L'objectif du TP est d'apprendre à programmer des tests unitaires, d'intégration et fonctionnels sur Android.

1. Application jouet

Vous allez travailler sur une application très simple, un convertisseur d'unités :

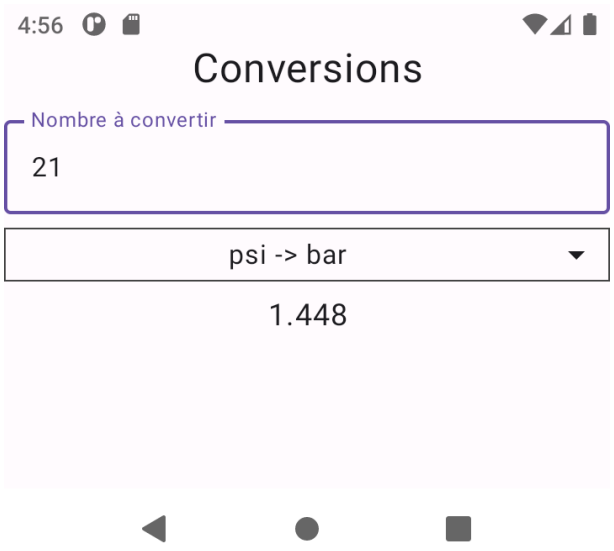

Figure 1: Application de conversion d'unités

1.1. Cahier des charges

Voici les grandes lignes du cahier des charges.

> L'interface est composée d'une zone de saisie pour le nombre à convertir, d'un sélecteur déroulant de conversion, d'un bouton pour lancer la conversion et d'une zone d'affichage pour le résultat ou un message d'erreur. L'utilisateur doit pouvoir saisir une valeur numérique réelle, en notation décimale (nnnn.nn) sans limite de chiffres. Si l'utilisateur saisit autre chose qu'un nombre correct, un message d'erreur sera affiché à la place du résultat. Le sélecteur de conversion permet de choisir parmi 4 conversions possibles (pouce->cm, psi->bar, mph->km/h, once->gramme). Le bouton de conversion lance le calcul du résultat. Le résultat affiché est la valeur convertie, avec 3 décimales.

NB: la régionalisation (« locale ») des nombres a été oubliée dans le cahier des charges, alors la saisie et l'affichage des nombres se fait avec un point.

1.2. Mise en place du projet

1. Relisez le début du TP4, §1.1 pour savoir comment on crée un projet Jetpack Compose.
2. Créez le projet TP7 exactement comme le TP4, dépendance incluse. ATTENTION au package: `fr.iutlan.tp7` impérativement, sinon vous ne pourrez pas faire la suite.

3. Vérifiez que la compilation se passe bien et corrigez les versions dans `libs.versions.toml` s'il y a un problème.
4. Ouvrez le dossier du projet, `~/AndroidStudioProjects/TP7`.
5. Descendez dans `app/src`. Vous devez voir trois dossiers : `main`, `test` et `androidTest`. Supprimez totalement ces trois dossiers et restez à ce niveau.
6. Téléchargez tp7.zip, ouvrez-le, vous voyez trois dossiers : `main`, `test` et `androidTest` qu'il faut faire glisser dans `app/src`. Ainsi ça les reconstruit dans `app/src` du projet.
7. Dépliez la catégorie `Gradle scripts` du projet et ouvrez le second fichier `build.gradle.kts`, celui du dossier `app`. Il y a deux sections à compléter, `plugins` au début et `dependencies` à la fin. Voici les modifications à faire (vérifiez ce que vous faites) :

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.jetbrains.kotlin.android)
    id("de.mannodermaus.android-junit5") version "1.12.0.0" // <--- AJOUTER
}

... ne rien modifier ...

dependencies {

    // tests unitaires JUnit5
    testImplementation("org.junit.jupiter:junit-jupiter:5.12.0")
    testImplementation("org.junit.jupiter:junit-jupiter-engine:5.12.0")
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.12.0")
    testImplementation("org.junit.jupiter:junit-jupiter-params:5.12.0")

    // assertions AssertJ
    testImplementation("org.assertj:assertj-core:3.21.0")
    androidTestImplementation("org.assertj:assertj-core:3.21.0")

    ... laisser les autres dépendances comme elles sont ...
}
```

8. Acceptez si Android Studio vous propose de passer à des références de catalogue.
9. Dans tous les cas, cliquez sur le lien `Sync Now` apparu en haut.

IMPORTANT Les tests fonctionnels ne peuvent être lancés que sur un AVD dont l'API est supérieure ou égale à 26. Donc :

- sur les postes fixes de l'IUT, supprimez tous les AVD et recréez-en un en API 26 x86_64 sans Google API. Ça sera le seul AVD et vous aurez moins de soucis de mémoire.
- sur un PC personnel, veillez à avoir un AVD en API ≥ 26.

Autant le signaler tout de suite, il y a quelques bugs dans cette application, et votre travail consiste à les trouver uniquement à l'aide de tests. C'est à dire que vous pourriez les trouver en analysant le code source, mais la démarche proposée, c'est de ne corriger quelque chose que si on a un test qui échoue.

Par exemple, si vous lancez l'exécution, que vous demandez la conversion « mph->km/h » et

mettez 10 en entrée, il vous affichera 0.689 au lieu de 16.093. Les autres conversions ne marchent pas mieux.

1.3. Structure de l'application

Cette application Jetpack Compose est construite autour des classes et fonctions suivantes :

- `State` représente la totalité des informations à un instant donné : nombre saisi, code de la conversion et résultat. Le code de la conversion indique quel coefficient il faut appliquer pour convertir d'une unité à l'autre. Il y a un objet compagnon qui définit deux tableaux, celui des coefficients et celui des noms des conversions, ainsi qu'une fonction de calcul. Enfin, il y a deux méthodes pour copier un état en changeant soit le nombre, soit le code de conversion. Le résultat est automatiquement calculé.
- `ConversionViewModel` gère le singleton `State` affiché dans la vue. Il y a des méthodes appelées par la vue quand l'utilisateur tape un nombre ou choisit une conversion.
- `MainActivity` crée une instance de `ConversionViewModel` et la fournit à la fonction `MainActivityView` qui affiche la vue principale.
- `MainActivityView` affiche une colonne contenant :
  - un titre.
  - une zone de saisie pour un nombre décimal. Cette zone de saisie est relativement complexe en interne. Il y a un label et la zone de saisie proprement dite avec un conseil lorsque le texte saisi est vide. Ces éléments supplémentaires poseront des problèmes lors des tests fonctionnels.
  - un `DropDownChoice` pour choisir la conversion parmi celles possibles, voir plus loin.
  - une zone de texte pour afficher le résultat.
- `DropDownChoice` affiche un texte, le choix courant, pouvant être cliqué pour faire apparaître une liste de choix. Cliquer sur l'un d'eux ferme la liste et déclenche une lambda dans la fonction appelante, en général pour valider le choix effectué. Le choix courant est fourni en paramètre et provient du *view model*.

👉 Cherchez les appels de ces fonctions composables pour comprendre leurs relations. Par exemple, `DropDownChoice` reçoit une lambda à un paramètre (qui s'appelle `it`), de la part de `MainActivityView` pour sélectionner la conversion. Ça la change dans le *view model*, ce qui la change aussi dans l'état, et en retour, il y a une recomposition qui fait réafficher le nouveau choix courant dans le `DropDownChoice`. Essyaez de bien comprendre ça.

Le source `AvisNombre.kt` est un tutoriel pour les tests fonctionnels. Ne vous en occupez pas maintenant.

1.4. Programmation et lancement des tests

Les tests sont programmés dans les dossiers `test` et `androidTest` qui sont dans `app/src` de votre projet. Dans `test`, on met les tests unitaires purs, de type JUnit, et dans `androidTest` on met les tests d'intégration et fonctionnels qui ont besoin d'être lancés sur un AVD.

Pour lancer un test, il suffit de déplier le package marqué `test` ou `androidTest` dans Android Studio, puis cliquer droit sur la classe de test et choisir `Run`. Voir la figure 2.

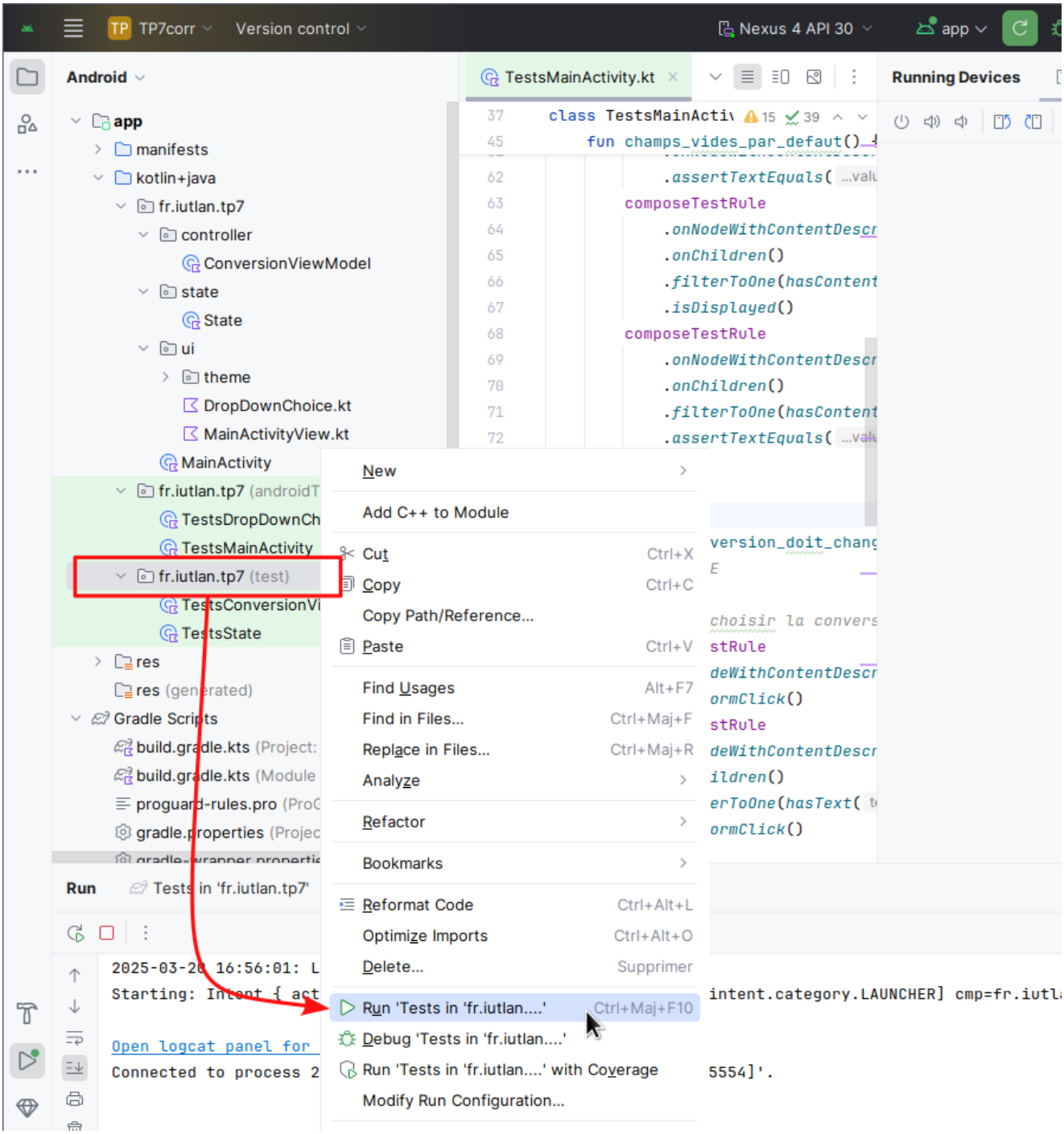

Figure 2: Lancement d'un test unitaire

Vous verrez que ce test s'est ajouté au bouton de lancement `app` à côté du triangle vert, figure 3.

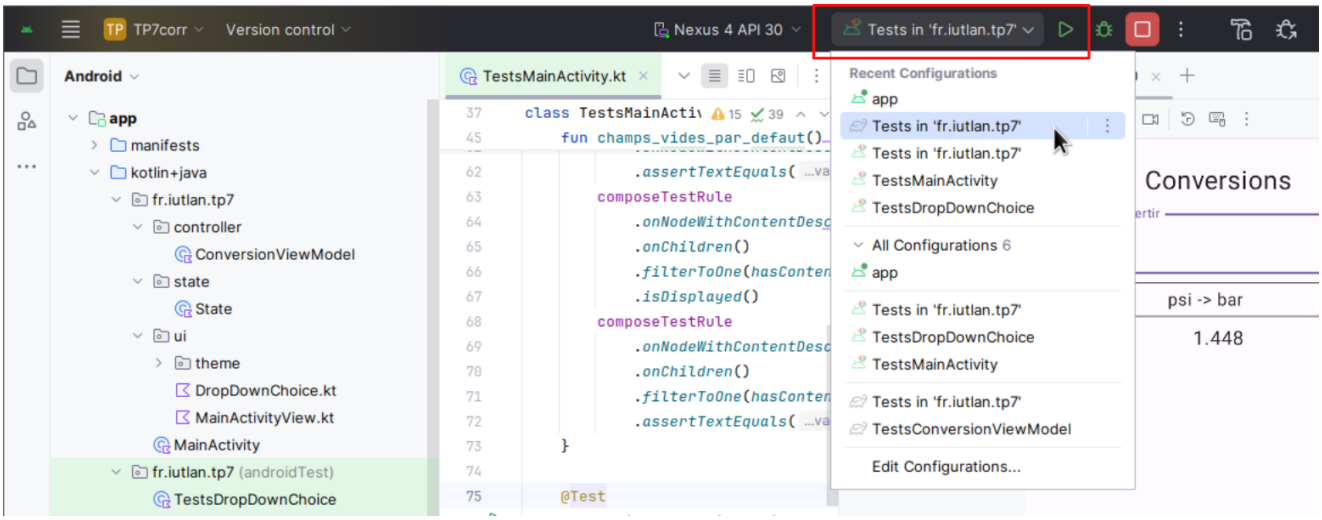

Figure 3: Configuration de lancement

NB: il est possible que les tests échouent par manque de mémoire RAM. La situation est extrêmement limite sur les postes fixes de l'IUT. Fermez les applications inutiles et relancez en espérant que ça s'arrange.

2. Tests unitaires sur `State`

Le but est de vérifier les fonctions de base du logiciel, en les isolant du reste. Dans notre projet, on voudra vérifier que la classe `State` fonctionne correctement avant de l'utiliser dans l'application. Comme c'est un bac à sable pour apprendre à faire des tests, forcément, cette classe a été mal programmée, et en plus de ça, mal conçue initialement. Mais c'est exactement ce que vous pourriez rencontrer dans votre vie professionnelle.

Attention On commence par des tests dans la partie « test », ne pas confondre avec « androidTest » de la fin du TP.

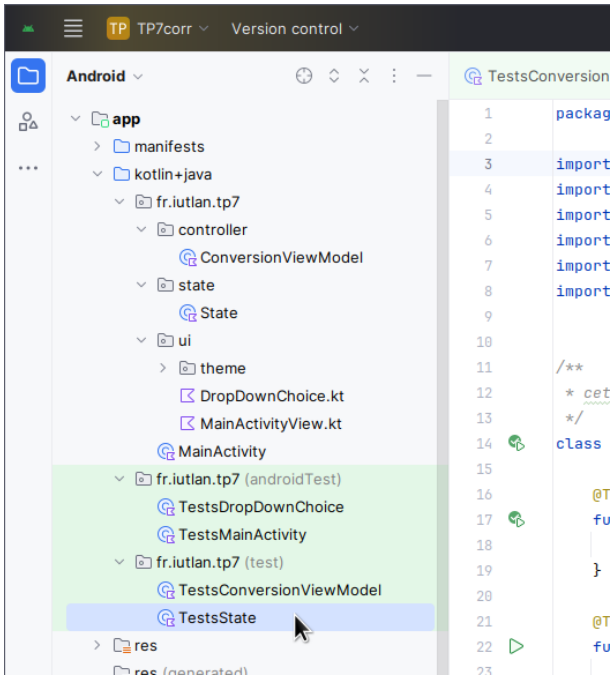

Figure 4: Tests unitaires

👉 Ouvrez `TestsState.kt` et `State.kt` (l'extension est masquée quand le fichier contient une classe du même nom).

👉 Juste pour vérifier que tout fonctionne, lancez cette classe de tests.

Il y a un premier test `addition_est_correcte()`. Il est fourni en tant qu'exemple. Vous reconnaissez les assertions AssertJ, et le cadre JUnit5 du cours R4.02 Qualité de développement. Les `import` sont ceux de ces deux API (Jupiter et AssertJ).

👉 Altérez le calcul $2+3$ en $2+7$ et recommencez. Vous verrez les tests réussis et échoués dans la fenêtre `Run` en bas de l'écran. Avant de continuer, soit vous supprimez ce test (seulement la méthode), soit vous la remettez en état.

Maintenant on va ajouter différents tests unitaires sérieux. Le but de cette suite de tests est de prouver que la classe `State` est conforme au cahier des charges, mais ce n'est pas le cas. Il y a déjà des tests, mais ils ne sont pas bien faits. Ils ne devraient pas tous réussir. C'est à vous de corriger ça.

2.1. Vérification du constructeur

La classe `State` contient un tableau de constantes. La variable membre `code` indique quelle est la constante à employer pour la conversion. Exemple, quand `codeConversion` vaut `0`, c'est une conversion « pouce vers cm ». Cette constante sera positionnée par le `DropDownChoice` de l'interface, dans `MainActivityView` et elle est initialisée par le constructeur.

👉 Complétez `constructeur_est_correct` et `constructeur_defauts_est_correct`.

Le test `constructeur_defauts_est_correct` échoue parce que l'une des valeurs par défaut est mauvaise, `1` à la place de `0`.

☛ Corrigez ce bug dans `State.kt`, car vous avez maintenant un test qui prouve qu'il y a un problème. On va continuer de cette manière : d'abord on prouve qu'il y a un problème avec un test qui échoue et ensuite, on le répare dans `State.kt`.

2.2. Noms des conversions

☛ Complétez `conversionNames_sont_corrects`. Si vous utilisez la bonne assertion, vous pouvez enlever `hasSize(4)`. Corrigez le bug que vous trouvez avec ce test.

☛ Complétez `conversionNames_coefficients_mm_taille`. Corrigez le bug que vous trouvez avec ce test.

Remarquez que le tableau des coefficients est privé. On ne pourra pas le tester. Ce sont des choses qui arrivent. Le client a voulu, sans doute pour des raisons de sécurité, qu'on ne puisse pas accéder à cette information hors de la classe. Donc on ne peut rien tester dessus directement.

2.3. Conversions

On arrive au plus critique, les calculs de conversion. La méthode `convertir_est_correct` est paramétrée et l'idée est de fournir des triplets (*code*, *entrée*, *sortie*) pour vérifier les calculs.

NB: les paramètres sont fournis par `@CsvSource` qui est plus simple ici que `@MethodSource` vu en R4.02.

☛ Complétez `convertir_est_correct` et fournissez au moins 3 valeurs pertinentes pour chaque conversion. Aidez-vous des convertisseurs en ligne, par exemple, celui de Google.

☛ Corrigez tous les bugs mis en évidence dans `State.kt`.

2.4. Copies

Les quatre derniers tests vérifient la copie avec changement de l'état. Il y a deux cas à chaque fois. Si la nouvelle valeur est incorrecte, alors l'état n'est pas changé, c'est la même instance d'objet. Si la nouvelle valeur est correcte, alors elle est prise en compte et le résultat de la conversion est aussi mis à jour dans la copie de l'état.

☛ Complétez les 4 tests `copySet*` et corrigez tous les bugs de `State.kt`.

Il n'est pas demandé de paramétrer les tests corrects. Notez que le test `copySetNombre_mauvais...` reçoit des chaînes en paramètre, et dans ce cas, `@CsvSource` demande des chaînes séparées par des apostrophes (*simples quotes*) à l'intérieur : `"'la chaine à fournir'"`.

À ce stade, tous les tests de `TestsState.kt` doivent passer et tous les bugs de `State.kt` doivent être corrigés.

3. Tests d'intégration sur `ConversionViewModel`

☛ Ouvrez `TestsConversionViewModel.kt` et `ConversionViewModel.kt`.

Il n'y a pas de bug dans `ConversionViewModel`, il est tellement simple, mais il faut s'en assurer. Les tests fournis ne sont pas terminés.

👉 Complétez tous les tests de `TestsConversionViewModel.kt` et faites-les réussir.

Ces tests vérifient la bonne intégration entre le *view model* et l'état.

4. Tests fonctionnels, explications

On arrive à des tests fonctionnels, ou *instrumentés*, qui ne peuvent se faire que sur un AVD, comme ceux qu'on fait avec Robot Framework sur une interface web. Les tests consistent à afficher la fonction composable, à vérifier son contenu visible ou non, et à faire des actions suivies à nouveau d'assertions sur le contenu, ses exécutions de lambda qu'on lui fournit et ses modifications sur un *viewmodel*. Ce sont des sortes de scénarios d'action sur l'interface utilisateur.

👉 IMPORTANT : sélectionnez l'AVD en API 26 et vérifiez constamment que c'est lui qui va être utilisé dans les tests.

Ces tests sont à la fois compliqués à mettre en place dans une application et compliqués à programmer. On va commencer par un petit tutoriel.

4.1. Classe à tester

👉 Ouvrez `AvisNombre.kt` et faites afficher sa prévisualisation. Si vous la survolez à la souris, vous verrez un petit menu au dessus à droite et vous pouvez cliquer le dernier bouton pour lancer la prévisualisation en tant qu'application dans l'AVD, voir la figure 5.

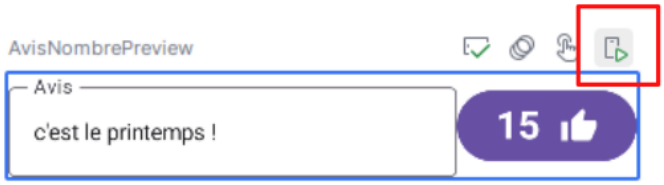

Figure 5: Menu prévisualisation dans l'AVD

On a une ligne comprenant un texte et un bouton, ce bouton possède lui aussi un texte.

Il y a une nouveauté, c'est le `Modifier.semantics`, sur les textes et le bouton. C'est une sorte d'annotation, de marqueur ajoutée aux éléments pour pouvoir les désigner dans les tests. Ces annotations servent aussi pour l'accessibilité des applications, à expliquer le rôle des composants. On place cette annotation sur tous les éléments mentionnés dans les tests : boutons, zones de saisie et d'affichage, etc.

4.2. Principes des tests

👉 Ouvrez `TestsAvisNombre.kt`. Il y a énormément de points à comprendre.

Le principe de ces tests est d'être lancés sur un AVD. Chacun met en place une interface composable et ensuite il y a des vérifications et des actions automatiques sur cette interface.

- Au début de la classe, il y a une constante spéciale `composeTestRule` qui est créée par un appel à `createComposeRule()`. Cette constante représente une sorte d'activité qui va être lancée sur l'AVD au début de chaque test.

- Dans chaque test, il y a une partie *arrange* qui prépare le contenu de `composeTestRule`, par exemple ici, en installant un `AvisNombre`. Cette étape ressemble à ce qu'on trouve dans `MainActivity`.
- Ensuite, on trouve des assertions et des actions. Elles ont toutes le même principe. D'abord on désigne l'un des *Nodes* de l'interface, c'est à dire un élément de l'interface, visible ou non, généré par l'initialisation de `composeTestRule`. Il y a plusieurs moyens de le désigner :
  - `composeTestRule.onNodeWithContentDescription(nom)` en lui passant le marqueur sémantique voulue,
  - `composeTestRule.onNodeWithText(texte)` si l'élément n'a pas de marqueur sémantique, par exemple parce qu'il est généré automatiquement (élément de liste).

Il faut savoir que les fonctions composables construisent parfois plusieurs *nodes*. Par exemple `OutlinedTextField` construit 3 éléments, dont une zone de saisie et un titre (label). La fonction `composeTestRule.onNodeWithContentDescription(nom)` sélectionne les trois à la fois, parce que le marqueur sémantique est appliquée à tous. Ça empêche de vérifier des assertions ou d'effectuer des actions. La solution consiste à ajouter l'option `useUnmergedTree = true` pour séparer les éléments et parfois, il faut désigner l'un des *nodes* plus précisément, on verra comment le cas échéant.

On doit connaître quelques assertions comme :

- `assertIsDisplayed()` vraie si l'élément est affiché, l'inverse : `assertIsNotDisplayed()`.
- `assertTextEquals(texte)` vraie si l'élément affiche ce texte, avec son opposée `assertTextNotEquals(texte)`.

Parmi les actions à connaître, on a :

- `performClick()` : clique sur l'élément
- `performTextReplacement(texte)` : remplace le texte de la zone de saisie par celui fourni.

Toutes ces fonctions de recherche, ces assertions et ces actions sont répertoriées dans cette page . Il y a un pdf imprimable qui récapitule tout ce qui existe.

👉 Affichez le document pdf qui est cité dans la page. Vous remarquerez que les noms des méthodes sont cliquables et ça amène sur leur documentation.

4.3. Tests `TestsMainActivityView.kt`

👉 Lancez les tests de ce fichier et regardez leur code source. Vous allez bientôt devoir vous en inspirer.

Dans le test `cliquer_bouton_incremente_nombre_affiche`, il y a trois appels à `Thread.sleep`. C'est un moyen pour avoir le temps de voir ce qui se passe, avant et après le clic sur le bouton. Regardez où ces instructions sont placées. La deuxième porte un commentaire qui signale qu'elle ne sert à rien. En effet, lorsqu'un test fait une action sur l'interface, ses conséquences ne sont pas appliquées immédiatement : une recomposition, mais seulement s'il y a une assertion.

Vous pouvez enlever ces pauses, mais retenez l'idée.

Il semble que cette catégorie de tests soit encore largement en travaux, parce que certaines opérations sont extrêmement difficiles à faire, et il y a des bugs dans Compose, ainsi que vous allez le constater dans ce TP.

5. Tests sur `MainActivityView`

C'est la fonction qui affiche l'interface complète de l'application. Elle contient une zone de saisie pour le nombre à convertir, un menu déroulant pour choisir la conversion et un texte pour afficher le résultat. Nous allons vérifier par exemple que l'état est correctement affiché.

👉 Ouvrez `MainActivityView.kt`.

Cette fonction composable est commentée. Étudiez un peu son fonctionnement pour avoir une idée des tests.

👉 Ajoutez les marqueurs sémantiques suivants :

- `"entree"` sur le `OutlinedTextField`,
- `"choix"` sur le `DropDownChoice`,
- `"sortie"` sur le `Text` qui affiche le résultat.

👉 Ouvrez `TestsMainActivityView.kt`. Ce sont les tests à faire passer. Ils sont incomplets.

👉 Dans `etat_est_correctement_affiche`, ajoutez les deux assertions manquantes.

L'assertion sur la liste est étrange. Vous avez placé un marqueur sémantique sur le choix, mais vous n'avez pas accès aux sous-éléments. Vous ne pouvez pas distinguer l'élément courant et la liste déroulante. C'est pour ça qu'il y a cette construction `onChildren.filterToOne(condition)` pour attraper le choix courant.

👉 Vérifiez que le test échoue avec un autre code, ou une autre chaîne à la place de `"psi -> bar"`.

6. Tests sur `DropDownChoice`

C'est une fonction importante dans le logiciel. Elle permet de choisir la conversion à effectuer, parmi les 4 possibles. On va vérifier qu'elle affiche correctement le choix courant et qu'on peut choisir un autre en cliquant sur le titre puis un élément.

👉 Ouvrez `DropDownChoice.kt`.

Cette fonction composable est commentée. Étudiez un peu son fonctionnement pour avoir une idée des tests.

👉 Ajoutez les marqueurs sémantiques suivants :

- `"courant"` sur le `Text` qui affiche le choix courant,
- `"liste"` sur le `DropdownMenu`.

👉 Ouvrez `TestsDropDownChoice.kt`. Ce sont les tests à faire passer. Ils sont incomplets.

Le premier test, `choix_courant_N_est_visible` est une sorte de test paramétré. On ne dispose pas de JUnit5 quand on fait les tests sur AVD. L'annotation `@ParameterizedTest` n'est pas accessible, alors on bricole avec une fonction privée qui fait le travail.

👉 Complétez les tests et testez-les.

7. Tests sur `MainActivity`

On termine avec les tests sur l'activité principale. Il n'y a pas grand chose dans la classe `MainActivity` mais on peut quand même vérifier que

👉 Ouvrez `TestsConversionViewModel.kt` et `MainActivity.kt` (pas `MainActivityView.kt`).

Il y a une petite différence avec les tests précédents concernant la création de `composeTestRule`. On crée une instance qui va s'occuper de `MainActivity`. L'autre écriture gère une activité quelconque.

Il n'y a aucun marqueur sémantique supplémentaire. Ceux que vous avez rajouté dans les fonctions composables sont tout à fait utilisables.

Les tests que vous trouvez ici sont de petits scénarios bâtis sur des cas d'utilisation. Ils comprennent beaucoup d'actions et de vérifications.

👉 Complétez les tests et testez tous les tests.

Remarquez comment on accède au *viewmodel* de l'activité dans `saisie_nombre_doit_changer_etat`, pour des assertions.

👉 Vérifiez que la totalité des tests passent.

Vous pouvez maintenant vendre cette application, elle ne contient aucun bug.

8. Travail à rendre

Important : votre projet doit se compiler et pouvoir être lancé. La note sera zéro si ce n'est pas le cas. Mettez donc en commentaire ce qui ne compile pas.

Avec le navigateur de fichiers, descendez dans le dossier `app` du projet, là où on voit les trois dossiers `build`, `libs` et `src`. C'est `src` qu'il faut rendre cette semaine, et non pas seulement `main` qui se trouve dedans.

Rajoutez un fichier appelé exactement `IMPORTANT.txt` dans le sous-dossier `src` si vous avez rencontré des problèmes techniques durant le TP : plantages, erreurs inexplicables, perte du travail, etc. mais pas les problèmes dus à un manque de travail ou de compréhension. Décrivez exactement ce qui s'est passé. Le correcteur pourra lire ce fichier au moment de la notation et compenser votre note.

Cliquez droit sur le dossier `src`, choisissez `Compresser...`, format `ZIP`, cliquez sur `Creer`. Déposez l'archive `src.zip` sur Moodle au bon endroit, voir sur la page Moodle R4.A11 Développement Mobile.