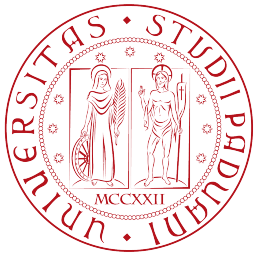


Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in
Ingegneria Elettronica

Caricamento dinamico di codice nella piattaforma Android

Relatore:
Prof. Carlo Fantozzi

Laureando:
Edi Linguanotto
1216274

Anno Accademico 2021/2022
Data Laurea: 21 settembre 2022

Abstract

In questa tesina viene presentata una applicazione Android in grado di caricare dinamicamente codice, seguendo la procedura di: selezione file.apk da parte dell'utente, verifica della compattibilità da parte dell'applicazione e poi effettivo caricamento del file di estensione. L'applicazione è in grado di caricare e usare le classi/metodi e altri contenuti dell'estensione.

Indice

Capitolo 1

Introduzione

L'obbiettivo del lavoro descritto in questa tesina, è stato quello di progettare e realizzare una applicazione che chiameremo "Bridge" che è in grado di usare il contenuto di altri file in formato apk, i quali non contengono una activity, ovvero che questi file.apk non fanno nulla presi singolarmente. Chiameremo questi file.apk con il nome "Estensioni", come suggerisce il nome, estenderanno le funzionalità del Bridge quando verranno implementate al suo interno. La procedura eseguita, ovvero l'utilizzo di un file esterno che contiene codice sconosciuto a chi lo legge (ovvero il bridge), viene chiamata con il nome "Caricamento di codice dinamico", infatti viene eseguito da parte del framework di Android una funzione chiamata "Load". Tale procedura è vantaggiosa solo nei casi in cui l'applicazione può eseguire molte funzioni ausiliarie e future, in modo tale che la base dell'applicazione risulti leggera ed esegue le funzioni essenziali, nel caso si vuole una funzione in particolare si ricorrere alla sua implementazione tramite estensione. Rimane comunque una procedura che in Android non si fa frequentemente, poichè le implementazioni in Android sono scomode, sia nel loro uso che realizzazione oltre ad altri problemi presenti anche in altre piattaforme che riguardano più che altro la sicurezza.
Il caricamento dinamico principalmente ha i seguenti vantaggi e svantaggi:

- Gli svantaggi riguardano la sicurezza poichè non si sa cosa si sta caricando, e quindi la possibilità di caricare codice malevolo.
- I vantaggi invece sono, come citato precendentemente, la possibilità di estendere le funzionalità dell'applicazione principale senza reinstallarla o avere versioni particolari di essa.

Per esemplificare questo sistema di caricamento del codice dinamico, è stata realizzata una calcolatrice la quale riesce a utilizzare estensioni che aggiungono funzionalità di esecuzione.

Capitolo 2

Caricamento di codice dinamico

Per eseguire il caricamento dinamico del codice, viene usato il contenuto del pacchetto “dalvik.system” un pacchetto di classi, le quali sono specializzate nel caricare il contenuto di un file dex, al loro interno viene realizzato quello che viene definito ”caricatore di classi”, che è un oggetto informatico resposabile del caricamento di esse, in Android però non è presente il bytecode di Java ma una sua rielaborazione: il bytecode Dalvik. Per caricare un file contenente bytecode Dalvik non si usa un caricatore di file generico fornito dal framework di Java, perchè funziona solo per caricare file.jar. Il pacchetto di classi “dalvik.system” fornisce la classe astratta “BaseDexClassLoader” che estende la definizione di ”ClassLoader” che è una classe astratta generica che definisce come deve essere fatta l’interfaccia pubblica di un Loader secondo il framework di Java. La classe “BaseDexClassLoader” sarà la base per i Loader presenti che caricano bytcode dalvik.

All’interno di un file ci sono più classi e risorse: quando si vuole caricare una classe bisogna specificare quale classe fra le presenti bisogna caricare, quindi specificare il percorso interno dei package e infine il nome della classe desiderata. La funzione di riferimento per fare ciò è

```
1 //Caricare la classe specificata
2 Class<?> classe = loader.loadClass("package.Nome_classe")
```

Listato 2.1: Esempio caricamento di una classe.

Essa ritorna come risultato un oggetto “Class<?>”. Tale oggetto descrive il contenuto della classe caricata: avendo quest’ultima, è possibile creare una istanza e quindi finalmente creare/allocare l’oggetto e poi usarlo.

2.1 Problemi

Nella realizzazione dell'applicazione sono stati riscontrati alcuni problemi i quali:

- "ClassLoader" è un classe astratta e quindi non utilizzabile direttamente bisogna utilizzare un oggetto che la concretizzi.
- Come so dov'è la classe da caricare?
- Il tipo della classe che carico qual'è?
- Dopo aver caricato tale classe come faccio ad usarla?
- Come caricare una interfaccia grafica?

2.2 Class<?>

Le istanze della classe Class rappresentano classi e interfacce in un'applicazione Java in esecuzione. Anche i tipi Java primitivi (boolean, byte, char, short, int, long, floate double) e la parola chiave void, sono rappresentati come oggetti Class. Il punto interrogativo in Class<?>, indica che non si sa a quale classe si riferisce quindi "Any", una qualsiasi, semmai verrà eseguita la sua allocazione, verrà ritornato l'oggetto generico di Java ovvero "Object", mentre se al posto del punto interrogativo ci fosse scritto il nome di una classe, quando verrà richiesta la sua instanziazione tramite la funzione "newInstance()", verrà ritornato come risultato l'oggetto specificato.
Tuttavia, quando si carica una classe tramite la funzione sopracitata, il loader non sa la natura della classe che sta caricando, quindi non si conosce l' interfaccia pubblica della classe caricata.

```
1 //Instanzia una classe generica
2 Class<?> classe = ... //ottenuta in un modo X
3 Object objA = classe.newInstance()
4
5 //Instanzia una classe nota "Calculator"
6 Class<Calculator> classe = ... //ottenuta in un modo Y
7 Calculator objB = classe.newInstance()
```

Listato 2.2: Esempi di instanziazione.

2.3 Si può caricare tutto di un apk?

Di per sè, si può leggere il contenuto di tutti i file, il problema sta nel corretto utilizzo.
Android Studio permette di creare un'interfaccia grafica in modo dichiarativo, un metodo molto comodo, però, gli elementi grafici, e anche altri elementi dichiarati nello stesso modo, vanno identificati tramite un ID. Per usare questi ID non si usa un numero ma un nome che lo identifica. L'associazione numerica viene fatta durante il build dell'apk partendo da 0 fino a N, quindi lo stesso ID di un apk potrebbe essere utilizzato da un'altro apk che però punta ad altro. Infatti gli ID sono "validi" solo all'interno dell'applicazione in uso.
Di conseguenza, non si può creare una interfaccia grafica in modo dichiarativo in una estensione poichè, anche se leggo il file.xml in cui c'è la descrizione del layout, quest'ultimo non potrà essere usato perchè ha ID probabilmente in conflitto con gli ID del bridge.

Capitolo 3

PathClassLoader

Il PatthClassLoader fornisce una semplice implementazione del ClassLoader che opera su un elenco di file e directory nel file system locale, ma non tenta di caricare classi dalla rete. Android usa questa classe per il caricatore di classi di sistema e per i caricatori di classi di applicazioni.
Verrà utilzzato questo oggetto fornito dali framework di Android per caricare l'estensione voluta, però per farlo bisogna specificare il percorso del file da caricare.

```
1 //Costruttori pubblici
2 PathClassLoader(String path, ClassLoader parent)
3
4 PathClassLoader(String path, String libPath, ClassLoader parent)
```

Listato 3.1: Costruttori per PathClassLoader.

In entrambi i casi bisogna avere un percorso ben definito che localizza il file.

```
1 //Caricamento di un file
2 File file = ... //ottenimento del file da caricare
3 PathClassLoader(file.path, activity.classLoader)
```

Listato 3.2: Caricamento di un file.

Nel costruttore per la variabile "parent" viene usato "activity.classLoader", come suggerisce il nome è un oggetto di tipo "ClassLoader" tale oggetto è quello utilizzato per l'applicazione "Bridge" all'avvio. Perchè, se si ottiene un Loader generico, ottenibile ad esempio con

```
1 //Un altro classLoader
2 ClassLoader loader = ClassLoader.getSystemClassLoader()
```

Listato 3.3: Ottenimento di un classLoader non corretto.

il caricamento apparentemente va a buon fine però, quando si vorrà eseguire l'instanziazione di una classe

```
1 //Un altro classLoader
2 File file = ...//ottini il file da caricare
3 ClassLoader sysLoader = ClassLoader.getSystemClassLoader()
4 PathClassLoader loader = PathClassLoader(file.path, sysLoader)
5 Class<out AbstractCalculator> calcClass = loader.loadClass("it.
    ConcreteCalculator")
6 AbstractCalculator calculator = calcClass.newInstance() //Verra
    lanciata una eccezione
```

Listato 3.4: Tentativo di caricare un file.

succede che poco dopo l'instanziazione la JVM (Java Virtual Machine) non riesce ad eseguire il Cast, difatti sulla console, comparirà un errore bizzarro ovvero, "ClassCastException". Sapendo che la classe da caricare estende "AbstractCalculator" quando la si instanzia si dovrebbe poterla parametrizzare con il tipo astratto, per via del polimorfismo, invece comparire come errore un "ClassCastException", perchè il loader ottenuto con "ClassLoader.getSystemClassLoader()" non è adatto per eseguire questo caricamento: in particolare, bisogna usare il loader come mostrato nel listato 3.2, altrimenti il caricamento e l'instanziazione non andrà a buon fine.

Questa è la soluzione del primo problema ovvero, quello che si riferisce a che "ClassLoader" concreto usare.

Capitolo 4

Struttura fissa

Utilizzando PathClassLoader ho un caricatore specifico di un file in formato apk, la loro struttura interna è ben definita Sono sempre presenti in un file.apk sono

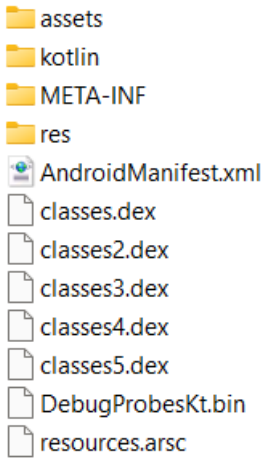


Figura 4.1: Struttura interna file.apk

gli elementi seguenti

- AndroidManifest.xml : file che contiene varie informazioni per il sistema Android, per esempio l'icona da usare, le modalità di avvio dell'app, quale classe chiamare per eseguire l'avvio dell'app, e altre specifiche.
- Cartella "res" : al suo interno ci sono svariate sottocartelle per tutte le risorse, come immagini, musica, video, file.xml per i layout/navigation/stringhe/eccetera. Le sotto cartelle hanno dei nomi ben definiti: da sottolineare che c'è né una in particolare che si chiama "drawable-v24" che verrà utilizzata più avanti. Le risorse possono essere raggruppate concettualemente in queste sotto cartelle:

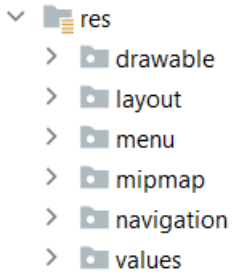


Figura 4.2: Raggruppamento Res.apk

Quando avviene il build vengono generate altre con un nome predefinito fra cui "drawable-v24". Sapere questo ci sarà utile più avanti.

- Cartella "assets" : al suo interno ci sono vari file a discrezione del programmatore.
- Cartella "kotlin" : al suo interno ci sono librerie riguardanti il linguaggio di programmazione "Kotlin"; non ci servirà il contenuto.
- Cartella "META_INF" : al suo interno ci sono le librerie utilizzate per realizzare l'app; non ci servirà il contenuto.
- File "classesX.dex"; X=1,2,3,4,5 : questi file contengono il bytecode di tutte le classi scritte per realizzare l'app.
- File "DebugProbestKt.bin" e "resources.arsc" : file di supporto per il funzionamento dell'app.

Il Loader conosce la struttura e quando cerca la classe desiderata, la cerca nei file.dex, però rimane il problema di come si chiama la classe e il percorso del package. Per risolvere a questo problema, si impone un' ulteriore struttura alle estensioni, sfruttando la cartella "assets". Poichè si può inserire al suo interno i file a discrezione del programmatore, si crea un file che verrà chiamato "Estensione.json",

```
1 {
2   "main": "com.example.calculatorresistance.
     ConscreteLoadClass",
3   "name": "CalculatorResistance"
4 }
```

Listato 4.1: Esempio Estensione.json.

così, in ogni caso, quando si cerca questo file, si sa esattamente dov'è, e si può leggerne il contenuto: al suo interno ci saranno scritti nome e percorso interno della classe, così da poter caricare la classe corretta voluta. Questo risolve il problema di dov'è la classe da caricare.

Per ralizzare l'estensione, si voleva anche avere un icona visualizzabile all'interno del Bridge, una immagine.png. Come per la classe, nasce il problema di quale immagine caricare, risolvibile usando la struttura interna della cartella "Drawable-v24" sopracitata, e in aggiunta il nome di questa immagine dovrà essere una in particolare, in modo da distinguerla dalla presenza di altre immagini questo nome deve essere "icona.png", questo risolve alla richiesta della presenza di una immagine rappresentativa dell'estensione dentro il Bridge.

Ora si sa dove sono tutti i file da caricare: se uno di questi dovesse mancare, l'estensione non è valida e quindi il caricamento viene annullato.
Ricapitolando una estensione deve contenere i seguenti file.

- asset\Estensione.json : contiene il prercorso della classe da caricare.
- res\Drawable-v24\icona.png : icona rappresentativa dell'estensione.
- Presenza della classe specificata nel file Estensione.json all'interno nei file.dex

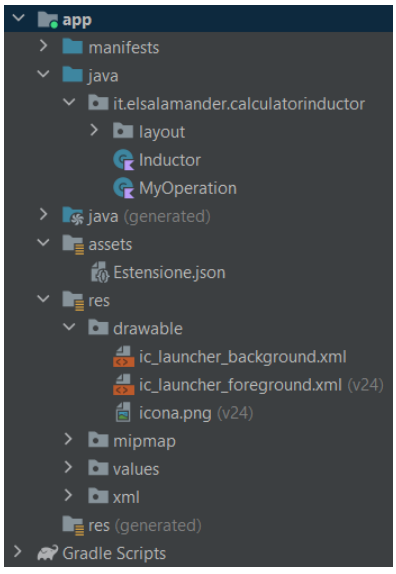


Figura 4.3: Progetto di una estensione

La struttura finale di una estensione realizzata è visibile in figura 4.3

Capitolo 5

Risoluzione interfacce pubbliche

Come utilizzare la classe caricata?
Per utilizzare una classe, bisogna conoscerne l'interfaccia pubblica, altrimenti non potrò invocare le funzioni volute. Per risolvere il problema dell'interfaccia è stata creata una libreria implementata sul Bridge e su tutte le estensioni, quindi, una estensione per essere creata, deve inanzi tutto implementare la libreria. All'interno della libreria, c'è la classe astratta "AbstractLoadClass", questa classe impone un'interfaccia pubblica a chi l'estende, si fa estendere questa classe a quella che viene caricata, ovvero la classe specificata nel file Estensione.json, imponendo la sua estensione, ora so l'interfaccia pubblica della classe che viene caricata.

```
try{
  Class<out AbstractLoadClass> classe = loader.loadClass("pck.
   nome") as Class<out AbstractLoadClass>
}catch(e : ClassNotFoundException){
  e.printStackTrace()
  throw ExtensionClassNotFound("Classe non trovata")
}catch (e : ClassCastException){
  e.printStackTrace()
  throw ExtensionClassNotFound("Classe trovata, ma non castabile"
   )
}
```

Listato 5.1: Caricamento classe con Cast.

In questo modo si può invocare la classe,tramite i metodi definiti da "AbstractLoadClass", che in questo caso è più che sufficente.
Quindi l'utilizzo di una libreria che tutti conoscono che definisce strutture preferibilmente astratte, in modo tale da imporre interfacce pubbliche da una parte, e poterle così leggere dall'altra poichè so che è stata imposta, risolve al problema

dell'interfaccia.

```
class ConscreteLoadClass : AbstractLoadClass(MyOperation()),
    OnStartUpExtension {

    override fun getTitle(): String {
        return "Calculator Resistance"
    }

    override fun getDescription(): String {
        return "Specializzata per resistenze"
    }

    override fun getFragment(context: Context): Fragment {
        return FragmentResistance(context)
    }

    override fun doOnStartUp(param: Holder, newLoad: Boolean) {
        Log.d("Test loading", "Invocazione eseguita solo dopo
    essere stata caricata")
    }
}
```

Listato 5.2: Esempio estensione creata.

Questo meccanisco risolve i problemi:

- La natura della classe che carico qual'è?
- Dopo aver caricato tale classe come faccio ad usarla?

Capitolo 6

Aggiungere interfaccie grafiche

Per risolvere al problema di: come aggiungere interfacce grafiche tramite le estensioni; ci sono dei requisiti da soddisfare nell'app che utilizza le estensioni, tali requisiti sono:

- Utilizzo del Navigator per l'interfaccia grafica;
- Un fragment vuoto, dovuto da un difetto del Navigator che viene discusso inseguito.

Il navigator, come per i layout grafici, è possibile crearlo in modo dichiarativo però, ciò significa che per forze di cose ci sono due fattori impossibili da soddisfare:

- Il layout deve essere contenuto nel Bridge;
- L'oggetto Fragment deve essere anch'esso presente nel Bridge;

Entrambe le cose però NON sono nel bridge ma nell'estensione!
La soluzione potrebbe essere quelle di manipolare il Navigator in modo programmatico: quando vengono caricate le estensioni si posso creare le "navigazioni", ovvero modificare il grafo che mappa i fragment e le loro azioni a piacimento. Il problema è che questo meccanismo vuole il nome della classe del fragment, che realizza l'interfaccia grafica, qui nasce un'altro problema, che non è proprio dargli il nome, poichè si può tranquillamente realizzare il fragment nell'estensione, ma il Navigator quando vuole caricare il fragment lo cerca all'interno del Bridge, ovvero nel posto sbagliato perchè non si trova lì, quindi modificare il grafo del Navigator è inutile perchè, è comunque richiesta la presenza dei fragment all'interno del bridge. Per ovviare a questo, è necessario usare un meccanismo che non necessita di un nome ma bensì di un oggetto fragment già allocato. Ci viene in

aiuto l'evento di navigazione: quando avviene una navigazione tra due fragment posso alterare il fragment di destinazione, inserendo il fragment già allocato. Per fare ciò si è crato un fragment vuoto, e quando si vuole aprire una interfaccia grafica di una estensione, si cambia il fragment in visualizzazione con quello vuoto tramite il Navigator, poi si cambia la destinazione con quella voluta: in questo modo verrà mostrato il fragment all'interno dell'estensione.

```
1  //Prendi la direzione di navigazione
2  val direction = NavGraphDirections.navigationMove(value)
3
4  //Naviga al fragment, in questo caso quello vuoto
5  navController.navigate(direction)
6
7  //ottieni il supporto per eseguire le modifiche
8  val fm: FragmentManager = supportFragmentManager
9  val fragmentTransaction : FragmentTransaction = fm.
       beginTransaction()
10
11 //prendi il fragment dell'estensione
12 extFrag = managerExtentions.extentions[value]!!.second.
       getFragment(this)
13
14 //aggiungilo alla destinazione
15 fragmentTransaction.add(R.id.nav_host_fragment,extFrag!!, null)
16 fragmentTransaction.addToBackStack(null)
17
18 //applica le modifiche
19 fragmentTransaction.commit()
```

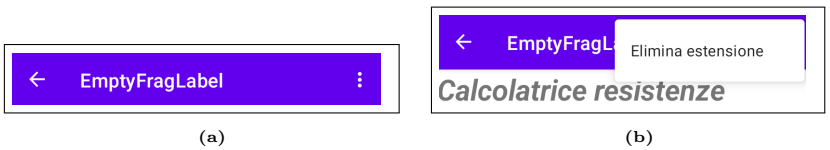
Listato 6.1: Caricamento classe con Cast.

Ora bisogna risolvere la problematica di come creare l'interfaccia grafica del fragment, perchè ora come ora, si è risolto solo il problema di come mostrare il fragment all'interno dell'estensione. Per realizzare l'interfaccia grafica, come discusso precedentemente, non si può fare uso del metodo dichiarativo, quindi bisogna ripiegare sul metodo programmatico. Di per sè questa è la soluzione, perchè si può creare una interfaccia grafica tranquillamente anche a codice, completa di eventi di interazione.

Capitolo 7

Un esempio completo

Per mostrare l'utilità del caricamente di codice dinamico è stata realizzata una calcolatrice, nella quale possono essere introdotti nuovi operatori e funzioni; in aggiunta, c'è la possbilità di aprire l'eventuale interfaccia grafica delle estensioni. Tali funzioni, sono descritte sempre nelle estensioni e devono rispettare una interfaccia pubblica definita dalla calcolatrice base, che è il bridge. Per aggiungere una estensione l'utente ricorre al picker di Android; una volta selezionato l'apk desiderato, viene verificato il contenuto dell'apk, poi si avvia il caricamento. Una volta caricata l'estensione, comparirà un messaggio di caricamento avvenuto con successo, vice versa se il file non era valido comparirà il messaggio un errore con un codice di errore che ne identifica la natura. Dopo il caricamento, se effettuato con successo, si torna alla view della calcolatrice e nella barra a scomparsa comparirà d'ora in poi anche l'estensione caricata (figura 7.2b) e si potrà usufruire di tutte le sue funzioni e applicazioni, senza riavviare l'app Bridge. Quando si apre la view di una estensione nella barra dell'applicazione comprarirà anche un menù aggiuntivo.



(a) (b)

Figura 7.1: Menu per eliminare l'estensione

Tramite questo menù è possibile eliminare l'estensione.

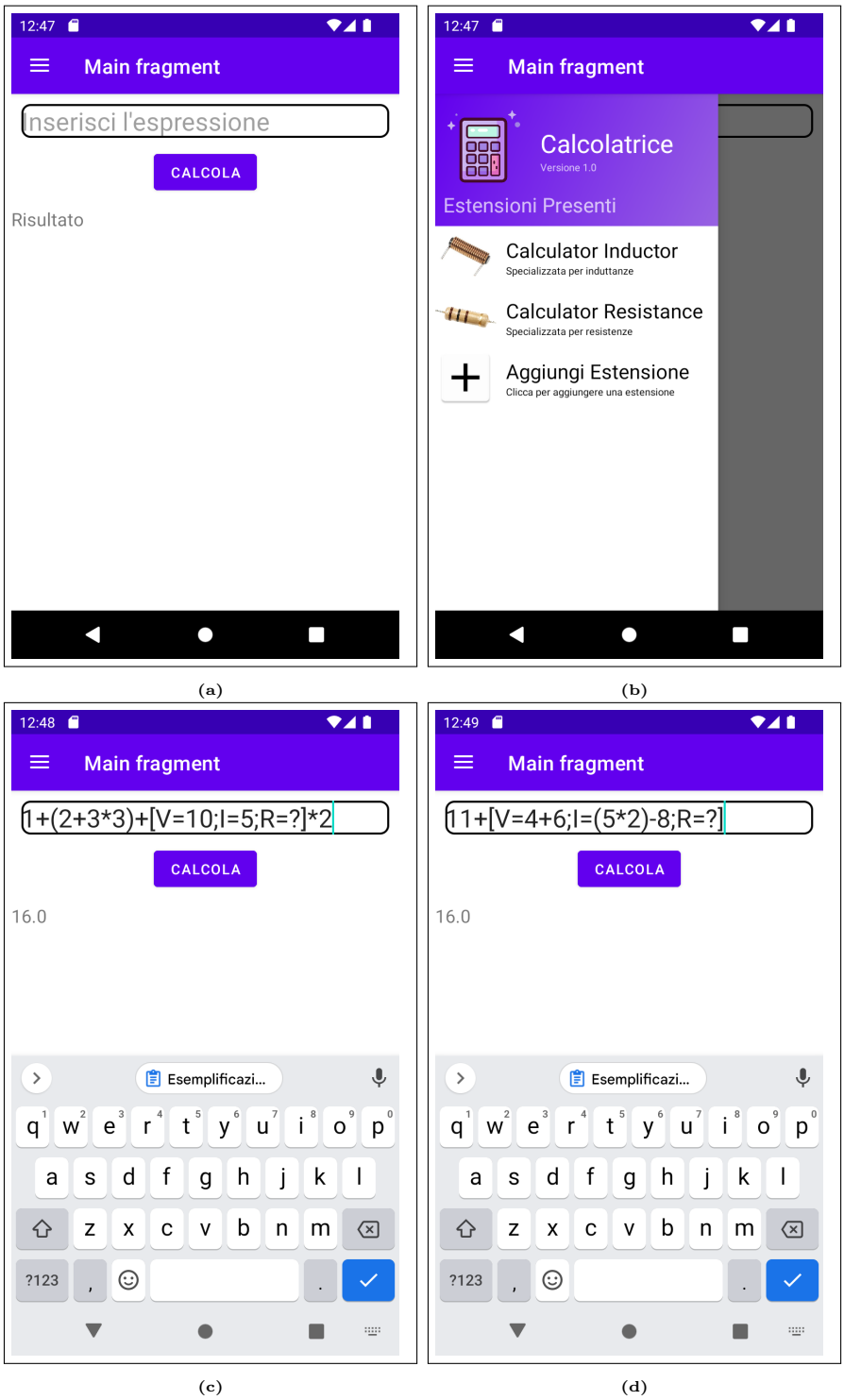


Figura 7.2: Screen calcolatrice in esecuzione

In figura 7.2c e 7.2d viene mostrata l'esecuzione di due espressioni differenti: da notare la parte di espressione singolare che recita come [V=*;I=*;R=*] con al posto dei * dei numeri, espressioni o un "?" per indicare che è la variabile di output richiesta, questa singolare espressione è implementata tramite l'estensione "Calculator Resistance" presente nella calcolatrice come mostrato in figura 7.2b

Bibliografia

Link di riferimento: https://developer.android.com/reference/java/lang/ClassLoader#:~:text=A%20class%20loader%20is%20an,a%20definition%20for%20the%20class. Link di riferimento: https://developer.android.com/reference/java/lang/ClassLoader#loadClass(java.lang.String,%20boolean)
Link di riferimento: https://developer.android.com/reference/dalvik/system/PathClassLoader