

Diseño de sistemas informáticos

Capa de red en Android

Santiago Munín

Universidade da Coruña

Mayo, 2013

Motivación

- Gran parte de las aplicaciones móviles de éxito emplean una arquitectura Cliente/Servidor.
- Estas arquitecturas utilizan la red como medio para comunicarse.
- **Ejemplos:** Twitter, Facebook, 4square, Evernote...

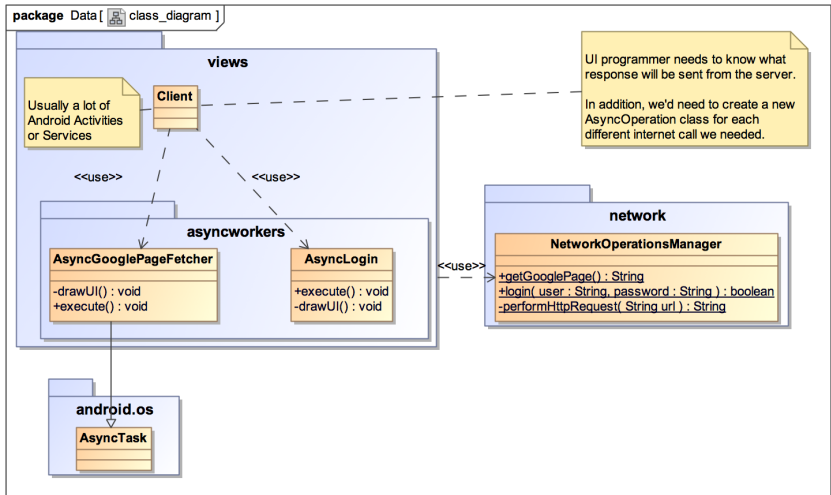
Arquitectura Cliente/Servidor

- La comunicación suele realizarse mediante HTTP (o HTTPS) contra servidores (habitualmente REST).
- Se emplean tecnologías como JSON o XML para transmitir información.
- Esta comunicación es potencialmente costosa → Si se realiza en el thread de la interfaz (thread principal) puede bloquear demasiado tiempo el teléfono y afectar a la experiencia del usuario.
- La solución a este problema es sencilla: ejecutar en segundo plano la petición y actualizar la interfaz una vez recibida la respuesta (mientras tanto, permitir al usuario interactuar con la aplicación o mostrar un diálogo de carga).

AsyncTask

- Android nos ofrece una clase que encapsula el manejo de threads.
- AsyncTask nos permite ejecutar una tarea en segundo plano y usar el resultado de la misma en primer plano (una vez haya acabado) sin bloquear la interfaz.
- **Más información:** <http://developer.android.com/reference/android/os/AsyncTask.html>

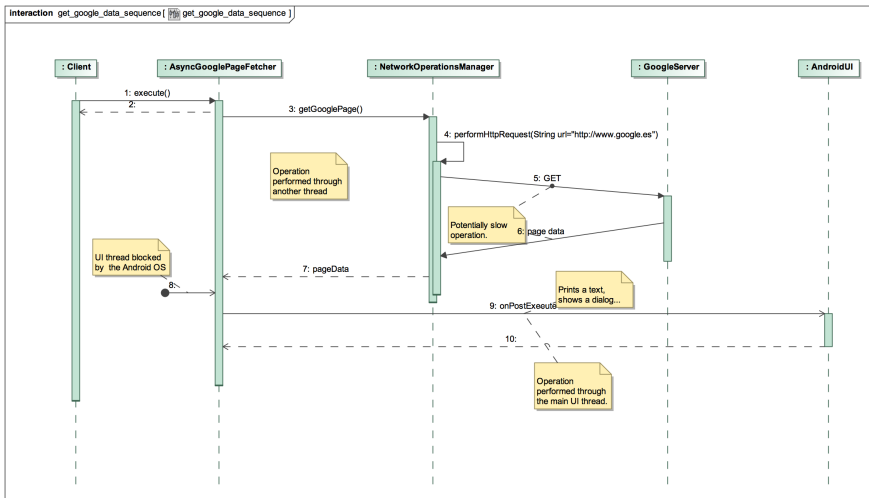
Diagrama de clases



Explicación

- El cliente (habitualmente "Activities" Android) utiliza clases que heredan de "AsyncTask" para no bloquear la interfaz.
- La clase "NetworkOperationsManager" contiene la lógica relacionada con las llamadas en red (direcciones, parámetros...). Podría ser responsable del cacheo de respuestas.

Diagrama de secuencia



Explicación

- ❶ **En el thread principal:** Cuando se llama a “execute” sobre la “AsyncTask” se devuelve el control inmediatamente.
- ❷ **Thread secundario:** Mientras tanto, se realiza la petición al servidor y se analiza la respuesta.
- ❸ **Thread principal:** Una vez acabado el trabajo en segundo plano, se vuelve a bloquear el thread principal y se realizan las acciones que se hayan indicado (por ejemplo, mostrar un diálogo con el resultado de la respuesta).
- **Nota:** Mientras el thread secundario realiza la petición, el thread principal puede estar haciendo cualquier cosa (muchas veces se muestra un diálogo de carga, pero no siempre).

Conclusiones

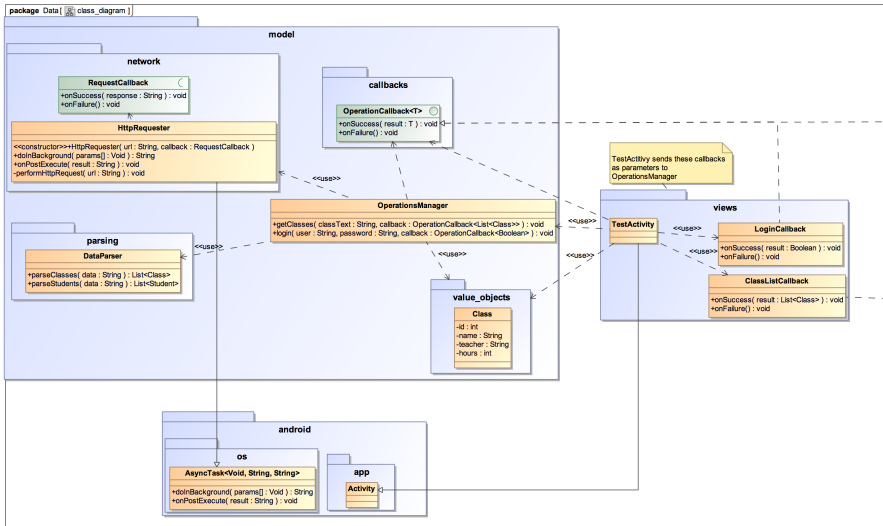
- Cada vez que se necesita hacer una petición (nueva o ya definida desde otra activity) y actualizar la interfaz, se define una clase que hereda de AsyncTask.
- Es necesario definir la petición a realizar (clase "NetworkOperationsManager"), el procesamiento de la respuesta y la actualización de la interfaz. Demasiadas responsabilidades.
- Hay demasiado acoplamiento (se viola el SRP) → resulta dificultoso dividir la programación de la aplicación entre interfaz y bajo nivel (operaciones de red y cachés).

¿Qué es un callback?

- Un callback es código ejecutable que se pasa como un argumento a otro código y se espera que sea llamado en un determinado momento.
- Los callbacks pueden ser síncronos o asíncronos según se ejecuten de forma inmediata o no.
- Los callbacks asíncronos exigen la existencia de más de un thread.
- Pueden considerarse una forma de patrón Observador donde se le dice a una función que, cuando realice su tarea, ejecute la reacción de la entidad que la llama.

Ejemplo de callback: Lectura de un fichero en Node.js

```
fs = require('fs');  
fs.readFile('path', 'utf8', function(err, data) {  
    if (err) {  
        return console.log(err);  
    }  
    console.log(data);  
});
```

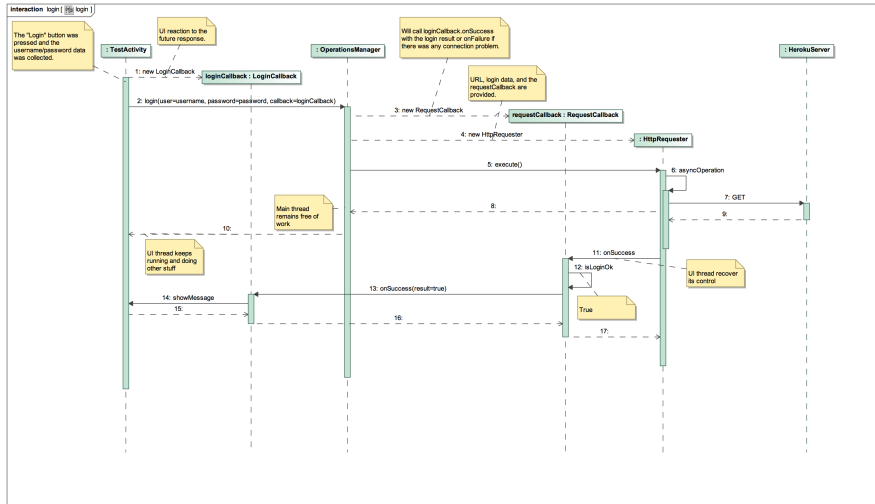


Explicación (capa interna del modelo)

- **Network:** Este es el paquete más interno, define la acción básica de realizar una consulta HTTP y hacer "algo" con la respuesta. Ese "algo" se define mediante el "RequestCallback".
- **Parsing:** Funcionalidad relativa al "parseo" de las respuestas.

Explicación (fachada del modelo)

- **OperationsManager:** Punto de entrada al subsistema, ofrece operaciones de red y requiere callbacks para saber qué hacer con las respuestas.
- **Callbacks:** Distintos tipos de callbacks que puedan requerir las operaciones del "OperationsManager".
- **Value_objects:** Aquí van las clases que representan objetos (sin funcionalidad, equivalentes a las "structs" de C).



Explicación

- 1 Se crea un `LoginCallback` (esta clase implementa la interfaz `OperationCallback` y `Boolean`)
- 2 Se llama a `OperationsManager.getInstance().login(user, password, LoginCallback)`
- 3 `OperationsManager` crea un `RequestCallback` y se lo manda al `HttpRequester` junto con la dirección y parámetros de la petición.
- 4 Aquí empieza la operación en segundo plano, mientras se devuelve el control de la interfaz.
- 5 Cuando la operación termina se ejecuta el código de `RequestCallback`, que a su vez llama a `LoginCallback` y realiza las operaciones pertinentes.

Conclusiones

- Esta aproximación se basa en tres conceptos: arquitectura MVC, patrón Observador (callbacks) y patrón Fachada.
- **Ventajas:**
 - Las responsabilidades están claramente separadas.
 - El cliente del módulo de red solo necesita conocer la fachada.
 - Esta aproximación permite dividir la programación de interfaz y red completamente. Conectarlas es inmediato.
- **Limitaciones:**
 - Hay cierta repetición de código al no poder generalizar métodos como "onFailure()".
 - La alternativa es utilizar una clase abstracta (pero no podríamos beneficiarnos de implementar la interfaz directamente en una "Activity" si fuese lo suficientemente sencilla).

Extra

- El módulo más interno es común a un sinfín de aplicaciones Android → Convertirlo en una librería.
- Alguien ha tenido la idea antes y ha hecho un muy buen trabajo: <http://loopj.com/android-async-http/>
- Este mismo diseño puede ser aplicado a cualquier otra aplicación Java (u otra plataforma) siempre y cuando se implemente un mecanismo similar al AsyncTask (muchas plataformas no requieren actualizar la interfaz desde el thread principal).