Testing

Michele Nardini

Michele.nardini2@studio.unibo.it

Introduzione

Il test è parte integrante del processo di sviluppo dell'app. Eseguendo test coerenti con la tua app, puoi verificare la correttezza, il comportamento funzionale e l'usabilità della tua app prima di rilasciarla pubblicamente.

Puoi testare manualmente la tua app navigando al suo interno. È possibile utilizzare dispositivi ed emulatori diversi, modificare la lingua del sistema e provare a generare ogni errore utente o attraversare ogni flusso utente.

Tuttavia, i test manuali si adattano male e può essere facile trascurare le regressioni nel comportamento dell'app. Il test automatizzato implica l'utilizzo di strumenti che eseguono test per te, il che è più veloce, più ripetibile e in genere ti fornisce un feedback più fruibile sulla tua app nelle prime fasi del processo di sviluppo.

L'automazione dei test è l'approccio migliore per rilevare i bug in anticipo. Il test automatizzato (d'ora in poi test) è un dominio ampio e Android offre molti strumenti e librerie che possono sovrapporsi. Per questo motivo, i principianti trovano spesso i test impegnativi.

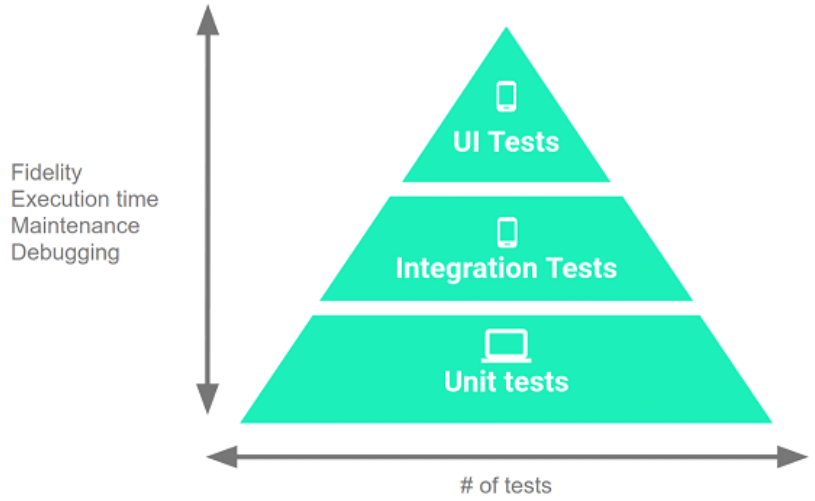
Tipi di test

- Test funzionali : la mia app fa quello che dovrebbe?
- Test delle prestazioni : lo fa in modo rapido ed efficiente?
- Test di accessibilità : funziona bene con i servizi di accessibilità?
- Test di compatibilità : funziona bene su ogni dispositivo e livello API?

Un'altra suddivisione

I test si suddividono anche a seconda delle dimensioni del test da effettuare:

- I test unitari o i test di piccole dimensioni verificano solo una parte molto piccola dell'app, ad esempio un metodo o una classe
- I test end-to-end o i test di grandi dimensioni verificano contemporaneamente parti più grandi dell'app, come uno schermo intero o un flusso utente.
- I test medi sono intermedi e controllano l' integrazione tra due o più unità.



Fattibilità dei testing

Una buona strategia di test trova un equilibrio appropriato tra la fedeltà di un test, la sua velocità e la sua affidabilità.

La somiglianza dell'ambiente di test con un dispositivo reale determina la fedeltà del test. I test di fedeltà più elevati vengono eseguiti sui dispositivi emulati o sul dispositivo fisico stesso. I test di fedeltà inferiore potrebbero essere eseguiti sulla JVM della workstation locale.

Con un'architettura di app testabile, il codice segue una struttura che consente di testare facilmente diverse parti di esso in isolamento.

Un'architettura non testabile produce quanto segue:

- Test più grandi, più lenti, più traballanti.
- Meno opportunità per testare scenari diversi. I test più grandi sono più lenti, quindi testare tutti i possibili stati di un'app potrebbe non essere realistico.

Testing in Android Studio

Quando si crea un nuovo progetto troviamo tre package all'interno della cartella java del modulo app:

- main: il codice della nostra applicazione.
- test: sono test che vengono eseguiti direttamente sulla Java Virtual Machine (JVM) e sono il modo più pratico per poter predisporre uno unit test con JUnit.
- androidTest: contiene test instrumented, eseguiti su dispositivo e attivati mediante le Instrumentation API.

Best practice

- Unit test per ViewModel.
- Unit test per il livello dati , in particolare i repository.
- Unit test per altri livelli indipendenti dalla piattaforma come il livello Dominio.
- Unit test per classi di utilità come la manipolazione di stringhe e la matematica.

I test unitari dovrebbero concentrarsi sia sui casi normali che su quelli marginali.

I casi limite sono scenari non comuni che è improbabile che i tester umani e i test più ampi riescano a catturare

Test che invece dovrebbero essere evitati sono quei test che verificano il corretto funzionamento del framework o di una libreria, non del tuo codice.

Testing con JUnit

Esempio:

```
public class ExampleUnitTest {
@Test
  public void addition_test() throws Exception{
      assertEquals(4, 2+2);
  }
}
```

@Test indica che quel metodo rappresenta un test.

AssertEquals in questo esempio consiste nel confrontare due valori, e il test si considera passato se tali valori coincidono

Colori associati ai singoli test:

- Verde: il test ha avuto successo;
- Giallo: l'esecuzione non ha riscontrato problemi ma il test non ha avuto successo;
- Rosso: durante il test è fallita l'esecuzione del codice per via di una eccezione non gestita.

Test delle interfacce utente

I test prodotti con Espresso risultano perfettamenti fluidi per questo tipo di test.

Ogni operazione sarà distribuita in tre fasi:

- Invocazione del componente visuale sul quale simulare l'interazione. Faremo ciò con il metodo onView(), mentre useremo il metodo onData() per gli elementi di un Adapter.
- Effetueremo l'azione tramite il metodo perform():
- Effettueremo controlli tramite ViewAssertions per verificare gli effetti dell'operazione eseguita.

```
@Test
public void deleteListviewItem() {
// 1. click sul pulsante
onData(anything()).inAdapterView(withId(R.id.listView))
.atPosition(0)
.onChildView(withId(R.id.btn_delete))
.perform(click());
// 2. conteggio iniziale del numero di righe della ListView
int initialCount=((ListView)mActivityRule.getActivity().findViewById(R.id.listVie
// 3. click sul pulsante "Sì" della finestra di dialogo
onView(withId(android.R.id.button2)).perform(click());
// 4. conteggio finale del numero di righe della ListView
int finalCount=((ListView)mActivityRule.getActivity().findViewById(R.id.listView)
// 5. Il numero di righe è diminuito di uno?
assertThat(finalCount, is(equalTo(initialCount-1)));
}
```

Test double in Android

Quando si testa un elemento o un sistema di elementi lo si fa in isolamento . Ad esempio, per testare un ViewModel non è necessario avviare un emulatore e avviare un'interfaccia utente perché non dipende (o non dovrebbe) dal framework Android.

Tuttavia, il soggetto in esame potrebbe dipendere da altri per il suo funzionamento. Ad esempio, un ViewModel potrebbe dipendere da un repository di dati per funzionare.

Quando è necessario fornire una dipendenza a un soggetto sottoposto a test, una pratica comune consiste nel creare un test double.

In seguito descriverò 3 varianti dei test double.

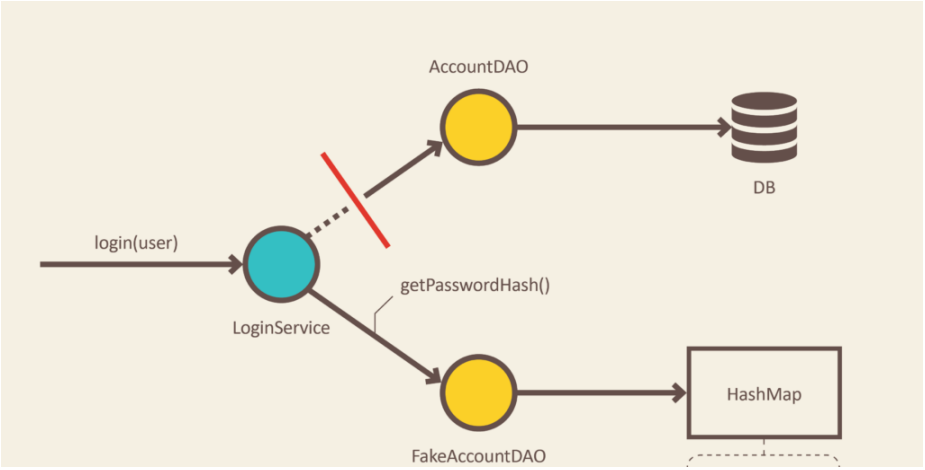
Fake

I fake sono oggetti che hanno implementazioni funzionanti, ma non uguali a quelle di produzione. Di solito prendono qualche scorciatoia e hanno una versione semplificata del codice di produzione.

```
Map<User, Account> accounts = new HashMap<>();

public FakeAccountRepository() {
        this.accounts.put(new User("john@bmail.com"), new UserAccount());
        this.accounts.put(new User("boby@bmail.com"), new AdminAccount());
}

String getPasswordHash(User user) {
        return accounts.get(user).getPasswordHash();
}
```



Stub

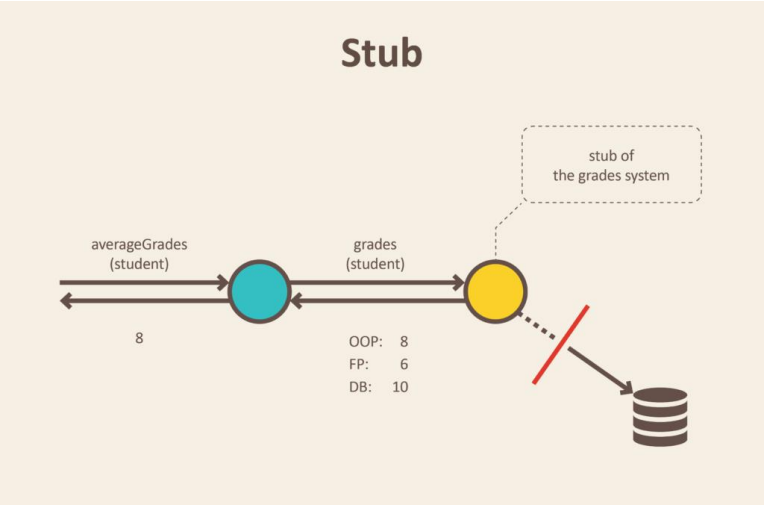
Stub è un oggetto che contiene dati predefiniti e li utilizza per rispondere alle chiamate durante i test.

Viene utilizzato quando non possiamo o non vogliamo coinvolgere oggetti che risponderebbero con dati reali o che avrebbero effetti collaterali indesiderati.

```java
public class GradesServiceTest {
    private Student student;
    private Gradebook gradebook;

    @Before
    public void setUp() throws Exception {
        gradebook = mock(Gradebook.class);
        student = new Student();
    }

    @Test
    public void calculates_grades_average_for_student() {
        when(gradebook.gradesFor(student)).thenReturn(grades(8, 6, 10)); //stubbing g
        double averageGrades = new GradesService(gradebook).averageGrades(student);
        assertThat(averageGrades).isEqualTo(8.0);
    }
}
```




```java
public class GradesService {
    private final Gradebook gradebook;

    public GradesService(Gradebook gradebook) {
        this.gradebook = gradebook;
    }

    Double averageGrades(Student student) {
        return average(gradebook.gradesFor(student));
    }
}
```

Mock

I mock sono oggetti che registrano le chiamate che ricevono.

Usiamo mock quando non vogliamo invocare il codice di produzione o quando non c'è un modo semplice per verificare che il codice previsto sia stato eseguito.

Un esempio può essere una funzionalità che chiama il servizio di invio di posta elettronica.

Non vogliamo inviare e-mail ogni volta che eseguiamo un test. Inoltre, non è facile verificare nei test che sia stata inviata un'e-mail corretta.

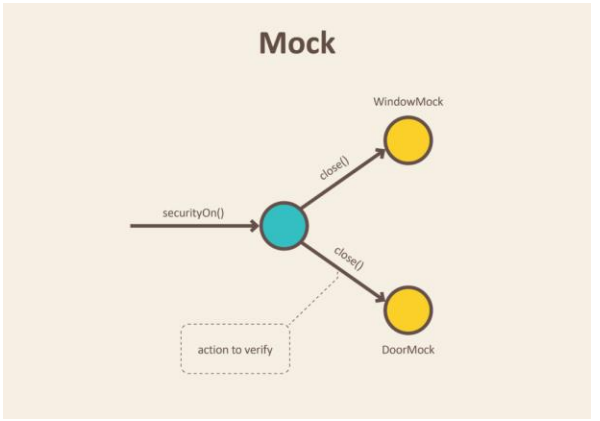
L'unica cosa che possiamo fare è verificare gli output della funzionalità che viene esercitata nel nostro test.

In altri mondi, verifica che sia stato chiamato il servizio di invio e-mail.

```java
public class SecurityCentral {
    private final Window window;
    private final Door door;

    public SecurityCentral(Window window, Door door) {
        this.window = window;
        this.door = door;
    }

    void securityOn() {
        window.close();
        door.close();
    }
}
```




```java
public class SecurityCentralTest {
    Window windowMock = mock(Window.class);
    Door doorMock = mock(Door.class);

    @Test
    public void enabling_security_locks_windows_and_doors() {
        SecurityCentral securityCentral = new SecurityCentral(windowMock, doorMock);
        securityCentral.securityOn();
        verify(doorMock).close();
        verify(windowMock).close();
    }
}
```

Reference


https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da
https://developer.android.com/
https://html.com
https://stackoverflow.com/