Depuración y pruebas

Índice

1. Depuración con Eclipse

1.1. Log y LogCat

Sin duda alguna el sistema más utilizado para depuraciones sencillas es la salida estándar y los logs. En Android podemos imprimir mensajes al log tras importar `android.util.Log`. Los mensajes de Log van clasificados por una etiqueta que les asignamos, aparte de otra información implícita, como la aplicación que los lanza, el PID, el tiempo y el nivel de debug. Se recomienda crear una constante con la etiqueta:

```
private static final String TAG = "MiActivity";
```

Para añadir al log un mensaje de nivel de información utilizaríamos `Log.i()`:

```
Log.i(TAG, "indice=" + i);
```

Según el nivel de depuración utilizaremos una llamada de las siguientes:

- Log.v(): Verbose
- Log.d(): Debug
- Log.i(): Info
- Log.w(): Warning
- Log.e(): Error

Con esta información el Log podrá ser mostrado filtrando los mensajes menos importantes, por ejemplo si establecemos la vista del Log a nivel de Info, los de Debug y los Verbose no se mostrarían, pero sí los de Warning y Error que son más graves que Info.

En Eclipse contamos con la vista LogCat (si no se muestra por defecto se puede añadir) donde podemos realizar dicho filtrado. También podemos realizar filtrado por etiquetas para ver sólo los mensajes que nos interesan. Los mensajes van apareciendo en tiempo real, tanto si estamos con un emulador como si estamos con un dispositivo móvil conectado por USB.

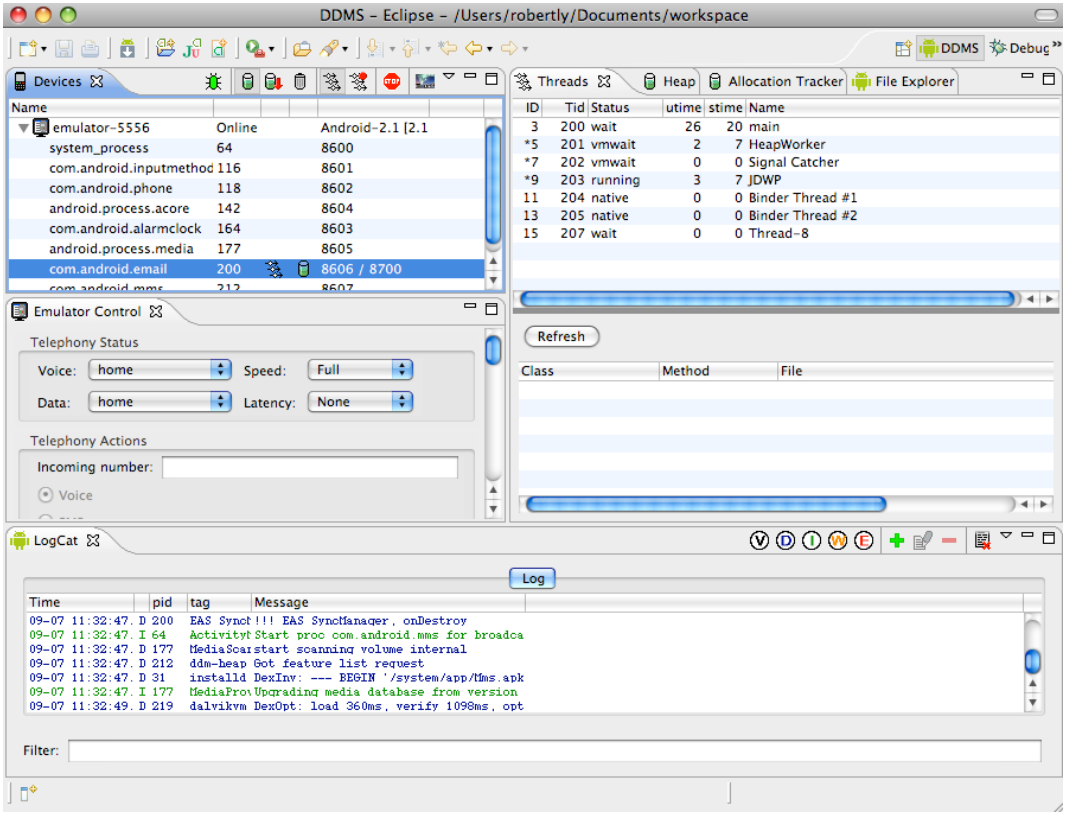
Se recomienda eliminar todas las llamadas a Log cuando se publica un programa en el Android Market, a pesar de que en los dispositivos de los usuarios no se vería ningún log ni salida estándar.

1.2. Dalvik Debug Monitor Server (DDMS)

Android viene con el servidor de depuración DDMS que puede ser ejecutado desde la terminal o desde Eclipse. Desde la terminal del sistema se tendría que ejecutar `./ddms` que se encuentra en la carpeta `tools`. No es necesario si utilizamos Eclipse, que cuenta

con una vista DDMS para interactuar y visualizar resultados de depuración.

En Android cada aplicación se ejecuta en su propia máquina virtual (VM) y cada VM tiene un puerto al que el debugger se conecta. Cuando se inicia DDMS, éste se conecta a `adb`. Cuando conectamos un dispositivo se crea un servicio de monitorización entre `adb` y DDMS, que notifica a DDMS cuando una VM del dispositivo arranca o termina. Cuando una VM arranca, DDMS recoge su PID a través de `adb` y abre una conexión con el debugger de la VM. Aunque cada depurador se puede conectar a un único puerto, DDMS maneja múltiples depuradores conectados.



Vista de DDMS en Eclipse

Como se puede observar en la figura, DDMS nos permite ver los hilos que están en ejecución en dada máquina virtual. También se permiten todas las características de depuración que proporciona Eclipse para Java, como puntos de ruptura, visualización de variables y evaluación de expresiones.

DDMS cuenta con las siguientes funcionalidades y controles:

- Visualización del uso de memoria heap
- Seguimiento de reservas de memoria para objetos
- Trabajar con el sistema de ficheros del emulador o del dispositivo
- Examinar la información de hilos

- Profiling de métodos: seguimiento de medidas tales como número de llamadas, tiempo de ejecución, etc.
- LogCat
- Emulación de operaciones de telefonía y localización
- Cambiar el estado de red y simular red lenta

2. Pruebas unitarias con JUnit para Android

Las pruebas unitarias consisten en probar métodos y partes concretas del código, intentando independizarlas del resto del código lo máximo posible. Los proyectos de Test de Android se construyen sobre JUnit y nos proveen de herramientas que permiten realizar no sólo pruebas unitarias sino también pruebas más amplias.

Cuando creamos un proyecto de Android con el asistente de Eclipse tenemos la opción de incluir casos de prueba en el propio proyecto. Otra manera muy común es la de separar las pruebas en un proyecto aparte. Para ello creamos un nuevo proyecto `Android Test Project` y seleccionamos el proyecto de nuestro workspace que queremos probar.

A continuación creamos casos de prueba en el mismo paquete donde se encuentra el proyecto original, o en subpaquetes de éste. Se crean con `New / JUnit Test Case` pero en lugar de que la clase padre sea la de JUnit, utilizaremos `ActivityInstrumentationTestCase2`, que es la versión actualizada de `ActivityInstrumentationTestCase` cuyo uso ya está desaconsejado. `ActivityInstrumentationTestCase2` se parametriza con la clase de la actividad que vamos a probar. Por ejemplo, si vamos a probar `MainActivity`, tendremos:

```
package es.ua.jtech.av.suma.test;

import android.test.ActivityInstrumentationTestCase2;
import es.ua.jtech.av.suma.MainActivity;

public class MainActivityTest extends
                ActivityInstrumentationTestCase2<MainActivity> {

        public MainActivityTest() {
                super("es.ua.jtech.av.suma", MainActivity.class);
        }

        protected void setUp() throws Exception {
                super.setUp();
        }

        public void test1(){
                // asserts
        }

        public void test2(){
                // asserts
        }

        // ...
```

```
        protected void tearDown() throws Exception {
                super.tearDown();
        }
}
```

En el código anterior se llama al constructor de la clase padre con el paquete `"es.ua.jtech.av.suma"` y la clase `MainActivity` de la actividad a probar.

Con esta clase ya se pueden escribir métodos con pruebas. Sus nombres deben comenzar por "test", por ejemplo `testMiPrueba1()`. Dentro de estos métodos de pruebas podemos utilizar los métodos `assert` de JUnit que provocan el fallo si la aserción resulta falsa. Por ejemplo, `assertEquals(a,b)` compara los dos parámetros y si son distintos la prueba falla. Otro ejemplo es `assertTrue(c)` que comprueba si es verdadero el parámetro que se le pasa.

Con JUnit se pueden probar métodos de las clases del código pero en Android podemos necesitar probar partes de la interfaz gráfica. Para acceder a los views de la actividad lo normal sería declarar referencias a éstos como campos de la clase e inicializarlos en el método `setUp()`. Para obtener las referencias hay que acceder a ellos con `getActivity().findViewById(id))`:

```
        private Button bt;

        protected void setUp() throws Exception {
                super.setUp();

                MainActivity activity = getActivity();
                bt =
(Button)activity.findViewById(es.ua.jtech.av.suma.R.id.button1);
        }
```

Una vez obtenidas las referencias se pueden comprobar los valores que contienen.

```
        assertEquals("32.3", miTextView.getText().toString());
```

Otra necesidad es la de simular eventos de introducción de texto, de selección, de pulsación. Para ello se utiliza la clase `TouchUtils`

```
                TouchUtils.tapView(this, miEditText);
                sendKeys("S");
                sendKeys("i");
                sendKeys("NUMPAD_DOT");
                TouchUtils.clickView(this, bt);
```

Aparte de `ActivityInstrumentationTestCase2` hay otras alternativas que también heredan de las clases de JUnit:

- `AndroidTestCase` que sólo ofrece el contexto local y no el de la aplicación.
- `ServiceTestCase` que se usa para probar servicios.
- `ActivityUnitTestCase` que crea la actividad pero no la conecta al entorno, de manera que se puede utilizar un contexto o aplicación mock.
- `ApplicationTestCase` para probar subclases propias de `Application`.

Se pueden utilizar todas las características de JUnit, tales como crear una Test Suite para agrupar distintos casos de pruebas.

3. Pruebas de regresión con Robotium

"Robotium es como Selenium pero para Android" es el eslogan de este software de pruebas que no está afiliado ni con Android ni con Selenium. Selenium es un software para pruebas de aplicaciones web que permite grabar una secuencia de acciones sobre una página web y después reproducirla. Una batería de pruebas de este tipo permite comprobar que tras realizar cambios en el software, éste sigue comportándose de manera idéntica en su interacción con el usuario.

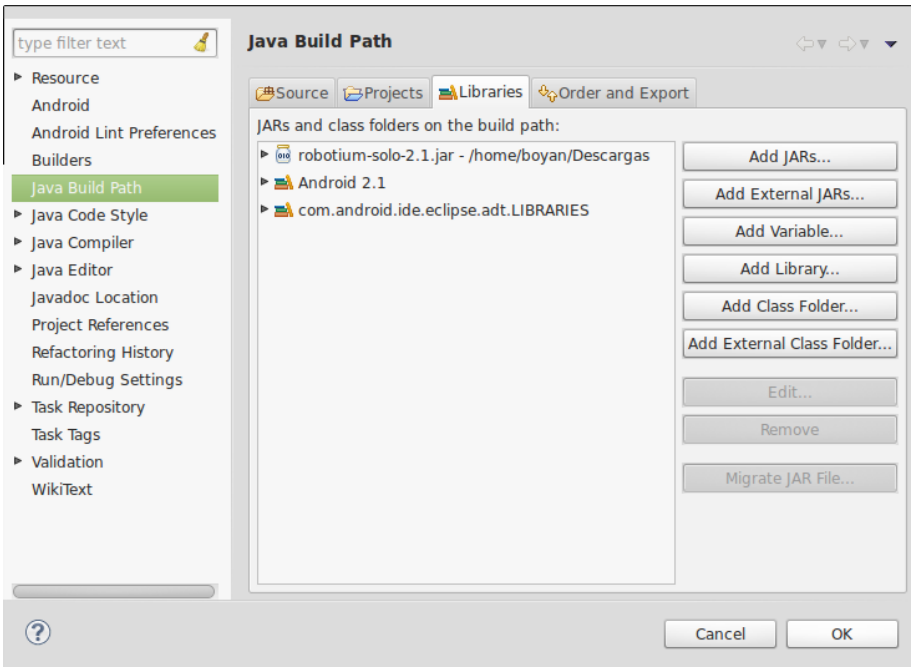
Robotium no permite grabar las acciones del usuario sino que la secuencia de acciones debe ser programada a través de sencillas llamadas a los métodos de Robotium. Da soporte completo para los componentes `Activity`, `Dialog`, `Toast`, `Menu` y `ContextMenu`.

Usar una herramienta como Robotium nos proporciona las siguientes ventajas:

- Desarrollar casos de prueba sin necesidad de conocer el funcionamiento interno de la aplicación probada.
- Automatizar el manejo múltiples actividades de Android.
- Pruebas realistas, al realizarse sobre los componentes GUI en tiempo de ejecución
- Integración con Maven y Ant para ejecutar pruebas como parte de una integración continua.

Para crear un conjunto de pruebas con Robotium debemos crear un nuevo proyecto de test de Android desde Eclipse: `New / Android Test Project`. Es importante crear los casos de prueba dentro del mismo paquete que el proyecto original, así que desde el asistente de creación del proyecto podemos introducir dicho paquete. También puede ser un subpaquete del original.

Añadiremos el JAR de Robotium en el build path del proyecto. Éste puede ser descargado de www.robotium.org y tendrá un nombre similar a `robotium-solo-3.1.jar` (según la versión). Lo añadiremos como jar tras incluírlo dentro del proyecto, en una carpeta `lib/`, por ejemplo.



Incluir Robotium en el Buildpath del proyecto de pruebas.

Una vez creado el proyecto de pruebas debemos crear un nuevo caso de prueba en el paquete correcto: `New / JUnit Test Case` en cuyo asistente pondremos el nombre del test que deseemos y cambiaremos la superclase del test por `ActivityInstrumentationTestCase2<Activity>` en lugar de `junit.framework.TestCase`. La Activity que pasamos como tipo puede ser directamente la clase de la actividad que vayamos a probar, por ejemplo `MainActivity`. Aunque pertenezca a otro proyecto Eclipse nos permite incluirla configurando automáticamente el path para referenciar al otro proyecto.

En el caso de prueba creado declaramos `Solo` como campo privado de la clase y, si no lo están ya, sobrecargamos los métodos `setUp()`, `tearDown` y el constructor.

```
package es.ua.jtech.av.miproyecto.test;

import android.test.ActivityInstrumentationTestCase2;
import es.ua.jtech.av.miproyecto.MainActivity;
import com.jayway.android.robotium.solo.Solo;

public class TestMainActivity extends
ActivityInstrumentationTestCase2<MainActivity> {
        private Solo solo;

        public TestMainActivity(){
                super("es.ua.jtech.av.miproyecto", MainActivity.class);
        }

        @Override
        protected void setUp() throws Exception {
                super.setUp();
```

```
                solo = new Solo(getInstrumentation(), getActivity());
        }

        public void test1(){
                //...
                //assertTrue(comprobacion);
        }

        //...

        @Override
        protected void tearDown() throws Exception {
                solo.finishOpenedActivities();
        }
}
```

Para ejecutar las pruebas programadas hay que hacer click sobre `TestMain` y en el menú contextual seleccionar `Run as / Android JUnit Test`. El emulador arrancará si no está arrancado, la aplicación a probar será instalada y ejecutada, y las acciones programadas en las pruebas serán realizadas sobre la actividad. Los resultados serán reconocidos por las pruebas y mostradas en el resumen de resultados de JUnit.

Un ejemplo de test podría ser el de una sencilla calculadora que cuenta con dos `EditText` y con un botón de sumar. El código de la prueba sería el siguiente:

```
        public void test1(){
                solo.enterText(0,"10");
                solo.enterText(1,"22.4");
                solo.clickOnButton("+");
                assertTrue(solo.searchText("32.4"));
        }
```

Los `EditText` de la actividad se corresponen (por orden) con los parámetros 0 y 1. El botón se corresponde con el que contiene la cadena indicada como etiqueta. Finalmente se busca el resultado en toda la actividad.

Exsiten diferentes métodos con los que `Solo` da soporte a los componentes gráficos. Algunos ejemplos son:

- `getView(id)`
- `getCurrentTextViews(textView)`
- `setActivityOrientation(Solo.LANDSCAPE)`
- `sendKey(Solo.MENU)`
- `clickOnButton(text)`
- `clickOnText(text)`
- `clickOnEditText(text)`
- `clearText(text)`
- `enterText(text)`
- `goBack()`
- `sleep(millis)`

La información sobre todos los métodos está en www.robotium.org.

4. Pruebas de estrés con Monkey

Monkey es un programa para pruebas que simula input aleatorio del usuario. Su propósito es realizar una prueba de estrés de la aplicación para confirmar que, haga lo que haga el usuario con la GUI, la aplicación no tendrá un comportamiento inesperado. El input que realiza Monkey no tiene por qué tener sentido alguno.

A continuación solicitamos 1000 eventos simulados cada 100 milisegundos obteniendo la lista de ellos (opción -v) y afectará a las aplicaciones del paquete `es.ua.jtech.av`.

```
adb shell monkey -p es.ua.jtech.av -v --throttle 100 1000
```

Monkey simulará eventos de teclado, tanto qwerty como teclas hardware especializadas, movimientos de trackball, apertura y cierre del teclado, rotaciones de la pantalla.

Si deseamos poder reproducir una serie de acciones aleatorias, podemos fijar la semilla de inicialización aleatoria con una propia, por medio de la opción `-s`. Así cada vez que ejecutemos con esta semilla, la secuencia de eventos aleatorios será la misma y podremos reproducir un problema tantas veces como haga falta hasta dar con la solución.