfacce standard per esporre le funzionalità hardware del dispositivo al framework API di livello superiore. HAL è formato da più moduli, ognuno dei quali implementa un'interfaccia per un tipo di componente hardware Linux . quando il framework fa una chiamata per accedere all'hardware del dispositivo Android carica il modulo di libreria di quella componente. I moduli sono scritti dal produttore.



Android RunTime

L'Android Runtime è composto dalla macchina virtuale **ART** (che ha sostituito Dalvik) e dalle librerie principali del sistema. ART è un ambiente di esecuzione in cui ogni app viene eseguita nel proprio processo, con una propria istanza di ART.

ART traduce il codice scritto in Java o Kotlin in bytecode, che viene poi convertito in codice nativo comprensibile dal processore del dispositivo. Utilizza principalmente un approccio di compilazione **Ahead-of-Time (AOT)**: quando si installa una nuova app, questa viene compilata in codice nativo già durante la fase di installazione, migliorando così le performance in fase di esecuzione.

Inoltre, ART può usare la modalità **Just-In-Time (JIT)** per compilare dinamicamente solo alcune parti del codice durante l'esecuzione dell'app, migliorando la reattività e riducendo i tempi di compilazione iniziale.

Native C/C++ Library NDK

Sempre sopra ad HAL troviamo le NDK strumenti per sviluppare app in C/C++. Sono utili per implementare funzionalità che richiedono altre performance o se dobbiamo implementare parte di codice che devono comunicare direttamente con l'hardware.

Java API Framework

Sopra ai livelli di Android Runtime e HAL troviamo il **Java API Framework**, un insieme di librerie, classi e interfacce scritte in Java/Kotlin che permettono di interagire facilmente con le funzionalità hardware e software del dispositivo. Questo livello è progettato per semplificare lo sviluppo delle app, fornendo un accesso ad alto livello alle componenti del sistema operativo Android. Le API mettono a disposizione implementazioni predefinite delle componenti fondamentali del sistema, che possono essere estese e personalizzate dagli sviluppatori per costruire le funzionalità specifiche dell'app.

Applicazioni

Sopra questo livello troviamo le applicazioni dell'utente.

Android SDK

L'**android software development kit ASDK** è un set di librerie e strumenti per sviluppare un app android. I tool si dividono in;

- **SDK platform**, tools platform dependent. Una nuova versione di queste è rilasciata ad ogni versione di android e include librerie, codici sorgenti ecc
- **SDK tools** che sono platform indipendent. Sono strumenti usati per sviluppare , farete bugno a disposizione un emulatore per simulate un dispositivo mobile. Ha un **android Debug Brigde** che permette di comunicare con il dispositivo per eseguire operazioni varie per fare testing e debug.

COMPILAZIONE APP ANDROID

Per quanto riguarda la compilazione di un app android prima di generare il bytecode, le risorse usate nel processo vengono analizzate indicizzate e compilate in bytecode, ottimizzato per la piattaforma punto la componente. La componente che si occupa di fare questo è **AAPT2** che opera il modo **incrementale** cioè compila le risorse e linka tutti i file intermedi generati dalla compilazione. Produce la classe **R.java** che contiene tutte le risorse compilate a cui sono associate un ID con il quale si può far riferimento alla risorsa.

Viene generato il bytecode del codice Java/Kotlin.

La componente **AIDL** definisce l'interfaccia di programmazione concordata dal client e server per comunicare fra loro usando IPC. Il motivo di ciò è che ogni app, anche se viene eseguita sullo stesso dispositivo, è un processo separato e quindi può accedere solo alle zone di memoria che le sono state riservate.

Android usa un meccanismo nativo chiamato **Binder** per gestire la comunicazione tra processi. È una tecnologia di basso livello, integrata nel kernel di Linux, che permette a un processo di:

- Fare richieste a un altro processo
- Ricevere risposte come se fossero normali chiamate di funzione

Con tutte queste parti si viene a creare il .class cioè il file bytecode Java che però non può essere ancora utilizzato ma deve essere ottimizzato dalla componente **DEX/ART** creando un file .dex eseguibile su ART VM.

Una volta pronti i file .dex e le risorse, lo strumento **apkbuilder** li unisce per creare il file .apk, cioè il pacchetto installabile dell'app Android. Dentro questo .apk troviamo:

- I file .dex
- Le risorse compilate (.arsc)
- I file non compilati (come asset, immagini, XML di layout)
- Il file AndroidManifest.xml

Prima che un'app venga installata su un dispositivo, **deve essere firmata digitalmente**. Questo garantisce due cose fondamentali:

- **Autenticità**: conferma che l'app è stata creata da chi dice di averla creata.
- **Integrità**: assicura che il file non sia stato modificato dopo la firma.

a) Firma con jarsigner

La firma avviene usando uno **keystore**, ovvero un contenitore sicuro di chiavi crittografiche. Con jarsigner, il pacchetto .apk viene firmato utilizzando una **chiave privata**.

- Si calcola un **hash** del contenuto dell'app.
- Questo hash viene poi cifrato con la chiave privata → questo è ciò che chiamiamo **firma digitale**.
- La firma viene allegata all'app.

b) Allineamento con zipalign

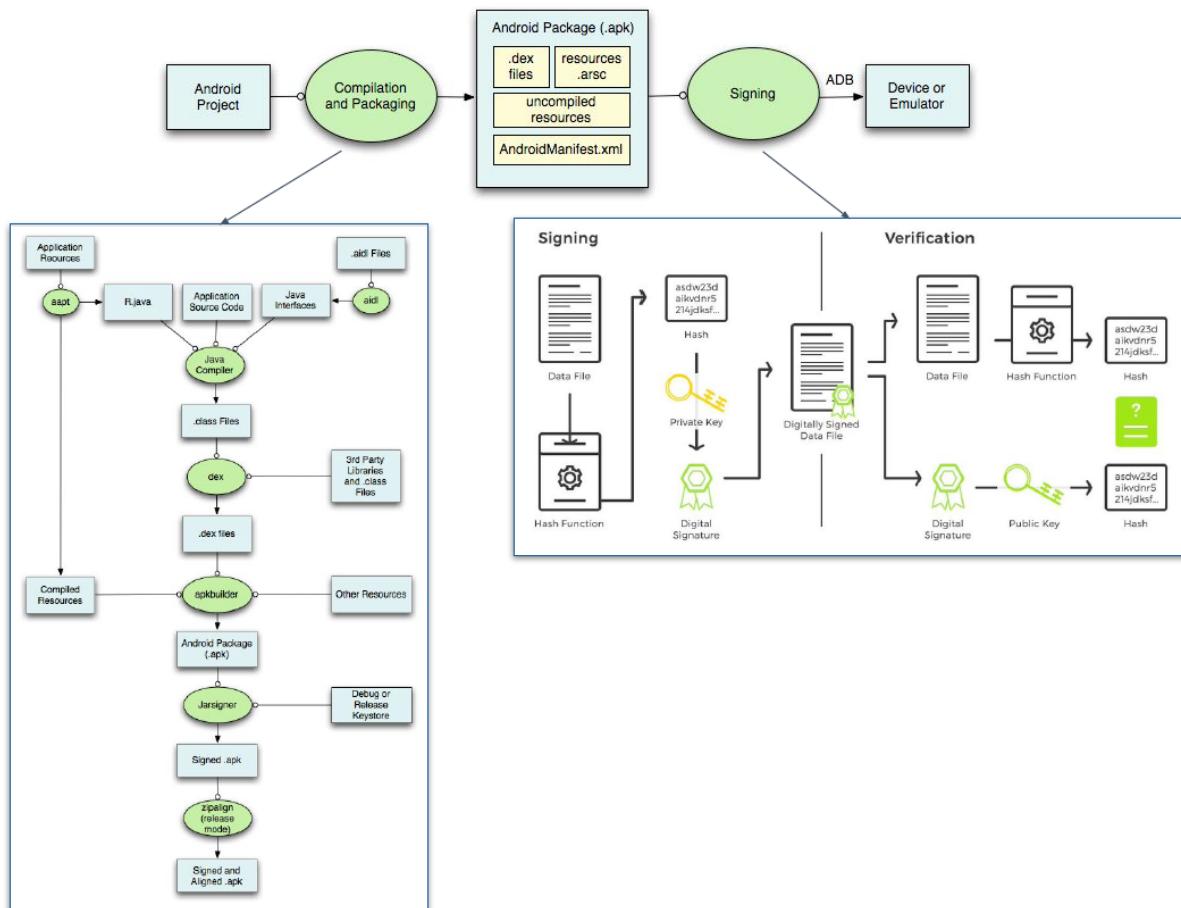
Dopo la firma, si utilizza lo strumento zipalign per **ottimizzare l'accesso alle risorse** del pacchetto. Questo è un passo importante, soprattutto per app in produzione, perché migliora le prestazioni e riduce il consumo di memoria.

Quando un dispositivo Android riceve un'app firmata, esegue la **verifica della firma**:

- Estraе l'hash dal file firmato.
- Ricalcola l'hash con il contenuto dell'app.
- Usa la **chiave pubblica** per verificare che la firma corrisponda all'hash originale.

Se i due hash corrispondono, l'app è autentica e non è stata manomessa → viene installata. Se **non combaciano**, il sistema rifiuta l'installazione.

- Durante lo sviluppo, Android Studio firma automaticamente l'app con una **chiave debug**.
- Per pubblicare l'app sul Play Store, devi firmarla con una **chiave di release** generata da te. È fondamentale non perdere questa chiave, perché ti servirà per aggiornare l'app in futuro.



AVVIO DI UN APPLICAZIONE ANDROID

Il processo di avvio di un'applicazione Android inizia quando l'utente o un altro componente di sistema richiede l'esecuzione di un componente dell'app. Se l'applicazione non è già in esecuzione, il sistema Android avvia un nuovo processo per essa.

Ogni app Android viene eseguita nel proprio processo Android, che è un processo Linux con un thread di esecuzione iniziale.

Il processo di avvio del sistema include l'avvio dell'**init process**, che a sua volta genera processi Linux di basso livello chiamati **daemon**. L'init process avvia un processo chiamato **Zygote** che inizializza la prima istanza della Dalvik Virtual Machine e precarica tutte le classi comuni utilizzate dal framework Android e dalle varie app installate.

In questo modo, si prepara a essere replicato e inizia ad ascoltare su un'interfaccia socket per richieste future di generare nuove macchine virtuali per i processi delle nuove applicazioni. Quando riceve una nuova richiesta, Zygote effettua un fork di se stesso per creare un nuovo processo che ottiene un'istanza VM pre-inizializzata. Dopo Zygote, l'init avvia il **runtime process**, e Zygote esegue un fork per avviare un processo ben gestito chiamato **system server**. Il system server avvia tutti i servizi principali della piattaforma, come l'Activity Manager Service e i servizi hardware, nel proprio contesto.

A questo punto, lo stack completo è pronto per avviare il primo processo dell'app: l'app Home, nota anche come **Launcher**.
Quando un utente fa clic sull'icona di un'app nel Launcher, l'evento di clic viene tradotto in *startActivity(intent)* e viene indirizzato all'**ActivityManagerService** tramite Binder IPC. L'ActivityManagerService esegue diverse operazioni:

1. Raccoglie informazioni sul target dell'oggetto intent utilizzando il metodo `resolveIntent()` sull'oggetto **PackageManager**, con i flag `PackageManager.MATCH_DEFAULT_ONLY` e `PackageManager.GET_SHARED_LIBRARY_FILES` utilizzati per impostazione predefinita.
2. Salva le informazioni sul target nell'oggetto intent per evitare di ripetere questo passaggio.
3. Verifica se l'utente ha privilegi sufficienti per richiamare il componente target dell'intent chiamando il metodo `grantUriPermissionLocked()`.
4. Se l'utente ha le autorizzazioni necessarie, l'ActivityManagerService verifica se l'attività target richiede di essere avviata in un nuovo task, in



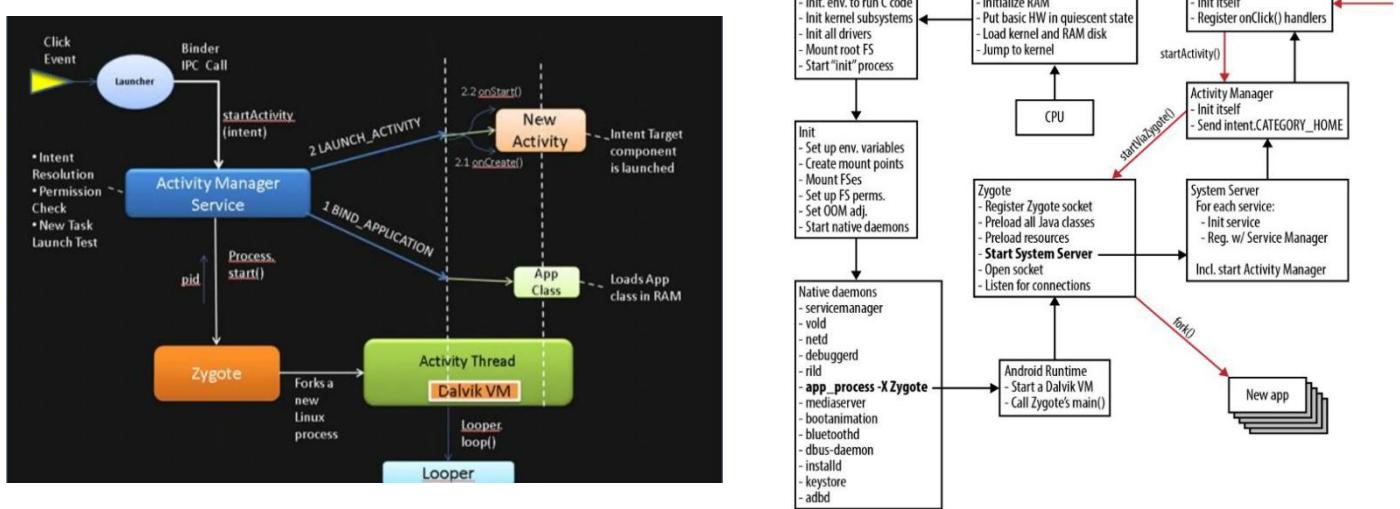
base ai flag dell'Intent come FLAG_ACTIVITY_NEW_TASK e FLAG_ACTIVITY_CLEAR_TOP.

- Verifica se esiste già un **ProcessRecord** per il processo. Se il ProcessRecord è null, l'ActivityManager deve creare un nuovo processo per istanziare il componente target.

L'ASM crea un nuovo processo invocando il metodo `startProcessLocked()`, che invia argomenti al processo Zygote tramite la connessione socket. Zygote esegue un fork di se stesso e chiama `ZygoteInit.main()`, che a sua volta istanzia un oggetto `ActivityThread` e restituisce l'ID del processo appena creato. Ogni processo ottiene un thread per impostazione predefinita. Il thread principale ha un'istanza **Looper** per gestire i messaggi da una coda di messaggi e chiama `Looper.loop()` in ogni iterazione del suo metodo `run()`. Il Looper estrae i messaggi dalla coda e richiama i metodi corrispondenti per gestirli. `ActivityThread` avvia quindi il message loop chiamando `Looper.prepareLoop()` e successivamente `Looper.loop()`.

Il passo successivo consiste nell'associare questo processo appena creato a un'applicazione specifica. Ciò avviene chiamando `bindApplication()` sull'oggetto `thread`. Questo metodo invia un messaggio `BIND_APPLICATION` alla coda dei messaggi. Questo messaggio viene recuperato dall'oggetto **Handler**, che quindi invoca il metodo `handleMessage()` per attivare l'azione specifica del messaggio: `handleBindApplication()`. Questo metodo invoca `makeApplication()`, che carica le classi specifiche dell'app in memoria.

Dopo il binding, il sistema contiene il processo responsabile dell'applicazione con le classi dell'applicazione caricate nella memoria privata del processo. Il processo effettivo di avvio inizia nel metodo `realStartActivity()`, che chiama `scheduleLaunchActivity()` sull'oggetto `thread` dell'applicazione. Questo metodo invia un messaggio `LAUNCH_ACTIVITY` alla coda dei messaggi. Il messaggio viene gestito dal metodo `handleLaunchActivity()`. L'**Activity** inizia il suo ciclo di vita .



ACTIVITY

Un'Activity è una componente visibile di un'applicazione Android che permette agli utenti di interagire con essa. Un'app può essere composta da più Activity, ciascuna delle quali svolge un compito specifico ed è indipendente dalle altre. Ogni Activity è formata da una serie di View, ovvero gli elementi dell'interfaccia utente.

CICLO DI VITA

Il sistema operativo Android gestisce il ciclo di vita di un'Activity, determinando quando questa viene creata, avviata, arrestata o distrutta. La classe Activity fornisce diversi metodi di callback che permettono agli sviluppatori di definire il comportamento dell'applicazione nei vari stati dell'Activity.

L'activity può essere avviata dall'utente che preme sull'icona dell'applicazione, oppure da un'altra activity tramite un intent.

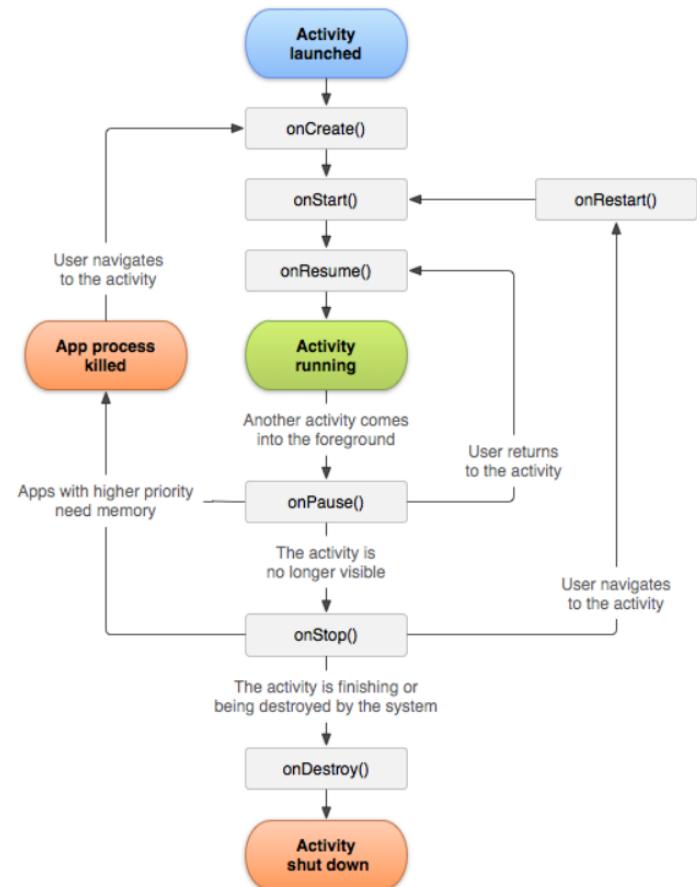
Quando un'app viene avviata Zygote **clona** se stesso per creare un nuovo processo, riducendo il tempo di avvio. Questo nuovo processo eredita le classi e le librerie già caricate, ottimizzando l'esecuzione.

L'ActivityManagerService (AMS) è il componente principale che gestisce il ciclo di vita delle **Activity** e coordina le transizioni tra di esse.

- Riceve la richiesta di avvio dell'app.
- Controlla se il processo dell'app è già in esecuzione o se deve essere creato.
- Comunica con il **Zygote** per avviare un nuovo processo se necessario.

Dopo l'avvio dell'app, vengono chiamate tre metodi per creare un activity.

1. **onCreate()** in questo momento l'activity viene **effettivamente creata** e in questa fase si inizializzano tutte le risorse necessarie. Viene caricato in memoria l'albero delle views che formano l'activity.
La **onCreate()** di un'Activity non deve essere bloccante perché viene



eseguita nel Main Thread, lo **UI Thread**. Questo thread è responsabile della gestione dell'interfaccia utente e dell'interazione con l'utente. Se il codice in `onCreate()` è troppo pesante Android **termina forzatamente** le app che bloccano il Main Thread.

2. Subito dopo, Android chiama il metodo `onStart()` che permette all'Activity di **essere visibile** all'utente, ma non è ancora interattiva. L'interfaccia viene renderizzata visualizzando le componenti che formano l'activity.
3. Con l'invocazione di `onResume()` l'Activity diventa **pienamente attiva e interattiva** e l'utente può effettuare interazioni con le componenti dell'applicazione. Gli eventi generati vengono catturati dal looper e fanno cambiare stato all'activity. Durante questa fase, l'Activity è in primo piano e qualsiasi altra Activity aperta in precedenza è stata messa in pausa o nascosta.

Dopo queste tre operazioni l'applicazione si trova nello stato di running e l'utente può interagire. Se l'utente non interagisce l'activity passa in uno stato di `onPause()`, che indica che l'activity non è più in primo piano, ma questo non vuol dire che debba essere distrutta.

Quando la tua attività non è più visibile all'utente, ma è ancora in memoria viene invocata la chiamata `onStop()` e da qui inizia il ciclo di distruzione e rilascio delle risorse dell'activity.

La differenza tra `onStop()` e `onPause()` è che nella `onPause()` il lavoro relativo all'interfaccia utente continua anche se l'utente sta visualizzando la tua attività in modalità multi-finestra.

Con la `onRestart()` tato transitorio si verifica solo se un'Activity fermata viene riavviata. `onRestart()` viene chiamato tra `onStop()` e `onStart()`.

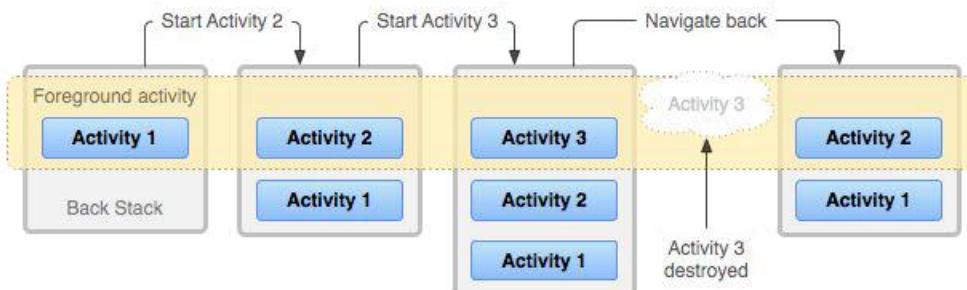
`onDestroy()` viene chiamato prima che l'Activity venga distrutta. Questo può avvenire perché l'Activity sta terminando, perché il sistema sta distruggendo l'Activity per risparmiare risorse, o a causa di un cambiamento di configurazione. È importante eseguire qui la pulizia finale delle risorse. Tuttavia, non si dovrebbe fare affidamento su `onDestroy()` per salvare dati importanti, in quanto potrebbe non essere sempre chiamato.

BACK STACK E NAVIGAZIONE FRA LE ACTIVITY

In Android, le Activity sono gestite attraverso uno stack di Activity, il **back stack**, gestito in modo FIFO.

Il back stack è un insieme di Activity che l'utente ha visitato e a cui può tornare premendo il pulsante Indietro del dispositivo. Una **task** è un insieme di Activity correlate all'applicazione o ad una specifica interazione dell'utente e ha il suo back stack e che possiamo gestire insieme.

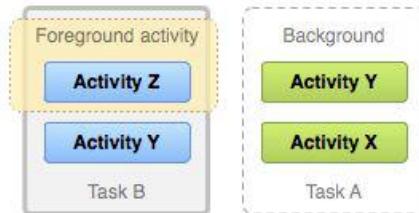
Quando si avvia una nuova Activity, essa viene **inserita in cima** al back stack e prende il focus dell'utente. L'Activity precedente viene fermata ma rimane disponibile nel back stack, conservando il suo stato attuale. Quando l'utente preme il pulsante Indietro, l'Activity in cima allo stack viene **rimossa** (e distrutta), e l'Activity precedente riprende.



Alla base dello stack sta il launcher e da questo si iniziano ad impilare le activity. Android non rimuoverà mai le activity tranne se si trova in situazioni critiche del sistema e allora rimuove prima le activity che si trovano in fondo allo stack, evitando di rimuovere l'activity attiva.

Il back stack **non è specifico di una singola applicazione**; può contenere Activity di diverse applicazioni lanciate dall'utente in ordine cronologico inverso. Ciò significa che premendo il pulsante Indietro, l'utente potrebbe ritrovarsi in un'Activity di un'altra app se l'ha avviata precedentemente.

Un'activity può spostarsi in **background** quando un utente avvia una nuova attività o passa alla schermata Home. In background, tutte le attività dell'attività vengono arrestate, ma lo stack precedente dell'attività rimane intatto, perde l'attenzione mentre è in corso un'altra attività. R l'attività può tornare in primo piano per consentire agli utenti di riprendere da dove hanno interrotto disattivata.



Le activity nel back-stack non vengono mai riorganizzate, ma è possibile modificare questo andamento lineare dello stack perché in certe situazioni questa gestione non va bene. Esistono cinque modalità di lancio che si possono assegnare all'attributo `launchMode`:

- 1. Standard** → La modalità predefinita. Il sistema crea una nuova istanza dell'attività nell'attività da cui è stato avviato e instrada l'intent a quest'ultimo. L'attività può essere creata più volte, ogni istanza può appartenere ad attività diverse un'attività può avere più istanze.

2. **SingleTop** → Se nella parte superiore dell'attività corrente esiste già un'istanza dell'attività, il sistema instrada l'intent a quell'istanza tramite una chiamata alla sua `onNewIntent()` anziché creare una nuova istanza dell'attività. L'attività è creata più volte, ogni istanza può appartenere ad attività diverse e un'attività può avere più istanze (ma solo se l'attività in alto dello stack posteriore *non* è un'istanza esistente dell'attività).
3. **singleTask** → Il sistema crea l'attività alla base di una nuova attività o individua la su un'attività esistente con la stessa affinità. Se un'istanza del componente esiste già un'attività, il sistema instrada all'istanza esistente tramite una chiamata alla sua `onNewIntent()` anziché creare una nuova istanza. Nel frattempo, tutti gli altri vengono distrutte.
4. **singleInstance** → Il comportamento è lo stesso di "singleTask", ad eccezione del fatto che il sistema non avvia nessun altro delle attività nell'attività che contiene l'istanza. L'attività è sempre l'unico e unico membro della sua attività. Tutte le attività iniziate da questa si aprono tra per un'attività a parte.
5. **singleInstancePerTask** → L'attività può essere eseguita solo come attività principale dell'attività, la prima all'attività che ha creato l'attività, perciò può esserci una sola istanza di questa attività in un'attività. A differenza della modalità di avvio di `singleTask`, questa l'attività può essere avviata in più istanze in diverse attività se `FLAG_ACTIVITY_MULTIPLE_TASK` o `FLAG_ACTIVITY_NEW_DOCUMENT` è stato impostato.

SingleTask e **SingleInstancePerTask** rimuovono tutte le attività superiori all'attività iniziale dell'attività.

I **cambiamenti di configurazione**, come la rotazione dello schermo, comportano la distruzione dell'Activity corrente e la creazione di una nuova Activity per adattarsi alla nuova configurazione. Per evitare la perdita di dati durante questi cambiamenti, è possibile **salvare lo stato dell'istanza dell'Activity**.

1. sovrascrivendo il metodo **onSaveInstanceState(Bundle)**. Lo stato viene salvato come una serie di coppie chiave/valore in un oggetto **Bundle** che viene passato a **onCreate()** quando l'Activity viene ricreata. È anche possibile utilizzare il callback `onRestoreInstanceState()` per ripristinare lo stato.
2. Oppure con il **model-view-controller**

FRAGMENT

Un **Fragment** è un componente utilizzato per dividere l'interfaccia utente in parti più piccole. Un Fragment rappresenta una porzione di UI o di comportamento che vive all'interno di un'Activity, e può essere riutilizzato o sostituito dinamicamente. Un Fragment riceve i propri eventi di input e ha una propria vista che viene composta da un file di layout apposito. I Fragment possono esistere solo all'interno di un'Activity. Un'attività può **delegare** a un fragment l'esecuzione di compiti.

Le Activity fungono da **host** per i Fragment queste devono conoscere i dettagli di come ospitare i loro Fragment, ma i Fragment non devono conoscere i dettagli delle Activity che le ospitano (**loosely couple**). Come le attività, i fragment sono "attivi" quando appartengono ad una activity focalizzata e in primo piano.

Quando un'attività viene messa in pausa o interrotta, anche i frammenti che contiene vengono messi in pausa e arrestati e anche i frammenti contenuti in un'attività inattiva sono inattivi.

Quando un'attività viene finalmente distrutta, ogni Fragment che contiene viene ugualmente distrutto. Poiché il gestore della memoria Android chiude regolarmente le applicazioni per liberare risorse, anche i frammenti all'interno di tali attività vengono distrutti.

Non dovrebbe esserci alcuna differenza nel passaggio di un frammento da uno stato scollegato, in pausa, interrotto o inattivo allo stato attivo, quindi è importante salvare tutto lo stato dell'interfaccia utente e conservare tutti i dati quando un frammento viene sospeso o interrotto. Come un'activity, quando un frammento diventa di nuovo attivo, dovrebbe ripristinare lo stato salvato.

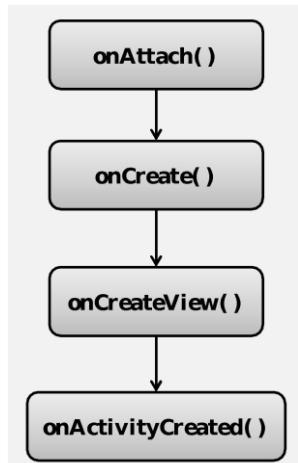


AUF, Always Use Fragments

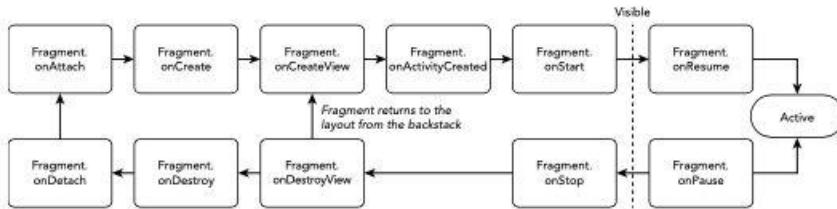
I Fragment hanno anche un proprio **ciclo di vita** che è simile a quello di un'attività. Questo è importante perché, visto che un fragment lavora per conto di un'attività, il suo stato dovrebbe riflettere lo stato dell'attività. Pertanto, ha bisogno di metodi del ciclo di vita corrispondenti per gestire il lavoro dell'attività.

I callback del ciclo di vita di un Fragment includono

- **onCreateView()**: il valore restituito di questo metodo deve essere un'istanza di View affinché il Fragment abbia un'interfaccia utente visibile
- **onDestroyView()**: Ci sono anche callback per onActivityCreated e callback che vengono attivati quando un Fragment viene aggiunto (onAttached) a o rimosso (onDetached) dall'UI utilizzando i metodi di FragmentTransaction.



- I metodi `onAttach(Activity)`, `onCreate(Bundle)` e `onCreateView(...)` vengono chiamati quando si aggiunge il fragment al FragmentManager.
- Il metodo `onActivityCreated(...)` viene chiamato dopo che il metodo `onCreate(...)` dell'attività ospitante è stato eseguito.
- Il FragmentManager di un'attività è responsabile della chiamata dei metodi del ciclo di vita dei fragment nella sua lista.



La vista di un fragment viene generalmente inflata nel metodo `onCreateView()`. Per aggiungere un fragment a un'attività nel codice, si effettuano chiamate esplicite al **FragmentManager** dell'attività che è responsabile della gestione dei fragment e dell'aggiunta delle loro viste alla gerarchia delle viste dell'attività.

Il FragmentManager gestisce due cose:

- un elenco di fragment
- stack di back delle transazioni di fragment.

Per ottenere il FragmentManager, si usa

- `getSupportFragmentManager()`, se si estende **AppCompatActivity** o **FragmentActivity**;
- `getFragmentManager()`, se si estende **Activity** e il fragment estende `android.app.Fragment` o `PreferenceFragment`).

Le **FragmentTransaction** vengono utilizzate per aggiungere, rimuovere, allegare, staccare o sostituire i fragment nell'elenco dei fragment in fase di runtime. Il FragmentManager mantiene uno stack di back delle transazioni di fragment su cui è possibile navigare.

Il metodo `FragmentManager.beginTransaction()` crea e restituisce un'istanza di **FragmentTransaction**. La classe `FragmentTransaction` utilizza una **fluent interface**: i metodi che configurano `FragmentTransaction` restituiscono un `FragmentTransaction` invece di void, il che consente di concatenarli.

- Il metodo `add()` della `FragmentTransaction` ha due parametri: un **ID del container view** e il **newFragment**. L'**ID del container view** è l'**ID risorsa** del `FrameLayout` definito nel layout dell'attività. Un **ID del container view** indica al FragmentManager dove nella vista dell'attività dovrebbe apparire la vista del fragment e viene utilizzato come identificatore univoco per un fragment nell'elenco del FragmentManager.

Quando è necessario recuperare il Fragment dal FragmentManager, lo si richiede tramite l'ID del container view utilizzando `fm.findFragmentById(R.id.fragment_container)`.

È prassi comune aggiungere un **metodo statico newInstance()** alla classe Fragment. Questo metodo crea l'istanza del fragment, raggruppa e imposta i suoi argomenti.

Stato di un Fragment

Ogni istanza di fragment può avere un oggetto **Bundle** allegato. Questo bundle contiene coppie chiave-valore,

Per creare gli argomenti di un fragment, si crea prima un oggetto Bundle. Quindi, si utilizzano i metodi "put" specifici del tipo di Bundle (simili a quelli di Intent) per aggiungere argomenti al bundle.

Per allegare il bundle degli argomenti a un fragment, si chiama

`Fragment.setArguments(Bundle)`. L'allegamento degli argomenti a un fragment deve essere fatto dopo che il fragment è stato creato ma prima che venga aggiunto a un'attività.

Quando un fragment deve accedere ai suoi argomenti, chiama il metodo `getArguments()` del Fragment e quindi uno dei metodi "get" specifici del tipo di Bundle.

Comunicazione tra Activity e Fragment

Per delegare funzionalità all'attività ospitante, un fragment definisce in genere un'**interfaccia di callback** denominata **Callbacks**. Questa interfaccia definisce il lavoro che il fragment deve far svolgere all'attività ospitante.

Qualsiasi attività che ospiterà il fragment deve implementare questa interfaccia. Con un'interfaccia di callback, un fragment è in grado di chiamare metodi sulla sua attività ospitante senza dover sapere nulla di quale attività lo stia ospitando.

L'attività viene assegnata nel metodo del ciclo di vita del Fragment:

`onAttach(Activity activity)`. Questo metodo viene chiamato quando un fragment viene collegato a un'attività, sia che sia stato conservato o meno.

Allo stesso modo, la variabile viene impostata su null nel corrispondente metodo del ciclo di vita in diminuzione: `onDetach()`. La variabile viene impostata su null qui perché in seguito non è possibile accedere all'attività o contare sulla sua continua esistenza.

Conservazione dei Fragment

Per impostazione predefinita, la proprietà `retainInstance` di un fragment è false. Ciò significa che non viene conservato e viene distrutto e ricreato alla rotazione insieme all'attività che lo ospita. La chiamata a `setRetainInstance(true)` conserva il fragment. Quando un fragment viene

conservato, il fragment non viene distrutto con l'attività. Invece, viene preservato e passato intatto alla nuova attività.

Quando si conserva un fragment, si può contare sul fatto che tutte le sue variabili d'istanza mantengano gli stessi valori.

Un fragment conservato non viene distrutto, ma viene **staccato dall'attività morente**. Questo mette il fragment in uno **stato di conservazione**. Il fragment esiste ancora, ma non è ospitato da alcuna attività. Lo stato di conservazione viene raggiunto solo quando si verificano due condizioni:

1. `setRetainInstance(true)` è stato chiamato sul fragment
2. l'attività ospitante viene distrutta per un cambio di configurazione

Riceverà l'evento **onDetach** quando l'attività padre viene distrutta, seguito dagli eventi **onAttach**, **onCreateView** e **onActivityCreated** quando viene creata un'istanza della nuova attività padre.

SERVICE

Un **service** in Android è un componente applicativo che esegue operazioni a lunga esecuzione, solitamente in background, senza fornire un'interfaccia utente (UI).

Un'applicazione può avviare un service anche se non è in primo piano e un service può continuare a essere eseguito anche dopo che l'attività che lo ha avviato è stata distrutta.

Un service può essere **avviato**, **collegato** (bound), o **entrambi**.

- Un **service avviato** è quello che un componente applicativo (come un'activity) avvia chiamando `startService()`. I service avviati vengono utilizzati per eseguire **task in background** che possono durare a lungo o per eseguire lavoro per processi remoti. Una volta avviato, un service può continuare a funzionare indefinitamente, anche se il componente che lo ha avviato viene distrutto. Solitamente, un service avviato esegue una singola operazione e non restituisce un risultato al chiamante. Quando l'operazione è completata, il service dovrebbe **fermarsi da solo chiamando `stopSelf()`**, oppure un altro componente può fermarlo chiamando `stopService()`.
- Un **service collegato** (bound) è un service a cui un componente applicativo si connette chiamando `bindService()`. I service collegati forniscono un'**interfaccia client-server** che permette ai componenti di interagire con il service, inviare richieste e ottenere risultati, a volte utilizzando la comunicazione interprocesso (IPC). Un service collegato è in esecuzione solo finché un altro componente applicativo è collegato ad esso. Più componenti possono collegarsi allo stesso service

contemporaneamente, ma quando tutti si disconnettono, il service viene distrutto. Un service collegato generalmente non consente ai componenti di avviarlo chiamando startService().

CICLO DI VITA DI UN SERVICE

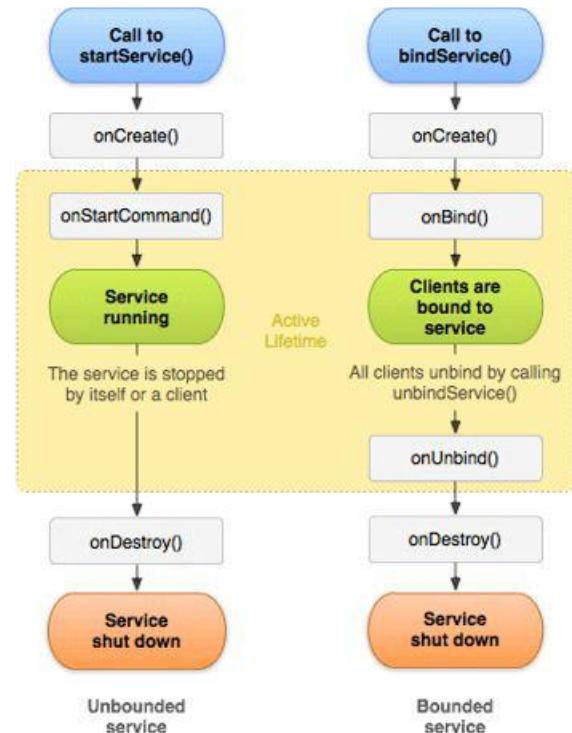
Il ciclo di vita di un service è più semplice di quello di un'activity. Tuttavia, è fondamentale prestare attenzione a come un service viene creato e distrutto, poiché un service senza UI può continuare a funzionare in background senza che l'utente ne sia a conoscenza, consumando risorse e batteria.

Come un'activity, un service ha dei **metodi di callback del ciclo di vita** che è possibile implementare per monitorare i cambiamenti nello stato del service ed eseguire operazioni nei momenti appropriati.

onCreate(): Questo metodo viene chiamato quando il service viene creato per la prima volta. Viene chiamato una sola volta per la durata del service. Qui è dove si dovrebbero eseguire le inizializzazioni di base, come la creazione di thread o la registrazione di listener.

onStartCommand(Intent intent, int flags, int startId): Questo metodo viene chiamato ogni volta che un componente avvia il service chiamando startService(). Riceve l'Intent fornito dal client (il componente che ha chiamato startService()) e due interi: flags, che forniscono informazioni aggiuntive sulla richiesta di avvio, e startId, un identificatore univoco per questa particolare richiesta di avvio. È in questo metodo che il service esegue il lavoro richiesto dall'Intent. Il valore restituito da onStartCommand() indica al sistema come dovrebbe comportarsi se il service viene interrotto inaspettatamente.

onBind(Intent intent): Questo metodo viene chiamato quando un client desidera collegarsi al service chiamando bindService(). Restituisce un oggetto IBinder che il client utilizza per comunicare con il service. I service che intendono essere collegati devono implementare questo metodo; tuttavia, un service può essere sia avviato che collegato (in tal caso implementerà sia onStartCommand() che onBind()). Un service collegato generalmente non implementa onStartCommand().



- `onUnbind(Intent intent)`: Questo metodo viene chiamato **quando tutti i client collegati al service si sono disconnessi** (hanno chiamato `unbindService()`). Il valore booleano restituito indica se `onRebind()` debba essere chiamato quando un nuovo client si collega al service.
- `onRebind(Intent intent)`: Questo metodo viene chiamato **quando un nuovo client si collega al service dopo che `onUnbind()` è già stato chiamato**. Viene chiamato se `onUnbind()` ha restituito true.
- `onDestroy()`: Questo metodo viene chiamato **quando il service non è più in uso e sta per essere distrutto**. Questo può avvenire quando il service si ferma da solo chiamando `stopSelf()` o quando un altro componente chiama `stopService()`, oppure quando tutti i client collegati si disconnettono (per i service collegati). È qui che si dovrebbero rilasciare tutte le risorse utilizzate dal service (come thread, listener, ecc.).

È importante notare che i service avviati continuano la loro esecuzione indipendentemente dal componente che li ha avviati e devono essere fermati esplicitamente. I service collegati, invece, esistono solo per servire il componente applicativo che è collegato ad essi e vengono distrutti quando non ci sono più componenti collegati.

Come le activity e altri componenti, **tutti i service devono essere dichiarati nel file manifest** dell'applicazione. Per dichiarare un service, è necessario aggiungere un elemento `<service>` come figlio dell'elemento `<application>`. Comprendere il ciclo di vita e le diverse tipologie di service è fondamentale per sviluppare applicazioni Android efficienti che eseguono operazioni in background in modo appropriato, senza impattare negativamente sull'esperienza utente o sulle risorse del dispositivo.

Quando un servizio è in esecuzione, può notificare gli eventi all'utente utilizzando le **notifiche snackbar** o le **notifiche della barra di stato**. Una notifica nella barra di app è un messaggio che viene visualizzato sulla superficie della finestra corrente solo per un breve istante prima di scomparire. Una notifica nella barra di stato fornisce un'icona con un messaggio, che l'utente può selezionare per eseguire un'azione (ad esempio avviare un'attività). In genere, una notifica nella barra di stato è la tecnica migliore da utilizzare quando un'operazione in background, come il download di un file, è stata completata e l'utente può ora intervenire. Quando l'utente seleziona la notifica dalla visualizzazione espansa, la notifica può avviare un'attività (ad esempio per visualizzare il file scaricato).

BROADCAST RECEIVER

Un **Broadcast Receiver** è un componente Android progettato per **rispondere a** messaggi provenienti da altre applicazioni o dal sistema operativo. Per implementare un broadcast receiver si estende la classe `BroadcastReceiver` e implementare il metodo `onReceive(Context context, Intent intent)` dal quale estraiamo l'evento di intent, i dati ecc.

Questi messaggi sono incapsulati negli intent per poter attraversare le applicazioni. Le app possono registrarsi per ascoltare specifici tipi di intent broadcast; quando un intent di quel tipo viene inviato, il sistema notifica il receiver in modo che possa eseguire un'azione.

Gli **intent broadcast** non sono indirizzati a destinatari specifici, ma a tutte le app interessate che hanno registrato un receiver per quel tipo di intent lo riceveranno. Le app anche in background possono ricevere messaggi dai broadcast receiver e svolgere delle attività.

Esistono due tipi di intent broadcast: quelli **inviai dal sistema (system broadcast intents)** e quelli **inviai dalla tua app (custom broadcast intents)**.

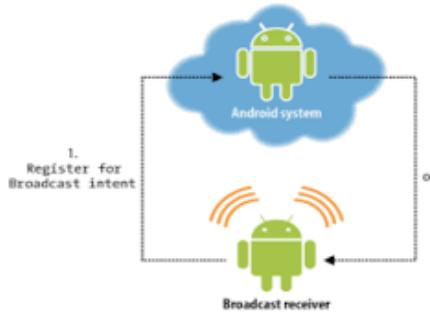
1. Gli **intent broadcast di sistema** vengono inviati quando si verifica un evento di sistema che potrebbe interessare la tua app, come l'avvio del dispositivo, la connessione o disconnessione dall'alimentazione ecc.
2. Gli **intent broadcast personalizzati** vengono utilizzati quando vuoi che la tua app esegua un'azione senza avviare un'attività, ad esempio per informare altre app che sono stati scaricati dei dati.

Ci sono due modi per registrare un broadcast receiver:

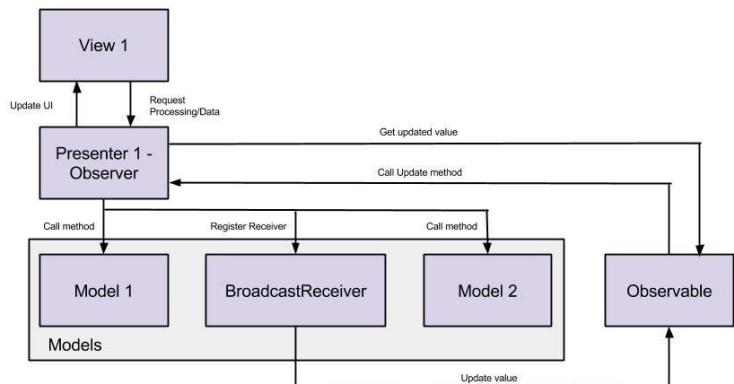
1. **staticamente** nel file manifest aggiungendo un elemento `<receiver>` al file `AndroidManifest.xml` e usare il percorso della tua sottoclasse `BroadcastReceiver` come attributo `android:name`.
I receiver registrati staticamente possono essere attivati anche se il processo della tua app non è in esecuzione e può accedere ad ogni eventi di sistema.

Infatti Android si accorse che questo causava problemi di sicurezza, visto che potevano essere inseriti dei malware quando si verificavano determinati eventi di sistema. Allora definirono una registrazione

2. **dinamica** sulla base del contesto con cui si chiamava il broadcast receiver. La registrazione avviene chiamando il metodo `registerReceiver()` e passando un oggetto `BroadcastReceiver` e un `IntentFilter`. I receiver registrati dinamicamente sono legati al ciclo di vita del componente in cui sono registrati, infatti è necessario gestire anche la loro cancellazione perché potrebbero contenere dati che sprecherebbero risorse



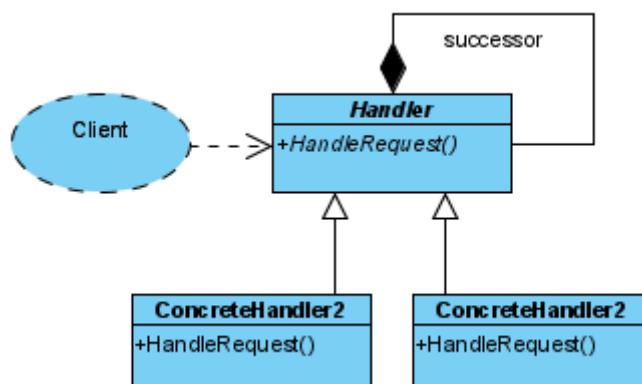
Il pattern che viene usato è
l'observer con tassonomia ad eventi



MODALITÀ DI INVIO DEI BROADCAST

Broadcast Normali, con cui il broadcast intente è prodotto da un singolo produce e viene inviato a tutti i receiver contemporaneamente. Essendo che vengono mandati sul thread principale, i receiver non vengono eseguiti in parallelo, quindi non è possibile fare affidamento su un ordine di esecuzione particolare né sapere quando tutti i receiver hanno completato l'esecuzione. Vengono inviati tramite il metodo `sendBroadcast()`. I receiver sono indipendenti fra loro e quindi non possono scambiarsi informazioni fra loro.

Broadcast Ordinati, sono principalmente eventi di sistema, e questi sono mandati ad una sequenza di broadcast che quando ricevono l'intent verificano se lo possono risolvere o se ne hanno autorizzazione a risolverlo, altrimenti viene mandato al broadcast successivo fino ad arrivare alla fine della sequenza dove viene lanciata un'eccezione per segnalare che non è stato possibile risolverlo. È possibile anche assegnare delle priorità, tramite `android:priority` per modificare l'ordine di esecuzione. Conviene nella catena mettere sempre quelli più specifici per quello che si deve fare. Questo implementa il **pattern di catena di responsabilità**.



LocalBroadcastManager Questi rimangono nell'app e quindi non hanno bisogno di intent e né della registrazione nel manifest. Si ottiene un'istanza singleton chiamando `getInstance(context)`, che è thread-safe. Per inviare un broadcast locale si usa il suo metodo `sendBroadcast()`, e per registrare un receiver si usa `registerReceiver()`. I receiver locali devono essere registrati dinamicamente nel codice; la registrazione statica nel manifest non è disponibile per **LocalBroadcastManager**. È importante annullare la registrazione del receiver locale quando non è più necessario per evitare memory leak.

Custom Broadcast sia il producer che il receiver devono avere lo stesso nome e devono avere un URI di riferimenti che deve essere univoco, infatti si usa il FQN del package in cui si trova

RESTIZIONE DEI BROADCAST

Bisogna evitare che altre applicazione estraggano dati dei broadcast. Infatti il sistema Android non è multiutente, ma per dare maggiori sicurezza android definisce delle linee guida per evitare minacce alla sicurezza. È necessario creare un sistema di permessi valido, implementando `setpackage()` per far consumare i messaggi broadcast solo da package specifici. I permessi che devono essere richiesti all'utente devono esser specificati nel manifest e non si possono mandare messaggi non confinati, l'unico che può è Android, che può mandare permessi null, cioè non confinati.

È possibile imporre permessi per il ricevente e lo si fa con `registerReceiver()` e nel manifest dobbiamo specificare nel `BroadcastReceiver android:permission` con i permessi di cui ha bisogno. Quando si hanno bisogno dei permessi, si chiede all'utente di darli o meno. Se l'utente accetta, allora il BroadcastReceiver sarà registrato. Non si devono fare operazioni long run perché impegnerebbero troppo il sistema, quindi una volta che abbiamo acquisito i dati nella `onReceiver()`, liberiamo la callback e poi elaboriamo in modo asincrono. (code)

CONTENET PROVIDER

Un **Content Provider** è un componente di Android che gestisce l'accesso a un repository di dati. L'applicazione non ha bisogno di sapere dove o come i dati sono archiviati, formattati o acceduti.

Un Content Provider separa i dati dal codice dell'interfaccia utente dell'app, fornendo un **modo standard** per accedere ai dati. Rende possibile la condivisione di dati tra diverse applicazioni. (Pattern Adapter)

Consente a più applicazioni di accedere, utilizzare e modificare in modo sicuro una singola origine dati fornita dalla tua app.

Per ottenere dati e interagire con un Content Provider, un'app utilizza un **ContentResolver** per inviare richieste al Content Provider.

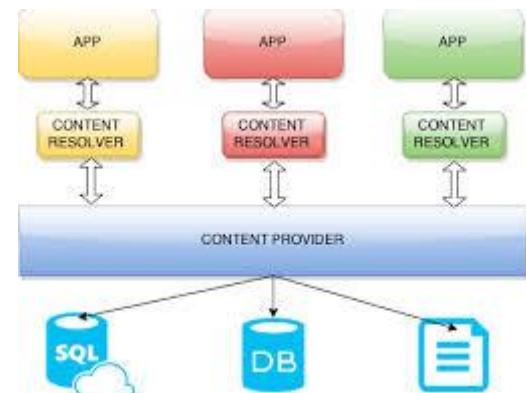
L'applicazione di un fornitore può specificare le autorizzazioni che altre applicazioni devono avere per accedere ai dati del fornitore. Queste autorizzazioni consentono all'utente di sapere a quali dati un'applicazione tenta di accedere. In base ai requisiti del fornitore, altre applicazioni richiedono le autorizzazioni necessarie per accedere al fornitore.

Gli utenti finali vedono le autorizzazioni richieste quando installano l'applicazione.

Se l'applicazione di un provider non specifica alcuna autorizzazione, le altre applicazioni non avranno accesso ai dati del provider, a meno che il provider non venga esportato.

Per rendere disponibile un ContentProvider all'interno di un'app Android, è necessario **dichiararlo esplicitamente** nel file `AndroidManifest.xml`, all'interno del tag `<application>` tramite l'elemento `<provider>`, che contiene una serie di attributi fondamentali.

- **android:name**, che specifica il nome completo della classe che implementa il ContentProvider.
- **android:authorities**, che rappresenta un identificatore **univoco** del provider. Questo valore viene utilizzato dalle app per costruire gli URI di accesso al provider. Per esempio, se l'authority è `com.miaapp.provider`, l'URI per accedere a una risorsa potrebbe essere `content://com.miaapp.provider/risorse`.
- **android:exported**. Questo valore determina se il provider può essere **accessibile da altre app**. Se impostato su `true`, il provider potrà essere utilizzato anche da app esterne, mentre se è `false`, sarà accessibile solo dall'interno dell'applicazione stessa. Questo è un parametro importante per la **sicurezza**, specialmente se i dati gestiti sono sensibili.



È anche possibile specificare dei **permessi di accesso** per controllare quali app possono leggere o scrivere dati tramite il provider. Questo si fa aggiungendo gli attributi `android:readPermission` e `android:writePermission`, dove si indicano i nomi dei permessi richiesti. Le app che vogliono accedere al provider dovranno dichiarare nel proprio manifest questi permessi, e l'utente li vedrà al momento dell'installazione.

L'architettura di un Content Provider include i seguenti componenti:

- **Data and Open Helper:** Il repository dei dati. Comunemente i dati sono archiviati in un database SQLite, ma possono essere anche file, dati sul web o dati generati dinamicamente. Per i database SQLite, si utilizza spesso un `SQLiteOpenHelper` per l'accesso ai dati.
- **Contract:** Una classe pubblica che espone informazioni importanti sul Content Provider ad altre app. Include solitamente gli schemi URI, costanti importanti e la struttura dei dati restituiti. L'uso di un contract separa le informazioni pubbliche da quelle private e fornisce un unico punto di riferimento per le altre app.
- **Content Provider:** Una classe che estende `ContentProvider` e implementa i metodi `query()`, `insert()`, `update()` e `delete()` per accedere ai dati. Fornisce un'interfaccia pubblica e sicura ai dati.
- **Content Resolver:** Un oggetto utilizzato dalle app per inviare richieste al Content Provider e ottenere i dati. Fornisce i metodi `query()`, `insert()`, `update()` e `delete()` che rispecchiano quelli del Content Provider.

Per implementare un Content Provider, è necessario:

- Avere i **dati** e un modo per accedervi
- Dichiarare il **Content Provider** nel file `AndroidManifest.xml` per renderlo disponibile. È importante impostare l'attributo `android:exported="true"` se si desidera che altre app possano accedervi.
- Creare una sottoclasse di `ContentProvider` che implementi i metodi per l'accesso e la manipolazione dei dati (`query()`, `insert()`, `delete()`, `update()`, `getType()`).
- Definire una classe **Contract** pubblica che esponga l'**URI scheme**, i nomi delle tabelle, i **tipi MIME** e altre costanti importanti.

URI e tipi MIME

Le app inviano richieste ai Content Provider utilizzando gli **Uniform Resource Identifier (URI)**. Un content URI ha la forma generale `scheme://authority/path/ID`, dove lo scheme è sempre `content://`, l'authority rappresenta il dominio del provider (solitamente il nome del package che

termina con .provider), il path è il percorso ai dati e l'ID identifica univocamente un set di dati. Il contract definisce costanti per l'AUTHORITY, il CONTENT_PATH e il CONTENT_URI.

Il **tipo MIME** indica il tipo e il formato dei dati restituiti dal Content Provider. Per i Content URI che puntano a righe di una tabella, si utilizzano tipi MIME specifici del vendor Android, con il formato generale type.subtype/provider-specific-part. Il type è vnd, il subtype è android.cursor.item/ per una singola riga e android.cursor.dir/ per più righe, e la parte specifica del provider include solitamente il nome del package e il nome della tabella. Il metodo getType() del Content Provider restituisce il tipo MIME dei dati per un dato URI. Il ContentResolver fornisce metodi (query(), insert(), delete(), update()) che corrispondono a quelli implementati nel ContentProvider.

- Il metodo query() viene utilizzato per recuperare dati e restituisce un oggetto **Cursor**, che è un puntatore a una riga di dati strutturati in formato tabellare, simile a un risultato di una query SQL. La query può includere una proiezione (le colonne da restituire), una clausola di selezione (il WHERE), gli argomenti di selezione e l'ordine di ordinamento (ORDER BY).
- I metodi insert(), delete() e update() vengono utilizzati per modificare i dati. Il metodo insert() riceve i valori da inserire come ContentValues e restituisce l'URI della nuova riga.

È buona pratica utilizzare un UriMatcher per gestire il matching degli URI all'interno del Content Provider, associando ogni URI supportato a un codice intero.

INTENT

In Android, un **Intent** è un meccanismo IPC per comunicare con il sistema operativo e fra le applicazioni. Gli Intent sono messaggi che effettuano una richiesta all'Android runtime per avviare un'activity o un altro componente dell'app o di un'altra app. Invece di avviare direttamente le activity, si costruiscono Intent con la classe Intent e si chiama il metodo **startActivity()** per inviare l'intent.

Un **Intent esplicito** specifica l'activity (o un altro componente) ricevente tramite il suo nome di classe completo. Si utilizzano Intent esplicativi per avviare componenti all'interno della propria applicazione. Per creare un Intent esplicito, si utilizza un Context e un oggetto Class. Il metodo **newIntent()** può essere utilizzato per configurare correttamente un Intent esplicito con gli extra necessari. L'activity ricevente ottiene l'Intent con **getIntent()** e recupera i dati dagli extra. È anche possibile avviare un'activity aspettandosi un risultato

utilizzando `startActivityForResult(Intent, requestCode)`. L'activity chiamata invierà un risultato tramite un altro Intent, e l'activity chiamante riceverà il risultato nel metodo `onActivityResult()`. Una Activity avviata tramite `startActivity` è indipendente da quella chiamante e non fornirà alcun dato di risposta alla chiusura.

Laddove è richiesto un feedback, è possibile avviare un'attività come subactivity che può restituire i risultati al suo genitore. Per fare questo si utilizza `startActivityForResult()`. Quando la sub-activity è pronta per essere restituita, si utilizza `setResult` prima della fine per restituire un risultato all'attività chiamante. Il metodo `setResult` accetta due parametri: **il codice del risultato e i dati del risultato stesso, rappresentati come un Intent**.

Il codice del risultato indica il successo dell'esecuzione dell'attività secondaria (`RESULT_OK`) o insuccesso (`Activity.RESULT_CANCELED`).

Un **Intent implicito** non specifica un'activity o un altro componente specifico per ricevere l'intent. Invece, si dichiara un'azione generica da eseguire nell'intent. Il sistema Android abbina la richiesta a un'activity o a un altro componente in grado di gestire l'azione richiesta (**late binding**). Se più activity corrispondono, all'utente viene presentata una finestra di dialogo di selezione dell'app. Le activity dichiarano la loro capacità di gestire Intent impliciti tramite i **filtri di intent** definiti nel file `AndroidManifest.xml`. Per inviare un **Intent implicito**, si crea un oggetto Intent specificando l'azione, i dati (se presenti) e il tipo MIME dei dati. Prima di chiamare un Intent implicito, è buona pratica verificare se ci sono activity in grado di gestirlo chiamando `resolveActivity(getApplicationContext())`. Se si desidera mostrare sempre all'utente un chooser di app quando più app possono gestire l'intent, si può utilizzare `Intent.createChooser(Intent target, String title)`.

Il passaggio da un'Activity ad un'altra coinvolge i cicli di vita di entrambe. La prima, quella messa a riposo, dovrà passare almeno per `onPause()` e `onStop()` mentre la seconda percorrerà la catena di creazione `onCreate-onStart-onResume`. La priorità del sistema è **il mantenimento della fluidità della user-experience**, quindi:

- La **prima Activity passa per `onPause`** e viene fermata in stato Paused;
- La **seconda Activity va in Running** venendo attivata completamente. In tale maniera l'utente potrà usarla al più presto non subendo tempi di ritardo;
- a questo punto, mentre l'utente sta già usando la seconda Activity, il sistema può invocare **`onStop sulla prima`**.

FILTRI

I **filtri di intent** sono dichiarazioni nel file manifest di un componente dell'app (solitamente un'attività, un servizio o un ricevitore di broadcast) che specificano i tipi di intent che il componente può gestire.

I filtri di intent sono fondamentali per il funzionamento degli **intent impliciti**. Quando un'applicazione invia un intent implicito, il sistema Android utilizza i filtri di intent dichiarati da tutte le app installate per determinare quale componente è il più adatto a gestire la richiesta.

I filtri di intent sono dichiarati nel file `AndroidManifest.xml` all'interno dell'elemento `<activity>`, `<service>` o `<receiver>` usando uno o più elementi `<intent-filter>` che può contenere tre tipi di elementi che corrispondono alle informazioni contenute in un oggetto Intent:

- `<action>`: Specifica l'**azione** generica che il componente può eseguire. Le azioni sono definite come costanti nella classe Intent. Un intent implicito deve contenere un'azione che corrisponda ad almeno una delle azioni dichiarate nel filtro. È possibile specificare più azioni all'interno dello stesso filtro.
- `<category>`: Fornisce informazioni aggiuntive sulla **categoria** del componente che dovrebbe gestire l'intent. Le categorie sono definite come costanti nella classe Intent. Un intent deve corrispondere a tutte le categorie specificate in un filtro per passare. È importante notare che tutte le attività che intendono ricevere intent impliciti devono includere la categoria `android.intent.category.DEFAULT`, in quanto questa categoria viene aggiunta implicitamente a tutti gli oggetti Intent impliciti dal sistema Android.
- `<data>`: Specifica il tipo di **dati** che il componente può gestire. Questo include il tipo MIME dei dati o altri attributi di un URI (come lo schema, l'host, la porta e il percorso). Un intent può specificare un URI di dati e/o un tipo MIME. Il filtro può dichiarare quali schemi URI e tipi MIME sono supportati.

Per un intent implicito affinché venga consegnato a un componente, l'intent deve superare i test di **tutti e tre gli elementi (azione, categoria e dati)** definiti in almeno uno dei filtri di intent dichiarati dal componente. Se un componente ha più filtri di intent, un intent che non corrisponde a un filtro potrebbe comunque corrispondere a un altro.

GLI EXTRAS

Gli Intent è che essi, nel recapitare questo messaggio, portano con se dati che possono essere letti dal destinatario. Questi valori condivisi mediante Intent vengono chiamati **Extras** e possono essere di varie tipologie, sia appartenenti a

classi più comuni che ad altre purché serializzabili. La gestione degli Extras negli Intent funziona in maniera simile ad una struttura dati HashMap: con dei metodi **put** viene inserito un valore etichettato con una chiave e con i corrispondenti metodi **get** viene prelevato il valore, richiedendolo mediante la chiave di riconoscimento.

PARAMetri INTENT

Activity class (per Intent espliciti): il nome della classe dell'activity o del componente che dovrebbe ricevere l'intent.

Intent action: l'azione generica che l'activity ricevente dovrebbe eseguire. Le azioni disponibili sono definite come costanti nella classe Intent e iniziano con ACTION_. Esempi includono ACTION_VIEW, ACTION_SEND e ACTION_PICK.

Intent category: fornisce informazioni aggiuntive sulla categoria del componente che dovrebbe gestire l'intent. Le categorie sono opzionali e possono essere aggiunte con il metodo addCategory(). Esempi includono CATEGORY_LAUNCHER, CATEGORY_DEFAULT e CATEGORY_BROWSABLE. Per rispondere a Intent impliciti, un filtro di intent deve impostare esplicitamente la categoria DEFAULT, che viene aggiunta implicitamente a ogni Intent implicito.

Intent data: contiene un riferimento ai dati su cui l'activity ricevente dovrebbe operare, come un oggetto Uri. Può rappresentare un URL web, un numero di telefono o un percorso a un file. Il tipo di dati può anche essere specificato con il metodo setType(), utilizzando un tipo MIME.

Intent extras: coppie chiave-valore che trasportano informazioni aggiuntive richieste dall'activity ricevente. I valori possono essere tipi primitivi o oggetti che implementano l'interfaccia Parcelable. I metodi putExtra() vengono utilizzati per aggiungere extra a un Intent, e getIntent().getExtras() o metodi specifici come getStringExtra() o getIntExtra() vengono utilizzati per recuperarli nell'activity ricevente.

Intent flags: metadati aggiuntivi che istruiscono il sistema Android su come avviare un'activity o come trattarla dopo l'avvio.

TIPI DI INTENT

Un **broadcast intent** è un tipo di intent che non avvia un'activity specifica, ma viene consegnato a tutti i **broadcast receiver** interessati registrati per tale intent. I broadcast receiver sono componenti che ascoltano specifici broadcast di sistema o personalizzati.

Un **PendingIntent** è un token che si concede a un'altra applicazione per utilizzare le autorizzazioni del proprio processo per eseguire un'azione in un secondo momento. Viene spesso utilizzato per le notifiche.

Un **IntentService** è una sottoclasse di Service che gestisce le richieste intent in background, una alla volta.

INTERFACCIA UTENTE

Quando sviluppiamo un'app Android, dobbiamo pensare all'interfaccia grafica, ovvero a come gli elementi vengono organizzati sullo schermo.

L'**interfaccia utente (UI)** di un'applicazione è costruita attraverso una **gerarchia di oggetti chiamati view**. Ogni elemento visibile sullo schermo è una view.

Tutti i componenti visivi in Android discendono dalla classe View e vengono indicati genericamente come Views.

La classe **ViewGroup** è un'estensione di View che supporta l'**aggiunta di visualizzazioni innestate** che hanno la responsabilità di decidere le dimensioni della view di ogni figlio e di determinare la loro posizione.

I ViewGroup che si concentrano principalmente sulla disposizione delle viste contenute sono indicati come **layout**. I ViewGroup sono visti quindi possono anche disegnare la propria interfaccia utente personalizzata e gestire le interazioni degli utenti.

Layouts → definiscono la struttura e la disposizione degli elementi sulla schermata

Views → Sono gli elementi base dell'interfaccia utente

Widgets → Sono particolari tipi di View con funzionalità specifiche

Per creare e disporre le viste all'interno della UI si possono utilizzare risorse di layout XML per creare e costruire lo scheletro statico della UI di un Activity da fare evolvere dinamicamente in modo programmatico.

Questo approccio consente di specificare diversi layout ottimizzati per diverse configurazioni hardware, in particolare variazioni delle dimensioni dello schermo, potenzialmente anche modificandoli in fase di esecuzione in base a modifiche hardware (come i cambiamenti di orientamento).

L'interfaccia utente è organizzata in una struttura ad **albero gerarchico**, chiamata **albero delle View**

- i nodi interni sono le ViewGroup
- le foglie sono le View effettive

Quando apriamo un'applicazione Android, una delle prime cose che il sistema fa è costruire l'interfaccia grafica dell'Activity in base alla struttura definita nel file XML del layout. Questo processo avviene all'interno del metodo **onCreate()**, dove viene caricato l'albero delle View.

Quando il metodo `onCreate()` viene eseguito, l'Activity richiama `setContentView(R.layout.activity_main)`, che ha il compito di leggere il file XML e tradurlo in un insieme di oggetti **View** e **ViewGroup** in memoria.

Android calcola la disposizione degli elementi, assegna loro le dimensioni e infine li disegna sullo schermo.

Android dà la possibilità di **modificare l'albero delle View in tempo reale**, aggiungendo, rimuovendo o modificando elementi dell'interfaccia mentre l'utente sta interagendo con l'app, senza dover ricaricare l'intera Activity

EVENTI

Le view permettono di acquisire gli eventi dalla l'oggetto con cui interagisce l'utente. Per un'app Android, l'interazione in genere consiste nel toccare, premere, digitare o parlare e ascoltare.

La View che **ha il focus** sarà la componente che riceve l'input dell'utente. Il focus può essere avviato dall'utente toccando una View. È possibile definire un ordine di focus in cui l'utente viene guidato da un controllo UI a un altro controllo UI.

Il focus può anche essere controllato a livello di programmazione; un programmatore può richiedereFocus() su qualsiasi View che è focalizzabile.

Un altro attributo di un controllo di input è **cliccabile**. Se questo attributo è true, allora la View può reagire agli eventi di clic. Come per il focus, cliccabile può essere controllato a livello di programmazione.

La differenza tra cliccabile e focalizzabile è che cliccabile significa che la view può essere cliccata o toccata, mentre focalizzabile significa che alla view è consentito ottenere il focus da un dispositivo di input come una tastiera.

I dispositivi di input come le tastiere non possono determinare a quale view inviare i loro eventi di input, quindi li inviano alla view che ha il focus.

per intercettarlo, devi estendere la classe e sostituire il metodo. Tuttavia, l'estensione di ogni oggetto View per gestire un evento del genere non sarebbe pratica. Per questo motivo la classe View contiene anche una raccolta di interfacce nidificate con callback che puoi definire molto più facilmente.

Un **listener di eventi** è un'interfaccia della classe View che contiene un singolo di callback. Questi metodi verranno chiamati dal framework Android quando la view a cui il listener ha registrato viene attivato dall'interazione dell'utente con l'elemento nell'interfaccia utente.

Le interfacce del listener di eventi includono i seguenti metodi di callback:

onClick() da `View.OnClickListener`. Questo viene chiamato quando l'utente tocca l'elemento. (in modalità touch) o si concentra sull'elemento con i tasti di navigazione o la trackball e preme il tasto "Invio" adatto o premi la trackball.

onLongClick() da `View.OnLongClickListener`. Questo viene chiamato quando l'utente tocca e tiene premuto l'elemento (in modalità tocco) o si concentra sull'elemento con i tasti di navigazione o la trackball e tiene premuto il pulsante "Invio" o tenere premuto sulla trackball (per un secondo).

onFocusChange() Da `View.OnFocusChangeListener`. Questo viene chiamato quando l'utente si avvicina o si allontana dall'elemento utilizzando i tasti di navigazione o la trackball.

onKey() da `View.OnKeyListener`. Viene chiamato quando l'utente si concentra sull'elemento e preme o rilascia un tasto hardware sul dispositivo.

onTouch() da `View.OnTouchListener`. Questo nome viene chiamato quando l'utente esegue un'azione qualificata come evento touch, ad esempio una stampa, un comunicato stampa o qualsiasi gesto di movimento sullo schermo (entro i limiti dell'elemento).

onCreateContextMenu() da `View.OnCreateContextMenuListener`. Questa operazione viene chiamata quando viene creato un menu contestuale (in seguito a un "clic lungo") prolungato.

Un **gesto di tocco** si verifica quando un utente posiziona una o più dita sul il touchscreen e l'app interpreta questa sequenza di tocchi come un gesto. Il rilevamento dei gesti prevede due fasi:

1. la raccolta di dati sugli eventi touch
2. l'interpretazione dei dati per determinare se soddisfano i gesti supportati dall'app.

Quando un utente posiziona una o più dita sullo schermo, viene attivata la **callback onTouchEvent()** sulla visualizzazione che riceve gli eventi touch. e per ogni sequenza di tocco viene memorizzata la posizione, pressione, dimensione, il numero di dita usate e altre info utili. Queste info vengono memorizzate nell'oggetto `MotionEvent`.

Il gesto inizia quando l'utente tocca per la prima volta lo schermo, poi continua mentre il sistema tiene traccia della posizione del dito o delle dita dell'utente e termina acquisire l'evento finale dell'ultimo dito dell'utente che lascia lo schermo.

Gli eventi, in particolare gli eventi di input come i tocchi, possono essere propagati attraverso la gerarchia delle view. Una View può consumare un evento o passarlo al suo parent ViewGroup per la gestione. Il sistema Android **distribuisce (dispatch)** questi eventi ai componenti appropriati per la gestione.

Quando si verifica un evento di interazione, si propaga dall'activity andando fino alla view.

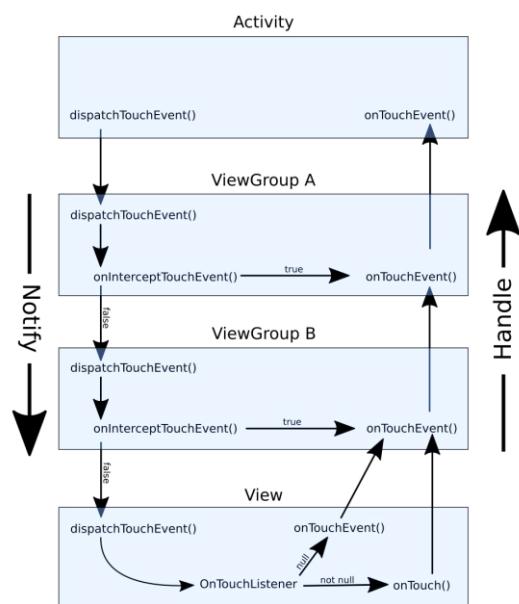
Quindi a tutti i componenti della gerarchia viene data la possibilità di gestire l'evento, iniziando con la vista in alto e tornando all'attività. Quindi l'activity è la prima a ricevere l'evento e l'ultima a cui viene data la possibilità di gestirlo.

Se qualche ViewGroup vuole gestire immediatamente l'evento touch, può restituire true nel suo **onInterceptTouchEvent()**.

Una Activity non ha **onInterceptTouchEvent()** ma può sovrascrivere **dispatchTouchEvent()** per fare la stessa cosa.

Se una vista (o un gruppo di viste) ha un **OnTouchListener**, l'evento tocco viene gestito da **OnTouchListener.onTouch()**. Altrimenti è gestito da **onTouchEvent()**. Se **onTouchEvent()** restituisce true per qualsiasi evento di tocco, la gestione si interrompe. Nessun altro ne ha la possibilità.

La classe **GestureDetector** può essere utilizzata per interpretare sequenze di **MotionEvent** e dispatchare eventi di gesto specifici (come tap, swipe, fling) a un listener **OnGestureListener**



VIEW PERSONALIZZATE

La creazione di nuove visualizzazioni dà la possibilità di modellare l'aspetto e il funzionamento delle applicazioni. Creando i propri controlli, si può creare una interfaccia utente che si adatta in modo perfetto ai requisiti.

Per creare nuovi controlli da un'area di disegno vuota, si usa la classe base **View** o **SurfaceView**.

La classe View fornisce un oggetto **Canvas** con una serie di metodi di disegno e classi Paint. Servono per creare un'interfaccia visiva con bitmap e grafica raster. È possibile catturare e sovrascrivere gli eventi utente, inclusi i tocchi dello schermo o la pressione dei tasti per fornire l'interattività.

La classe **SurfaceView** fornisce un oggetto Surface che supporta il disegno da un thread in background e, facoltativamente, l'uso di OpenGL per implementare la grafica.

Per creare una view custom dobbiamo fornire la classe:

1. Costruttori
 - a. Uno a cui passiamo solo il contesto
 - b. Contesto e attributi
 - c. Contesto attributi e stile
2. **onMeasure()** che viene chiamato quando la view deve essere disegnata. L'incarico di disegnare la view spetta al genitore che chiede al figlio le sue dimensioni: altezza e larghezza, chiamando i metodi della classe del figlio.
 - a. Questa funzione viene chiamata ogni volta che la view viene invalidata.
3. **onDraw()** che serve per fare effettivamente il disegno della view. Questa funzione viene chiamata frequentemente e in modo sincrono in base alla frequenza di refresh impostata dal dispositivo. Se la frequenza di refresh è minore del tempo con cui questa funzione viene chiamata si hanno dei problemi nel rendering.

LAYOUT

Alcuni ViewGroup sono designati come **layout** perché organizzano le View figlie in un modo specifico e sono tipicamente usati come **ViewGroup** radice.

I ViewGroup sono definiti nei file di **layout XML**, che si trovano nella cartella layout all'interno della cartella res del progetto Android. La gerarchia delle View, con un ViewGroup alla radice, può diventare complessa in app con molte View sullo schermo. Comprendere questa gerarchia è importante per l'efficienza del rendering e la visibilità delle View. È possibile esplorare la gerarchia delle View di un'app utilizzando l'**Hierarchy Viewer**.

1. LinearLayout

Il **LinearLayout** organizza le sue View figlie in una singola direzione: **verticale** o **orizzontale**. Ogni elemento viene disposto uno dopo l'altro, rispettando l'ordine di dichiarazione. È utile quando si desidera una disposizione semplice e sequenziale degli elementi.

2. RelativeLayout

Il **RelativeLayout** permette di posizionare le View figlie in relazione tra loro o rispetto al contenitore padre. Ad esempio, un elemento può essere posizionato alla destra di un altro o allineato al bordo superiore del layout. Questo offre una maggiore flessibilità nella disposizione degli elementi rispetto al LinearLayout.

3. FrameLayout

Il FrameLayout è progettato per contenere una singola View, ma può gestire anche più elementi sovrapposti. Le View aggiunte successivamente si posizionano sopra le precedenti, creando un effetto di stratificazione. È spesso utilizzato per visualizzare una View principale con elementi sovrapposti, come pulsanti flottanti o indicatori di stato.

4. ConstraintLayout

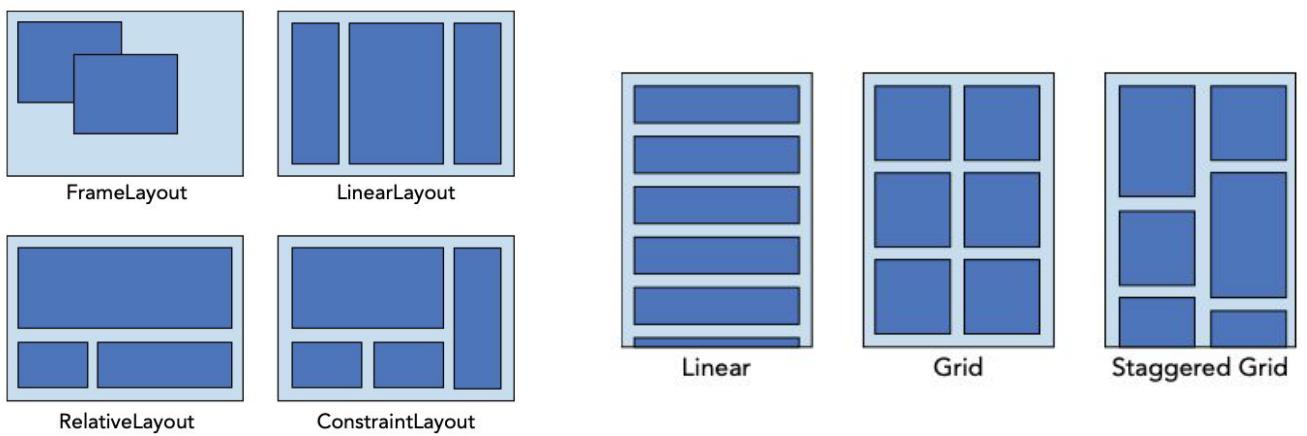
Il **ConstraintLayout** offre un alto grado di flessibilità, permettendo di definire vincoli tra le View per determinare la loro posizione e dimensione. Questo layout è stato introdotto per semplificare la creazione di interfacce complesse senza nidificare più layout, migliorando le prestazioni dell'applicazione.

6. GridLayout

Un **GridLayout** (o GridLayoutManager per le liste) organizza gli elementi in **righe e colonne**, creando una struttura a griglia. È utile quando si devono mostrare più elementi con lo stesso peso visivo, come immagini o pulsanti.

7. StaggeredGridLayout

Uno **StaggeredGridLayout** è simile a un GridLayout, ma con **colonne o righe di altezze/larghezze diverse**. Questo lo rende perfetto per layout più dinamici e visivamente accattivanti.



Il **ConstraintLayout** è un **ViewGroup** in Android che offre un modo flessibile per disporre e allineare le View figlie all'interno di un layout. Utilizza un sistema di **vincoli (constraints)** per determinare la posizione e le dimensioni delle sue View figlie.

Il ConstraintLayout organizza le View figlie utilizzando **punti di ancoraggio, bordi e linee guida** per controllare come le View sono posizionate rispetto ad altri elementi nel layout. Un **vincolo** è una connessione o un allineamento a un'altra View, al layout genitore o a una linea guida invisibile.

Ciascun vincolo definisce la posizione della vista lungo l'asse verticale o orizzontale; quindi ogni vista deve avere almeno un vincolo per ogni asse, ma spesso ne sono necessari di più.

Fissa: specifica una dimensione specifica nella casella di testo seguente o ridimensionando la vista nell'editor.

A capo: la visualizzazione si espande solo per adattarsi ai contenuti.

Vincoli di corrispondenza: la vista si espanderà il più possibile per soddisfare i vincoli su ogni lato, dopo aver tenuto conto dei margini della vista.

MODELLO NAVIGAZIONALE

Quando progettiamo l'interfaccia grafica e dobbiamo pensare all'UX perché svolto in azioni troppo complesse tutte in una schermata può portare l'utente a compiere errori.

Può essere utile dividere una task complessa in più task semplici e per questo è possibile raggruppare task simili in N schermate diverse in modo che l'utente si focalizzi meglio di una singola parte.



È necessario progettare come l'utente nave di chi e promuovere il **wayfinding** cioè dare alle persone la possibilità di orientarsi e di muoversi all'interno delle schermate.

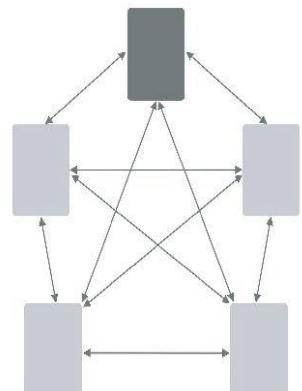
La navigazione si era voluta anche a capire le informazioni, cosa può fare, dov'è ora, dove può andare e come tornare indietro.

Per aiutare gli utenti a navigare necessario usare **elementi di segnaletica** per aiutare l'utente a orientarsi.

- Indicatori di avanzamento
- Barre di stato
- Breadcrumbs

La navigazione è un overhead perché richiede tempo e risorse per realizzarla in quanto si potrebbe realizzare tutto in una singola schermata ma il motivo per cui si venne a realizzare è di ridurre il carico cognitivo, lo sforzo mentale è caduta anni fa per elaborare le informazioni. Dobbiamo raggruppare task simili perché troppe task che svolgono operazioni diverse ma anche troppe che ne svolgono simili portano all'utente a compiere errori e lapsus perché secondo le scienze cognitive l'uomo non riesce a concentrarsi più su più di sette oggetti alla volta.

Fully Connected Navigation Ogni sezione dell'interfaccia può portare direttamente a qualsiasi altra. Questo tipo di navigazione è utile quando si vuole offrire **massima libertà** all'utente. Tuttavia, c'è un rischio che se tutto è collegato a tutto, può diventare difficile capire dove si è e dove si sta andando. Serve dunque una forte coerenza visiva, una buona architettura dell'informazione e magari qualche aiuto visivo come breadcrumb o menu ben organizzati.



The fully connected model

Multilevel Navigation Si basa su una struttura gerarchica dei contenuti, dividendolo in categorie

Aiuta l'utente a **orientarsi** e a trovare ciò che cerca seguendo un percorso logico però troppi livelli possono diventare dispersivi, quindi è bene non esagerare con la profondità della struttura.

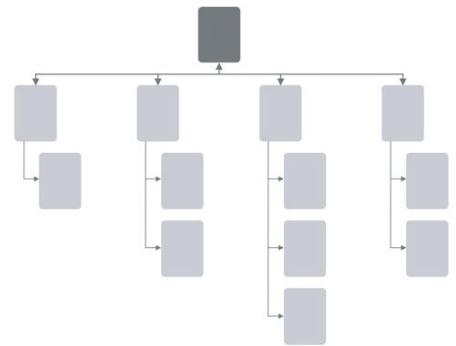


Figure 3-3. Multilevel navigation

Step-by-Step Flows Utilizzati per guidare l'utente in un processo lineare, come una registrazione, un acquisto online, o la configurazione di un profilo. Semplifica l'interazione perché ogni passo è chiaro, focalizzato, e l'utente sa sempre cosa deve fare. L'importante è rendere visibile il progresso, con indicatori e consentire di tornare indietro senza perdere i dati inseriti.

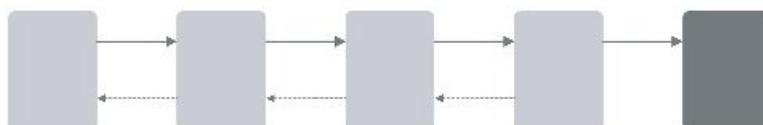


Figure 3-4. Step-by-step flows

Pyramidal Model Si parte da un'informazione generale e consente all'utente di esplorare via via contenuti sempre più specifici. Funziona bene quando si vuole **gestire l'attenzione** dell'utente e accompagnarla in una lettura a livelli, senza sommergerlo subito di dettagli.

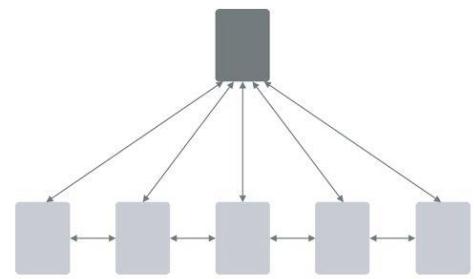
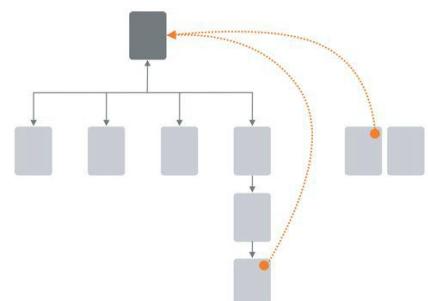


Figure 3-5. Pyramid

Deep Link A volte non si parte dalla cima ma ci si ritrova direttamente "all'interno" dell'app, grazie a un **deep link**, cioè un collegamento diretto a una pagina interna. Devono essere progettati con cura, perché l'utente che ci arriva potrebbe **non avere il contesto**: è quindi importante che la pagina sia autonoma, comprensibile da sola.



Clear Entry Point Cioè una porta d'ingresso ben visibile e comprensibile per l'utente. Appena apre l'app o il sito, l'utente deve capire cosa può fare, dove può andare e quali sono le opzioni principali. Questo vale soprattutto per esperienze nuove o complesse, dove una schermata iniziale troppo carica o confusa rischia di scoraggiare. Meglio semplificare, dare una direzione e lasciare che il resto venga scoperto poco alla volta.

Hub and Spoke L'utente parte da un **hub** e da lì può accedere a diverse **spoke**, sezioni o attività isolate. Una volta completata un'attività, torna all'hub per decidere cosa fare dopo. Questo approccio è utile quando si vuole **Mantenere un senso di controllo** e di orientamento.

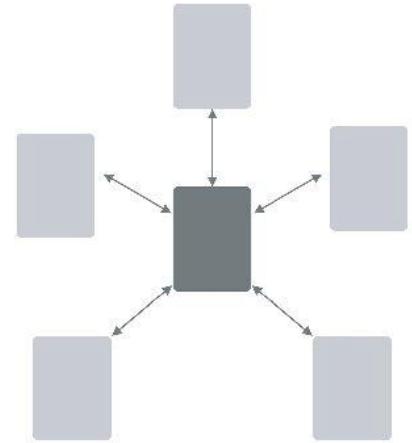
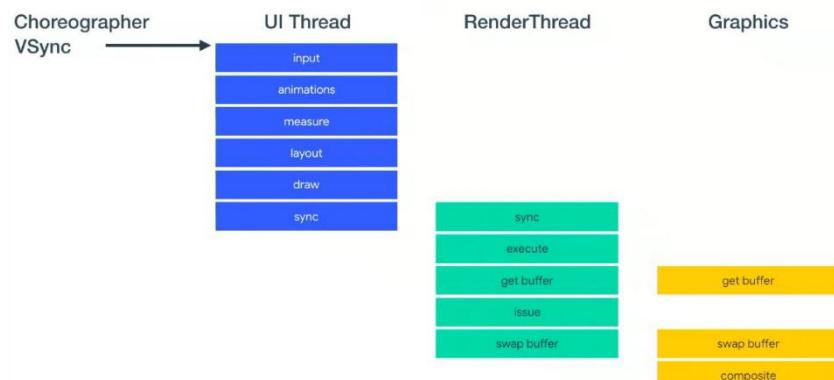


Figure 3-1. Hub and spoke architecture

RENDERING PIPELINE

Il processo di rendering è sincronizzato con il segnale **VSync (vertical synchronization)** generato dall'hardware del display scandendo il ritmo del rendering, sincronizzando ogni frame con il refresh dello schermo. Android sfrutta questo segnale per organizzare il lavoro su tre livelli principali:

1. **UI Thread**
2. **RenderThread**
3. **Graphics Pipeline.**



Il **UI Thread**, è responsabile della gestione degli eventi di input, dell'aggiornamento dell'interfaccia e delle animazioni.

Quando il sistema riceve un segnale di **VSync**, il **Choreographer**, un componente di Android che orchestra il rendering, sveglia il UI Thread, che esegue una serie di operazioni:

1. **Input Handling** → Il primo compito è elaborare gli input dell'utente, come tocchi, scroll o gesti. Se premiamo un pulsante, il sistema deve registrare l'evento e prepararsi a cambiare la UI.
2. **Animations** → Se ci sono animazioni in corso (ad esempio, un pulsante che si ingrandisce quando viene premuto), vengono aggiornate e

interpolate. Android calcola la nuova posizione, opacità o dimensione dell'elemento animato.

3. **Measure & Layout** → Ora il sistema deve capire **quanto spazio occupano le View e dove devono essere posizionate**. Questo processo avviene in due fasi:

Measure → Ogni View calcola la propria dimensione basandosi sui suoi genitori e sul contenuto.

Layout → Una volta note le dimensioni, le View vengono posizionate nella finestra dell'app.

4. **Draw** → Dopo aver determinato le posizioni e le dimensioni delle View, il sistema esegue il disegno vero e proprio. Questo avviene nel metodo `onDraw(Canvas canvas)`, che dipinge gli elementi grafici sullo schermo.
5. **Sync** → Infine, il **UI Thread** invia i dati al **RenderThread**, il quale si occuperà di trasformarli in qualcosa che la GPU può elaborare.

Se una proprietà di una view cambia in modo da influire sul suo aspetto viene chiamato il metodo **invalidate()** sulla vista. Questa chiamata non causa un redraw immediato, ma marca la vista come "sporca" e segnala che deve essere ridisegnata. Una volta chiamato `invalidate()` su una vista, questa chiamata si propaga **verso l'alto nella gerarchia delle viste**, chiamando una serie di metodi, come **invalidateChild()**, sui suoi parent.

Questo processo continua fino a raggiungere la radice della gerarchia delle viste, in particolare la **ViewRootImpl**.

Quando la dimensione o la posizione di una vista deve cambiare viene chiamato **requestLayout()**. Questo è simile all'invalidazione, ma innesca un processo di misurazione e layout per determinare le nuove dimensioni e posizioni delle viste coinvolte.

Quando la chiamata `invalidateChild()` raggiunge il **ViewRootImpl**, quest'ultimo non esegue immediatamente il ridisegno ma chiama il metodo **scheduleTraversals()** con cui pianifica l'esecuzione del processo di **traversal** (misurazione, layout e disegno) in un momento successivo, tipicamente in sincronia con il prossimo segnale VSync gestito dal Coreograph.

Per ottimizzare questo processo, Android utilizza un meccanismo chiamato **Display List**. Quando viene chiamato `draw()`, la vista (o i suoi antenati) ottiene (o rigenera) una **display list** tramite il metodo **getDisplayList()**. Una **display list** è una registrazione di tutte le operazioni di disegno (ad esempio, `drawBackground()`, `drawText()`) che la vista deve eseguire per rendersi. L'intera gerarchia delle viste è rappresentata da una gerarchia di display list.

Dopo che l'UI thread ha completato il traversal (misura, layout e disegno) e ha generato la gerarchia delle **Display List**, queste informazioni vengono sincronizzate con il **Render Thread**.

Per evitare di sovraccaricare il **UI Thread**, Android introduce il **RenderThread**, un thread separato che gestisce il disegno e l'invio dei comandi alla GPU.

Una volta che il **UI Thread** ha terminato il suo lavoro, il **RenderThread** prende in carico il frame e segue questi passaggi:

1. **Sync** → Riceve i dati dal **UI Thread** e li sincronizza con lo stato attuale della grafica.
2. **Execute** → Processa i comandi di rendering, preparando la scena da inviare alla GPU.
3. **Get Buffer** → Recupera un **buffer di rendering**, ovvero un'area di memoria in cui verrà disegnato il frame.
4. **Issue** → Converte i comandi in istruzioni per la GPU.
5. **Swap Buffer** → Una volta che tutto è pronto, il **RenderThread** invia il frame alla GPU e chiede il prossimo buffer per il frame successivo.

Il Render Thread prende queste Display List, che sono una rappresentazione delle operazioni di disegno a livello Java, e le trasforma in qualcosa che può effettivamente elaborare per la GPU. Queste rappresentazioni native delle operazioni di disegno sono chiamate **Display List Operations (DL ops)**.

Le DL ops sono quindi la forma in cui le intenzioni di disegno dell'applicazione, espresse tramite le API Canvas a livello Java, vengono tradotte in operazioni concrete che il Render Thread può ottimizzare e inviare alla GPU.

Un esempio di DL op è una **fill operation** che corrisponde all'operazione di riempire una determinata area con un colore. Altre DL ops rappresenterebbero il disegno di testo, linee, bitmap, ecc..

In sostanza, le DL ops sono un intermediario tra la rappresentazione astratta del disegno (Display List) e i comandi concreti inviati alla GPU.

Le DL ops vengono ottimizzate tramite **riordinamento (reordering)** delle operazioni di disegno.

L'obiettivo del riordinamento è di raggruppare operazioni di disegno simili che non si sovrappongono per minimizzare i **cambi di stato** della GPU, che sono operazioni molto costose in termini di prestazioni.

Attraverso il riordinamento, il Render Thread analizza le DL ops e, se possibile, le raggruppa. Quindi, tutte le operazioni di disegno dei rettangoli verrebbero eseguite insieme, seguite da tutte le operazioni di disegno del testo. In alcuni casi, operazioni simili possono anche essere **batching (raggruppate)** ulteriormente in una singola chiamata alla GPU per maggiore efficienza.

Questa ottimizzazione può portare a **miglioramenti significativi nelle prestazioni del rendering**, specialmente in scenari con molte operazioni di disegno simili, come ad esempio nel rendering di liste complesse.

Il riordinamento non può essere applicato se le operazioni di disegno si sovrappongono, poiché in tal caso l'ordine è cruciale per rispettare il blending e l'alpha blending.

A questo punto, entra in gioco la **Graphics Pipeline**, ovvero la parte hardware responsabile del rendering finale. Il suo compito è prendere i comandi ricevuti dal **RenderThread** e trasformarli in pixel sullo schermo.

1. **Get Buffer** → La GPU recupera il buffer con i dati da disegnare.
2. **Swap Buffer** → Il buffer contenente il nuovo frame viene inviato allo schermo.
3. **Composite** → Infine, la GPU compone il frame con altri layer grafici e lo mostra all'utente.

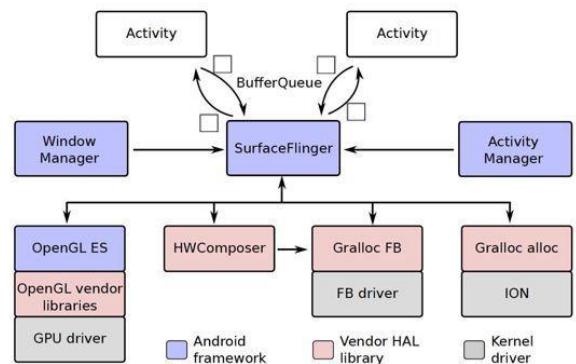
A questo punto interviene il **SurfaceFlinger** è un servizio di sistema responsabile della composizione di tutti i **window** (finestre) visibili sullo schermo.

Ogni finestra ha associata una **BufferQueue**, una coda di buffer grafici dove risiedono i dati pixel prodotti dall'applicazione (o da altri servizi di sistema). La BufferQueue ha due estremità: un produttore, il **Window Manager** che inserisce i buffer nella coda e un consumatore, l'**Activity Manager** che li preleva.

Quando il render thread chiama swapBuffers, accodando nel queue buffer il buffer renderizzato nella BufferQueue. SurfaceFlinger acquisisce questi buffer pronti per essere composti.

SurfaceFlinger comunica con l'**Hardware Composer (HWC)**, che è un'astrazione hardware specifica del dispositivo in grado di comporre più layer (bitmap) in modo molto efficiente, spesso senza utilizzare la GPU, per risparmiare energia. L'HWC decide come gestire ciascun layer:

- Se l'HWC supporta il formato pixel del layer e non ci sono troppe trasformazioni complesse, può gestirlo come un **overlay**, componendolo direttamente sull'hardware.
- In alcuni casi, l'HWC potrebbe non essere in grado di gestire un layer come overlay, allora il SurfaceFlinger deve utilizzare la **GPU** per comporre questi layer in un **frame buffer** (un buffer di rendering temporaneo) tramite comandi OpenGL, un'API utilizzata in Android per il rendering 2D e 3D.



Indipendentemente dal fatto che la composizione avvenga tramite HWC o GPU, SurfaceFlinger combina tutti i layer visibili (le finestre di tutte le applicazioni, la barra di stato, la barra di navigazione) e invia il risultato all'hardware del display per essere visualizzato.



ARCHITECTURAL UI AND DATA MANAGEMENT PATTERNS

Prima ancora di scrivere codice, è importante **definire l'architettura del sistema**. Progettare l'architettura in anticipo permette di avere una visione chiara di come sarà strutturata l'applicazione, come interagiranno i diversi componenti tra loro e quali saranno le dipendenze principali.

Questo ha un impatto diretto sulla qualità del progetto nel lungo periodo. Se in futuro sarà necessario apportare modifiche, aggiungere funzionalità, sostituire una tecnologia, risolvere bug o adattarsi a nuove esigenze... avere un'architettura solida e ben pensata renderà tutto più semplice, veloce e sicuro. È necessario strutturare il sistema in modo che sia indipendente dai framework, librerie e linguaggi che usiamo perché cambiano nel tempo: nuove versioni, nuove API, a volte addirittura vengono abbandonati o sostituiti da soluzioni migliori. Se il nostro sistema dipende troppo da queste tecnologie, ogni cambiamento porta ad errori o riscritture pesanti del codice.

a **modularità** è uno dei concetti più importanti. Significa **suddividere un sistema in parti indipendenti e riutilizzabili**, chiamate moduli o componenti, ognuno dei quali ha una responsabilità chiara e ben definita.

Conviene pensare alla **modularità** cioè dividere il codice in moduli in modo che sia più semplice da capire, da testare, da manutenere e da evolvere. Ogni modulo può essere sviluppato (e perfino sostituito) senza dover riscrivere tutto il resto. Dobbiamo fare attenzione alle **dipendenze circolari** che si verificano quando due (o più) moduli dipendono l'uno dall'altro direttamente o indirettamente. Questo crea una **situazione di interdipendenza**, dove nessuno dei moduli può essere isolato, testato o riutilizzato in modo indipendente.

Il principio che si trova alla base di ogni pattern architettonico è la **separazione dei problemi** in livelli in modo da avere le dipendenze verso un'unica direzione ed evitare le dipendenze circolari. I vari livelli non devono avere riferimenti ai livelli più esterni di loro, questo violerebbe la modularità e renderebbe quindi i livelli dipendenti fra loro.

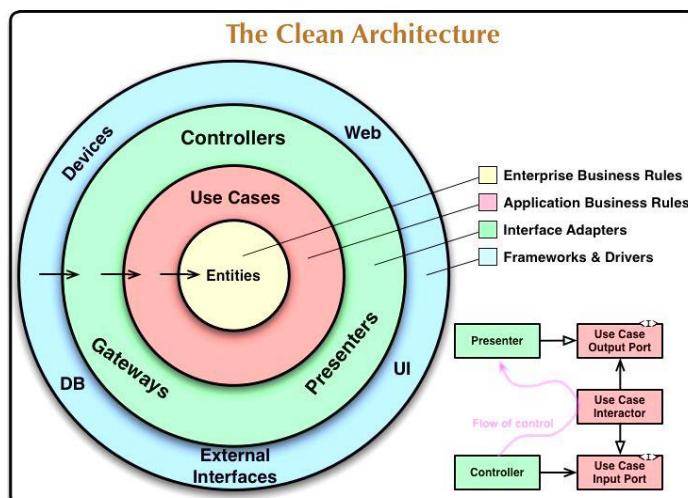
CLEAN ARCHICTURE

La **Clean Architecture** mira a creare sistemi **intercambiabili** con una forte **separazione delle preoccupazioni** e un **accoppiamento debole** tra i livelli. L'idea fondamentale di Clean Architecture, concepita da Robert Martin (Uncle Bob), è rendere **intercambiabile** ogni elemento all'interno di un certo confine architettonico, senza richiedere modifiche nei livelli sottostanti.

Le dipendenze del codice sorgente possono puntare solo verso l'interno. Questo significa che i livelli interni non devono dipendere da livelli. Questa direzione delle dipendenze è fondamentale per ottenere un basso accoppiamento e facilitare il test e la manutenzione.

La Clean Architecture divide un'applicazione in diversi livelli con responsabilità distinte:

1. **Domain/Application Core:** Contiene le entità e la logica di business dell'applicazione. Questo livello è indipendente da qualsiasi framework o dettaglio di implementazione esterna.
2. **Use Cases:** Il software in questo livello contiene regole aziendali specifiche dell'applicazione ed incapsula e implementa tutti i casi d'uso del sistema. Questi casi d'uso orchestrano il flusso di dati da e verso le entità e indirizzano tali entità per raggiungere gli obiettivi del caso d'uso. I cambiamenti in questo livello non devono influenzare le entità e questo livello non è influenzato da modifiche alle componenti esterne come il database, l'interfaccia utente o qualsiasi framework utilizzato. Questo strato è isolato da tali problematiche. Tuttavia, modifiche alle funzioni di dominio dell'applicazione influenzano i casi d'uso e quindi il software in questo livello.
3. **Infrastructure/Frameworks:** Contiene i dettagli di implementazione dei sistemi esterni, come database, framework UI, librerie esterne e API. Le implementazioni delle interfacce definite nel livello Application risiedono qui.
4. **Interface Adapters:** Questo livello funge da ponte tra i livelli Application e Infrastructure. Contiene adapter, presentatori e controller che convertono i dati da un formato all'altro per soddisfare le esigenze dei diversi livelli.
5. **API/UI:** Il livello più esterno che presenta l'applicazione all'utente (tramite un'API web o un'interfaccia utente grafica).

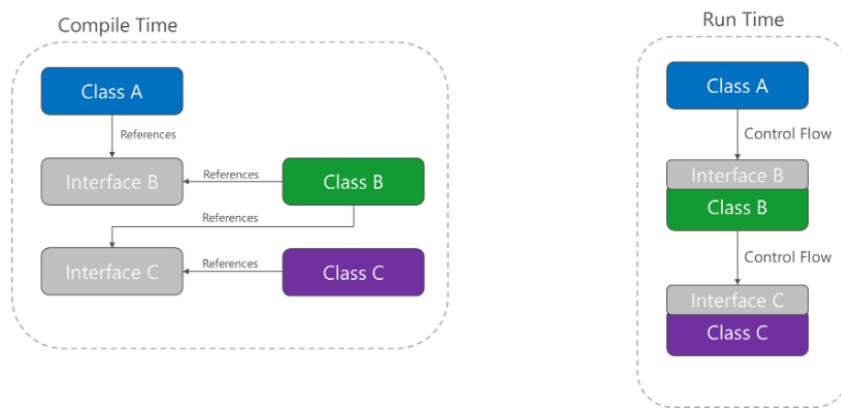


Per attraversare i confini (cerchi concentrici) si usano i controller e i Presenter che comunicano con i casi d'uso nel livello successivo.

1. l'esecuzione inizia nel controller
2. si sposta attraverso il caso d'uso
3. finisce per essere eseguito nel presenter.

Per implementare questa logica si utilizza il **principio di inversione delle dipendenze** che consente ad A di chiamare metodi su un'astrazione implementata da B, rendendo possibile per A chiamare B in fase di esecuzione, ma per B di dipendere da un'interfaccia controllata da A in fase di compilazione (invertendo così la tipica dipendenza in fase di compilazione). In fase di esecuzione, il flusso di esecuzione del programma rimane invariato, ma l'introduzione di interfacce significa che diverse implementazioni di queste interfacce possono essere facilmente collegate. La stessa tecnica viene utilizzata per attraversare tutti i confini nelle architetture.

Sfruttiamo il polimorfismo dinamico per creare dipendenze del codice sorgente che si oppongono al flusso di controllo in modo che possiamo rispettare la regola delle dipendenze indipendentemente dalla direzione in cui sta andando il flusso di controllo.



Il **Repository Pattern** è spesso integrato in Clean Architecture come un meccanismo per **astrarre l'accesso ai dati**. L'interfaccia del repository è definita nel livello Application, mentre l'implementazione concreta che interagisce con il database si trova nel livello Infrastructure. Questo rispetta la regola della dipendenza, poiché il livello Application dipende solo dall'interfaccia del repository, non dalla specifica implementazione del database.

La separazione dei livelli e la regola della dipendenza rendono le applicazioni Clean Architecture più **facili da testare**. I casi d'uso e la logica di business nel livello Application possono essere testati unitariamente senza dipendere da database o UI. Le interfacce nel livello Application facilitano il **mocking** delle dipendenze esterne durante i test. L'accoppiamento debole tra i livelli rende le

applicazioni Clean Architecture più **scalabili e manutenibili**. Le modifiche a un livello hanno meno probabilità di influenzare altri livelli, facilitando l'aggiornamento e l'aggiunta di nuove funzionalità.

I ViewModel spesso interagiscono con un **Repository** per l'accesso ai dati. Questa struttura (UI - ViewModel - Repository - Data Source) in Android può essere vista come un'implementazione pratica di alcuni principi di Clean Architecture, in particolare la separazione delle preoccupazioni e l'astrazione dell'accesso ai dati.

Una potenziale violazione della regola della dipendenza in Clean Architecture quando si implementa il Repository pattern utilizzando direttamente un ORM (come Entity Framework Core) nel livello Infrastructure. Questo introduce una dipendenza del livello Interface Adapters (dove si troverebbe l'implementazione del repository) verso un framework esterno e verso l'I/O diretto, violando la dipendenza verso l'interno. Per risolvere questo, si suggerisce di spostare l'implementazione del repository nel livello Frameworks/Infrastructure e di definire le interfacce nel livello Application.

LIVEDATA

LiveData è una classe di dati **osservabile** che è legata al ciclo di vita dei componenti UI, come **Activity** o **Fragment**. Questo significa che **LiveData** permette di aggiornare la UI automaticamente quando i dati cambiano, ma solo quando la UI è in uno stato attivo, evitando aggiornamenti inutili o errori quando la UI non è visibile (ad esempio, se l'**Activity** è in background). Implementa il pattern Observer.

ROOM

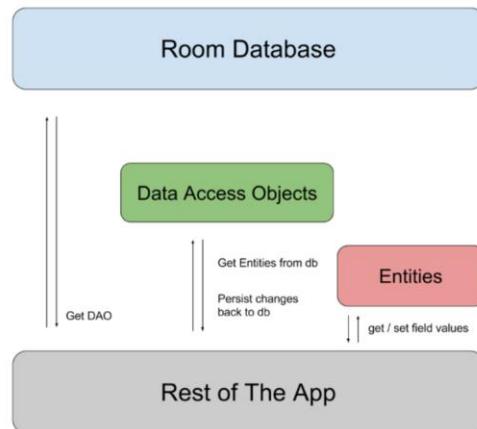
Room è una libreria di persistenza fornita da Google che si colloca all'interno dei componenti dell'architettura Android, con l'obiettivo di semplificare l'interazione con il database **SQLite** del sistema operativo, fornendo un **livello di astrazione**.

Room è descritta come un modo efficace per creare database e salvare dati. Fornisce un accesso al database SQLite **mappando oggetto-relazionale (ORM)** basato su annotazioni.

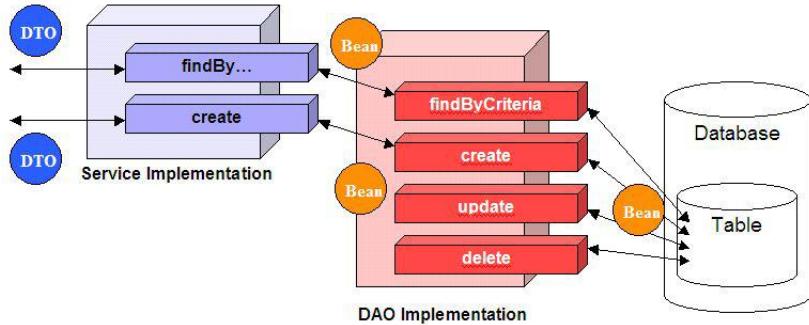
L'utilizzo di Room si articola principalmente su tre tipi di classi:

- **Entità:** Sono **Plain Old Java Objects (POJO)** che modellano i dati da trasferire verso e dal database. Sono annotate con **@Entity**, specificando la tabella del database a cui corrispondono. Almeno una proprietà deve essere designata come **chiave primaria** usando l'annotazione **@PrimaryKey**. Room crea una tabella nel database per ogni entità.

- **DAOs - Data Access Objects:** Definiscono l'**API per interagire con i dati**, contenendo metodi per operazioni come query, inserimenti, aggiornamenti ed eliminazioni. I DAOs sono interfacce o classi astratte annotate con `@Dao`. Le operazioni sul database sono definite all'interno dei metodi del DAO tramite annotazioni come `@Query`, `@Insert`, `@Update` e `@Delete`. Room genera l'implementazione concreta di queste interfacce o classi astratte in fase di compilazione.
- **Database:** Rappresenta l'**interfaccia principale al database SQLite sottostante**. È una classe astratta che estende RoomDatabase ed è annotata con `@Database`, elencando tutte le entità utilizzate dal database e la sua versione. Contiene metodi astratti che restituiscono istanze dei DAO. Si ottiene un'istanza del database tramite un RoomDatabase.Builder.
- **Relazioni tra Entità:** Sebbene Room supporti **relazioni tramite chiavi esterne** definite con l'annotazione `@ForeignKey`, **non supporta riferimenti diretti tra entità**. Per rappresentare relazioni uno-a-molti o molti-a-molti e accedere agli oggetti correlati, si utilizza l'annotazione `@Relation` all'interno di una classe POJO separata che contiene i campi per le entità correlate. Per le relazioni molti-a-molti è necessario implementare una "**join entity**" che crea la tabella di join associata.



In Room, le **entità sono pensate più come Data Transfer Objects (DTO)** oggetti concepiti come un **mezzo per trasferire dati** tra diversi punti di un'applicazione. Modellano una risposta o sono ottimizzati per la creazione e la persistenza dei dati. Le Entity in Room rappresentano i dati che si desidera memorizzare nel database e sono anche l'unità tipica di un set di risultati recuperato dal database.



Room può essere integrato con **LiveData** per osservare i cambiamenti nel database. Un DAO può restituire un oggetto **LiveData** da una query, consentendo all'interfaccia utente di aggiornarsi automaticamente quando i dati sottostanti cambiano.

Room si integra bene con **ViewModel**, che gestisce i dati relativi alla View in modo indipendente dai cambiamenti di configurazione. Il ViewModel può interagire con il database tramite un repository che utilizza Room.

È una buona pratica utilizzare il **pattern Repository** come livello intermedio tra Room e il resto dell'applicazione. Questo permette di **astrarre l'accesso ai dati** e, se in futuro vorrai cambiare il tipo di persistenza (ad esempio passare da Room a una sorgente di rete o cache), potrai farlo **senza modificare i** ViewModel o altri componenti della business logic.

MVC

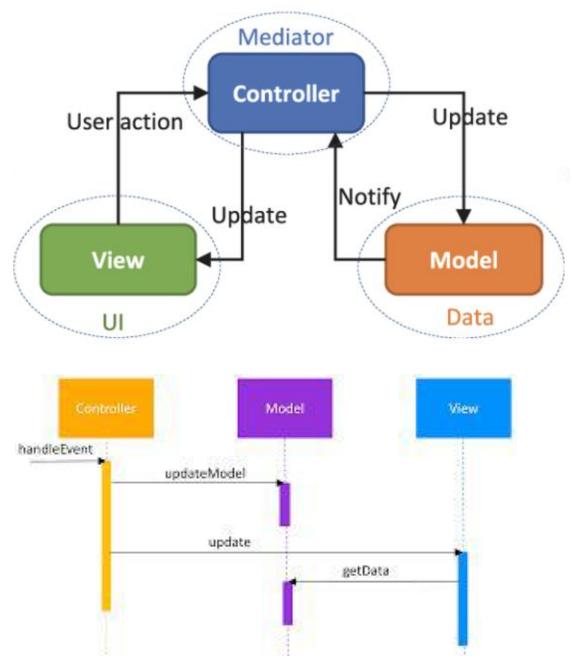
Il **MVC Model View Controller** è un modo per organizzare le funzionalità di un'applicazione separando gli oggetti in tre ruoli distinti:

Model Il modello contiene i dati dell'applicazione e la logica di business.

Le classi del modello sono progettate per rappresentare le entità con cui l'app lavora, come una domanda vero/falso.

Gli oggetti del modello non hanno alcuna conoscenza dell'interfaccia utente (UI). Il loro unico scopo è la gestione e la detenzione dei dati.

In Android, il livello del modello è generalmente costituito da classi personalizzate create dallo sviluppatore. Il modello può anche specificare la struttura dei dati dell'app e il codice per accedervi e manipolarli.



View La vista è responsabile della visualizzazione dei dati all'utente e della risposta alle azioni dell'utente.

Ogni elemento visibile sullo schermo è una vista. Android fornisce molteplici tipi di viste. Le viste sanno come disegnarsi sullo schermo e come rispondere all'input dell'utente, come i tocchi.

Controller Il controllore agisce come un intermediario tra il modello e la vista.

Contiene la logica dell'applicazione .I controllori rispondono agli eventi innescati dalle viste e gestiscono il flusso di dati da e verso il modello e la vista.

In Android, un controllore è tipicamente una sottoclasse di Activity, Fragment o Service.

È importante notare che il modello e la vista non comunicano direttamente. Il controllore si trova al centro, ricevendo messaggi da un lato e inviando istruzioni all'altro.

Benefici dell'MVC

- **Separazione delle responsabilità:** Aiuta a progettare e comprendere l'applicazione come un insieme di classi distinte.
- **Migliore organizzazione del codice:** La separazione in livelli (modello, vista, controllore) facilita la progettazione e la comprensione dell'applicazione a un livello superiore.
- **Riutilizzabilità del codice:** Le classi con responsabilità limitate sono più riutilizzabili.

Observer Synchronization

Questo approccio, strettamente associato all'MVC, si basa sul concetto che **le viste e i controllori osservano il modello**.

Quando il **modello subisce una modifica**, viene notificato a tutti i suoi osservatori (le viste e potenzialmente i controllori).

Le **viste reagiscono a queste notifiche** aggiornando la propria visualizzazione in base ai nuovi dati del modello.

Il **controllore**, in questo modello, è **relativamente "ignorante"** di quali altre viste potrebbero aver bisogno di essere aggiornate quando l'utente interagisce con una specifica vista. Il controllore si limita a modificare il modello, lasciando che il meccanismo di osservazione si occupi di propagare i cambiamenti alle viste interessate.

Flow Synchronization

Nella sincronizzazione tramite flusso, è l'applicazione (spesso il controllore) che manipola direttamente le viste per riflettere i cambiamenti nel modello. Ad

esempio, quando si apre una schermata o si preme un pulsante di salvataggio, il codice dell'applicazione si occupa di aggiornare esplicitamente i vari controlli (viste) con i dati del modello. In questo caso, il form (o l'attività/il fragment in Android) deve tenere traccia di quali controlli devono essere aggiornati in seguito a un cambiamento, il che può diventare complesso in schermate elaborate.

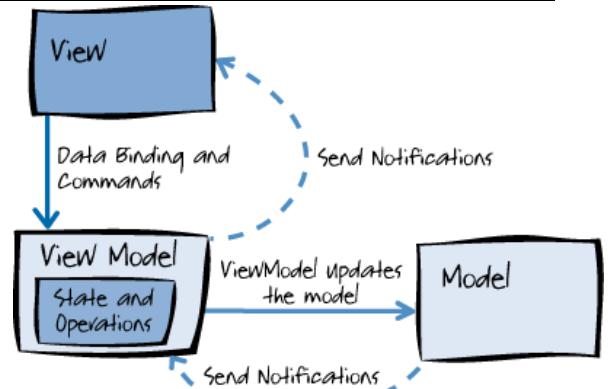
MVVM

Il pattern **MVVM Model-View-ViewModel** permette una gestione più fluida della UI e una separazione più chiara tra la logica di business e l'interfaccia utente.

Il pattern MVVM è composto da tre componenti principali:

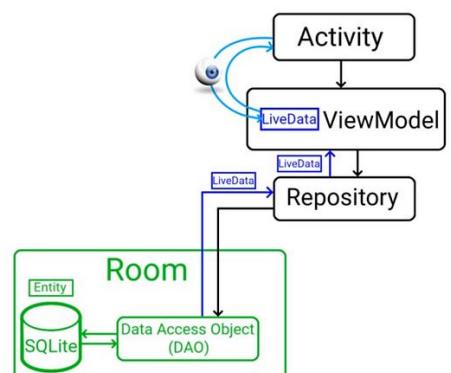
1. **Model**
2. **View**
3. **ViewModel** che da intermediario tra la

View e il **Model**. È responsabile della gestione dei dati da visualizzare nella UI e dell'elaborazione della logica necessaria per presentarli. La ViewModel fornisce i dati alla View tramite **LiveData**, che consente di osservare i cambiamenti dei dati e aggiornare automaticamente la UI quando necessario. La ViewModel non ha conoscenza diretta della View. Comunica con il Model per recuperare i dati e li prepara in una forma che la View può facilmente consumare.



Interazione tra i componenti nel pattern MVVM

1. L'utente interagisce con la **View** (ad esempio, tocca un pulsante).
2. La **View** invia un'azione al **ViewModel** (ad esempio, invoca un metodo che cambia i dati).
3. Il **ViewModel** interagisce con il **Model** per recuperare o modificare i dati. In caso di operazioni asincrone, il ViewModel gestisce il flusso di lavoro.
4. Quando il **Model** restituisce i dati (ad esempio tramite una chiamata API), il **ViewModel** li prepara (ad esempio, li converte in un formato adatto alla visualizzazione).
5. Il **ViewModel** aggiorna un **LiveData**, che è osservato dalla **View**.
6. La **View** riceve i nuovi dati tramite **LiveData** e aggiorna automaticamente l'interfaccia utente.



In questo modo, ogni componente ha un compito preciso:

- la **UI** si concentra solo sulla presentazione e l'interazione,
- il **ViewModel** gestisce la logica di visualizzazione,
- il **Repository** coordina l'accesso ai dati,
- e il livello **Model** (Room + Retrofit) fornisce le sorgenti dati reali.

NOTIFICHE

Una **notifica** è un messaggio che un'app visualizza all'utente al di fuori dell'interfaccia utente dell'applicazione. Quando si indica al sistema di emettere una notifica, questa appare per la prima volta all'utente come un'icona nell'area di notifica, sul lato sinistro della barra di stato. Per vedere i dettagli della notifica, l'utente apre il **drawer** di notifica o visualizza la notifica sulla schermata di blocco se il dispositivo è bloccato. L'area di notifica, la schermata di blocco e il drawer di notifica sono aree controllate dal sistema che l'utente può visualizzare in qualsiasi momento.

Le notifiche sono il modo in cui Android consente di visualizzare informazioni al di fuori della propria applicazione. Consentono all'utente di essere avvisato delle informazioni dalle applicazioni installate, anche se non le sta utilizzando al momento. Questo è particolarmente utile quando nuove informazioni, come email o messaggi, arrivano sul dispositivo in modo **asincrono**. Quando progettiamo le notifiche dobbiamo evitare di disturbare l'utente per notizie inutile, perché l'utente distraendosi può decidere di cancellare l'applicazione. Quando si progettano le notifiche, è fondamentale ricordare che **interrompono sempre l'utente**. Pertanto, devono essere,:

1. **Brevi:** Utilizza il minor numero possibile di parole. Sii conciso. Evita di irritare gli utenti inviando troppe notifiche o notifiche con contenuti inutili o fastidiosi.
2. **Tempestive:** Le notifiche devono apparire quando sono utili, altrimenti se arrivano i ritardo potremmo non servire più.
3. **Pertinenti:** l'informazione deve essere utile per l'utente. Notifiche che disturbino inutilmente l'utente o che lo possono trarre in inganno posso partare la cancellazione dell'applicazione.

La classe **NotificationCompat.Builder** permette di creare le notifiche ed è costruita secondo il pattern Builder. La notifica deve avere:

- Un **ID** che la identifica;
- Il **contesto**;
- **Icona piccola** da mostrare nella barra di stato. Questa è impostata con **setSmallIcon()**.
- **Titolo** mostrato sopra il testo dettagliato. È impostato con **setContentTitle()**.
- **Testo dettagliato** cioè il messaggio della notifica, un breve testo che descrive alcuni aspetti importanti È impostato con **setContentText()**.

Questi metodi setter fanno parte della classe Builder e sono costruiti usando l'**interfaccia fluent** un **modello di progettazione** che permette di scrivere il

codice in modo più leggibile e "fluente", simile a un linguaggio naturale, concatenando più metodi attraverso il **method chaining**. Quindi tutti i metodi di settaggio della notifica, in questo caso, restituiscono l'oggetto Builder in modo da apportare più modifiche. Poi alla fine del settaggio si dovrà invocare un metodo di chiusura per ottenere l'oggetto.

La notifica può essere estesa mostrando più informazioni. Si usano le **viste espande** nel drawer di notifica

1. **NotificationCompat.BigTextStyle**: Utilizzata per notifiche di grande formato che includono molto testo.
2. **NotificationCompat.InboxStyle**: Utilizzata per notifiche di grande formato che includono una lista di fino a cinque stringhe.
3. **NotificationCompat.BigPictureStyle**: Utilizzata per notifiche di grande formato che includono un **grande allegato immagine**. Puoi impostare l'immagine grande con `bigPicture()` e anche un titolo per il contenuto grande con `setBigContentTitle()`.
4. **Notification.MediaStyle**: Utilizzata per notifiche di **riproduzione multimediale**.

Per applicare uno di questi stili a una notifica, utilizzi il metodo `setStyle()` sull'oggetto `NotificationCompat.Builder`. Quindi oltre ad implementare un pattern Builder, ne implementa anche una decorator.

Oltre al contenuto testuale o multimediale aggiuntivo, puoi mostrare più informazioni aggiungendo **azioni** alla notifica che l'utente può eseguire sulla notifica stessa, resa disponibile tramite un pulsante di azione. Queste azioni aggiungono funzionalità e permettono all'utente di interagire direttamente con la notifica per accedere o manipolare le informazioni.

Pending Intent

Quando usiamo delle notifiche modifichiamo il pattern navigazionale dell'applicazione, perché per evitare di far percorrere tutta la strada per arrivare al punto di interesse, possiamo permettere all'utente di cliccare sulla notifica e di essere indirizzato direttamente lì. Quindi aggiungiamo dei deep link per arrivare direttamente nelle zone di interesse.

Questo è realizzato tramite `PendingIntent` un oggetto che **incapsula un Intent**. Il suo scopo è permettere a un'altra applicazione o al sistema Android di eseguire un Intent per conto della tua applicazione, in un momento futuro.

Questo è particolarmente utile perché garantisce che il sistema possa consegnare l'Intent anche se la tua app non è in esecuzione nel momento in cui l'utente interagisce con esso.

Quando si crea un `PendingIntent` si ottiene un **token** che non è l'Intent stesso, ma una **riferimento o un permesso** mantenuto dal sistema operativo. Quando

consegni questo PendingIntent (token) a un altro componente, quel componente può utilizzare il token per chiedere al sistema di eseguire l'Intent originale come se fosse stato avviato dalla app che l'ha creato.

I PendingIntent sono comunemente utilizzati in vari scenari in cui un'azione deve essere posticipata o eseguita da un altro componente di sistema o un'altra app per conto della tua:

- **Notifiche:** Quando l'utente tocca una notifica, solitamente si vuole che venga avviata un'Activity della tua app. Per fare ciò, si incapsula l'Intent che avvia l'Activity in un PendingIntent e lo si imposta sulla notifica usando `setContentIntent()`.
- **Allarmi:** L'`AlarmManager` può attivare un PendingIntent dopo un certo periodo di tempo o a intervalli regolari.
- **Broadcast:** Un PendingIntent può incapsulare un Intent da inviare come broadcast,

Per creare un'istanza di un PendingIntent, si utilizzano metodi statici appropriati a seconda di come si desidera che l'Intent contenuto venga consegnato:

- `PendingIntent.getActivity()`: Per un Intent che dovrebbe essere consegnato usando `startActivity()`. Si passa un Intent esplicito per l'Activity che si desidera avviare.
- `PendingIntent.getService()`: Per un Intent che dovrebbe essere passato a `startService()`
- `PendingIntent.getBroadcast()`: Per un Intent broadcast consegnato con `sendBroadcast()`.

Ciascuno di questi metodi per creare PendingIntent richiede solitamente i seguenti argomenti:

1. Il contesto dell'applicazione.
2. Un codice di richiesta (un ID per distinguere i PendingIntent).
3. L'Intent da consegnare.
4. Un flag PendingIntent che determina come il sistema gestisce più oggetti PendingIntent dalla stessa applicazione.

Se richiedi un PendingIntent due volte con gli stessi parametri si ottiene lo stesso oggetto PendingIntent.

Notification Channel

Un altro principio di progettazione è quello di **dare agli utenti la possibilità di scegliere** tramite le impostazioni dell'app i tipi di notifiche che desiderano ricevere e come desiderano riceverle. Questo è possibile realizzarlo a partire dall'API26 che permette di creare dei **NotificationChannel** per offrire all'utente un maggiore controllo sui tipi di notifiche che riceve dall'applicazione. Quando

si crea un canale, si definiscono alcune impostazioni iniziali, ma l'utente può personalizzare ciascun canale e decidere come si comporta.

- **Azioni:** Un'azione che l'utente può intraprendere sulla notifica. L'azione è resa disponibile tramite un pulsante adiacente al contenuto della notifica. Un'azione utilizza un PendingIntent per completare l'azione. Si aggiunge un'azione usando il metodo `addAction()` passando l'icona, la stringa del titolo e il PendingIntent da attivare.
- **Importance**
- **Priorità:** Influenzano il modo in cui il sistema Android consegnerà la notifica. Le notifiche hanno una priorità tra MIN (-2) e MAX (2). Le priorità disponibili includono PRIORITY_MAX (critiche/urgenti), PRIORITY_HIGH (comunicazioni importanti come messaggi), PRIORITY_DEFAULT (quelle che non rientrano nelle altre categorie), PRIORITY_LOW (informazioni non urgenti) e PRIORITY_MIN (informazioni di sfondo "nice-to-know"). La priorità è impostata con `setPriority()`. Sono rimaste più per un fatto di compatibilità con le versioni precedenti.
- **Notifiche continue/Servizi in foreground (setOngoing(), startForeground()):** Le notifiche continue non possono essere eliminate dall'utente e devono essere esplicitamente cancellate dall'app. Vengono utilizzate per indicare attività in background con cui l'utente interagisce attivamente (come la riproduzione di musica) o attività che occupano il dispositivo (come download). Per rendere una notifica continua, si imposta `setOngoing()` su true. Un servizio in foreground richiede che visualizzi una notifica visibile all'utente. Per avviare un servizio in modalità foreground e visualizzare la notifica, si utilizza `startForegroundService()` e `startForeground()`, passando un ID di notifica univoco e l'oggetto notifica. L'ID intero passato a `startForeground()` non deve essere 0. Per rimuovere un servizio dal foreground, si chiama `stopForeground()`, specificando se rimuovere la notifica.

Gestione delle notifiche

Il **NotificationManager** è un **servizio di sistema** utilizzato per **consegnare** o attivare le notifiche. Per ottenere un'istanza del **NotificationManager**, devi chiamare il metodo `getSystemService()`, passando la costante **NOTIFICATION_SERVICE**. Non si dovrebbe mai chiamare questo direttamente, ma usare il **NotificationMangerCompact** che garantisce anche la retrocompatibilità con le versioni precedenti.

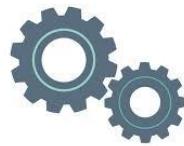
La funzione principale del **NotificationManager** è quella di **mostrare la notifica all'utente**. Ciò si fa chiamando il metodo `notify()`. Il metodo `notify()` richiede due parametri:

1. Un **ID di notifica** (un numero intero), utilizzato per aggiornare o annullare la notifica in seguito. Dovrebbe essere unico all'interno della tua applicazione. Se chiavi `notify()` più volte con lo stesso tag e ID, **sostituirà qualsiasi notifica esistente con lo stesso tag e ID**. Questo è il modo in cui puoi implementare, ad esempio, una barra di avanzamento o altre visualizzazioni dinamiche in una notifica.
2. L'**oggetto Notification** stesso. Questo oggetto viene tipicamente costruito utilizzando la classe `NotificationCompat.Builder`.

A partire dall'API 26 , il `NotificationManager` è anche responsabile della gestione dei **canali di notifica**, infatti è obbligatorio associare la notifica ad un canale. Puoi utilizzare il `NotificationManager` per controllare se un canale esiste già e, in caso contrario, creare un nuovo oggetto. Questo dà all'utente un maggiore controllo sui tipi di notifiche che riceve dall'applicazione, poiché può personalizzare le impostazioni per ciascun canale.

Le notifiche rimangono visibili fino a quando non si verifica una delle seguenti condizioni:

- L'utente le ignora individualmente o tramite "Cancella tutto"
- Chiavi `setAutoCancel(true)` sul builder della notifica, in tal caso la notifica scompare quando l'utente ci clicca sopra.
- Chiavi `cancel()` per un ID di notifica specifico.
- Chiavi `cancelAll()` per far sparire tutte le notifiche che hai emesso.



APP SETTING

Le app settings si riferiscono alle **preferenze dell'utente** che consentono di **modificare le caratteristiche e i comportamenti dell'applicazione**. Sono controlli che catturano le preferenze degli utenti che influiscono sulla maggior parte di essi o forniscono supporto critico a una minoranza. Esempi includono l'abilitazione delle notifiche o la frequenza di sincronizzazione dei dati con il cloud. In iOS, concetti simili ("user defaults") sono usati per preferenze come lingua, stile UI o unità di misura.

Gli utenti dovrebbero poter navigare alle impostazioni dell'app toccando un'opzione **Settings**. Questa opzione è solitamente collocata nella **navigazione laterale** o nel **menu opzioni**. Secondo le linee guida di design, l'opzione "Settings" dovrebbe trovarsi al di sotto di tutti gli altri elementi (eccetto Help e Send Feedback) sia nella navigazione laterale che nel menu opzioni.

Le impostazioni sono solitamente accessibili **infrequentemente**, poiché una volta che l'utente cambia un'impostazione, raramente ha bisogno di modificarla di nuovo. Se un controllo o una preferenza necessita di accesso frequente, è meglio spostarlo nel menu opzioni della app bar o in un menu di navigazione laterale. È importante impostare dei **valori di default** per le impostazioni che siano familiari agli utenti e migliorino l'esperienza dell'app. I default dovrebbero rappresentare la scelta più comune, usare meno batteria, comportare il minor rischio per la sicurezza/perdita di dati, e interrompere solo quando importante. Informazioni sull'app come numero di versione o licenze dovrebbero essere spostate in una schermata di Help separata.

Settings UI

Per costruire la UI delle impostazioni, si usano **sottoclassi della classe Preference** piuttosto che oggetti View. La classe **Preference** fornisce la View da visualizzare per ogni impostazione e si associa a un'interfaccia **SharedPreferences** per memorizzare/recuperare i dati.

Per visualizzare una lista di impostazioni, si usa una **Activity o un Fragment specializzato**. La best practice per Android 3.0+ è usare una **SettingsActivity** che ospita un **PreferenceFragment**. Per compatibilità con la v7 appcompat library, si estende la Settings Activity con **AppCompatActivity** e il fragment con **PreferenceFragmentCompat**. Per versioni più vecchie di Android (<3.0), si usa **PreferenceActivity**. L'uso di **PreferenceFragment** è generalmente più flessibile rispetto alla sola **PreferenceActivity**.

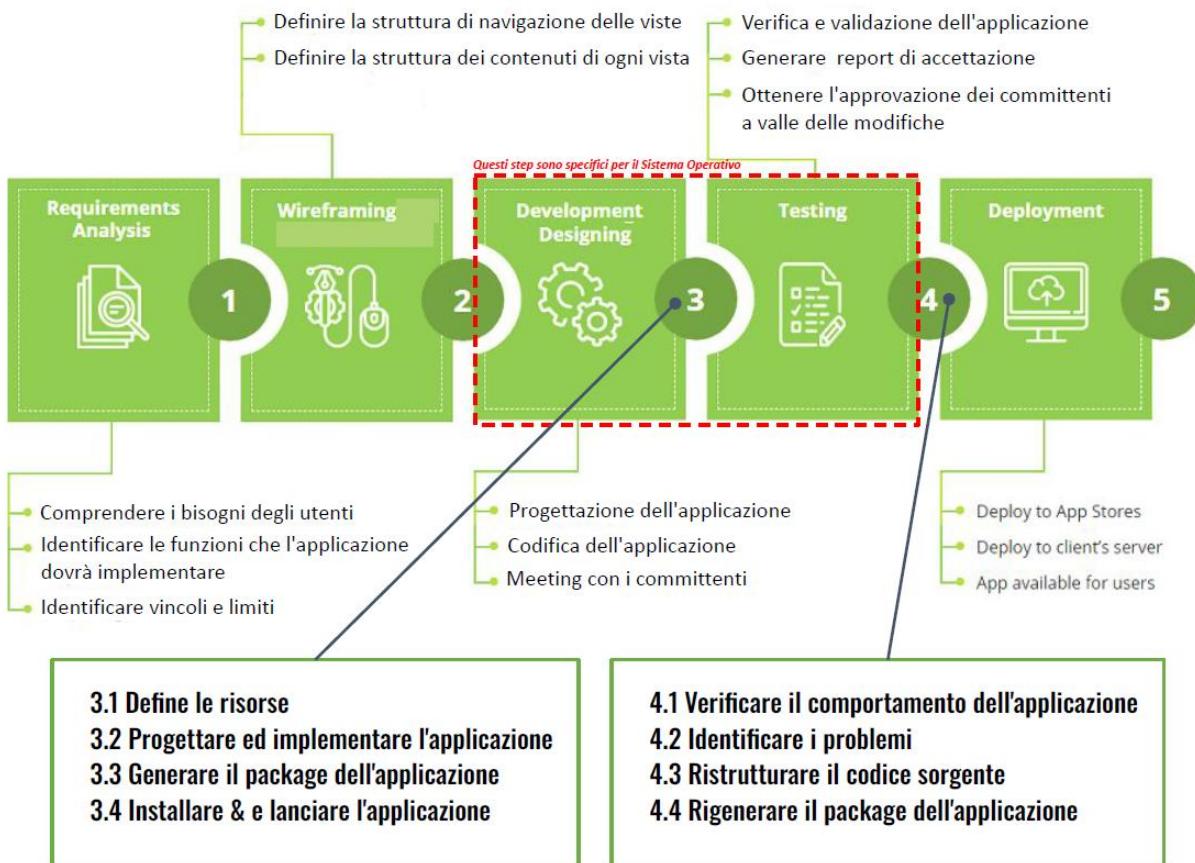
Android Studio fornisce un **template Settings Activity**. Questo template facilita la creazione di schermate di impostazioni, specialmente se si hanno più gruppi di impostazioni. Fornisce layout diversi per smartphone (header links) e tablet (master/detail view). Il template crea i file XML (res/xml/) che definiscono le impostazioni (pref_data_sync.xml, pref_general.xml, pref_headers.xml, pref_notification.xml), risorse stringa (strings.xml), e le classi Activity/Fragment necessarie (SettingsActivity, AppCompatPreferenceActivity). Il template include anche la funzionalità per ascoltare i cambiamenti delle impostazioni e aggiornare il riassunto (summary).

La gerarchia delle impostazioni inizia con un layout **PreferenceScreen**.

1. **Tipi di Controlli UI:** Diverse sottoclassi di Preference offrono UI appropriate per modificare le impostazioni:
 - CheckBoxPreference: Per impostazioni abilitate/disabilitate (valore booleano).
 - ListPreference: Mostra una lista di pulsanti radio in un dialogo.
 - SwitchPreference: Un'opzione toggle con due stati (on/off, true/false).
 - EditTextPreference: Un campo di testo in un dialogo per l'inserimento di stringhe.
 - RingtonePreference: Consente all'utente di scegliere una suoneria.

I valori delle impostazioni vengono memorizzati in un file **SharedPreferences**. Questo è un modo comune per persistere piccole quantità di dati primitivi come coppie chiave-valore tra le sessioni dell'app. Ogni oggetto Preference (impostazione) ha una coppia chiave-valore corrispondente che il sistema usa per salvare il valore in un file SharedPreferences predefinito.

Dopo aver salvato le impostazioni, è necessario **leggere i valori** dallo SharedPreferences file per usarli nella logica dell'app. Si può anche implementare un listener (Preference.OnPreferenceChangeListener) per reagire immediatamente ai cambiamenti di un'impostazione.



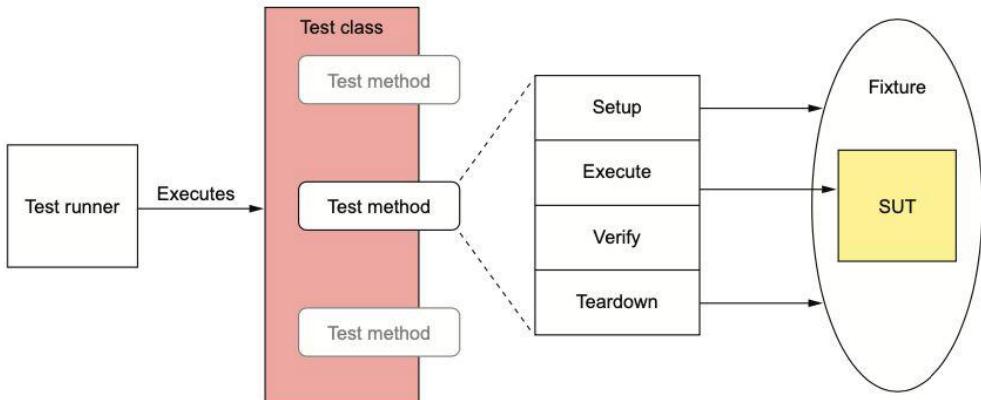
Il testing è fondamentale per assicurarsi che un'app si comporti come previsto in ogni situazione, specialmente man mano che l'app cresce e cambia.

Aiuta a **identificare i problemi nelle prime fasi dello sviluppo**, quando sono meno costosi da risolvere, e migliora la robustezza del codice. Ci sono vari tipi di test:

- **unit test** è una verifica che si concentra sul comportamento di una singola unità di codice. Questa "unità" di codice è solitamente una singola funzione o un metodo. L'obiettivo è garantire che quella specifica parte del codice funzioni correttamente in isolamento, senza dipendenze esterne.
- **Component test** si verifica se le classi fra loro collaborano bene
- **integration test** si concentrano sull'interazione tra più unità di codice, per verificare che diversi componenti funzionino insieme come previsto. Mentre i unit test testano singole unità in isolamento, gli integration test verificano la corretta integrazione di più unità, come moduli, librerie o servizi, e se queste interazioni producono il comportamento desiderato.

Il testing automatizzato rende più facile eseguire test su diverse configurazioni di dispositivi e stati. Scriverli da tutti sarebbe una follia. 😊

Bisogna individuare i test suite e usare dei framework per poter scrivere test in modo automatico. Lo scopo dei test non è di rimuovere gli errori, ma di capire cosa non funziona per poterli risolvere il problema.



I test suite vengono dati ad un **test runner** che gli esegue e crea delle **test class** con i **method test** eseguendo per ogni metodo delle operazioni:

- **Setup:** Prima dell'esecuzione dei metodi di test, il test runner può invocare metodi di setup che sono utilizzati per configurare le risorse necessarie per il test
- **Esecuzione dei Test:** I metodi di test vengono eseguiti uno alla volta.
- **Tear Down:** Dopo l'esecuzione di un test, possono essere invocati metodi di teardown per liberare risorse, come la chiusura di connessioni al database o la pulizia di dati temporanei.

Una volta che i test sono stati eseguiti, il test runner crea un **report dei risultati**.

Questo report mostra:

- I test che sono passati.
- I test che sono falliti.
- I test che hanno generato errori (ad esempio, eccezioni non gestite).

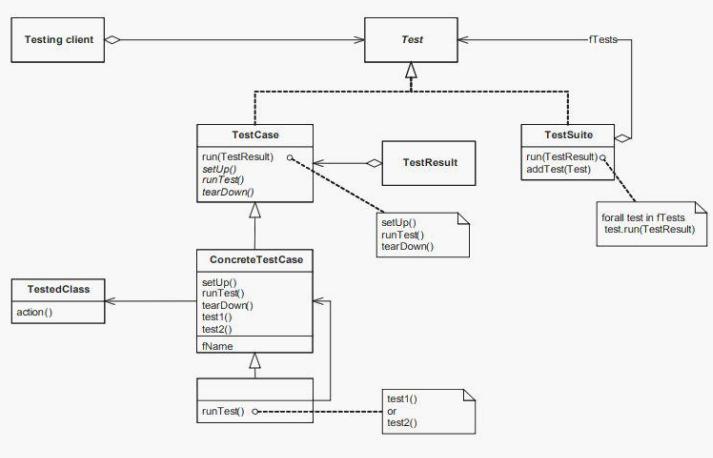
Android supporta diverse tipologie di test e framework di testing.

JUnit 4 è un framework comune per scrivere unit test in Java per Android.

Il framework JUnit implementa il composite, quindi possiede il **TestSuite** (compose) e i **TestCase** (leaf).

Per creare gli unit Test si crea una classe test che implementa i metodi:

- `setUp()`
- `tearDown`
- `TestXXX`
- `Suite()`



Nei metodi di test ci mettiamo le **asserzioni** per verificare che il comportamento del codice corrisponda a quanto previsto.

La libreria **Hamcrest** è uno strumento utile per scrivere asserzioni nei test, permettendo di creare regole di matching in modo dichiarativo.

1. Framework di Unit Test di "Prima Generazione"

Nei framework di unit test di prima generazione, il concetto di base è che un test deve verificare che una condizione sia vera durante l'esecuzione del codice. Questo avviene tramite l'istruzione assert, che verifica una condizione booleana e, se la condizione non è vera, fa fallire il test.

I framework di "prima generazione" forniscono un messaggio di errore generico, come "Assertion failed", che non è particolarmente utile per capire cosa sia andato storto.

2. Framework di Unit Test di "Seconda Generazione"

I framework di unit test di seconda generazione migliorano la situazione fornendo una serie di **asserzioni specializzate** che sono in grado di produrre messaggi di errore più chiari e specifici, adattandosi a diversi tipi di confronti (ad esempio, uguaglianza, differenza, ecc.).

Alcuni esempi includono:

- **assert_equal(x, y)**: Verifica che x e y siano uguali.
- **assert_not_equal(x, y)**: Verifica che x e y non siano uguali.

Queste asserzioni specializzate sono più esplicite e forniscono messaggi di errore più significativi. Tuttavia, il problema principale di questo approccio è che il numero di macro di asserzione tende a crescere rapidamente, perché ogni tipo di confronto richiede una macro di asserzione.

3. Framework di Unit Test di "Terza Generazione"

I framework di unit test di **terza generazione** (come **Hamcrest**, un popolare framework di asserzione in Java) risolvono questo problema introducendo un approccio più **generico e compositibile**. In questi framework, l'operatore assert_that viene combinato con **matcher** (oggetti che eseguono verifiche su una condizione).

Un **matcher** è una classe o un oggetto che definisce una condizione che può essere verificata. In questo caso, assert_that viene utilizzato per affermare che un oggetto soddisfi una determinata condizione definita dal matcher.



MOCKITO (mock a mamm't) ☺

Mockito è un framework di mocking che può essere utilizzato per isolare le unità di codice durante l'esecuzione di unit test.

Permette di creare oggetti **mock** (fittizi o simulati) che si comportano come le dipendenze di un'unità di codice che stai testando. Questo è utile perché consente di **testare la**

logica specifica di un componente in isolamento, senza doversi preoccupare del comportamento reale delle sue dipendenze.

Mockito è specificamente menzionato nel contesto della creazione di unit test. Quando si esegue un unit test, l'obiettivo è verificare la logica di una piccola porzione di codice (come un metodo o una classe).

Android permette due tipi di test:

- **Local Unit Test** → Questi test vengono compilati ed eseguiti interamente sulla tua macchina locale utilizzando la Java Virtual Machine (JVM). Sono utilizzati per testare la logica interna dell'app che non richiede l'accesso all'Android framework o a un dispositivo/emulatore.
- **Instrumented Test** → Questi test vengono eseguiti su un dispositivo o emulatore Android.

ESPRESSO

Il UI testing si concentra sul test degli aspetti dell'interfaccia utente e delle interazioni con gli utenti.

Espresso lavora con il test runner `AndroidJUnitRunner` e richiede instrumentation.

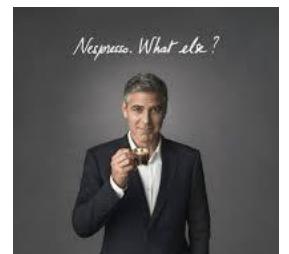
I test Espresso si basano sulla simulazione delle azioni che un utente potrebbe compiere: **trovare una view, eseguire un'azione** (es. clic) e **verificare il risultato** (asserzione sullo stato della view).

I test Espresso vengono eseguiti su dispositivi Android reali o emulatori. Richiedono l'strumentazione e funzionano con il test runner `AndroidJUnitRunner`.

- L'app restituisce l'output UI corretto in risposta a una sequenza di azioni dell'utente.
- I controlli di navigazione e input dell'app aprono le Activity, le View e i campi corretti.
- L'app risponde correttamente con dipendenze "mockate" (false) o può lavorare con metodi backend "stubbed out" (simulati).



Nepresso. What else?



Un vantaggio fondamentale di Espresso è l'accesso alle informazioni di strumentazione, come il contesto dell'app. Sincronizza automaticamente le azioni di test con l'UI dell'app, rilevando quando il thread principale è inattivo. Ciò permette ai test di essere eseguiti al momento opportuno, migliorando l'affidabilità ed evitando la necessità di workaround basati sul tempo, come i ritardi (sleep) nel codice di test.

La scrittura dei test Espresso si basa su ciò che farebbe un utente. I passaggi fondamentali sono:

1. **Trovare una vista (Match a view):** Individuare l'elemento UI con cui interagire, spesso usando il metodo onView().
2. **Eseguire un'azione (Perform an action):** Interagire con la vista trovata, ad esempio cliccando (click()).
3. **Asserire e verificare il risultato (Assert and verify the result):** Controllare lo stato della vista o l'output per vedere se corrisponde allo stato o al comportamento atteso.

Espresso ha una sintassi fluida e un paradigma funzionale. Il framework Hamcrest è comunemente usato con Espresso per le asserzioni. Permette di creare **matcher** personalizzati e combinare espressioni per definire regole di corrispondenza in modo dichiarativo. Questo porta a messaggi di errore più utili rispetto alle semplici asserzioni booleane.