DAA - Développement d'applications Android

Tests automatisés

04 January 2026

Table des matières

1 Tests automatisés

Les tests automatisés permettent de vérifier le bon fonctionnement d'une application sans intervention humaine. Ils sont essentiels pour garantir la qualité du code, détecter les bugs et assurer la stabilité des fonctionnalités au fil des évolutions du projet.

On catégorise généralement les tests automatisés en 3 types principaux :

- **Tests unitaires**: Validation des classes et des méthodes individuelles.
- **Tests d'intégration**: Vérification des interactions entre plusieurs composants.
- **Tests de bout en bout** ou **tests système**: Simulation du comportement de l'utilisateur pour tester l'application dans son ensemble.

1.1 JUnit

JUnit est un framework de test unitaire pour le langage Java. Il permet d'écrire et d'exécuter des tests unitaires de manière simple et efficace.

Il est possible d'utiliser JUnit dans les projets Android via Android Studio. Les tests JUnit sont généralement placés dans le répertoire `src/test/java` de l'application.

1.2 Tests unitaires

Dans le cas ou nous souhaitons tester une classe Java, nous allons créer une classe de test dédiée. Par exemple, pour tester une classe `Calculator`, nous allons créer une classe `CalculatorTest`.

Imaginons que nous avons développé une méthode `pow` et une méthode `factorial` dans notre classe `Calculator`. Nous allons écrire des tests unitaires pour vérifier que ces méthodes fonctionnent correctement.

```
import org.junit.Test
import org.junit.Assert.*

class CalculatorTest {

    private val calculator = Calculator()

    @Test
    fun testPow() {
        assertEquals(8.0, calculator.pow(2.0, 3.0), 0.001)
        assertEquals(1.0, calculator.pow(5.0, 0.0), 0.001)
        assertEquals(0.25, calculator.pow(2.0, -2.0), 0.001)
    }

    @Test
    fun testFactorial() {
        assertEquals(120, calculator.factorial(5))
        assertEquals(1, calculator.factorial(0))
        assertEquals(1, calculator.factorial(1))
    }
}
```

1.3 Tests instrumentalisés

Les tests instrumentalisés sont des tests qui s'exécutent sur un appareil Android ou un émulateur. Ils permettent de tester les interactions avec l'interface utilisateur et les composants spécifiques à Android.

Ces tests nous permettent de tester des scénarios complets avec l'intéraction à la base de données. Pour ce faire, plusieurs éléments clés sont nécessaires :

- `@RunWith(AndroidJUnit4::class)`: Indique que le test doit être exécuté avec le runner AndroidJUnit4.
- Héritage de `TestCase` premet de fournir la structure pour créer un « banc d'essai »
- `@setUp`: Méthode exécutée avant chaque test pour initialiser les ressources nécessaires.
- `@tearDown`: Méthode exécutée après chaque test pour libérer les ressources utilisées.

1.3.1 Tests sur une base de données Room

Définition de la base de données Android Room

```
@Entity(tableName = "history",
        indices = [Index("date")])
data class History(
    @PrimaryKey(autoGenerate = true) var id:
Long? = null,
    var expression: String,
    var result: Long? = null,
    var date: Date? = Date()
)
```

DAO associé à notre base de données

```
@Dao
interface HistoryDAO {
    @Transaction
    fun insert(entry: History) {
        entry.id = _insert(entry)
        deleteOldests() // remove
supernumerary entries
    }

    @Insert
    fun _insert(entry: History) : Long

    @Query("SELECT * from history ORDER BY
date DESC")
    fun fullHistory() :
LiveData<List<History>>

    @Query("DELETE FROM history")
    fun deleteAll()

    @Query("DELETE FROM history WHERE id NOT
IN (SELECT id FROM history ORDER BY date
DESC LIMIT :itemToKeep)")
    fun deleteOldests(itemToKeep: Int =
NBR_ITEM_TO_KEEP)
}
```

Puis nous pouvons définir notre classe de tests destinée à la DB Room :

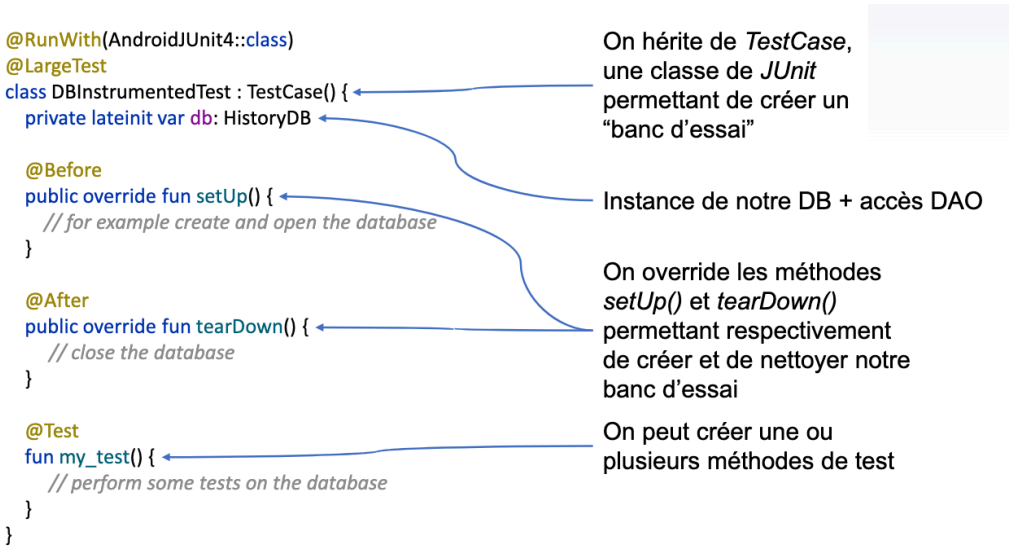

Fig. 1. – Capture des slides du cours – Classe de test instrumentalisé pour la DB Room

```
@Before
public override fun setUp() {
    val context = ApplicationProvider.getApplicationContext<Context>()
    db = Room.inMemoryDatabaseBuilder(context, HistoryDB::class.java).build()
    dao = db.historyDao()
}

@After
public override fun tearDown() {
    db.close()
}
```

Android Room prpose une API spécifique pour les tests, notamment la classe `Room.inMemoryDatabaseBuilder` qui permet de créer une base de données en mémoire pour les tests.

1.4 Tests sur l'interface utilisateur avec Jetpack Compose

Jetpack Compose propose des outils spécifiques pour tester les composants d'interface. La configuration de base nécessite l'utilisation d'une règle de test dédiée :

```
class MyComposeTests {
    @get:Rule
    val composeTestRule = createComposeRule()

    @Test
    fun editorScreenTest() {
        composeTestRule.setContent {
            val emptyTestPerson = Person("", "")
            EditorScreen(emptyTestPerson)
        }

        val fname = "Toto"
        val name = "Tata"

        composeTestRule.onNodeWithText("Bienvenue", substring = true)
            .assertTextEquals("Bienvenue !")
        composeTestRule.onNodeWithText("Name").performTextInput(name)
        composeTestRule.onNodeWithText("Firstname").performTextInput(fname)
        composeTestRule.onNodeWithText("Bienvenue", substring = true)
            .assertTextEquals("Bienvenue $fname $name !")
    }
}
```

Limitation importante : Les tests basés sur le texte posent plusieurs problèmes :

- Sensibilité au refactoring
- Problèmes de traduction/internationalisation
- Ambiguïté lorsque plusieurs éléments contiennent le même texte

Solution : Utiliser des identifiants de test

Il est préférable d'ajouter des `testTag` aux composables :

```
Text(
    modifier = Modifier.testTag("welcome-msg"),
    text = "Bienvenue ${person.firstname} ${person.name} !"
)
```

Puis de les référencer dans les tests :

```
composeTestRule
    .onNodeWithTag("welcome-msg")
    .assertTextEquals("Bienvenue Jean Neige !")
```

Une cheat-sheet complète est disponible sur la documentation officielle Android pour découvrir toutes les assertions et actions disponibles : https://developer.android.com/jetpack/compose/testing-cheatsheet

1.5 Intégration Continue et Déploiement (CI/CD)

Les tests unitaires et instrumentalisés peuvent être intégrés dans un pipeline CI/CD. Cependant, plusieurs prérequis sont nécessaires :

- Le SDK Android pour compiler le projet
- Un émulateur pour exécuter les tests instrumentalisés

1.5.1 Mise en place avec Docker

Bien qu'il n'existe pas d'image Docker officielle maintenue à jour, il est possible de créer sa propre image en suivant ces étapes :

1. Partir d'une image `openjdk:11-jdk`
2. Télécharger et dézipper le SDK Android
3. Ajouter les exécutables au PATH

4. Accepter les licences via la ligne de commande
5. Télécharger les versions nécessaires du SDK, platform-tools et build-tools
6. Utiliser Gradle pour exécuter les tâches de build et test :
   - `app:lintDebug`
   - `app:assembleDebug`
   - `app:testDebug`

Un tutoriel détaillé est disponible : https://howtodoandroid.com/setup-gitlab-ci-android/

1.5.2 Configuration de l'émulateur

Pour les tests instrumentalisés, l'émulateur nécessite une configuration spécifique :

- **Installation via ligne de commande** : entièrement automatisable
- **Accélération matérielle** : requise pour les versions récentes, nécessite une configuration au niveau de l'hôte
- **Options de démarrage** : `--no-audio` et `--no-windows` pour l'exécution en CI
- **Temps de démarrage** : plusieurs minutes, nécessite d'utiliser `adb` pour monitorer l'avancement
- **Désactivation des animations** : requis avant l'exécution des tests

Commande pour lancer les tests sur l'émulateur :

```
gradlew app:connectedAndroidTest
```

N'oubliez pas de stopper l'émulateur à la fin des tests pour libérer les ressources.

1.6 Tests automatisés supplémentaires

1.6.1 Monkey Testing sur le Play Store

Lors de la publication d'une nouvelle version sur le Play Store, Google effectue automatiquement des tests :

- Installation et interactions aléatoires avec l'application
- Possibilité de fournir des identifiants pour passer les écrans de connexion
- Tests sur plusieurs appareils différents
- Vérification de l'accessibilité (taille et contraste des textes)
- Détection de failles de sécurité connues

1.6.2 Firebase Robo Tests

Les tests du Play Store sont basés sur les **Robo Tests** du Test Lab de Firebase. Ces tests offrent des fonctionnalités avancées :

Fonctionnalités :

- Cartographie automatique de l'application
- Captures d'écran et vidéos des exécutions
- Logs détaillés
- Profilage de l'application
- Vérification de l'utilisation des niveaux d'API

Tarification :

- Version gratuite avec limitations (nombre de tests quotidiens et d'appareils)
- Version payante pour des tests à plus large échelle
- Possibilité de scripter les tests via API pour intégration CI/CD

Ces outils permettent d'assurer une couverture de test complète et automatisée, de l'unité de code jusqu'au comportement global de l'application sur différents appareils.