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 aquitectura Cliente/Servidor.
- Estas arquitecturas utilizan la red como medio para comunicarse.
- **Ejemplos**: Twitter, Facebook, 4square, Evernote...

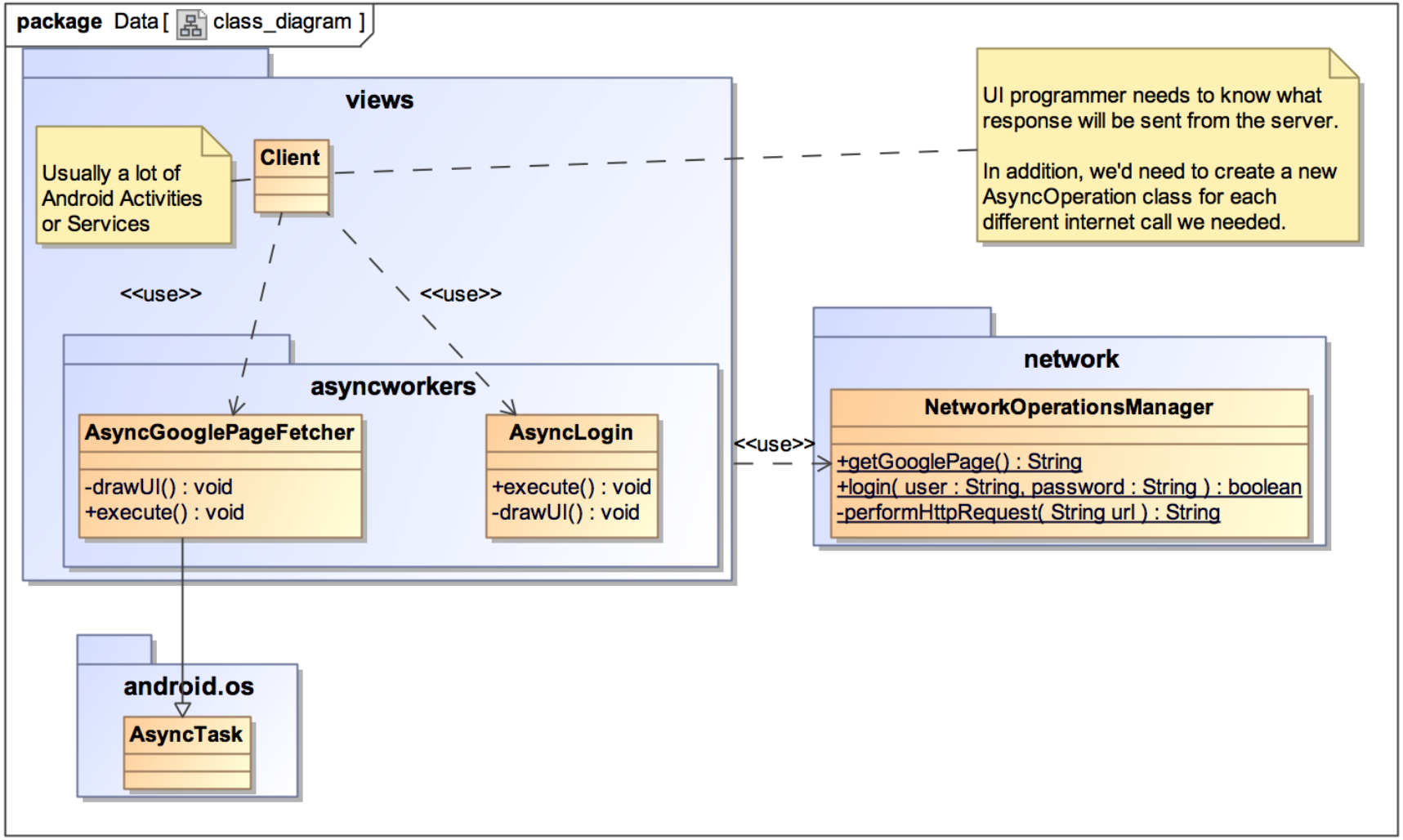
Arquitectura Cliente/Servidor

- 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 $\rightarrow$ 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 recibiba 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`

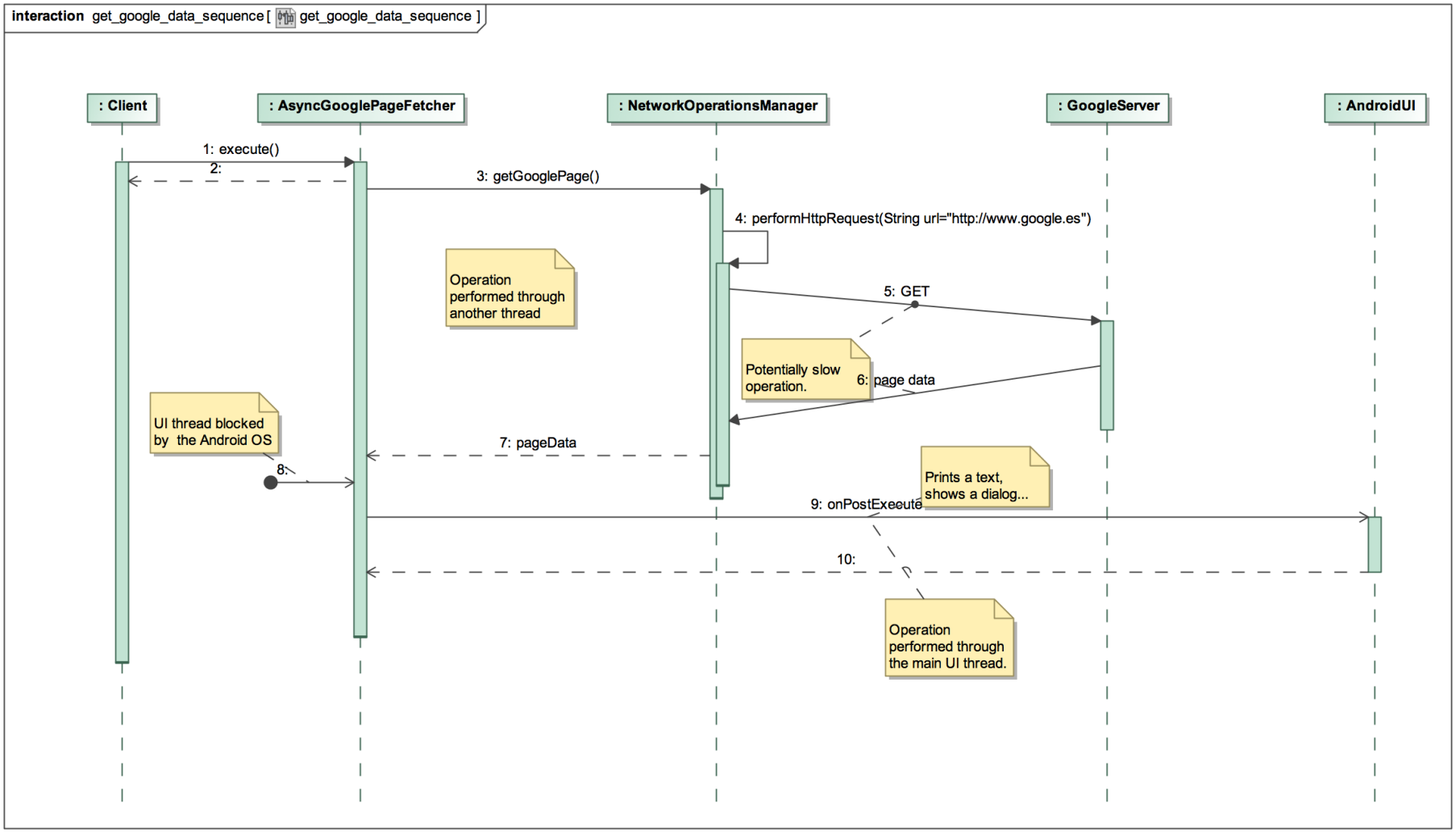
Diagrama de clases



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

1. **En el thread principal**: Cuando se llama a "execute" sobre la "AsyncTask" se devuelve el control inmediatamente.
2. **Thread secundario**: Mientras tanto, se realiza la petición al servidor y se analiza la respuesta.
3. **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

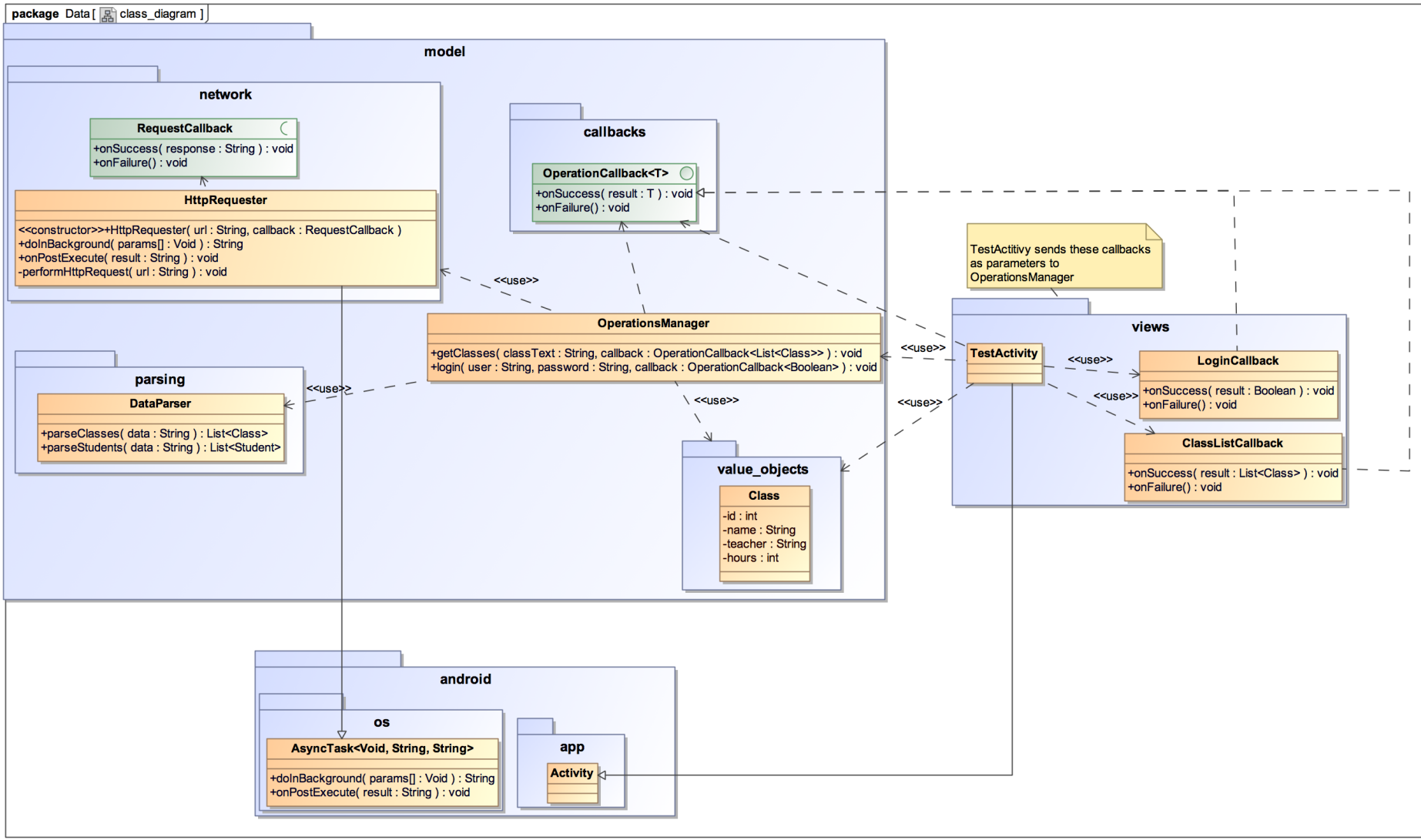
- Cada vez que se necesita hacer una petición (nueva o ya definida desde otra activity) y actualizar la interfaz, se define una clase que hereda de AsyncTask.
- Es necesario definir la petición a realizar (clase "NetworkOperationsManager"), el procesamiento de la respuesta y la actualización de la interfaz. Demasiadas responsabilidades.
- Hay demasiado acoplamiento (se viola el SRP) $\rightarrow$ 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 realize 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);
});
```

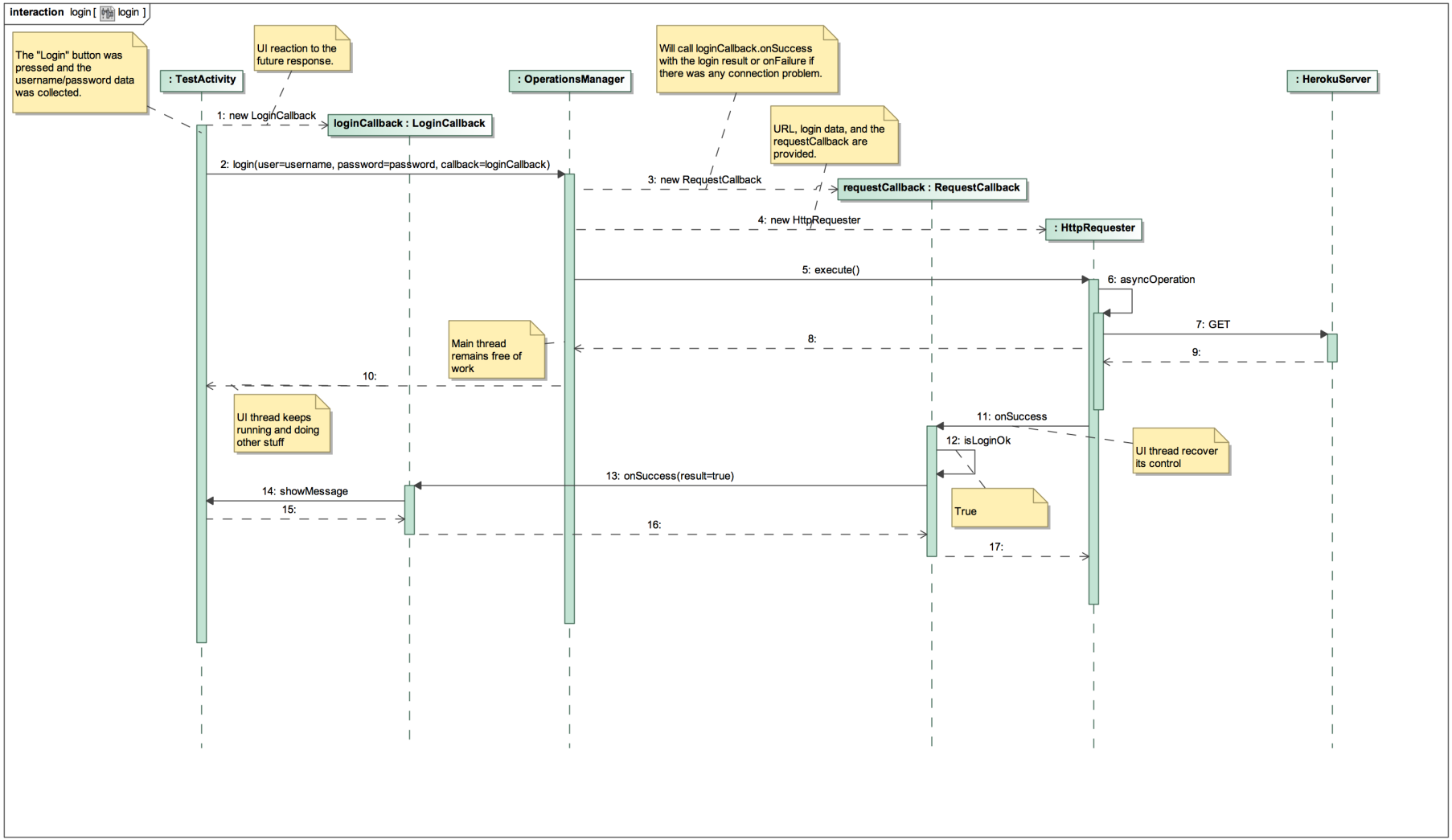

package Data [class_diagram]
model
network
RequestCallback
+onSuccess(response : String) : void
+onFailure() : void
HttpRequester
<<constructor>>+HttpRequester(url : String, callback : RequestCallback)
+doInBackground(params[] : Void) : String
+onPostExecute(result : String) : void
-performHttpRequest(url : String) : void
callbacks
OperationCallback<T>
+onSuccess(result : T) : void
+onFailure() : void
<<use>>
OperationsManager
+getClasses(classText : String, callback : OperationCallback<List<Class>>) : void
+login(user : String, password : String, callback : OperationCallback<Boolean>) : void
parsing
DataParser
+parseClasses(data : String) : List<Class>
+parseStudents(data : String) : List<Student>
value_objects
Class
-id : int
-name : String
-teacher : String
-hours : int
TestActitivy sends these callbacks as parameters to OperationsManager
views
TestActivity
LoginCallback
+onSuccess(result : Boolean) : void
+onFailure() : void
ClassListCallback
+onSuccess(result : List<Class>) : void
+onFailure() : void
android
os
AsyncTask<Void, String, String>
+doInBackground(params[] : Void) : String
+onPostExecute(result : String) : void
app
Activity

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).


interaction login [login]
The "Login" button was pressed and the username/password data was collected.
UI reaction to the future response.
: TestActivity
: OperationsManager
Will call loginCallback.onSuccess with the login result or onFailure if there was any connection problem.
: HerokuServer
1: new LoginCallback
loginCallback : LoginCallback
URL, login data, and the requestCallback are provided.
2: login(user=username, password=password, callback=loginCallback)
3: new RequestCallback
requestCallback : RequestCallback
4: new HttpRequester
: HttpRequester
5: execute()
6: asyncOperation
7: GET
8:
9:
Main thread remains free of work
10:
UI thread keeps running and doing other stuff
11: onSuccess
12: isLoginOk
UI thread recover its control
13: onSuccess(result=true)
14: showMessage
15:
True
16:
17:

Explicación

1. Se crea un LoginCallback (esta clase implementa la interfaz OperationCallback¡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 podriamos benificiarnos de implementar la interfaz directamente en una "Activity" si fuese lo suficientemente sencilla).

Comentarios

- El módulo más interno es común a un sinfín de aplicaciones Android $\rightarrow$ 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).

Repositorio

- Diseño e implementación publicados en GitHub.
- https://github.com/SantiMunin/android-observer-pattern-example