

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in
Ingegneria Informatica

Anno Accademico 2015 – 2016

 POLITECNICO DI MILANO



Lifecycle and Event-Based Testing for Android Applications



Candidato: Simone Graziussi (836897)

Relatore: Luciano Baresi

Correlatore: Konstantin Rubinov



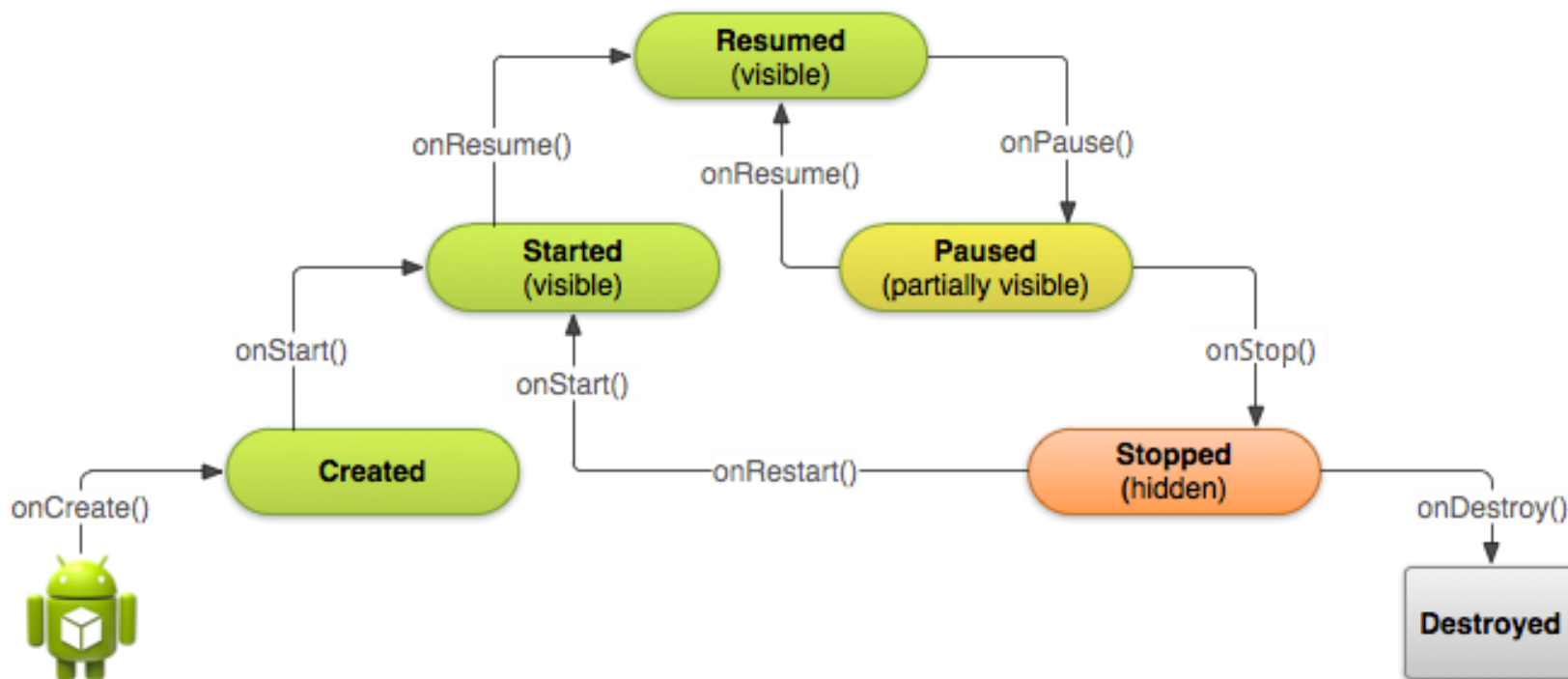
- I dispositivi mobili sono un ambiente molto dinamico
 - Continui cambi di applicazione attiva
 - Centinaia di eventi al secondo, spesso concorrenti
- Problema: i meccanismi di testing attuali non permettono una verifica completa della sequenza di eventi generati durante un'esecuzione
- Obiettivi:
 - Facilitare la gestione di uno dei più importanti gruppi di eventi, le transizioni del ciclo di vita
 - Permettere di esprimere condizioni sul flusso di eventi in modo più naturale



- Introduzione e Obiettivi del Lavoro
- **Testing per Lifecycle**
- Testing basato sugli Eventi
- Conclusioni



- Componenti dell'applicazione come Activity e Fragment
- Diversi stadi di funzionamento
- Sviluppatore definisce le azioni ad ogni transizione





- Evitare spreco di risorse
 - es. rilascio sensori quando in background
- Fermare l'esecuzione se l'utente lascia l'applicazione
 - es. gioco si ferma se arriva una chiamata
- Mantenere lo stato se l'utente lascia l'applicazione
 - es. messaggio scritto parzialmente
- Adattarsi ai cambi di configurazione
 - es. rotazione



- Analisi statica del codice per controllare la gestione di componenti in base al lifecycle
- Controllo di rilascio, best practices e doppia acquisizione
- Esempio: Broadcast Receiver
 - Rilascio: il metodo `unregisterReceiver()` è da chiamare sempre dopo `registerReceiver()`, ma non durante `onSaveInstanceState()`
 - Best Practices: durante `onStart()` e `onStop()`
 - Doppia Acquisizione: in questo caso non causa problemi, ma utile controllare il doppio rilascio
- Controlli implementati con Lint



Controlli Statici per Lifecycle - Valutazione

7

App “InTheClear”

```
53 public class SMSThread extends Thread {  
54  
55     @Override  
56     public void run() {  
57         c.registerReceiver(smsconfirm, new IntentFilter(SENT));  
58         c.registerReceiver(smsconfirm, new IntentFilter(DELIVERED));  
59     }  
60 }  
61  
62 public void exitWithResult(boolean result, int process, int status) {  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100
```

Found a BroadcastReceiver registerReceiver() but no unregisterReceiver() calls in the class [BroadcastReceiverLifecycle] [more...](#) (Ctrl+F1)

```
143     }  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200
```

../src/main/java/com/byteshaft/trackbuddy/LocationService.java:142: The best practice is to call the GoogleApiClient connect() during onStart()

```
139         .addOnConnectionFailedListener(this)  
140         .addApi(LocationServices.API)  
141         .build();  
142     mGoogleApiClient.connect();  
143 }  
144
```

App “TrackBuddy”

Priority: 5 / 10

Category: Performance

Severity: Warning

Explanation: Incorrect GoogleApiClient lifecycle handling.

You should always disconnect a GoogleApiClient when you are done with it. For activities and fragments in most cases connection is done during onStart and disconnection during onStop().

More info: <https://developers.google.com/android/reference/com/google/android/gms/common/api/GoogleApiClient#nested-class-summary>




- Analisi dinamica dell'applicazione
- Libreria che permette di controllare facilmente le transizioni del lifecycle
- Lo sviluppatore definisce solo dei callback, il resto delle transizioni del ciclo di vita è gestito dalla libreria
 - es. azioni/controlli prima di mettere in pausa, controlli durante la pausa e azioni/controlli dopo la pausa
- Disponibile per
 - Unit Testing tramite Instrumentation
 - UI Testing con Android Espresso
 - Unit Testing tramite Robolectric





- Test per WordPress definito in Espresso

```
public RotationCallback testRotation() {  
    return new RotationCallback() {  
        private String name;  
  
        @Override  
        public void beforeRotation() {  
            onView(withId(R.id.first_name_row))  
                .check(matches(isDisplayed()))  
                .perform(click());  
  
            name = "MyFirstName" + (new Random().nextInt(100));  
            onView(withId(R.id.my_profile_dialog_input))  
                .check(matches(isDisplayed()))  
                .perform(replaceText(name));  
  
            onView(withText("OK"))  
                .perform(click());  
  
            onView(withId(R.id.first_name))  
                .check(matches(allOf(isDisplayed(), withText(name))));  
        }  
  
        @Override  
        public void afterRotation() {  
            onView(withId(R.id.first_name))  
                .check(matches(allOf(isDisplayed(), withText(name))));  
        }  
    };  
}
```

Lo sviluppatore
deve solo definire
un callback



Azioni e controlli
standard di Espresso





- Introduzione e Obiettivi del Lavoro
- Testing per Lifecycle
- **Testing basato sugli Eventi**
- Conclusioni



- App possono registrare anche centinaia di eventi al secondo
 - es. sensori, richieste/risposte via internet, click sul touchscreen, lifecycle, ecc.
- Spesso gestiti da diversi thread, e quindi concorrenti
- Se registrati in ordine o quantità inaspettati dal programmatore, possono causare problemi
- Esempio: Race Condition
 - Lo sviluppatore assume la causalità $E1 \rightarrow E2$ tra due eventi, ma il sistema genera E2 prima di E1
 - Possibili crash o comportamenti inaspettati

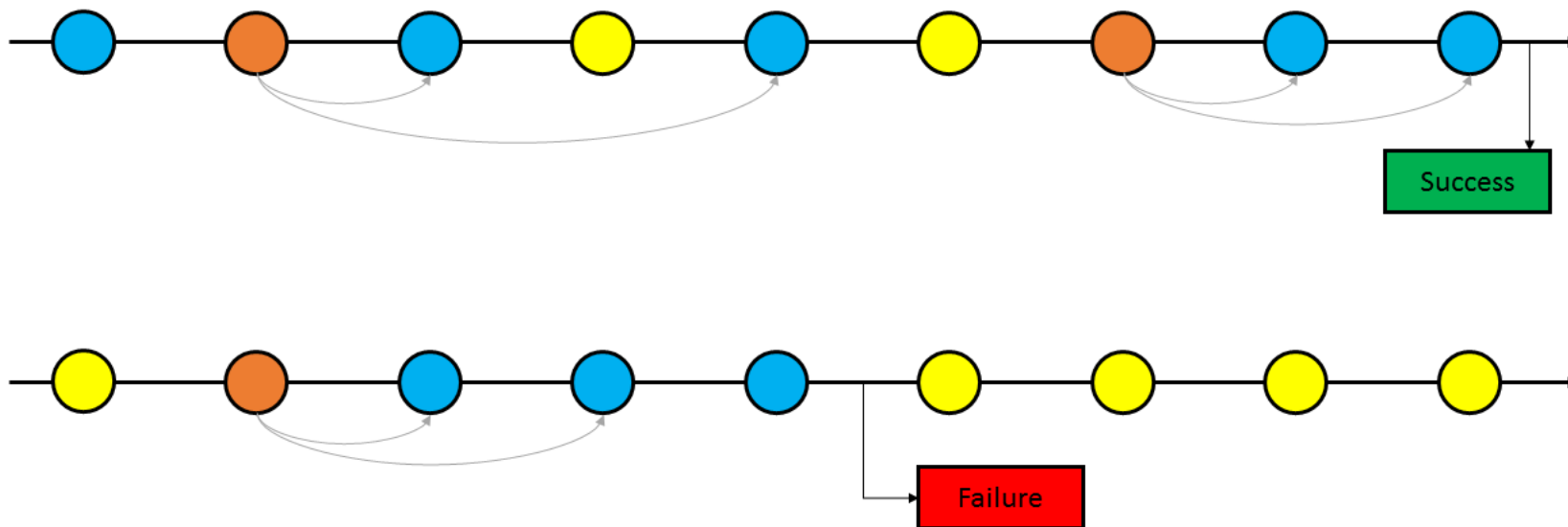


- In Android, gli eventi sono complessi da testare con le tecnologie disponibili al momento
- Soluzione: specificare delle asserzioni temporali per verificare le relazioni tra due o più eventi
- Esprimere, sul flusso di eventi generati da un'esecuzione, condizioni di
 - Esistenza
 - Ordinamento
 - Causalità
 - Quantificazione
- Possibilità di correlare più condizioni tramite connettivi logici



- Esempio: causalità tra eventi

`exactly(2).eventsWhereEach(●).mustHappenAfter(anEventThat(●))`

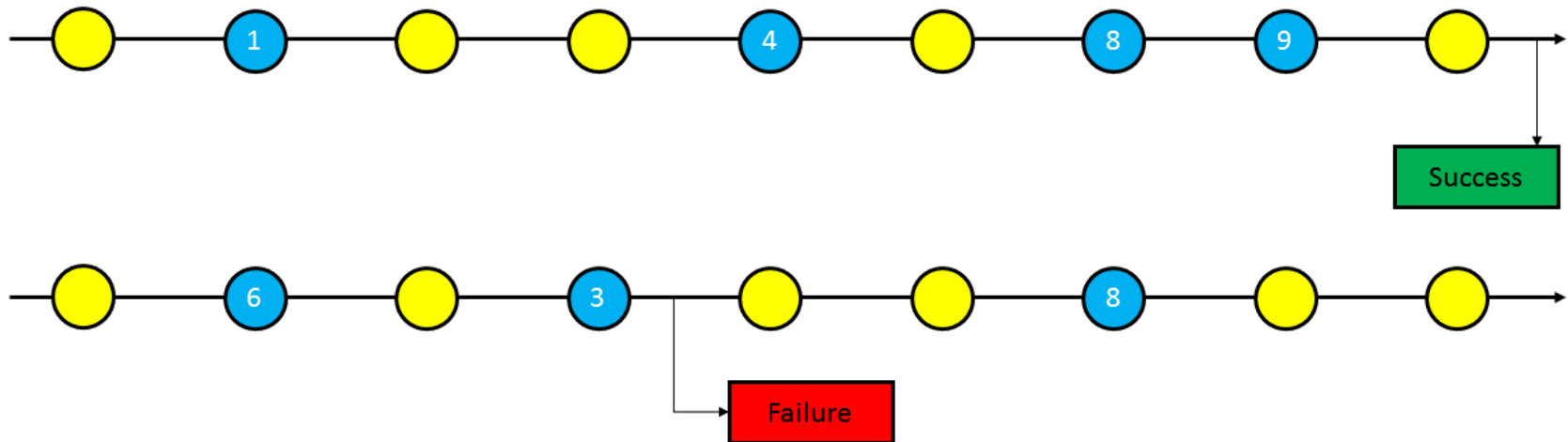




Asserzioni Temporalì - Esempi

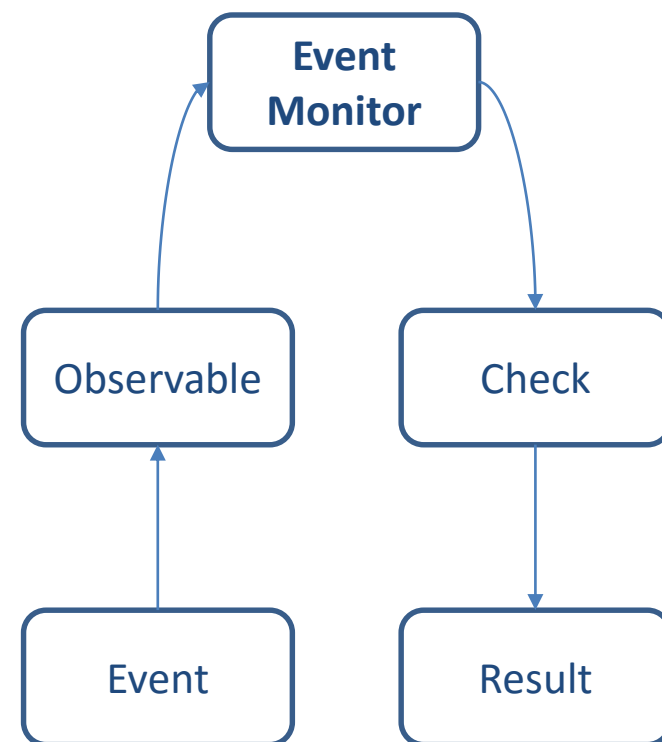
- Esempio: ordinamento di eventi di un determinato tipo

`allEventsWhereEach(●).areOrdered(\leq)`





- Interfaccia principale è l'Event Monitor:
 - lo sviluppatore definisce
 - Eventi da osservare
 - Asserzioni temporali da verificare
 - Come reagire ai risultati delle asserzioni
- Tool implementato con la libreria ReactiveX: RxJava e RxAndroid
- Utilizzabile in ogni framework di test





- Utilizzo della libreria nell'applicazione WordPress, per mostrarne il funzionamento in un contesto reale
- Sezione dell'app che permette di scrivere un post all'interno del blog e pubblicarlo
- Esempi di asserzioni temporali:
 - Contenuto del post non può cambiare dopo l'inizio della procedura di pubblicazione
 - Click su "Pubblica" genera sempre o un messaggio di errore o l'inizio della procedura di pubblicazione
 - Gli aggiornamenti sul progresso dell'upload di immagini devono essere inviati in ordine crescente



- Esempio di eventi osservati durante il test

```
EventMonitor.getInstance().observe(  
    EventUtils.postChanges(editorTitleView,  
                           editorContentView));
```

- Esempio di asserzione temporale

```
EventMonitor.getInstance().checkThat(  
    "Post content changed after the upload started!",  
    providedThat(  
        existsAnEventThat(isPostUploadStart()))  
    .then(  
        anEventThat(isPostChange())  
        .canHappenOnlyBefore(  
            anEventThat(isPostUploadStart()))));
```



- Risultato asserzione temporale

[SUCCESS] IF (Exists an event that is post upload start)
THEN (Every event that is post change happens before an
event that is post upload start)

REPORT: Every event that is post change was found before
{Post upload start}

- Risultato della stessa asserzione con fault seeding

[FAILURE] IF (Exists an event that is post upload start)
THEN (Every event that is post change happens before an
event that is post upload start)

ERROR: Post content changed after the upload started!

REPORT: Event {Post change on view 2131820907} was found
after every event that is post upload start



- Introduzione e Obiettivi del Lavoro
- Testing per Lifecycle
- Testing basato sugli Eventi
- **Conclusioni**



- Contributi
 - Controlli statici per il lifecycle, integrati nell'IDE di sviluppo
 - Libreria per controllare le transizioni del lifecycle, per permettere un testing più approfondito delle applicazioni
 - Libreria che permette di esprimere asserzioni temporali sul flusso di eventi generato durante un test
- Possibilità di usare i tre tool in contemporanea, per una verifica ancora più approfondita



Fine

21

