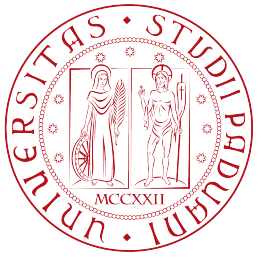


Università degli Studi di Padova

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

Abstract

Realizzare una applicazione 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 deve essere in grado di carcare e usare le classi/metodi e altri contenuti dell'estensione.

Indice

Capitolo 1

Introduzione

L'obbiettivo è realizzare una applicazione che chiameremo "Bridge" che è in grado di usare il contenuto di altri file.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", poichè 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 sconoscuto 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". Questo porta vantaggi e svantaggi che sono:

- I 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 può estenere 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 aggiungo funzionalità di esecuzione.

Capitolo 2

Caricamento di codice dinamico

Per eseguire il caricamento dinamico del codice, viene usato quello che viene definito "caricatore di classi", è un oggetto informatico resposabile del caricamento di esse, questo oggetto si chiama "ClassLoader".
Link di riferimento: `https://developer.android.com/reference/java/lang/ClassLoader#:~:text=A%20class%20loader%20is%20an,a%20definition%20for%20the%20class.`
Essa in realtà è una classe astratta ovvero definisce come deve essere fatta l'interfaccia pubblica di un "Loader" concrettizzabile, nel framework di android sono presenti vari "Loader" che estendono "ClassLoader" in modo da esserci molti modi in cui la JVM (=Java VirtualMachine) carica dinamicamente le classi. All'interno di un file ci sono più classi e risorse, quando si vuole caricare una classe bisogna spcificare quale classe fra le presenti bisogna caricaricare, quindi, specificare il percorso interno dei package e in fine 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")
```

Listing 2.1: Esempio caricamento di una classe.

Link di riferimento: `https://developer.android.com/reference/java/lang/ClassLoader#loadClass(java.lang.String,%20boolean)`
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 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, 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.
Ma, quando si carica una classe tramite il 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()
```

Listing 2.2: Esempi di instanziazione.

2.2 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.
Per creare un interfaccia grafica, Android Studio, permette di crearla comodamente in modo dichiarativo, un metodo molto comodo, però, ha un problema perchè ogni elemento grafico, e anche per altre variabile dichiarate nello stesso modo, vanno identificate 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.
Quindi, non si può creare una interfaccia grafica in modo dichiarativa in una estensione poichè, anche se leggo il file.xml in cui c'è la descrizione del layout,

quest'ultimo non potrò usarlo perchè ha ID probabilmente in conflitto con gli ID del bridge.

2.3 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?
- La natura della classe che carico qual'è?
- Dopo aver caricato tale classe come faccio ad usarla?
- Come caricare una interfaccia grafica?

Capitolo 3

PathClassLoader

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.
Link di riferimento: `https://developer.android.com/reference/dalvik/system/PathClassLoader`
Verrà utilzzato questo oggetto fornitosi dal 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)
```

Listing 3.1: Costruttori per PathClassLoader.

Come dimostrato 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)
```

Listing 3.2: Caricamento di un file.

Viene usato come "ClassLoader" la variabile "activity.classLoader", come suggerisce il nome è un oggetto di tipo "ClassLoader" questo è quello utilizzato per l'applicazione "Bridge".
Deve essere utilizzato quello dell'applicazione perchè, se si ottiene un Loader generico, ottenibile ad esempio con

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

Listing 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

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

Listing 3.4: Tentativo di caricare un file.

succede che poco dopo l'instanziazione non riesce ad eseguire il Cast, difatti sulla console, comparirà un errore bizzarro ovvero, "ClassCastException" un errore assurdo poichè sapendo che la classe da caricare estende "AbstractCalculator" quando la si instanzia, si dovrebbe poter parametrizzarla con il tipo astratto, per via del polimorfismo, invece comparirà come errore un "ClassCastException", questo perchè il loader ottenuto con "ClassLoader.getSystemClassLoader()" non è adatto per eseguire questo caricamento in particolare, bisogna usare il loader come mostrato in 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, questi file, al loro interno hanno una struttura ben definita Di fisso in un file.apk c'è

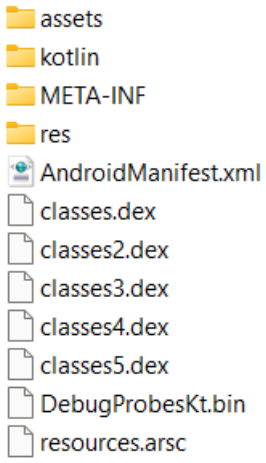


Figura 4.1: Struttura interna file.apk

- AndroidManifest.xml : File che contiene varie informazioni per il sistema android, per esempio l'icona da usare, le modalità di avvio dell'app, come subspecifica dell'avvio 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/-stringe/..., importante che le sotto cartelle hanno dei nomi ben definiti, da sottolineare che c'è né una in particolare che si chiama "drawable-v24". Le Resources possono essere raggruppate concettualemente in queste sotto cartelle:

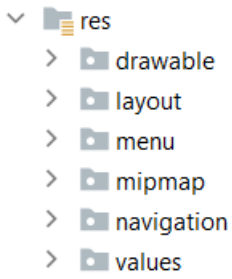


Figura 4.2: Raggruppamento Res.apk

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 libreria 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 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 all'interno del file.dex, però rimane il problema di come si chiama la classe e com'è posizionata all'interno, 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 }
```

Listing 4.1: Esempio Estensione.json.

così, in ogni caso, quando si cerca questo file, si sa esattamente dov'è, e leggerne il contenuto, al suo interno ci sarà scritto il 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, qui come per la classe, nasce il problema, risolvibile usando la struttura interna della cartella "Drawable-v24" sopracitata, 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 sà 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 ci devono essere 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 del file.dex

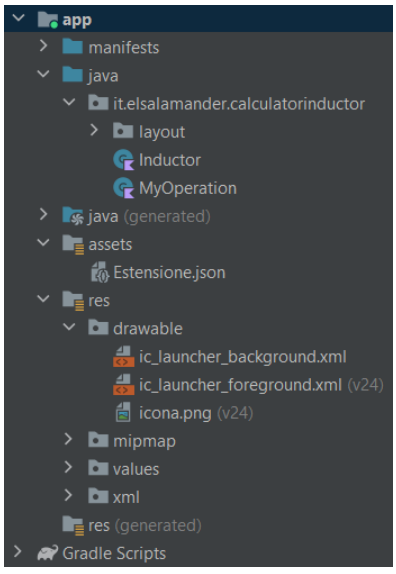


Figura 4.3: Progetto di una estensione

Questà è la struttura finale di una estensione realizzata.

Capitolo 5

Risoluzione interfaccie pubbliche

Come utilizzare la classe caricata?
Per utilizzare una classe, bisogna conoscerne l'interfaccia pubblica, altrimenti non potrò invocare le funzioni volute, per risolvere a ciò, si fa uso di una libreria che, nel caso di android, si traduce in un file.AAR, per risolvere al 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 quest'ultima. Al suo interno, c'è una classe astratta molto importate chiamata "AbstractLoadClass", questa classe impone un'interfaccia pubblica a chi l'estende, si fa estendere questa classe a quella che viene caricata nell'estensione come citato nei punti precendenti, imponendo la sua estensione, ora so l'interfaccia pubblica della classe che carico.

```
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 non trovata, ma non
   castabile")
}
```

Listing 5.1: Caricamento classe con Cast.

In questo modo ora, si può utilizzare la classe, ma solo le funzioni imposte da "AbstractLoadClass" che, in questo caso è più che sufficente.
Per risolvere al problema dell'interfaccia, si utilizza una libreria che tutti cono-

scono che definisce strutture preferibilmente astratte, in modo tale da imporre interfaccie pubbliche da una parte, e poterle così leggere dall'altra poichè so che è stata imposta.

```
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")
    }
}
```

Listing 5.2: Esempio estensione creata.

Questo meccanisco risolve ai 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 a questo problema bisogna prima sapere come è fatta quella del Bridge. Infatti, il Bridge, per quanto sembra, semplice non lo è, tralasciando la barra a scomparsa, si ha una schermata che è gestita tramite un Navigator e sarà proprio l'utilizzo di un navigator che, ci permette di cambiare interfaccia grafica, però nasce il problema. Il navigator, come per i layout grafici, ha la comodità di crearlo in modo dichiarativo però, ciò significa che per forze di cose ci sono 2 fattori impossibili da soddisfare:

- Il layout deve essere contenuto nel Bridge
- L'oggetto Fragment deve essere anchesso 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 eventuali "navigazioni" a piacimento, il problema è che questo meccanismo vuole il nome della classe del fragment, che realizza l'interfaccia grafica, il problema non è proprio dargli il nome, poichè si possono tranquillamente realizzare il fragment, nell'estensione, ma il Navigator quando vuole caricare il fragment lo cerca all'interno del Bridge..., ovvero lo cerca nel posto sbagliato perchè non si trova li, quindi modificare le navigazioni è inutile perchè, anche questo sistema, richiede 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, ovvero, quando avviene una navigazione tra due fragment posso alterare il fragment di destinazione, inserendo il fragment già allocato come si vuole, per fare ciò si è crato un fragment vuoto, e quando si vuole aprire una interfaccia grafica di una estensione, prima si fa

navigare nel fragment vuoto, si intercetta la navigazione e si cambia la destinazione con quella voluta, in questo modo verrà mostrato il fragment all'interno dell'estensione.

```
//Prendi la direzione di navigazione
val direction = NavGraphDirections.navigationMove(value)

//Naviga al fragment, in questo caso quello vuoto
navController.navigate(direction)

//ottieni il supporto per eseguire le modifiche
val fm: FragmentManager = supportFragmentManager
val fragmentTransaction : FragmentTransaction = fm.
    beginTransaction()

//prendi il fragment dell'estensione
extFrag = managerExtentions.extentions[value]!!.second.
    getFragment(this)

//aggiungilo alla destinazione
fragmentTransaction.add(R.id.nav_host_fragment,extFrag!!, null)
fragmentTransaction.addToBackStack(null)

//applica le modifiche
fragmentTransaction.commit()
```

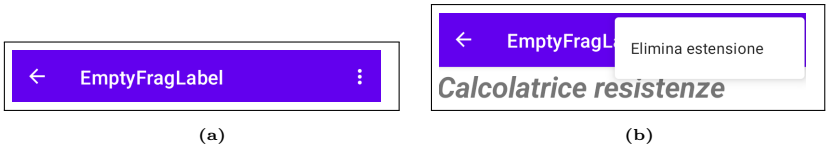
Listing 6.1: Caricamento classe con Cast.

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 al metodo programmatico, di per sè questa è la soluzione ne più ne meno, perchè si può creare una interfaccia grafica tranquillamente anche a codice, completa di eventi di interazione.

Capitolo 7

Esemplificazione

Per mostrare l'utilità del caricamente di codice dinamico è stata realizzata la seguente app:
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 come per il problema della natura e interfaccia pubblica, devono rispettare una interfaccia pubblica descritta nella libreria.
Per aggiungere una estensione si fa uso del picker di Android, una volta selezionato l'apk desiderato, inizierà la fase di verifica del contenuto dell'apk, poi si avvierà il caricamento di essa, una volta caricata, comparirà un messaggio di caricamento avvenuto con successo, vice versa se il file non era valido comparirà il messaggio di errore con un possibile codice di errore che ne identifica la natura. Dopo il caricamento, effettuato con successo, si torna alla view della calcolatrice e nella barra a scomparsa comparirà d'ora in poi anche l'estensione caricata(come mostrato nelle immagini successive), una volta caricata 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.

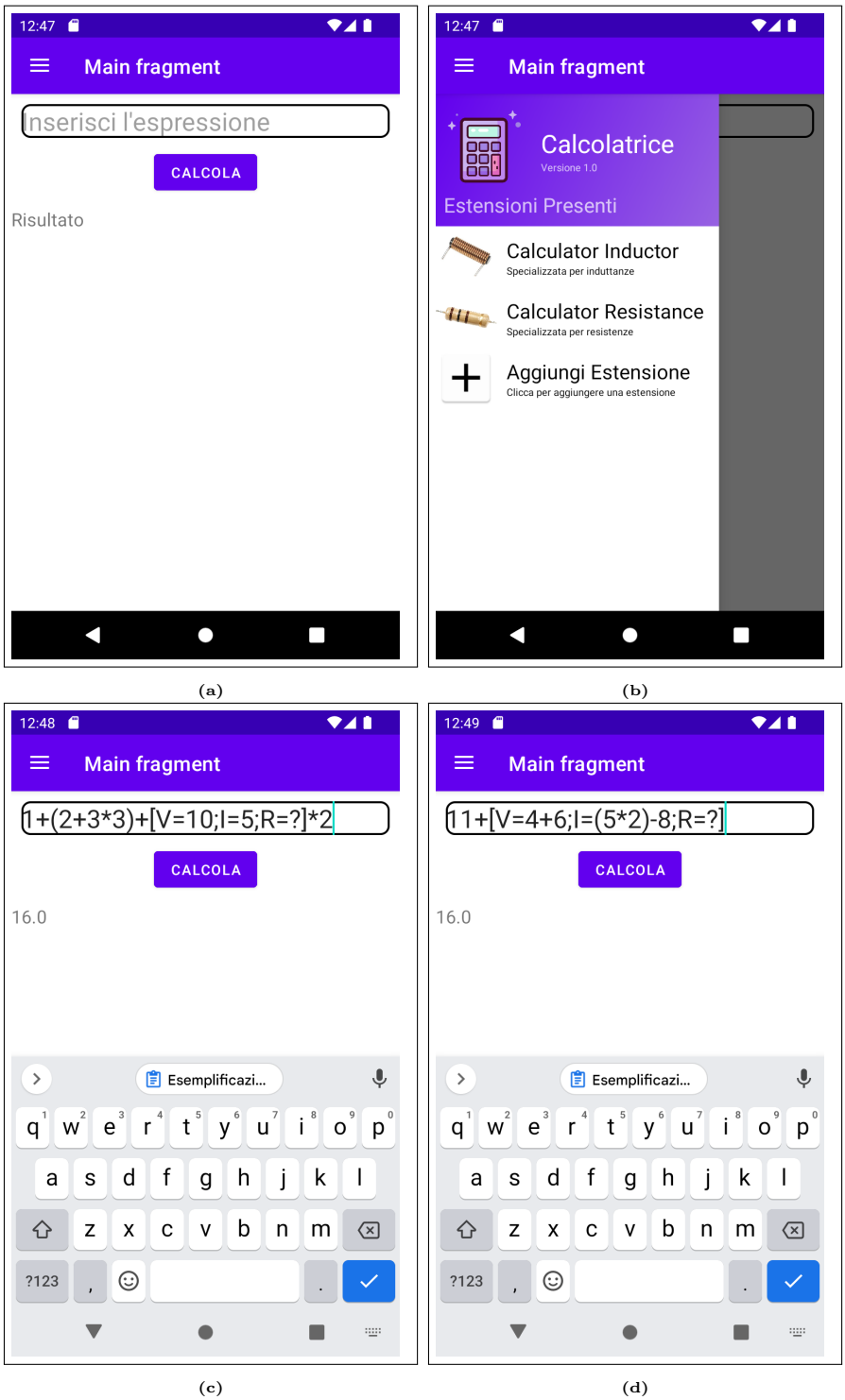


Figura 7.2: Screen calcolatrice in esecuzione

Nell'immagine (c) e (d) viene mostrata l'esecuzione di 2 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 colcolatrice come mostrato nell'immagine (b)

Bibliografia