Android
Network layer design

Santiago Munín

Universidade da Coruña

May, 2013

Motivation

- A huge number of successful mobile applications use an Client/Server architecture.
- These architectures use the network as a tool to communicate.
- **Examples**: Twitter, Facebook, 4square, Evernote...

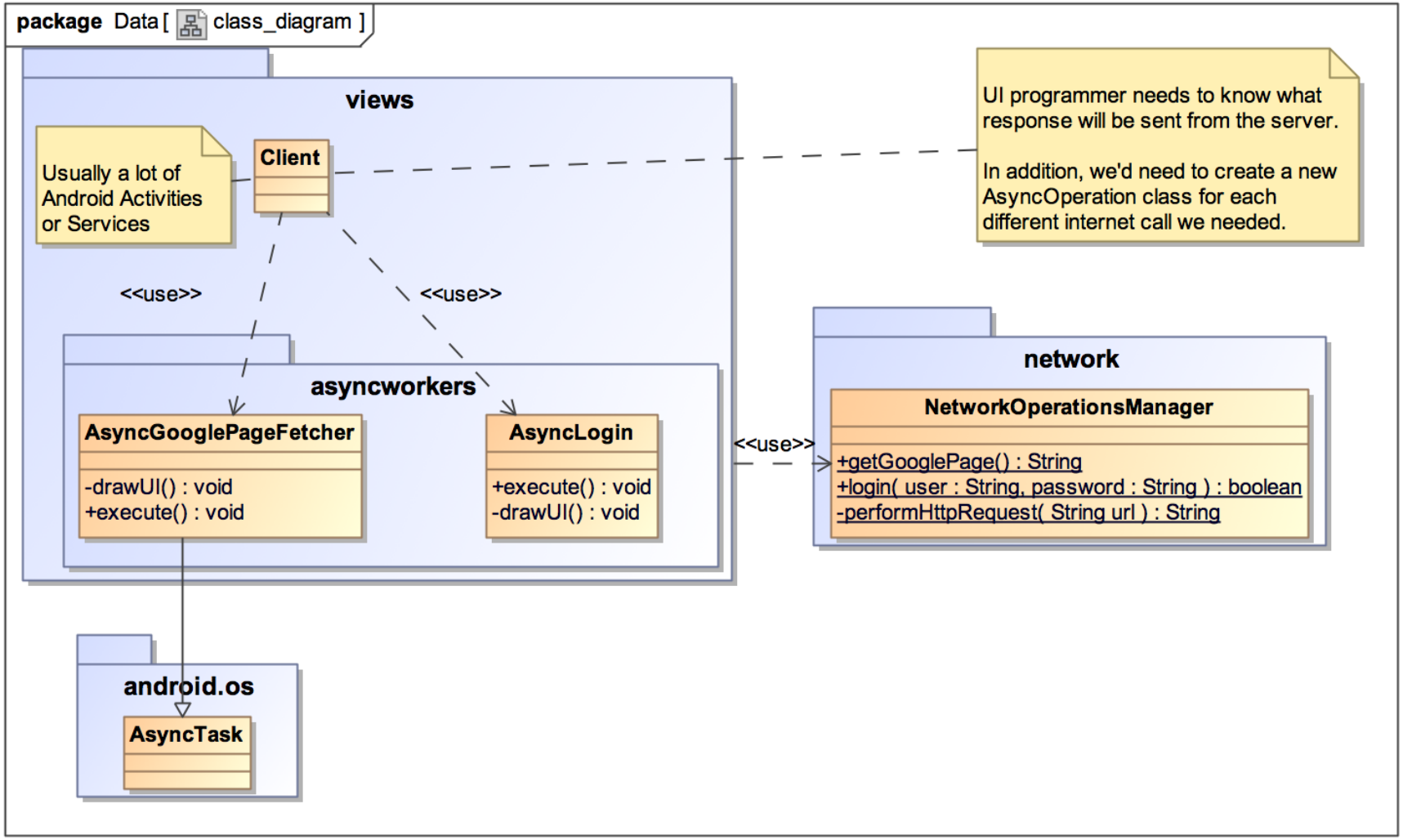
Client/Server architecture

- Communications are usually performed through HTTP (or HTTPS) with servers (usually REST servers).
- Markup languages (such as JSON or XML) are used in order to exchange information between nodes.
- These are costly communications→ They have to be done through another thread or they would block the UI (decreasing the user experience).
- Simple solution: the network operation should be excuted in background and use the UI thread only after getting the response without blocking it (a loading dialog could be showed meanwhile).

AsyncTask

- Android provides us with a class which wraps the thread handling.
- AsyncTask allows us to execute a task in background and use the result in foreground without blocking the UI.
- **More information**: `http://developer.android.com/reference/android/os/AsyncTask.html`

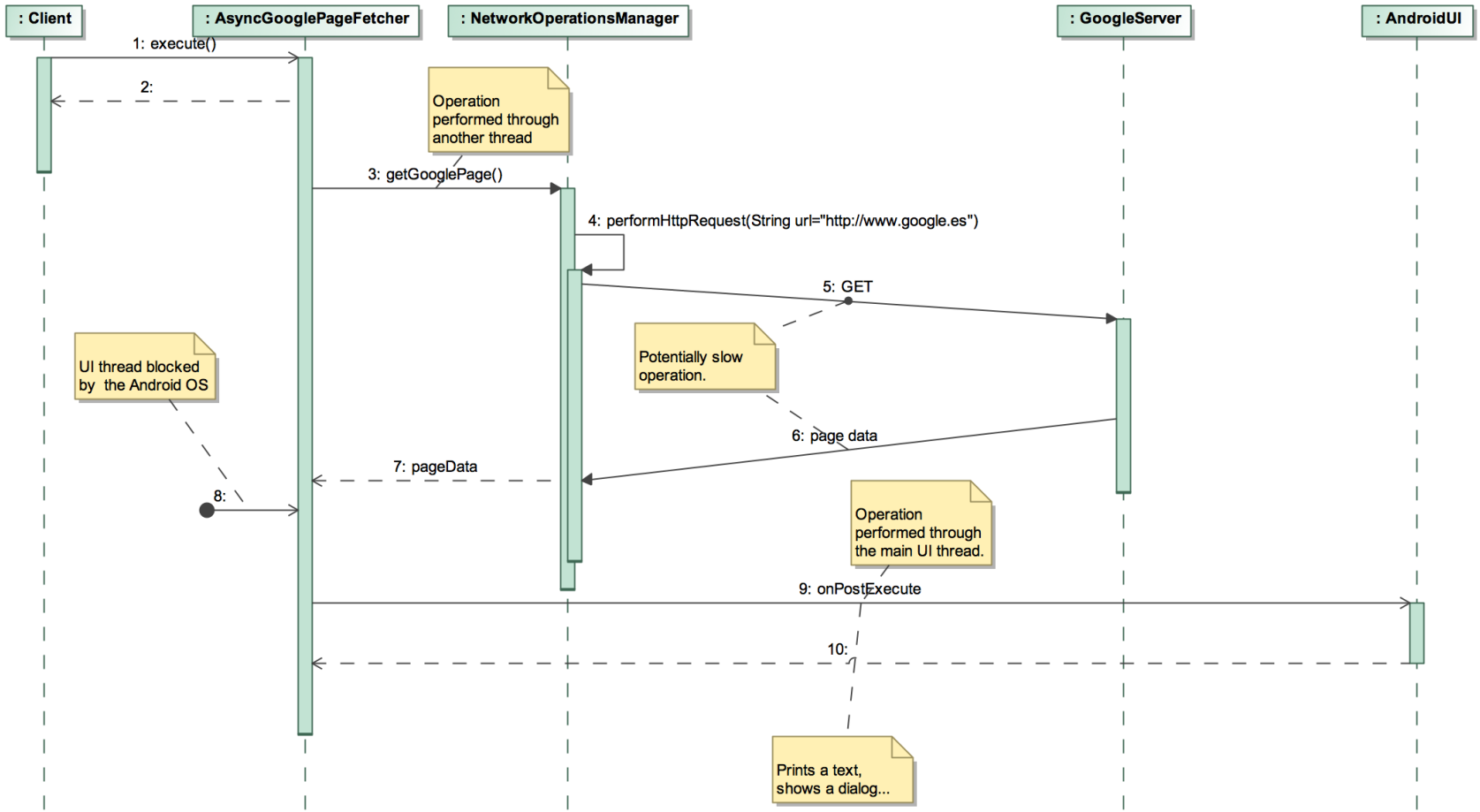
Class diagram



Explanation

- The client (usually "Activities" Android) uses classes which inherit from "AsyncTask" in order to not block the UI.
- The "NetworkOperationsManager" class contains all the logic relative to the network calls (url, parameters). It may be responsible of the response parsing and caching.

Sequence diagram



Explanation

1. **Main thread**: Once "execute" is called it returns the control inmediately.
2. **Secondary thread**: Meanwhile, this thread does the http request and analizes the response.
3. **Main thread**: Once the background job is finished, the main thread is blocked again and the UI actions are performed (for example, show a dialog or update a textview).

- **Note**: While the secondary thread performs the request, the main thread is able to execute any instructions (is usual to show a loading dialog).

Conclusions

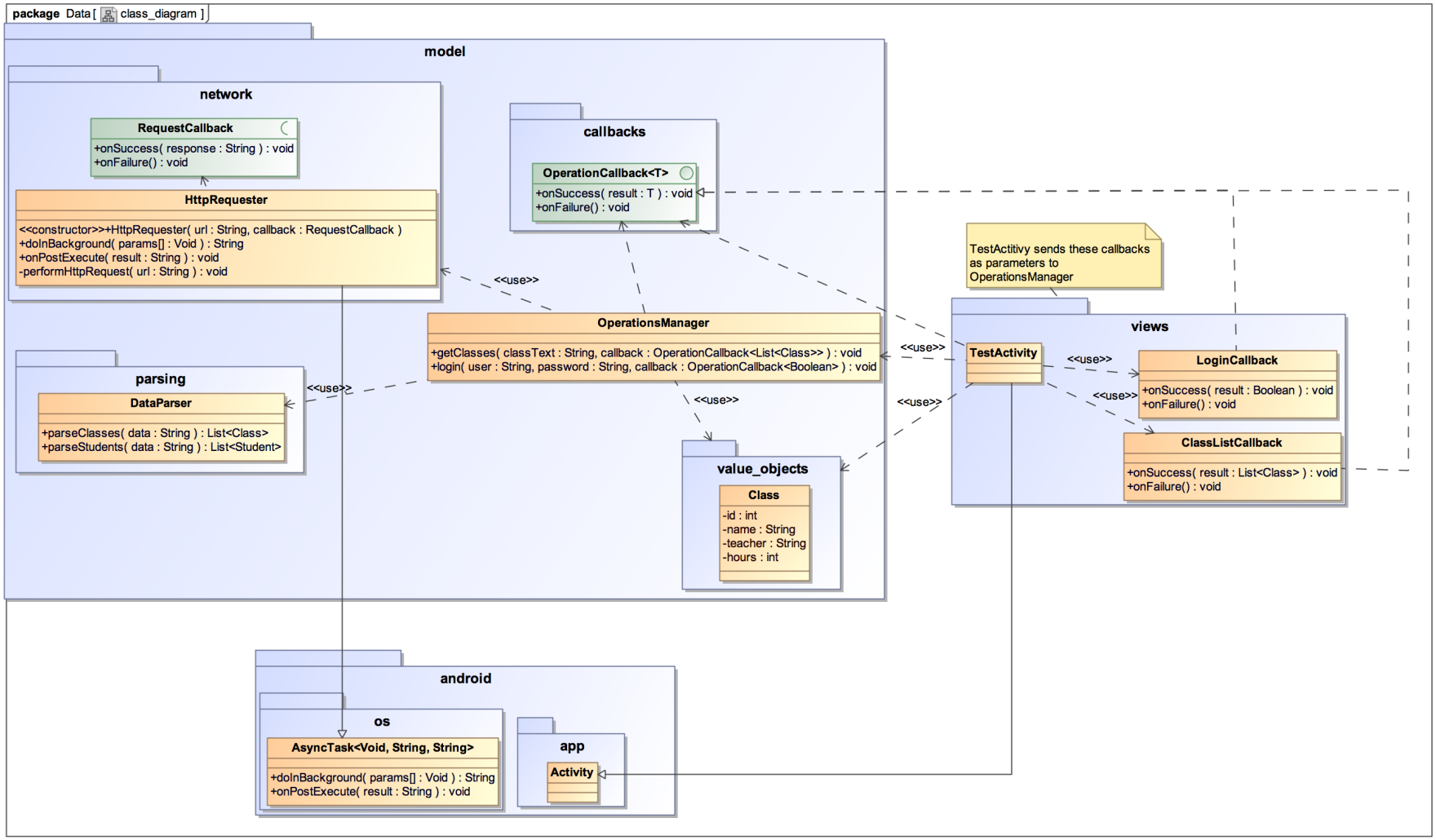
- Whenever you need to make a request (a new or an already defined one but from another activity) and update the UI, you have to define a class that inherits from AsyncTask.
- It's necessary to define the request to be performed ("NetworkOperationsManager" class), the processing of the response and the UI update. Too many responsibilities.
- There is too much coupling (it violates the SRP) $\rightarrow$ It is difficult to distribute the programming between UI and low-level (network operations and caches).

What is a callback?

- A callback is executable code that is passed as an argument to another code and is expected to be called at a given time.
- There are two types of callbacks: blocking callbacks (also known as synchronous callbacks or just callbacks) and deferred callbacks (also known as asynchronous callbacks).
- Deferred callbacks imply the existence of multiple threads.
- They are a sort of “Observer” pattern.

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);
});
```

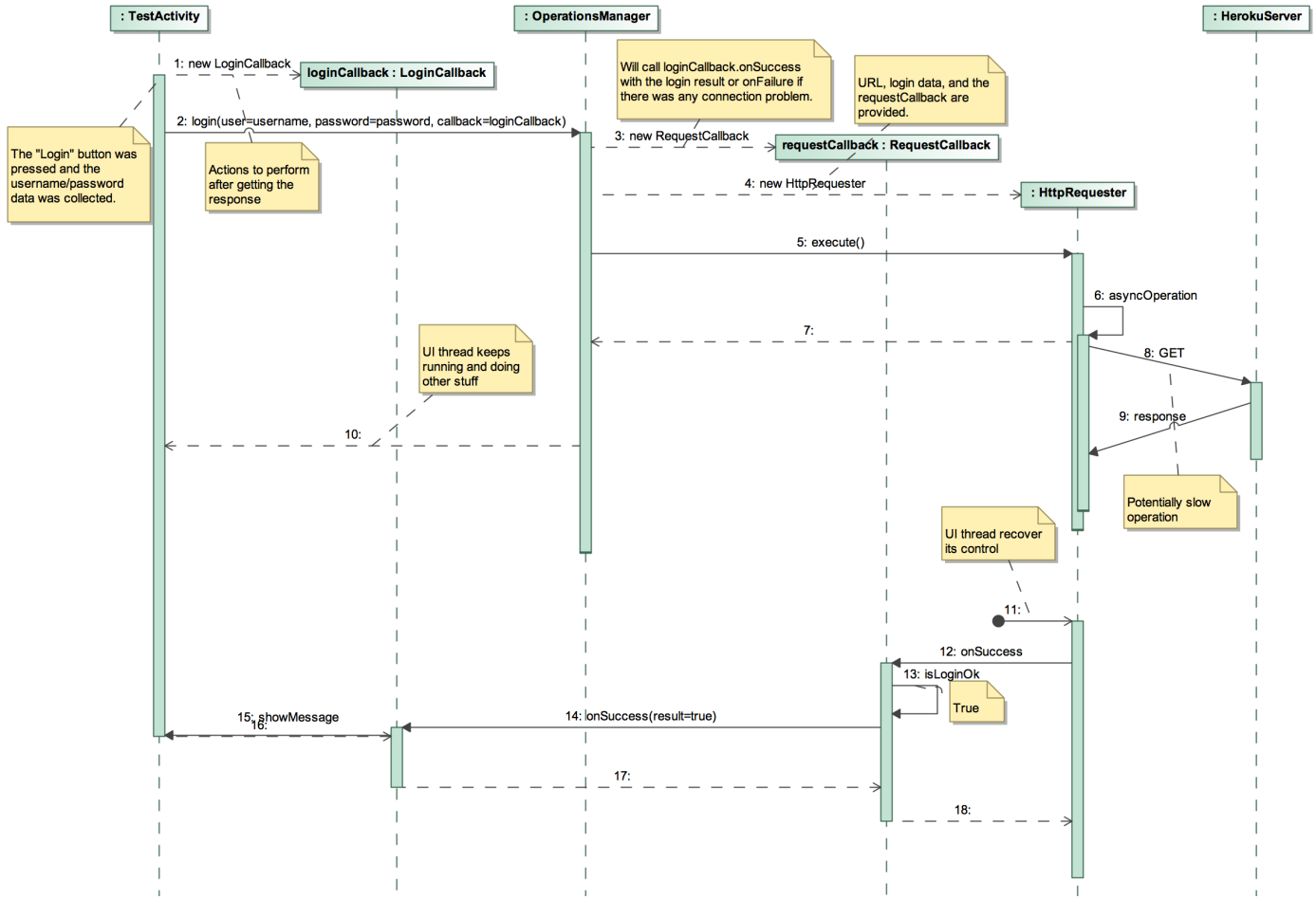

package Data [class_diagram]
model
network
RequestCallback
+onSuccess(response : String) : void
+onFailure() : void
HttpRequester
<<constructor>>+HttpRequester(url : String, callback : RequestCallback)
+doInBackground(params[] : Void) : String
+onPostExecute(result : String) : void
-performHttpRequest(url : 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

Explanation (model inner layer)

- **Network**: This is the innermost package. It defines the basic action of making an HTTP request and "doing something" with the response. That "something" is defined by the "requestCallback".
- **Parsing**: Response parsing methods.

Explanation (model facade)

- **OperationsManager**: Entry point into the subsystem, provides network operations and requires callbacks for what to do with the answers.
- **Callbacks**: Different types of callbacks that may be required by the Operations Manager.
- **Value_objects**: Classes that represent objects (equivalents to the "C" structs).


: TestActivity
: OperationsManager
: HerokuServer
1: new LoginCallback
loginCallback : LoginCallback
Will call loginCallback.onSuccess with the login result or onFailure if there was any connection problem.
URL, login data, and the requestCallback are provided.
2: login(user=username, password=password, callback=loginCallback)
3: new RequestCallback
requestCallback : RequestCallback
The "Login" button was pressed and the username/password data was collected.
Actions to perform after getting the response
4: new HttpRequester
: HttpRequester
5: execute()
6: asyncOperation
7:
UI thread keeps running and doing other stuff
8: GET
9: response
10:
Potentially slow operation
UI thread recover its control
11:
12: onSuccess
13: isLoginOk
True
15: showMessage
16:
14: onSuccess(result=true)
17:
18:

Explanation

1. It creates a LoginCallback (this class implements OperationCallback¡Boolean¿).
2. OperationsManager.getInstance().login(user, password, LoginCallback) is called.
3. Operations Manager creates a requestCallback and sends it to HttpRequester along with the address and request parameters.
4. Here begins the operation in the background. The UI thread is released.
5. When the operation is finished, it executes the requestCallback code.

Conclusions

- This approach is based on three concepts: MVC architecture, Observer pattern (callbacks) and Facade pattern.
- **Advantages**:
  - All the responsibilities are clearly separated.
  - The client only needs to know the facade of the network module.
  - his approach allows to divide the network and UI programming completely. Connecting them is straightforward.
- **Hándicaps**:
  - There is some repetition of code which cannot be generalized in the “onFailure” methods.
  - The alternative is to use an abstract class (but we could not implement the interface directly into a “ Activity” if it were simple enough).

Extra

- The innermost module is common to many Android apps *rightarrow* Turn it into a library.
- Someone had the idea before and has done a very good job: `http://loopj.com/android-async-http/`
- This same design can be applied to any other Java application (or other platform) as long as it implements a mechanism similar to AsyncTask (many platforms do not need to update the interface from the main thread).