Junit Android pour les PROs

Nous allons ajouter la possibilité d'effectuer des tests unitaires intégrants plusieurs terminaux Android, comme dans le cas d'une communication entre téléphones, pour communiquer en NFC, lors d'un scan d'un QRCode présenté par un autre device, etc. Android ne propose rien pour cela, mais nous allons voir qu'avec beaucoup d'astuces et de lecture des sources d'Android, nous allons y parvenir simplement.

Par Philippe PRADOS - 2012
www.prados.fr

Les tests unitaires sous Android

Android propose toute une batterie d'outils pour faciliter les tests unitaires des applications. Il est possible de tester du code POJO (*Plain Old Java Object*) dans une application Android, de tester une activité, un service ou un fournisseur de contenu (**Content Provider**). Les activités peuvent être testées en dehors de tous contexte d'exécution (**ActivityInstrumentationTestCase**), ou classiquement présenté à l'écran (**ActivityInstrumentationTestCase2**).

Pour effectuer des tests unitaires, il est important de pouvoir isoler chaque test les uns des autres, pour ne pas avoir d'effet de bord pouvant, soit provoquer des erreurs normalement impossible, soit faire croire à un fonctionnement correct, alors que le traitement n'est valide qu'après un autre traitement.

Pour éviter cela, une déclinaison du framework JUnit version 3, bien connu des développeurs Java, est proposé dans le framework Android. Il est possible d'injecter du code de test dans une application Android, pour effectuer des scénarios d'utilisations des composants. Cela permet d'avoir un accès direct aux composants de l'activité, pour injecter des traitements comme la prise de focus, ou pour analyser le contenu d'un composant graphique.

Afin de ne pas devoir ré-installer l'application complètement entre chaque test, il est possible de fonctionner avec des MockObjects (package **android.test.mock**), des simulations d'objets standards Android. Toute une batterie est disponible avec le framework. Par exemple, il est facile de simuler un état particulier du **Context**. C'est d'ailleurs pour cela qu'il existe cette notion et le **Context** d'application (**getApplicationContext()**). Cela permet de rédiger un code sans jamais avoir d'objet statiques (Conforme au modèle d'injection de dépendance). Par injection, chaque objet peut être isolé de tous les autres. Il n'est pas nécessaire d'avoir des variables statiques, comme des caches ou autres données partagées. Le framework Android respect cela.

Les applications pas toujours. Cela complexifie les tests unitaires, car un test peut avoir des effets de bords. Il est parfois difficile de rétablir une situation propre entre chaque test. Pour pouvoir tester correctement votre application, préférez l'utilisation du contexte applicatif après un *cast*, pour toutes les données partagées. Vous pourrez ainsi écrire des tests unitaires, sans effet de bord.

```
class MonApp extends Application
{
}
...
MonApp getApplicationContext()
{
  return (MonApp)super.getApplicationContext() ;
}
```

Comment tester des applications Client / serveur ?

Les applications client/serveurs utilisent au moins deux composants. Une application Android comme client, et un serveur, généralement un site internet. Mais, il existe également d'autres situations, comme par exemple, la communication directe entre deux terminaux Android (Bluetooth, Wifidirect, NFC, Scan de QRCode, ou plus simplement par WIFI). La communication par 3G est plus complexe, car nos chères opérateurs ne proposent toujours pas une adresse IPV6 pour nos terminaux, alors que le 6 juin est maintenant dernier nous :-((Pour rappel à ceux qui ne suive pas, le 6 juin a été le jour officiel d'ouverture de l'IPV6 dans le monde).

Comment tester ce type de situation ? Les outils de tests unitaires d'Android permettent d'exécuter des scénarios de tests, mais seulement limités à un seul terminal. Il n'est pas possible de demander le démarrage d'un service sur un terminal avant l'exécution d'un test depuis un autre. Et encore moins de contrôler tous cela lors de l'exécution. Impossible de vérifier que les traitements sont correctes des deux cotés de la communication.

Nous souhaitons pouvoir tester tous les scénarios impliquant plusieurs terminaux. Cela intègre les communications réseaux, NFC, Caméra/écran, réseaux sociaux, etc. Et pourquoi pas, les situations impliquant de nombreux terminaux.

Avec cet objectif en tête, nous nous plongeons dans les outils proposés par Android pour rédiger des tests unitaires.

Les outils proposés par Android

Android permet de rédiger une application de test, lié à une application classique. Cette application spécifique possède deux caractéristiques dans le fichier **AndroidManifest.xml**.

Il faut d'abord indiquer que l'application possède un ou plusieurs composants d'instrumentation.

```
<instrumentation
  android:name="android.test.InstrumentationTestRunner"
  android:targetPackage="fr.prados.clientserver" />
```

Le paramètre **targetPackage** indique alors l'application où doit être injecté le code de l'application de test, pour pouvoir invoquer tous les services de cette dernière. Le paramètre **name** indique une classe dérivée de **Instrumentation**. Android propose une implémentation par défaut, s'occupant du lien vers les autres classes permettant de tester les différents composant d'une application. Généralement, on ne modifier pas ce paramètre. Nous allons voir qu'il est parfois utile de le faire, mais il est trop tôt pour l'évoquer.

Il manque encore un paramètre dans le fichier **AndroidManifest.xml** : la déclaration de la librairie de test dans le marqueur **<application/>**.

```
<uses-library android:name="android.test.runner" />
```

En effet, les librairies de tests unitaires sont sensibles, car elles permettent d'injecter des touches ou des actions de l'utilisateur. Elles ne sont donc pas disponible dans la librairie par défaut des applications. Un constructeur peut même la supprimer, pour interdire tous les tests unitaires sur le terminal. Imaginez une application malveillante qui envoie des simulations de touché à l'application d'envoi de SMS ? Il ne serait plus nécessaire d'avoir de privilège pour envoyer des messages surtaxés.

Avant même le démarrage de l'application par l'invocation de la méthode **onCreate()** de l'instance **Application**, l'instance **InstrumentationTestRunner** est invoquée. À chaque événement important du cycle de vie de l'application correspond une méthode d'appel en retour permettant de prendre le contrôle. Il est alors possible d'invoquer les différentes méthodes du cycle de vie d'une activité par exemple, puis de manipuler l'instance activité et tous ses composants.

Comment déclencher les tests ?

Il existe plusieurs possibilités pour démarrer un test ou une suite de tests unitaires. La première approche, la plus simple, consiste à utiliser notre ami Éclipse et de demander « *Run As / Android Junit Test* ». Les tests unitaires sont alors déclenchés les uns après les autres. Éclipse affiche la progression et l'arbre de tous les

tests. Il est possible de rejouer les tests ayant échoués ou seulement certains d'entre-eux unitairement.

Il est possible de déclencher des tests unitaires JUnit classique, pour qualifier des algorithmes ou des traitements hors des composants Androids ou des tests unitaires spécialisés pour les composants de notre système préféré.

En étudiant les paramètres de démarrage du test (*Debug configuration...*), on constate qu'il est possible de sélectionner d'autres composant déclarées dans les marqueurs **<instrumentation/>**. Si vous en déclarez plusieurs, pour pouvez sélectionner celui que vous souhaitez. Nous utiliserons cela plus tard.

Am instrument

Il est également possible de démarrer les tests depuis la ligne de commande d'Android. En générale, on utilise **adb** avec le paramètre **shell** pour demander l'exécution depuis le Desktop.

L'outil **am** (Android Manager) propose un module instrument qui permet de démarrer un test unitaire. Plusieurs paramètres sont disponibles. Il faut systématiquement indiquer le point de départ de l'instrumentation. Cela se décompose en deux paramètres séparé d'un slash : le nom du package de l'application de test et le type de la classe d'instrumentation à utiliser (voir le marqueur correspondant dans le fichier **AndroidManifest.xml**).

```
> adb shell am instrument -w com.android.foo/android.test.InstrumentationTestRunner
```

Le paramètre **-w** permet d'indiquer que l'on souhaite attendre la fin du traitement avant de rendre la main. Les tests sont alors exécutés. Une traces indique les résultats. Il est alors possible de déclencher les différents tests depuis le Desktop. C'est très pratique pour systématiser l'exécution des tests sur plusieurs émulateurs différents, simulant différentes combinaisons de taille d'écran et de version de l'OS.

Il est possible de limiter les tests possédants des annotations spécifiques, de ne déclencher qu'un test particulier ou qu'un niveau de test (**small**, **medium**, **large**). Je vous invite à consulter la documentation de la classe **InstrumentationTestRunner** pour plus de détail.

En ajoutant **-e debug true**, le dévermineur d'Eclipse prend la main. D'autres paramètres peuvent être ajoutés via le paramètre **-e**. Nous verrons plus tard comment récupérer ces paramètres qui ne sont pas, par défaut, accessibles aux tests unitaires. En effet, les méthodes existes dans le code d'Android, mais déclarées **@hide**.

Monkey

Il existe également un composant amusant pour tester une application Android : le singe. C'est un composant permettant d'injecter aléatoirement des événements de tous types dans une application. Cela permet de simuler l'utilisation du terminal par un utilisateur classique, non, plutôt un primate. Enfin, c'est pareil.

Essayez la commande suivante, vous risquez d'être surpris ! Ou pas.

```
> adb shell monkey 10
```

Nous allons analyser et comprendre comment fonctionne ce composant un peu plus loin. Cela sera une source d'inspiration pour répondre à notre besoin.

Monkeyrunner

Il nous reste à évoquer un dernier composant sympathique : **monkeyrunner**. C'est un composant permettant d'installer un APK, de démarrer une session d'instrumentation et d'injecter des événements. Tous cela via des scripts rédigés en Python, ou plutôt en Jython. Ce composant permet l'injection d'événement, mais pas de contrôler l'impact de ces événements sur l'application.

C'est également un composant que nous allons étudier pour répondre à notre objectif.

Pour le moment, nous constatons que tous ces outils sont sympathique, mais ne permettent pas d'invoquer une méthode d'une application pour contrôler un scénario de test impliquant plusieurs téléphones, tablettes ou téléviseurs. Nous pouvons lancer des tests unitaires en parallèles sur plusieurs périphériques, injecter des événements, mais nous ne pouvons rien sortir d'un périphérique. Par exemple, il n'est pas possible de

demander l'adresse IP ou un autre paramètre d'un Android pour livrer l'information à un test unitaire d'un autre périphérique afin de tester le processus de connexion.

L'étude du code d'Android

Comment **monkeyrunner** fait pour envoyer des événements à un périphérique ? Nous entrons dans le code présent dans le répertoire **sdk** des sources d'Android. Nous constatons que le code démarre l'application **monkey** sur le device, avec le paramètre **--port 12345** (**adb shell monkey –port 12345**). Une étude de **monkey** montre que dans ce cas, **monkey** se met à l'écoute d'un socket sur ce port. Pour des raisons de sécurités, les clients sont limités aux connexions venant de **localhost**.

Pour que **monkeyrunner** puisse communiquer avec **monkey**, il faut demander au serveur ADB d'effectuer un transfert de port pour exposer un port local vers le port **12345** de **monkey** (**adb forward tcp:12345 tcp:12345**).

Ainsi, **monkeyrunner** peut envoyer des ordres à **monkey** sur **localhost:12345**. Les ordres sont alors capturés par **adb** et transférés à l'agent **adb** présent dans le terminal Android. Ce dernier se connecte alors vers le port local **12345**. **monkey** accepte alors les connexions car elles viennent de **localhost**.

Ainsi, limiter les accès à **localhost** revient à limiter les accès à l'USB. Et c'est exactement ce que nous souhaitons. En effet, nous voulons pouvoir tester les cas ou le réseau Wifi n'est plus disponible, le mode avion, etc.

Les ordres sont composés de lignes de texte du type « **instrument <param>\n** ». Le traitement est exécuté, l'événement injecté, puis un statu composé également d'une ligne est retourné à **monkeyrunner**.

Nous ébauchons alors une piste pour communiquer avec une application, avec pour objectif de pouvoir invoquer une méthode statique avec des paramètres et récupérer le résultat. Ainsi, il devrait être possible de rédiger un test unitaire cross device, récupérant des informations de l'un pour les donner à un autre. Par exemple, demander à l'application à tester de nous donner l'image de l'écran qu'elle présente, puis de l'envoyer à un autre Android pour lui faire croire qu'il s'agit de l'image capturé par la caméra.

monkeyrunner permet de rédiger des scripts en Python. Nous voulons des scripts JUnit java. Nous étudions plus en profondeur le code, pour voir s'il est facile de proposer une petite API pour offrir plus de service de manipulation, avec un JUnit qui sera exécuté sur le Desktop, donc dans Éclipse et non dans un Android.

En étudiant un peu plus le code, nous constatons que ce dernier utilise des APIs permettant de communiquer directement avec le serveur ADB. Cool ! Nous allons pouvoir les récupérer pour aller plus loin.

En effet, la librairie **ddmlib.jar** présente dans le répertoire **<android_sdk>/tools/libs** nous donne un accès plus facile à ADB. Quelques lignes permettent d'ouvrir la connexion.

```
AndroidDebugBridge.init(false /* debugger support */);
mBridge = AndroidDebugBridge.createBridge(
        adbLocation, true /* forceNewBridge */);
```

La méthode **getDevices()** permet alors de récupérer la liste des périphériques connectés dans des objets de type **IDevice**. Cette dernière classe offre tous un ensemble de méthode pour communiquer via le port USB ou Wifi, si le terminal accepte le déverminage via ce canal.

Nous ajoutons également **sdklib.jar** pour avoir accès à quelques constantes du SDK.

Architecture

Nous pouvons alors imaginer une architecture pour notre composant. Avec pour objectif de rédiger le moins de code possible, comme toujours. (*La fainéantise est mère de la créativité*).

Le composant **monkey** nous permet d'injecter des événements ou de démarrer des tests unitaires. Mais, il ne permet par d'invoquer une méthode particulière présente dans l'application. Comme le fait **monkey**, nous allons alors ouvrir un « *serveur de méthodes* » en écoute sur un socket, pour recevoir des ordres depuis le port USB. Le socket va lire des objets sérialisés pour connaître la méthode à invoquer ainsi que ses paramètres, et retourner le résultat ou l'exception dans un autre objet sérialisé.

Ce serveur de méthodes ne doit pas être présent dans l'application, mais uniquement lors des tests unitaires. C'est maintenant que nous nous rappelons du marqueur **<instrument/>** de

AndroidManifest.xml. Il doit être possible d'utiliser une classe à nous à la place de **android.test.InstrumentationTestRunner** pour ajouter le démarrage du serveur de méthodes.

Notre classe **DesktopInstrumentationTestRunner** hérite de **InstrumentationTestRunner** pour démarrer automatiquement le serveur de méthodes dans le constructeur. Il suffit d'utiliser notre classe dans le marqueur **<instrumentation/>** de **AndroidManifest.xml**.

Le serveur de méthodes s'écrit en quelques lignes de code.

```
public class Server extends Thread
{

	@Override
	public void run()
	{
		ServerSocket srvSocket;
		try
		{
			Log.i(TAG,"Start server on port 1088)");
			srvSocket=new ServerSocket(1088,0,InetAddress.getLocalHost());
			ObjectInputStream input;
			ObjectOutputStream output;

			for (;;)
			{
				try
				{
					Socket clientSocket=srvSocket.accept();
					Log.i(TAG,"Server accept client)");
					input = new

ObjectInputStream(clientSocket.getInputStream());
					output = new

OjectOutputStream(clientSocket.getOutputStream());
					for (;;)
					{
						Log.i(TAG,"Execute...");
						execute(input,output);
					}
				}
				catch (EOFException e)
				{
					// Ignore
				}
				catch (IOException e)
				{
					Log.e(TAG,"Exception",e);
				}
				catch (ClassNotFoundException e)
				{
					Log.e(TAG,"Exception",e);
				}

			}
		}
		catch (IOException e)
		{
			e.printStackTrace();
		}
	}
}
```

La méthode **execute()** va simplement récupérer un objet **Request** sérialisé possédant le nom de la méthode à invoquer (format **<package>.<method>** pour les méthodes statiques, ou **<method>** pour les méthodes de la classe dérivé de **InstrumentationTestRunner**) ainsi qu'un tableau avec tous les paramètres. Merci à java de permettre la sérialisation entre un code Desktop et un code Android ;-)

Nous profitons de cette classe pour exposer les paramètres qui peuvent être ajoutés lors du lancement d'un test unitaire par la ligne de commande sous un **shell** dans Android. Cela se fait en une minute, en

surchargeant cette méthode :

```
@Override
public void onCreate(Bundle arguments)
{
        super.onCreate(arguments);
        mArguments=arguments;
}
```

Ainsi, un getter nous donne accès aux paramètres. Cela nous sera utile plus tard.

Coté Desktop maintenant, nous allons encapsulé les classes de **ddmlib.jar** pour simplifier les accès et pouvoir les utiliser simplement depuis un JUnit test classique.

Comme le fait **monkeyrunner**, nous encapsulant ces méthodes dans les classes **Adb** et **AdbDevice**. Nous nous inspirons très fortement des classes équivalentes de **monkeyrunner**. Une fois la connexion établie avec **adb**, puis avec un ou plusieurs Androids, nous pouvons lancer un test unitaire pour démarrer le serveur de méthodes. Pour pouvoir communiquer avec le serveur de méthode, il faut demander un transfert de port à **adb**. Rien de plus simple avec la libraire proposée par Android.

```
mDevice.createForward(mLocalPort, DEST_PORT);
```

Nous pouvons alors démarrer des tests unitaires. Ces derniers déclenchent le lancement du serveur de méthode sur le port **1088**. Comme nous pouvons nous connecter sur plusieurs périphérique, nous devons mapper chacun sur des ports locaux différents. Un simple compteur va nous aider à distribuer les ports.

Pause/continue

Il nous reste à régler un dernier problème. Lors du déclenchement d'une séance d'instrumentation, l'application est d'abord tuée (**kill**) avant d'être re-démarrée. Cela permet de supprimer tous les effets de bords. Puis, les tests unitaires sont exécutés et l'application est interrompu à nouveau. Il ne va pas être facile de démarrer deux tests sur deux Androids en même temps sans que l'un se termine avant l'autre ! Nous allons alors proposer un mécanisme pour mettre en pause un test unitaire. Charge au Desktop de reprendre le traitement lorsque cela est nécessaire.

Nous ajoutons une méthode **pauseTest()** à **DesktopInstrumentationTestRunner**. Elle consiste simplement en un **wait()**. Nous proposons également une méthode **continueTest()** qui devra être déclenchée par notre serveur de méthode. Ainsi, nous allons pouvoir écrire un scénario qui demande l'exécution d'un test unitaire sur l'Android jouant le rôle de serveur. Ce test unitaire démarre un traitement, puis se met un pause, via la récupération de notre classe d'instrumentation.

Pour simplifier l'utilisation, nous écrivons une petite classe dérivé de **ActivityInstrumentationTestCase2<T>** pour modifier le type de retour de la méthode **getInstrumentation()**.

```
public abstract class DesktopActivityInstrumentationTestCase2<T extends Activity>
        extends ActivityInstrumentationTestCase2<T>
{
        public DesktopActivityInstrumentationTestCase2(Class<T> activityClass)
        {
                super(activityClass);
        }
        @Deprecated
        public DesktopActivityInstrumentationTestCase2(String pkg,
                Class<T> activityClass)
        {
                super(pkg,activityClass);
        }
        public DesktopInstrumentationTestRunner getInstrumentation()
        {
                return (DesktopInstrumentationTestRunner)super.getInstrumentation();
        }
        protected Bundle getBundle()
        {
           return getInstrumentation().getBundle();
        }
        protected void pauseTest()
        {
                getInstrumentation().pauseTest();
```

```
        }
}
```

Un test unitaire peut alors facilement avoir accès à la méthode **pauseTest()** et aux paramètres de la session (via l'exposition du **Bundle** comme décrit précédemment).

Implémentation

Pour résumer tous nos travaux nous avons fait une petite librairie avec moins de dix classes, quelques méthodes et c'est tous.

Coté Android, nous avons la classe **DesktopInstrumentationTestRunner** qui n'est rien d'autre qu'une autre classe d'instrumentation. Elle offre quatre nouveaux service :

- un serveur de méthodes en écoute d'un socket pour pouvoir exécuter des méthodes de l'instance courante ou des méthodes statiques ;
- Une méthode **getBundle()** pour donner accès aux paramètres de la session de test ;
- et deux méthodes pour mettre en pause et continuer un test unitaire.

Nous avons également une classe **DesktopActivityInstrumentationTestCase2** pour faciliter la colle avec la classe précédente.

Coté Desktop, nous avons :

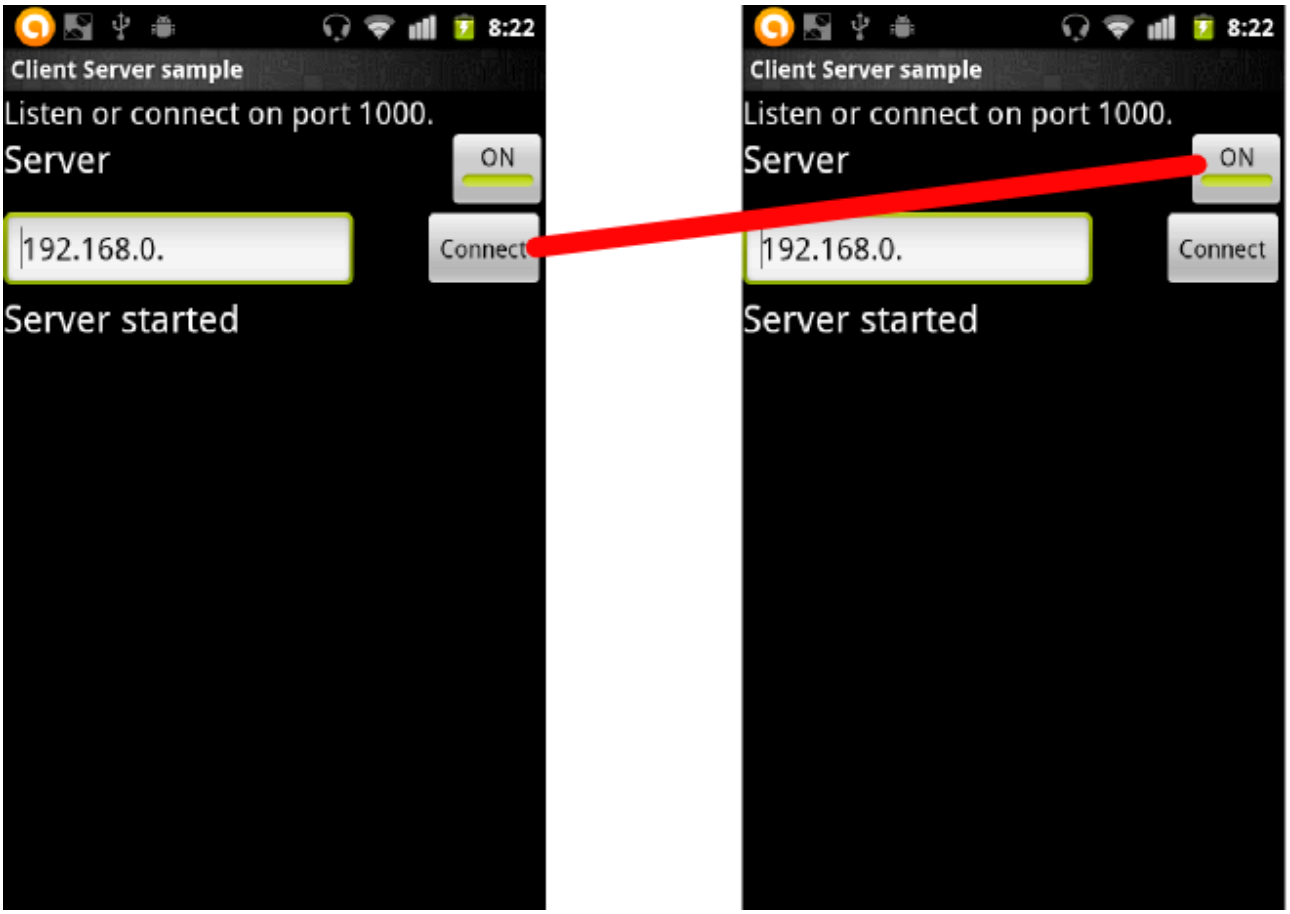
- La classe **Adb** pour se connecter à **adb** ;
- La classe **AdbDevice** pour pouvoir installer des APKs, lancer des tests ou des shells, se connecter au serveur de méthode de l'application et demander l'exécution d'une méthode distante.

Je vous fait grâce des classes **Request** et **Response** servant de transport pour le serveur de méthodes.

Finalement, cela ne fait pas grand chose à se mettre sous la dent.

Cas de test

Nous allons maintenant faire une petite application Android qui va nous permettre de faire jouer tous cela. Cette application est très simple. Un *toggle button* pour lancer ou arrêter un serveur en écoute sur un port ; un champ de saisie pour saisir l'adresse ip d'un autre Android où le serveur est démarré et un bouton pour ouvrir la connexion. Cela envoi le type du terminal au serveur. Ce dernier retourne la chaîne « **hello <paramètre>** » que nous affichons dans une ligne de statu.



C'est une application toute simple, de type client/serveur. Nous pouvons écrire un test unitaire pour vérifier que le serveur est bien activé sur le clic du toggle button, mais c'est tous.

Comment tester le scénario impliquant deux Androids ? Sur le premier, nous voulons rédiger un test unitaire pour démarrer le serveur, puis se mettre en pause avant de l'arrêter proprement.

Sur le deuxième Android, nous voulons valoriser l'adresse IP avec l'adresse du premier, puis cliquer sur le bouton de connexion et vérifier ensuite le statu.

Impossible de faire cela sans nos quelques classes magiques.

Testons

La première étape consiste à créer un projet **ClientServerForDesktopTest** de test associé à notre projet **ClientServerSample** (tous les sources sont disponibles en ligne). Nous ajoutons la librairie nécessaire coté Android, et nous modifions le **AndroidManifest.xml** pour utiliser notre classe lors de l'instrumentation.

```
<instrumentation
  android:name=
    "fr.prados.clientserver.test.fordesktop.ClientServerDesktopInstrumentationTestRunner"
  android:targetPackage="fr.prados.clientserver"
 />
```

Voilà. Lorsque le test est démarré, le serveur en attente des méthodes à invoquer l'est également.

Nous rédigeons alors notre premier test unitaire que nous appelons **testStartServer()**. Ce dernier demande le focus sur le toggle bouton via le UI Thread, puis simule l'appui sur **Enter**.

Si tous s'est bien passé, le test se met en pause. Au réveil, il reclique sur le bouton pour interrompre le serveur.

```
public void testStartServer()
{
        // Start server
        mActivity.runOnUiThread(
            new Runnable()
```

```
            {
              public void run()
              {
                 mToggleServer.requestFocus();
              }
            }
           );
        try { Thread.sleep(2000); } catch (Exception e){}
        sendKeys(KeyEvent.KEYCODE_ENTER);
        getInstrumentation().waitForIdleSync();
        try { Thread.sleep(2000); } catch (Exception e){}
        assertNotNull(mActivity.mServer);
        assertTrue(mActivity.mServer.isAlive());

        Log.d(TAG,"wait next notification");
        pauseTest();
        Log.d(TAG,"receive notification");

        // Stop server
        sendKeys(KeyEvent.KEYCODE_ENTER);
        getInstrumentation().waitForIdleSync();
        try { Thread.sleep(2000); } catch (Exception e){}
        assertFalse(mActivity.mServer.isAlive());
        assertNull(mActivity.mServer);
}
```

Nous allons devoir récupérer l'adresse IP du serveur pour la fournir à l'Android client. Pour cela, nous écrivons une classe **ClientServerDesktopInstrumentationTestRunner** qui étend notre **DesktopInstrumentationTestRunner**. Nous n'oublions pas de modifier le fichier **AndroidManifest.xml** en conséquence. Nous ajoutons alors la méthode **getIp()** comme ceci :

```
public String getIp()
{
        WifiManager wifiManager =
                (WifiManager) getContext().getSystemService(Context.WIFI_SERVICE);
        WifiInfo wifiInfo = wifiManager.getConnectionInfo();
        int i = wifiInfo.getIpAddress();
        return
                ( i & 0xFF) + "." +
                ((i >> 8 ) & 0xFF) + "." +
                ((i >> 16 ) & 0xFF) + "." +
                ((i >> 24 ) & 0xFF );
}
```

Cette méthode pourra être appelée à partir du Desktop, grace à notre serveur de méthode. Il suffit d'utiliser l'instance **AdbDevice** et notre méthode **execute()**.

Justement, nous créons un projet JUnit classique avec Éclipse, et nous ajoutons les archives nécessaires, dont la fameuse archive **ddmlib.jar** présente avec le SDK.

Nous écrivons alors un cas de test. Il est important de bien comprendre chaque étape, c'est pour cela que nous allons les décomposer.

```
public void testConnection() throws Throwable
{
        try
        {
```

Après s'être connecté à deux Android, nous déclarons l'un comme **Server**, et l'autre comme **Client**. Sur le serveur, nous souhaitons démarrer l'instrumentation du test **testStartServer()**. Comme décrit dans la documentation de Google, cela consiste à envoyer un ordre **am** au périphérique. Nous n'utilisons pas la méthode **instrumentation()** de notre API, car la méthode sera mise en pause, ce qui peut être interprété comme une erreur du test.

```
getServer().shell("am instrument "+
        "-w "+
```

Si le debugger est utilisé pour lancer le test sur le Desktop, et bien faisons de même sur le test unitaire dans

l'Android !

```
(isDebug() ? "-e debug true " : "")+
```

Nous indiquons précisément la méthode que nous voulons déclencher.

```
"-e class fr.prados.clientserver.test.fordesktop."+
"ForDesktopActivtyTest#testStartServer "+
```

Puis le nom du projet de test, ainsi que la classe d'instrumentation à utiliser.

```
"fr.prados.clientserver.test.fordesktop/"+
"fr.prados.clientserver.test.fordesktop."+
"ClientServerDesktopInstrumentationTestRunner");
```

Ce traitement va tuer éventuellement l'application si elle est présente en mémoire avant de la relancer. Donc, la connexion éventuelle à notre serveur de méthodes est perdu. Nous devons nous reconnecter.

```
getServer().connectApp(PACKAGENAME);
```

Maintenant, nous pouvons enfin utiliser notre mécanique pour invoquer la méthode **getIp()** et obtenir ainsi l'adresse IP de l'Android jouant le rôle de serveur.

```
String ip=(String)getServer().execute("getIp");
```

Maintenant, plusieurs solutions. Soit nous utilisons la même approche pour injecter l'adresse IP dans le test unitaire client, soit nous utilisons les paramètres de la session d'instrumentation. Nous choisissons cette approche, car elle nous permet d'utiliser l'instrumentation plus classiquement.

```
Map<String,Object> result=getClient().instrument(
```

Le paramètre pourra être récupéré via **getBundle().getString("target")**

```
"-e target "+ip+" "+
```

Nous indiquons précisément la méthode de test à invoquer.

```
"-e class fr.prados.clientserver.test.fordesktop.ForDesktopActivtyTest#testConnect "+
```

Puis le package et la classe d'instrumentation.

```
"fr.prados.clientserver.test.fordesktop/"+
"fr.prados.clientserver.test.fordesktop."+
"ClientServerDesktopInstrumentationTestRunner");
```

Le résultat est récupéré dans une **Map** sous la clef **stream**

```
String stream=(String)result.get("stream");
```

Une petite expression régulière nous permet de savoir si tous c'est bien passé

```
assertTrue(Pattern.compile("^OK.*",Pattern.MULTILINE).matcher(stream).find());
}
finally
{
```

Il ne faut pas oublié de continuer le test sur le serveur avant de partir pour l'arrêter proprement. Pas d'effet de bord je vous dis.

```
try
{
        getServer().continueTest();
}
catch (Throwable e)
{
}
}
```

La méthode **continueTest()** est une simple utilisation de notre méthode **execute()**.

```
adbDevice.execute("continueTest");
```

Cela invoque la méthode de notre instrumentation.

Il nous reste à écrire le test unitaire client cette fois-ci. Ce dernier consiste à récupérer l'adresse IP des paramètres de la session d'instrumentation, de saisir cette valeur dans le champ, puis de déclencher la connexion. Il reste à vérifier que le traitement s'est bien déroulé en consultant la ligne de statu.

```
public void testConnect()
{
        final String ip=getBundle().getString("target");
```

```
        mActivity.runOnUiThread(
            new Runnable()
            {
              public void run()
              {
                mEditIp.setText(ip);
                mConnect.requestFocus();
              }
            }
          );
        sendKeys(KeyEvent.KEYCODE_DPAD_CENTER);
        try { Thread.sleep(1000); } catch (Exception e) {}
        getInstrumentation().waitForIdleSync();
        assertTrue(mStatus.getText().toString().startsWith("Hello "));
}
```

Et voilà.

Avant de démarrer tous cela, il faut bien vérifier que les applications de tests pour le Desktop sont bien installées sur tous les périphériques. Il est possible de les installer par du code exécuté dans le test unitaire Desktop. Vous pouvez paramétrer l'exécution d'un test utilitaire Android pour sélectionner l'instrumentation à utiliser. Attention, le plugin d'Android de génère le package APK que juste avant de l'installer sur le terminal. Il y a donc le risque d'installer une version précédente, présente dans le répertoire **bin** du projet de test.

Nous pouvons alors utiliser Éclipse pour démarrer sur le Desktop notre test unitaire. Celui-ci commence par lancer un test unitaire sur le serveur qui simule l'appui sur le toggle button pour lancer son serveur. Puis, le test lance le test unitaire de connexion pour vérifier s'il est possible, avec un autre Android, de se connecter au premier. Lorsque tous est terminé, le test unitaire est valide, il est possible de passer au suivant.

Objectif atteint

Voilà ce qui achève notre étude des codes d'Android pour résoudre une difficulté, qui à notre connaissance, ne possède pas de solution à ce jour. Avec beaucoup d'astuces, un peu de code, nous avons codé le framework et écrit cet article en trois jours.

Il est ainsi possible de demander à un Android un QRCode pour le présenter à un autre, de demander une trame NFC pour l'injecter dans un autre, d'initier une connexion Bluetooth, Wifi Direct, Internet ou 3G, et de vérifier tous cela dans des tests unitaires.

Tous le code et les exemples sont disponibles ici : **http://code.google.com/p/articles-glmf/**

Philippe PRADOS article@prados.fr
Architect, Smart Mobility, AtoS