

Android pour les développeurs J2EE : un modèle asynchrone pour les clients web

serge.tahe at istia.univ-angers.fr
mars 2013

Table des matières

1 INTRODUCTION.....	5
 1.1 CONTENU DU DOCUMENT.....	5
 1.2 LES OUTILS UTILISÉS.....	8
 1.3 LES CODES SOURCE.....	9
 1.4 PRÉ-REQUIS.....	9
2 LE MODÈLE AVAT.....	10
 2.1 LE PROJET ECLIPSE.....	10
 2.2 LE COEUR DU MODÈLE.....	11
 2.3 L'IMPLÉMENTATION ANDROID DU MODÈLE AVAT.....	17
 2.4 ÉLÉMENTS D'IMPLÉMENTATION.....	26
 2.4.1 L'ACTIVITÉ ANDROID.....	26
 2.4.2 LA VUE.....	27
 2.4.3 L'ACTION.....	29
 2.4.4 LA TÂCHE ASYNCHRONE.....	31
3 AVAT- EXEMPLE 1.....	33
 3.1 LE PROJET.....	33
 3.2 L'ARCHITECTURE.....	34
 3.3 LE PROJET ECLIPSE.....	35
 3.4 LE MANIFESTE DE L'APPLICATION ANDROID.....	35
 3.5 LES DÉPENDANCES MAVEN.....	36
 3.6 L'ACTIVITÉ ANDROID.....	37
 3.7 LA FABRIQUE D'OBJETS.....	39
 3.8 LA CLASSE D'EXCEPTION.....	40
 3.9 LA VUE [VUE_01].....	41
 3.9.1 MÉTHODE [DOEXÉCUTER].....	45
 3.9.2 MÉTHODE D'ANNULATION DES TÂCHES.....	47
 3.10 L'ACTION [ACTION_01].....	47
 3.11 LA TÂCHE ASYNCHRONE [TASK_01].....	50
 3.12 LES TESTS.....	52
4 AVAT- EXEMPLE 2.....	54
 4.1 LE PROJET.....	54
 4.2 LE MANIFESTE DE L'APPLICATION ANDROID.....	54
 4.3 LE SERVEUR.....	55
 4.3.1 LA COUCHE [MÉTIER].....	55
 4.3.1.1 Le projet Eclipse.....	55
 4.3.1.2 Les dépendances Maven.....	56
 4.3.1.3 L'interface [IMetier].....	56
 4.3.1.4 L'implémentation [Metier].....	56
 4.3.1.5 La classe d'exception.....	57
 4.3.2 LE SERVICE REST.....	58
 4.3.2.1 Le projet Eclipse.....	58
 4.3.2.2 Les dépendances Maven.....	58
 4.3.2.3 Configuration du service REST.....	60
 4.3.2.4 SpringMVC.....	61
 4.3.2.5 Le serveur REST.....	64
 4.3.2.6 Déploiement et test du service REST.....	65
 4.4 LE CLIENT ANDROID.....	66
 4.4.1 LA COUCHE [METIER].....	67
 4.4.1.1 Le projet Eclipse.....	67
 4.4.1.2 Les dépendances Maven.....	67
 4.4.1.3 Implémentation de la couche [métier].....	68
 4.4.2 LA COUCHE [DAO].....	69
 4.4.2.1 Le projet Eclipse.....	70
 4.4.2.2 Les dépendances Maven.....	70
 4.4.2.3 Implémentation de la couche [DAO].....	71
 4.4.3 LA COUCHE [AVAT].....	74
 4.4.3.1 Le projet Eclipse.....	74
 4.4.3.2 Les dépendances Maven.....	75
 4.4.3.3 La tâche [Task_01].....	75
 4.4.3.4 La vue [Vue_01].....	77
 4.4.3.5 La configuration de l'application.....	78

4.4.3.6 La fabrique d'objets.....	79
5 AVAT- EXEMPLE 3.....	81
 5.1 LE PROJET.....	81
 5.2 LE PROJET ECLIPSE.....	82
 5.3 L'ACTION [ACTION_01].....	82
 5.4 LA VUE [VUE_01].....	83
6 AVAT- EXEMPLE 4.....	85
 6.1 LE PROJET.....	85
 6.2 LE PROJET ECLIPSE.....	86
 6.3 LA SESSION.....	86
 6.4 L'ACTIVITÉ ANDROID.....	87
 6.5 L'ACTION [ACTION_01].....	88
 6.6 LA VUE [VUE_01].....	88
 6.7 LA VUE [VUE_02].....	89
 6.8 LA FABRIQUE.....	90
7 AVAT- EXEMPLE 5.....	93
 7.1 LE PROJET.....	93
 7.2 LE PROJET ECLIPSE.....	93
 7.3 LA VUE [VUE_01].....	93
 7.4 LA VUE [VUE_02].....	94
8 AVAT- EXEMPLE 6.....	96
 8.1 LE PROJET.....	96
9 AVAT- EXEMPLE 7.....	99
 9.1 LE PROJET.....	99
 9.2 L'ARCHITECTURE DU PROJET.....	104
 9.3 LA BASE DE DONNÉES.....	105
 9.3.1 LA TABLE [MEDECINS].....	105
 9.3.2 LA TABLE [CLIENTS].....	105
 9.3.3 LA TABLE [CRENEAUX].....	106
 9.3.4 LA TABLE [RV].....	106
 9.3.5 GÉNÉRATION DE LA BASE.....	107
 9.4 LES PROJETS ECLIPSE DE L'APPLICATION.....	109
 9.5 LA COUCHE [MÉTIER] DU SERVEUR.....	109
 9.6 LE SERVEUR REST.....	114
 9.6.1 LE PROJET ECLIPSE.....	114
 9.6.2 LES DÉPENDANCES MAVEN.....	115
 9.6.3 CONFIGURATION DE LA COUCHE [REST].....	116
 9.6.3.1 Le fichier [web.xml].....	116
 9.6.3.2 Le fichier [rest-services-config.xml].....	117
 9.6.4 IMPLÉMENTATION DE LA COUCHE [REST].....	119
 9.6.4.1 Liste des clients.....	121
 9.6.4.2 Liste des médecins.....	122
 9.6.4.3 Liste des créneaux d'un médecin.....	122
 9.6.4.4 Liste des rendez-vous d'un médecin.....	123
 9.6.4.5 Obtenir un client identifié par son n°.....	124
 9.6.4.6 Obtenir un médecin identifié par son n°.....	124
 9.6.4.7 Obtenir un rendez-vous identifié par son n°.....	125
 9.6.4.8 Obtenir un créneau horaire identifié par son n°.....	125
 9.6.4.9 Ajouter un rendez-vous.....	126
 9.6.4.10 Supprimer un rendez-vous.....	126
 9.6.4.11 Obtenir l'agenda d'un médecin.....	127
 9.7 LA COUCHE [DAO] DU CLIENT ANDROID.....	128
 9.7.1 LE PROJET ECLIPSE.....	128
 9.8 LA COUCHE [MÉTIER] DU CLIENT ANDROID.....	129
 9.8.1 LE PROJET ECLIPSE.....	129
 9.8.2 IMPLÉMENTATION.....	129
 9.9 LA COUCHE [AVT].....	132
 9.9.1 LE PROJET ECLIPSE.....	133
 9.9.2 L'ACTIVITÉ ANDROID.....	133
 9.9.3 LA CLASSE DE CONFIGURATION.....	134
 9.9.4 LE PATRON DES VUES.....	135
 9.9.5 LA VUE [CONFIG].....	137
 9.9.6 LA VUE [LOCALVUE].....	141

<u>9.9.7</u> LA FABRIQUE.....	146
<u>9.9.8</u> LA TÂCHE [GETALLCLIENTSTASK].....	148
<u>9.9.9</u> LA TÂCHE [GETALLMEDECINSTASK].....	149
<u>9.9.10</u> LA VUE [ACCUEILVUE].....	149
<u>9.9.11</u> LA TÂCHE [GETAGENDAMEDECINTASK].....	152
<u>9.9.12</u> LA VUE [AGENDAVUE] - 1.....	152
<u>9.9.13</u> L'ADAPTATEUR [LISTCRENEAUXADAPTER].....	154
<u>9.9.14</u> LA VUE [AGENDAVUE] - 2.....	158
<u>9.9.15</u> LA TÂCHE [SUPPRIMERRVTASK].....	160
<u>9.9.16</u> LA VUE [AJOUTRVTUE].....	161
<u>9.9.17</u> LA TÂCHE [AJOUTERRVTASK].....	164
<u>9.10</u> CONCLUSION DE L'EXEMPLE 7.....	164
10 CONCLUSION GÉNÉRALE.....	167
11 ANNEXES.....	168
<u>11.1</u> LA BIBLIOTHÈQUE JACKSON.....	168
<u>11.2</u> LA BIBLIOTHÈQUE GSON.....	169
<u>11.3</u> INSTALLATION DE L'IDE ECLIPSE.....	170
<u>11.4</u> INSTALLATION DU SDK MANAGER D'ANDROID.....	177
<u>11.5</u> INSTALLATION DES Outils ANDROID POUR ECLIPSE.....	177
<u>11.6</u> CRÉATION D'UN PROJET MAVEN ANDROID.....	181
<u>11.7</u> EXÉCUTION DES EXEMPLES DU DOCUMENT AVEC UN ÉMULATEUR.....	183
<u>11.8</u> EXÉCUTION DES EXEMPLES DU DOCUMENT SUR UNE TABLETTE ANDROID.....	185
<u>11.9</u> INSTALLATION DE [WAMP SERVER].....	186

1 Introduction

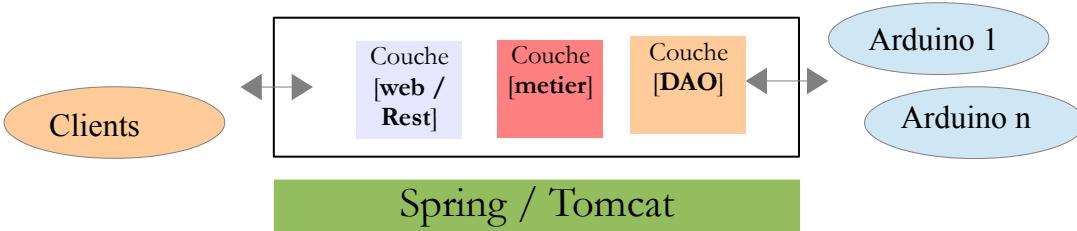
1.1 Contenu du document

Le point de départ de ce document a été un projet étudiant de dernière année de l'école d'ingénieurs ISTIA de l'université d'Angers [istia.univ-angers.fr] : piloter des éléments de la maison (lampe, chauffage, porte de garage, ...) avec un mobile, smartphone ou tablette. Ce projet a été proposé par Sébastien Lagrange, enseignant-chercheur de l'ISTIA. Il a été réalisé par six étudiants : Thomas Ballandras, Raphaël Berthomé, Sylvain Blanchon, Richard Carrée, Jérémy Latorre et Ungur Ulu. Le projet a été décliné en trois sous-projets :

1. application web mobile avec JSF2 / Primefaces ;
2. application web mobile avec HTML5 / Javascript / Windows 8 ;
3. application native Android.

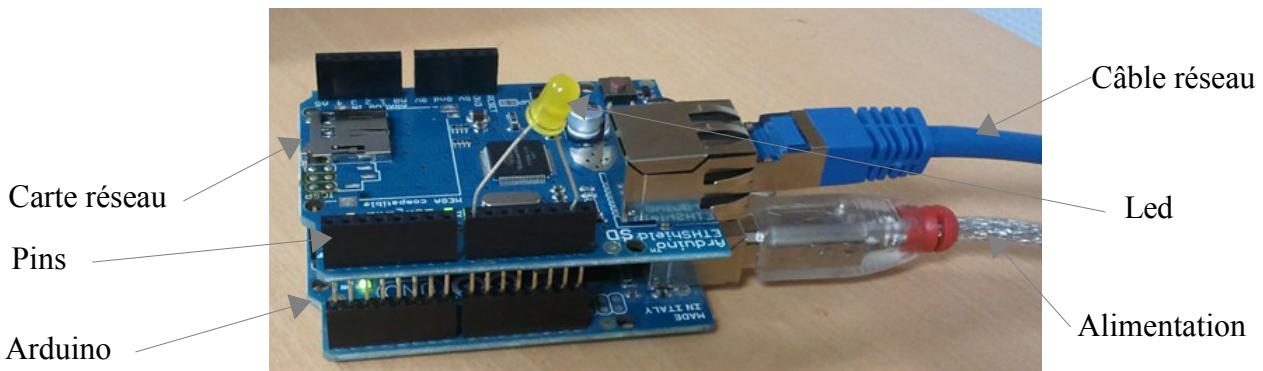
J'ai participé à ce projet en tant que support Java. J'ai réalisé en parallèle des étudiants les projets 1 et 3. C'est ce dernier projet qui a donné naissance aux idées exposées dans ce document.

L'architecture du projet Android est celle d'une application client / serveur. L'architecture du serveur est la suivante :



Wikipedia donne la définition suivante des Arduinos :

Arduino est un [circuit imprimé](#) en [matériel libre](#) (dont les plans sont publiés en [licence libre](#)) sur lequel se trouve un [microcontrôleur](#) qui peut être [programmé](#) pour analyser et produire des [signaux électriques](#), de manière à effectuer des tâches très diverses comme la charge de batteries, la [domotique](#) (le contrôle des appareils domestiques - éclairage, chauffage...), le pilotage d'un [robot](#), etc. C'est une plateforme basée sur une interface entrée/sortie simple et sur un environnement de développement utilisant la technique du [Processing/Wiring](#).



La couche [DAO] est reliée aux arduinos via un réseau. Elle est constituée

- d'un serveur TCP/IP pour enregistrer les Arduinos qui se connectent ;
- d'un client TCP/IP pour envoyer les commandes à exécuter aux Arduinos connectés.

La couche [DAO] peut commander plusieurs Arduinos.

Côté Arduino, on trouve les mêmes composantes mais programmées en langage C :

- un client TCP/IP pour s'enregistrer auprès du serveur d'enregistrement de la couche [DAO] ;
- un serveur TCP/IP pour exécuter les commandes que la couche [DAO] envoie.

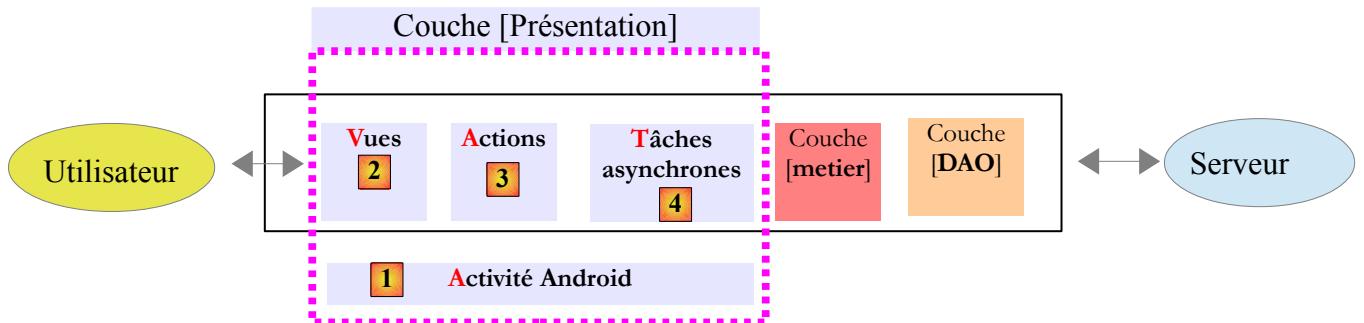
Un Arduino peut servir plusieurs clients. Ils sont alors servis séquentiellement.

Les échanges entre la couche [DAO] et un Arduino se font par lignes de texte :

- la couche [DAO] envoie une ligne de texte contenant une commande au format JSON (**JavaScript Object Notation**) ;
- l'Arduino interprète puis exécute cette commande et renvoie une ligne de texte contenant une réponse également au format JSON.

La couche métier du serveur est exposée au client via un service REST (**REpresentational Rest Transfer**) assuré par le framework Spring MVC.

L'architecture du client Android est elle la suivante :

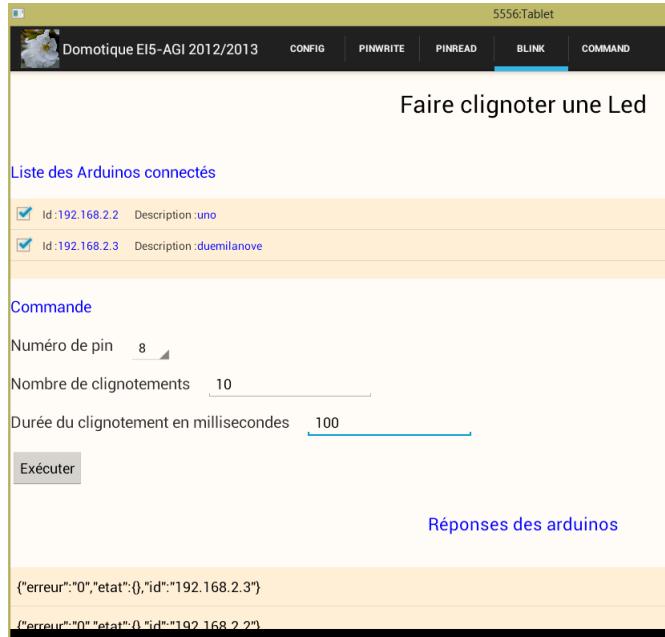


- la couche [DAO] communique avec le serveur REST. C'est un client REST implémenté par **Spring-Android** ;
- la couche [métier] reprend l'interface de la couche [métier] du serveur ;

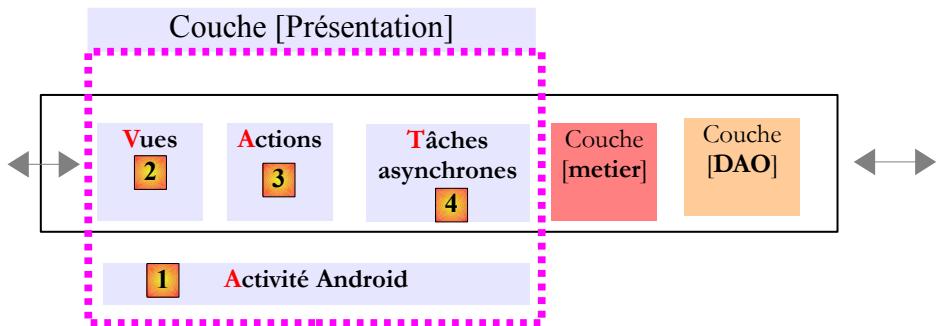
Les versions actuelles d'Android exigent que les connexions réseau soient faites dans un autre thread que celui qui gère les interfaces visuelles. C'est ce qui explique la présence du bloc [4]. Les méthodes de la couche [métier] sont exécutées au sein d'un thread différent de celui de l'UI (User Interface). Lorsqu'on fait exécuter une méthode de la couche[métier] dans un thread, on peut attendre ou non la réponse de la méthode. Dans le premier cas, synchrone, on bloque l'UI en attendant la réponse de la tâche. Dans le second cas, asynchrone, l'UI reste disponible pour l'utilisateur qui peut lancer d'autres actions. Ici, on a choisi la solution asynchrone pour deux raisons :

- le client Android doit pouvoir commander plusieurs Arduinos simultanément. Par exemple, on veut pouvoir faire clignoter une led placée sur deux Arduinos, en même temps et non pas l'une après l'autre. On ne peut donc pas attendre la fin de la première tâche pour commencer la seconde ;
- les connexions réseau peuvent être longues ou ne pas aboutir. Pour cette raison, on veut donner à l'utilisateur la possibilité d'interrompre une action qu'il a lancée. Pour cela, l'UI ne doit pas être bloquée.

Les vues [2] sont les interfaces visuelles présentées à l'utilisateur. Sur une tablette, elles ressemblent à ceci :

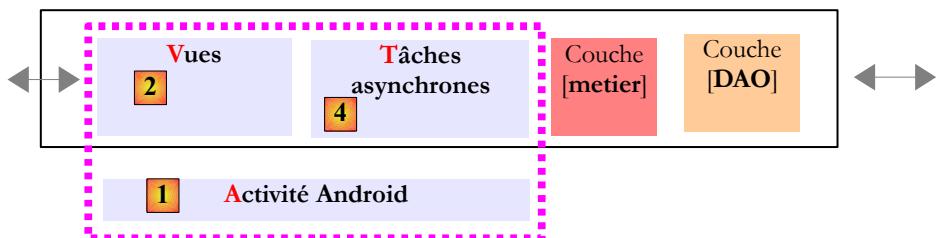


A partir de cette vue, l'utilisateur lance une action avec le bouton [Exécuter].



Le bloc [3] regroupe les actions exécutées par les vues. La vue ne fait que saisir des données et contrôler leur validité. On part du principe qu'elle ne sait pas à quoi elles vont servir. Elle sait simplement à quelle action elle doit transmettre les données saisies. L'action lancera les tâches asynchrones nécessaires, mettra en forme leurs résultats et rendra à la vue un modèle pour que celle-ci se mette à jour. La vue n'est pas bloquée par l'action, afin de donner à l'utilisateur la possibilité de l'interrompre.

Les activités [1] sont le coeur d'une application Android. Ici on n'en a qu'une et elle ne fait quasiment rien si ce n'est d'assurer les changements de vues. On appellera ce modèle, **AVAT** (Activité – Vues – Actions – Tâches) et c'est ce modèle que nous allons décrire et illustrer dans ce document. On verra qu'il peut être allégé de la façon suivante :



Dans le modèle AVAT, la vue est complètement ignorante de l'utilisation faite des données qu'elle a saisies. Dans le modèle **AVT** ci-dessus (**A**ctivité – **V**ues – **T**âches), la logique de l'action est transférée dans la vue. On trouve donc un peu de logique dans la vue. Celle-ci doit organiser les appels des tâches asynchrones elle-même. Cette logique est forcément faible. Si elle était importante, elle

serait normalement transférée dans la couche [métier]. Ce modèle AVT est suffisant dans tous les cas. Le modèle AVAT est pour les puristes comme moi qui cherchent à limiter la vue à la saisie / affichage de données.

Ce document décrit et illustre le modèle AVAT tout d'abord avec des exemples d'école. Puis nous créerons un client Android pour l'application de rendez-vous décrite dans le document " Introduction aux frameworks JSF2, Primefaces et Primefaces Mobile " [<http://tahe.developpez.com/java/primefaces/>]. Dans ce document, est décrite une application web mobile. Nous ferons la même chose que celle-ci avec cette fois une application native Android.

Ce document n'est pas celui d'un expert Android. Il y a moult choses que je ne connais pas. On y trouvera simplement la démarche d'un développeur J2EE qui arrivant dans le monde Android cherche à retrouver deux habitudes :

- la structuration en couches du code ;
- le modèle MVC (Modèle – Vue - Contrôleur)

Lorsque j'ai découvert Android il y a deux ans, je l'ai fait avec un livre qui ne devait pas être le bon. Une application Android était un ensemble d'activités. Chaque activité gérait un écran et une activité pouvait passer des informations à l'activité suivante. Le livre décrivait des situations singulières mais il n'y avait nulle part de vision globale d'une application qui aurait regroupé ces situations singulières. Finalement, comment structure-t-on une application Android ? C'est une question à laquelle un enseignant doit pouvoir répondre.

J'ai laissé tomber Android parce que ce n'était pas ma préoccupation du moment. A cause du projet étudiant évoqué au début du document, j'ai du y revenir. Il s'est trouvé que le plugin ADT (Android Developer Tools) de Google pour Eclipse qui permet de gérer des projets Android, génère par défaut des squelettes d'application pour différents cas de navigation entre les vues. Dans le cas où l'on veut naviguer à l'aide d'onglets, l'activité générée est de type [FragmentActivity] et les onglets de type [Fragment]. L'activité se charge d'afficher l'onglet cliqué par l'utilisateur. J'ai tout de suite vu dans l'activité un contrôleur et dans les fragments des vues et j'ai alors cherché à implémenter le modèle MVC du web.

Je suis parti d'un article écrit en 2005 pour créer des interfaces Swing respectant le modèle MVC du web : " Construction d'une application Swing MVC à trois couches avec Spring " à l'URL [<http://tahe.developpez.com/java/swing3tier>]. Dans ce modèle, la vue Swing lançait des actions sans les attendre comme dans le modèle AVAT. L'action était exécutée par un contrôleur mais celui-ci attendait la fin de celle-ci avant d'exécuter la suivante. Cela ne correspondait pas à ce qui était souhaité ici. Après de nombreuses itérations, je me suis arrêté au modèle AVAT qui me convenait.

Je ne sais pas si ce modèle est pertinent mais je l'ai trouvé utile sur les quelques exemples où il a été utilisé. Débutant Android, je n'ai pas exploré certains points dont celui qui me paraît important du cycle de vie de l'activité et des fragments. Que se passe-t-il lorsque l'application est recouverte visuellement par une autre ou lorsque je change l'orientation de la tablette ?

Donc l'utilité et la fiabilité du modèle AVAT restent à démontrer. Il est possible également qu'on peut faire la même chose de façon plus simple encore...

1.2 Les outils utilisés

Les exemples qui suivent ont été réalisés avec l'IDE Spring Tool Suite (STS), un IDE basé sur Eclipse :



Ils ont été testés sur une tablette Samsung Galaxy Tab 2 10.1. Les copies d'écran proviennent d'exécutions sur un émulateur logiciel de la tablette.

L'installation de l'IDE STS et des plugins nécessaires au développement Android est présentée en annexes, page 170, paragraphe 11.3.

1.3 Les codes source

Les codes source des exemples sont disponibles à l'URL [<http://tahe.developpez.com/android/avat-01>].

Nom
android-avat
android-avat-exemple1
android-avat-exemple2
android-avat-exemple3
android-avat-exemple4
android-avat-exemple5
android-avat-exemple6
android-avat-exemple7

Ce sont des projets Maven qu'on peut importer dans la plupart des IDE Java.

1.4 Pré-requis

La bonne compréhension du document nécessite certaines connaissances :

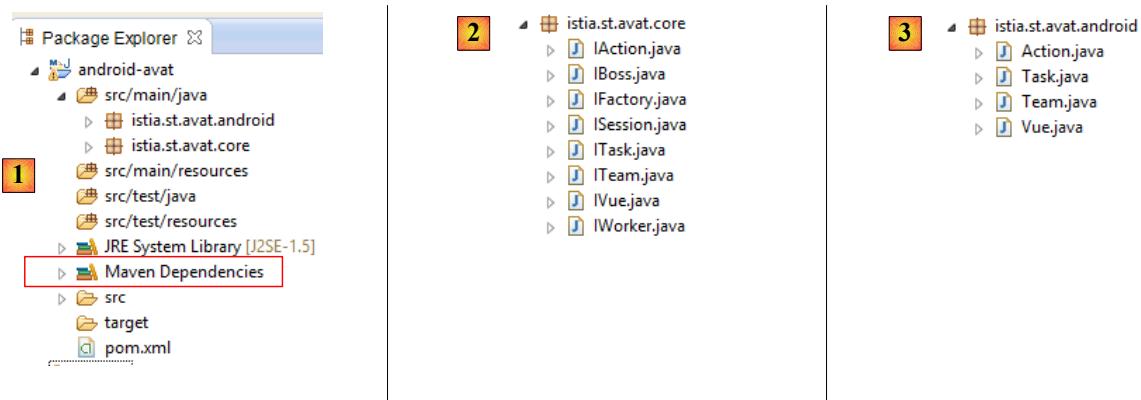
- le langage Java, les architectures en couches, la programmation par interfaces ;
- la construction d'interfaces visuelles Android ;
- la construction de projets Maven.

Le document présente des architectures REST client / serveur. Elles seront expliquées.

2 Le modèle AVAT

2.1 Le projet Eclipse

Le projet Eclipse du modèle AVAT est le suivant :



- en [1], le projet est un projet Maven ;
- en [2], les interfaces du modèle ;
- en [3], leurs implémentations Android.

Le fichier [pom.xml] du projet Maven est le suivant :

```
1. <project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
  4.0.0.xsd">
3.   <modelVersion>4.0.0</modelVersion>
4.   <groupId>istia.st.android</groupId>
5.   <artifactId>android-avat</artifactId>
6.   <version>1.0-SNAPSHOT</version>
7.   <properties>
8.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
9.     <platform.android.version>4.1.1.4</platform.android.version>
10.    </properties>
11.
12.   <name>android-avat</name>
13.
14.   <dependencies>
15.     <dependency>
16.       <groupId>com.google.android</groupId>
17.       <artifactId>android</artifactId>
18.       <version>${platform.android.version}</version>
19.       <scope>provided</scope>
20.     </dependency>
21.     <dependency>
22.       <groupId>com.google.android</groupId>
23.       <artifactId>support-v4</artifactId>
24.       <version>r6</version>
25.       <scope>provided</scope>
26.     </dependency>
27.   </dependencies>
28.
29. </project>
```

Le projet Maven dépend d'Android :

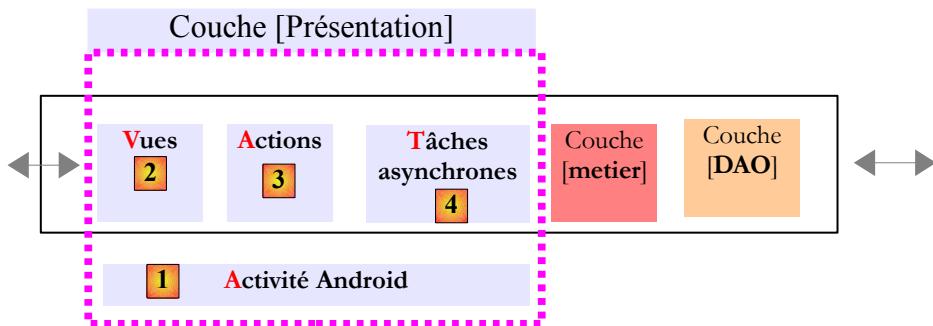
- ligne 9 : la version d'Android ;
- lignes 15-20 : la dépendance vers Android ;
- lignes 21-26 : une dépendance vers des bibliothèques de support d'Android. Ces bibliothèques permettent à des applications conçues pour une certaine version d'Android de s'exécuter sur des versions plus anciennes. Ces bibliothèques de support contiennent les classes manquantes à la version ancienne d'Android.

2.2 Le cœur du modèle

```

    istia.st.avat.core
      ▷ IAction.java
      ▷ IBoss.java
      ▷ IFactory.java
      ▷ ISession.java
      ▷ ITask.java
      ▷ ITeam.java
      ▷ IVue.java
      ▷ IWorker.java
  
```

Revenons à l'architecture AVAT :



- l'activité [1] affiche des vues [2] ;
- la vue [2] lance des actions [3] ;
- l'action [3] lance des tâches [4] ;
- une tâche [4] exécute une ou plusieurs méthodes de la couche [métier].

Le modèle AVAT a été conçu pour séparer les différentes tâches (separation of concern) de la couche [Présentation] d'une application Android :

- l'activité gère l'affichage des vues. Elle n'accède à celles-ci que via un n° et leur interface [IVue] ;
- la vue sert à saisir des données et à en afficher. Elle vérifie la validité des données saisies. Elle transmet celles-ci à une action identifiée par un n° et son interface [IAction] ;
- l'action normalement devrait appeler une méthode de la couche [métier] là également via une interface. Cette méthode peut déboucher par l'ouverture d'une connexion réseau dans la couche [DAO]. Avec les versions récentes d'Android, celle-ci doit se faire dans un thread séparé de celui de l'UI (User Interface). Nous appelons ce thread une tâche asynchrone. Dans ce cas, l'action va lancer une ou plusieurs tâches asynchrones identifiées par un n° et leur interface [ITask] ;
- la tâche asynchrone va exécuter une ou plusieurs méthodes de la couche [métier] via l'interface de celle-ci.

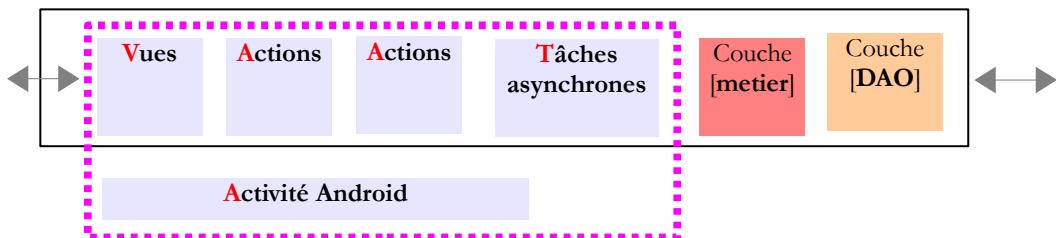
Dans le modèle AVAT, certaines entités font travailler d'autres entités :

- la vue fait travailler des actions ;
- l'action fait travailler des tâches.

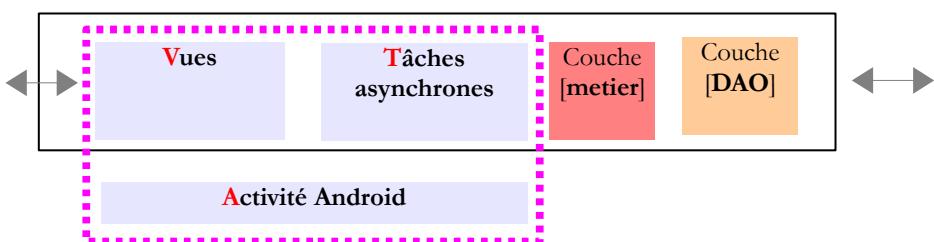
L'entité qui fait travailler d'autres entités est un patron. Elle implémentera l'interface [IBoss]. L'entité qui travaille est un travailleur. Elle implémentera l'interface [IWorker]. Ainsi,

- la vue est un patron. Elle implémentera l'interface [IBoss] ;
- l'action est à la fois un travailleur et un patron car elle fait travailler des tâches. Elle implémentera les interfaces [IBoss] et [IWorker] ;
- la tâche est un travailleur. Elle implémentera l'interface [IWorker].

Ce modèle est extensible. On peut l'élargir :



ou le rétrécir :



Une entité patron a une équipe de travailleurs. Cette équipe implémentera l'interface [ITeam].

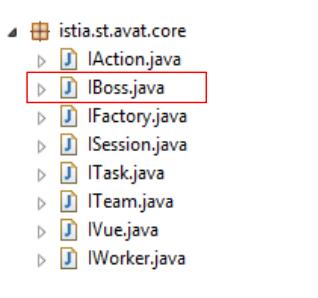
Les entités du modèle VAT seront générées par une fabrique d'objets (factory) d'interface [IFactory]. Aucune implémentation n'est proposée car elle est spécifique à chaque application. Nous en ferons une dans tous nos exemples. La fabrique est centrale à l'application. Le concept est tiré de celui d'[ApplicationContext] du framework Spring.

Les vues partagent une même activité Android. Celle-ci peut donc être utilisée pour stocker des informations partagées entre les vues. De façon alternative, nous pouvons utiliser le concept de session du web. Celle-ci implementera l'interface [ISession]. Dans nos exemples, nous utilisons systématiquement la session pour partager des données entre vues mais c'est une pratique qui peut être discutée. Nous verrons pourquoi.

Passons en revue la définition de ces interfaces :

Interface [IBoss]

Modélise un patron.



```

1. package istia.st.avat.core;
2.
    
```

```

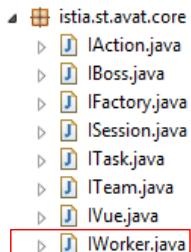
3. public interface IBoss {
4.
5.     // les types d'événements
6.     static final int WORK_STARTED = 0;
7.     static final int WORK_INFO = 1;
8.     static final int WORK_TERMINATED = 2;
9.     static final String[] STATUS = { "WORK_STARTED", "WORK_INFO", "WORK_TERMINATED" };
10.
11.    // le boss a une identité
12.    public void setBossId(String id);
13.
14.    public String getBossId();
15.
16.    // un boss a accès à la factory
17.    public void setFactory(IFactory factory);
18.
19.    // le boss gère une équipe de workers
20.    public void setTeam(ITeam team);
21.
22.    // il reçoit des notifications de ses workers
23.    public void notifyEvent(IWorker worker, int eventType, Object event);
24.
25.    // il peut annuler un travail
26.    public void cancel(IWorker worker);
27.
28.    public void cancel(String workerId);
29.
30.    // il peut annuler tous les travaux
31.    public void cancelAll();
32.
33.    // on lui signale la fin des travaux
34.    public void notifyEndOfTasks();
35.
36.    // il monitore ses travailleurs
37.    public void beginMonitoring();
38.
39.    // le boss peut travailler en mode verbeux
40.    public void setVerbose(boolean verbose);
41. }

```

- lignes 12, 14 : le boss a une identité pour des besoins de traçabilité dans les logs ;
- ligne 17 : comme toutes les entités du modèle VAT, le boss a accès à la fabrique d'objets ;
- ligne 20 : le boss a une équipe de travailleurs ;
- lignes 26, 28, 31 : le boss ou quelqu'un d'autre peut annuler le travail d'un travailleur particulier (lignes 26, 28) ou de toute l'équipe (ligne 31) ;
- ligne 23 : les travailleurs rendent compte à leur boss du déroulement de leur travail. Les paramètres de la méthode sont :
 - **IWorker worker** : le travailleur qui rend compte,
 - **int eventType** : le type d'information que donne le travailleur : WORK_STARTED (le travailleur vient de commencer son travail), WORK_TERMINATED (le travailleur a terminé son travail), WORK_INFO (le travailleur transmet une information à son boss),
 - **Object event** : une information transmise par le travailleur. Peut être **null**.
- ligne 34 : cette méthode est appelée pour signaler au boss que toutes les tâches qu'il a lancées sont terminées ;
- ligne 37 : le boss lance d'abord toutes les tâches puis par cette méthode signale à son équipe qu'il attend la fin des tâches. Lorsque celle-ci arrivera, la méthode *notifyEndOfTasks* sera appelée ;
- ligne 40 : on peut demander au boss de se mettre en mode verbeux. Les événements WORK_STARTED, WORK_INFO et WORK_TERMINATED sont alors logués ainsi que les opérations d'annulation. C'est utile pour déboguer l'application.

Interface [IWorker]

Modélise un travailleur.



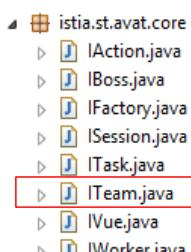
```

1. package istia.st.avat.core;
2.
3. public interface IWorker {
4.
5.     // a accès à la factory
6.     public void setFactory(IFactory factory);
7.
8.     // un worker sait faire un travail
9.     public void doWork(Object... params);
10.
11.    // il a un patron
12.    public void setBoss(IBoss boss);
13.
14.    // il a une identité
15.    public void setWorkerId(String id);
16.
17.    public String getWorkerId();
18.
19.    // son travail peut être annulé
20.    public void cancel();
21.
22. }
  
```

- ligne 12 : un travailleur a un patron à qui il envoie des notifications via la méthode `IBoss.notifyEvent`. Il envoie les notifications `WORK_STARTED`, `WORK_INFO` et `WORK_TERMINATED`.
- lignes 15-17 : il a une identité. Cela permet de le tracer dans les logs. Cela permet également au patron d'annuler une tâche identifiée par son Id `IBoss.cancel(String workerId)` ;
- ligne 9 : il sait faire un travail. Pour préciser celui-ci, il peut recevoir une suite de paramètres ;
- ligne 20 : ce travail peut être annulé, généralement par le patron ;
- ligne 6 : le travailleur a lui aussi accès à la fabrique d'objets.

Interface [ITeam]

Modélise une équipe de travailleurs.



```

1. package istia.st.avat.core;
2.
  
```

```

3. // une équipe de travailleurs
4. public interface ITeam {
5.
6.     // l'équipe gère une équipe de travailleurs
7.     // elle exécute les ordre d'un boss
8.
9.     // l'équipe a un monitor
10.    public void setMonitor(IBoss boss);
11.
12.    // l'équipe reçoit des notifications des travailleurs via le boss
13.    public void notifyEvent(IWorker worker, int eventType, Object event);
14.
15.    // le boss peut annuler le travail d'un travailleur
16.    public void cancel(IWorker worker);
17.
18.    public void cancel(String workerId);
19.
20.    // le boss annule tout
21.    public void cancelAll();
22.
23.    // le boss envoie cet ordre lorsqu'il veut être prévenu de la fin des tâches
24.    public void beginMonitoring();
25.
26. }

```

- ligne 10 : l'équipe a un patron, un surveillant (monitor). C'est à lui que l'équipe envoie la notification " endOfTasks " lorsque tous les travailleurs ont terminé leurs tâches ;
- le boss délègue à son équipe de travailleurs certains traitements qu'on lui demande. C'est le cas des méthodes des lignes 13, 16, 18, 21, 24.

Nous présentons maintenant les interfaces des Vues, Actions et Tâches du modèle VAT :

L'interface [IVue]

```

istia.st.avat.core
  ↳ IAction.java
  ↳ IBoss.java
  ↳ IFactory.java
  ↳ ISession.java
  ↳ ITask.java
  ↳ ITeam.java
  ↳ IVue.java (highlighted)
  ↳ IWorker.java

```

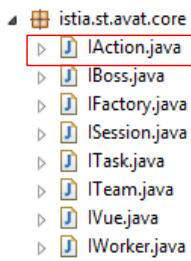
Une vue est un patron car elle gère une équipe d'actions. Elle implémente donc l'interface [IBoss] :

```

1. package istia.st.avat.core;
2.
3.
4. public interface IVue extends IBoss {
5. }

```

L'interface [IAction]



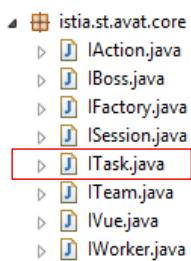
Une action est à la fois :

- un travailleur : elle travaille pour une vue ;
- un patron : elle fait travailler des tâches.

Elle implémente donc les interfaces [IBoss] et [IWorker] :

```
1. package istia.st.avat.core;
2.
3. public interface IAction extends IBoss, IWorker {
4.
5. }
```

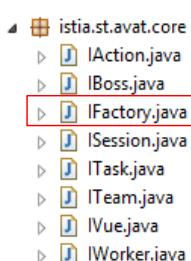
L'interface [ITask]



Une tâche asynchrone est un travailleur. Elle travaille pour une action.

```
1. package istia.st.avat.core;
2.
3. public interface ITask extends IWorker {
4.
5. }
```

L'interface [IFactory]



L'interface [IFactory] permet de demander la référence d'un objet identifié par un nombré :

```

1. package istia.st.avat.core;
2.
3. public interface IFactory {
4.     // fabrique d'objets
5.     public Object getObject(int id, Object... params);
6.
7. }

```

- ligne 5, l'objet identifié par **id** est créé ou recyclé (singleton). Pour sa construction initiale, une suite de paramètres peut être passée.

L'interface [ISession]

```

    ▲ istia.st.avat.core
      ▷ IAction.java
      ▷ IBoss.java
      ▷ IFactory.java
      ▷ ISession.java
      ▷ ITask.java
      ▷ ITeam.java
      ▷ IVue.java
      ▷ IWorker.java

```

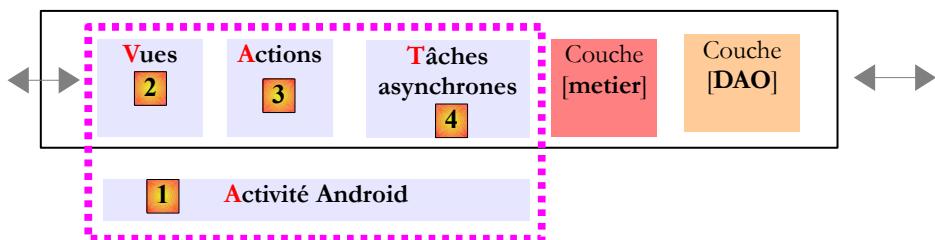
La session permet de stocker des objets, de les retrouver et de les supprimer. Les objets sont identifiés par une chaîne de caractères.

```

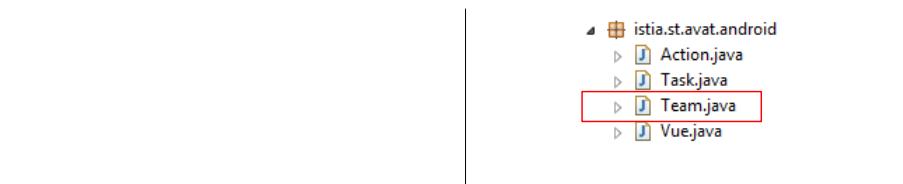
1. package istia.st.avat.core;
2.
3. public interface ISession {
4.     // ajoute un élément dans la session
5.     void add(String id, Object value);
6.     // enlève un élément de la session
7.     void remove(String id);
8.     // recherche d'un élément
9.     Object get(String id);
10.    // vide la session
11.    void clear();
12. }

```

2.3 L'implémentation Android du modèle AVAT



Le modèle VAT [2, 3, 4] ci-dessus est implémenté par trois classes abstraites [Vue, Action, Task] auxquelles s'ajoute la classe [Team] implémentant l'interface [ITeam].



Classe [Team]

Elle implémente l'interface [ITeam]. C'est elle qui fait l'essentiel du travail du modèle VAT.

```

1. package istia.st.avat.android;
2.
3. import istia.st.avat.core.IBoss;
4. import istia.st.avat.core.ITeam;
5. import istia.st.avat.core.IWorker;
6.
7. import java.text.SimpleDateFormat;
8. import java.util.ArrayList;
9. import java.util.Date;
10. import java.util.List;
11. import java.util.Locale;
12.
13. import android.util.Log;
14.
15. public class Team implements ITeam {
16.
17.     // toutes les méthodes de cette classe s'exécutent dans le thread de l'UI
18.     // c'est pourquoi elles ne sont pas synchronisées
19.
20.     // la liste des travailleurs de l'équipe
21.     private List<IWorker> workers = new ArrayList<IWorker>();
22.     // le monitoring
23.     private Boolean isMonitoring = false;
24.     // le monitor de l'équipe
25.     private IBoss monitor;
26.     // état verbeux ou non
27.     private boolean verbose;
28.
29.     // recherche d'un travailleur à partir de son id
30.     private IWorker getWorkerById(String workerId) {
31.         for (IWorker worker : workers) {
32.             if (worker.getWorkerId().equals(workerId)) {
33.                 // on a trouvé
34.                 return worker;
35.             }
36.         }
37.         // on n'a pas trouvé
38.         return null;
39.     }
40.
41.     // annulation du travail d'un travailleur
42.     public void cancel(IWorker worker) {
43.         // on demande au travailleur d'arrêter son travail
44.         worker.cancel();
45.         // on le supprime de l'équipe
46.         workers.remove(worker);
47.         // verbose ?
48.         if (verbose) {
49.             Log.i(worker.getWorkerId(), String.format("taskId=%s canceled at time=%s\n",
50.                 worker.getWorkerId(),
51.                 new SimpleDateFormat("hh:mm:ss:SS", Locale.FRANCE).format(new Date())));
52.         }
53.     }
54. }
```

```

51.     }
52. }
53.
54. public void cancel(String workerId) {
55.     // on cherche le travailleur
56.     IWorker worker = getWorkerById(workerId);
57.     // on annule son travail si on l'a trouvé
58.     if (worker != null) {
59.         cancel(worker);
60.     }
61. }
62.
63. // annulation de tous les travaux
64. public void cancelAll() {
65.     // on les annule en partant de la fin de la liste
66.     int nbWorkers = workers.size();
67.     for (int i = nbWorkers - 1; i >= 0; i--) {
68.         cancel(workers.get(i));
69.     }
70. }
71.
72. // on gère certains événements
73. public void notifyEvent(IWorker worker, int eventType, Object event) {
74.     // verbose ?
75.     if (verbose) {
76.         Log.i(worker.getWorkerId(), String.format("eventType=%s, taskId=%s, time=%s\n",
77.             IBoss.STATUS[eventType],
78.             worker.getWorkerId(), new SimpleDateFormat("hh:mm:ss:SS",
79.                 Locale.FRANCE).format(new Date())));
80.     }
81.     // on s'intéresse à certains événements
82.     switch (eventType) {
83.         case IBoss.WORK_STARTED:
84.             // un nouveau travailleur
85.             workers.add(worker);
86.             break;
87.         case IBoss.WORK_TERMINATED:
88.             // un travailleur a terminé son travail
89.             workers.remove(worker);
90.             // fin des travailleurs ?
91.             if (isMonitoring && workers.size() == 0) {
92.                 // on l'indique au monitor
93.                 monitor.notifyEndOfTasks();
94.             }
95.             break;
96.     }
97.     // le boss veut être prévenu de la fin des travaux
98.     public void beginMonitoring() {
99.         // on monitore
100.        isMonitoring = true;
101.        // on vérifie tout de suite s'il reste des travailleurs
102.        if (workers.size() == 0) {
103.            // on l'indique au monitor
104.            monitor.notifyEndOfTasks();
105.        }
106.    }
107. }
108.
109. // getters et setters
110. public void setMonitor(IBoss monitor) {
111.     this.monitor = monitor;

```

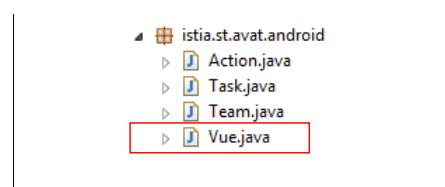
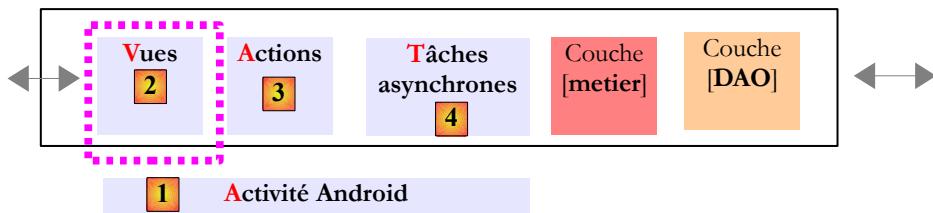
```

112.    }
113.
114.    public void setVerbose(boolean verbose) {
115.        this.verbose = verbose;
116.    }
117.
118. }

```

- ligne 21 : une équipe gère une équipe de travailleurs ;
- ligne 25 : elle a un patron ou monitor. C'est à ce patron qu'elle signale la fin des tâches ;
- ligne 23 : elle surveille ou non la fin des tâches. C'est à l'initiative du patron qu'elle commence cette surveillance ;
- ligne 27 : on peut demander à l'équipe de loguer ce qu'elle fait ;
- ligne 73 : l'équipe maintient la liste des travailleurs en activité. Lorsqu'elle reçoit la notification WORK_STARTED d'un travailleur, elle ajoute celui-ci dans la liste (ligne 83). Lorsqu'elle recevra la notification WORK_TERMINATED elle l'enlèvera (ligne 87). Dans ce dernier cas, si son boss lui a demandé de surveiller la fin des tâches et qu'il n'y a plus de travailleurs dans l'équipe (tous ont terminé leur tâche), elle appellera la méthode [notifyEndOfTasks] de son boss (ligne 91) ;
- ligne 98 : l'équipe reçoit une demande de surveillance de la fin des tâches. Elle commence cette surveillance immédiatement (ligne 102). La raison en est la suivante : le boss demandera la surveillance de la fin des tâches après les avoir toutes lancées. Celles-ci vont venir s'accumuler dans l'équipe au moyen des notifications. Entre le moment où le boss lance la dernière tâche et celui où il demande la surveillance de la fin des tâches, celles-ci peuvent être toutes terminées. Si on ne fait pas le test de la ligne 102, on va attendre une notification WORK_TERMINATED qui ne viendra jamais. La demande de surveillance peut-elle intervenir alors qu'une tâche n'a pas encore envoyé sa notification WORK_STARTED ? Auquel cas, trouver une liste de travailleurs vide ne signifierait pas nécessairement la fin des tâches. Il faudra être vigilant sur le moment où la notification WORK_STARTED est envoyée au boss ;
- ligne 42 : la méthode *cancel* permet d'annuler une tâche. Le message *cancel* est transmis à la tâche elle-même. Le travailleur est ensuite retiré de la liste des travailleurs (ligne 46). Si la tâche est aussi un patron, le message *cancel* va être transmis à tous les membres de son équipe et ainsi de suite. En bout de chaîne, un certain nombre de tâches asynchrones va être annulée.

Voilà pour la classe [Team]. Passons maintenant à la classe [Vue] :



La classe abstraite [Vue]

```

1. package istia.st.avat.android;
2.
3. import istia.st.avat.core.IBoss;
4. import istia.st.avat.core.IFactory;
5. import istia.st.avat.core.ITeam;
6. import istia.st.avat.core.IVue;
7. import istia.st.avat.core.IWorker;
8. import android.app.Activity;
9. import android.app.Fragment;

```

```

10.
11. public abstract class Vue extends Fragment implements IVue {
12.
13.     // son identité
14.     protected String bossId;
15.     // la vue a une équipe d'actions qui travaillent pour elle
16.     private ITeam team;
17.     // la vue travaille avec une activité
18.     protected Activity activity;
19.
20.     // la factory
21.     protected IFactory factory;
22.
23.     // la vue gère des événements
24.     public void notifyEvent(IWorker worker, int eventType, Object event) {
25.         // on passe l'événement à l'équipe d'actions
26.         team.notifyEvent(worker, eventType, event);
27.     }
28.
29.     public void cancel(IWorker worker) {
30.         // on délègue à l'équipe
31.         team.cancel(worker);
32.
33.     }
34.
35.     public void cancelAll() {
36.         // on délègue à l'équipe
37.         team.cancelAll();
38.
39.     }
40.
41.     public void cancel(String workerId) {
42.         // on délègue à l'équipe
43.         team.cancel(workerId);
44.
45.     }
46.
47.     // le monitor de l'équipe
48.     public void setMonitor(IBoss monitor) {
49.         team.setMonitor(monitor);
50.
51.     }
52.
53.     // début du monitoring
54.     public void beginMonitoring() {
55.         team.beginMonitoring();
56.     }
57.
58.     // état verbeux
59.     public void setVerbose(boolean verbose) {
60.         team.setVerbose(verbose);
61.     }
62.
63.     // méthodes implémentées par la classe fille
64.     abstract public void notifyEndOfTasks();
65.
66.     // getters et setters
67.     ...
68. }
```

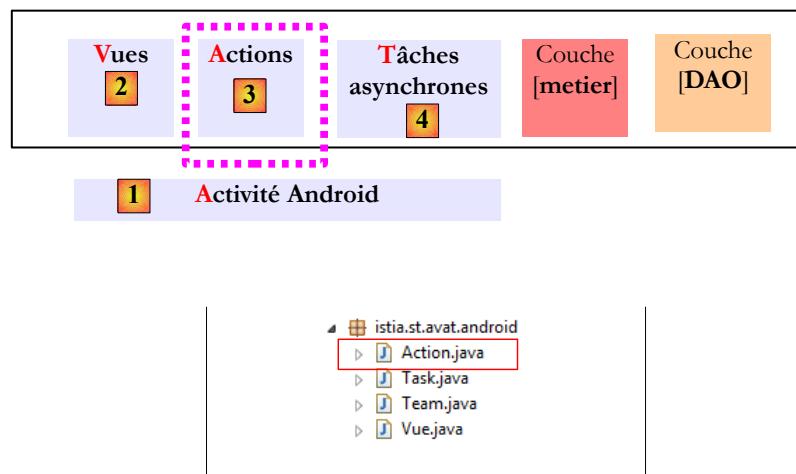
- ligne 11 : la classe abstraite [Vue] implémente la classe [IVue] déjà décrite. Elle étend la classe Android [Fragment]. L'idée est d'utiliser une activité de type [FragmentActivity] capable d'afficher des fragments ;
- ligne 14 : l'identité de la vue pour la tracer dans les logs ;

- ligne 16 : la vue est un [IBoss]. Elle a donc une équipe de travailleurs. Dans le modèle AVAT (Activité-Vues-Actions-Tâches), cette équipe est composée d'actions [IAction]. Dans le modèle AVT (Activité-Vues-Tâches), cette équipe est composée de tâches [ITask]. On notera que ce champ est privé. Les classes filles n'auront donc pas de notion d'équipe. Elles devront passer par leur classe mère pour y avoir accès. Ce choix peut être discuté ;
- ligne 18 : on est dans Android. On a donc la notion d'activité. Celle-ci sera de type [FragmentActivity]. La vue l'utilisera pour demander un changement de vue ;
- ligne 21 : la fabrique d'objets de l'application Android. Elle est injectée dans tous les composants VAT ;
- ligne 24 : la vue en tant que [IBoss] reçoit des notifications de ses travailleurs ;
- ligne 26 : l'événement est passé à l'équipe. On notera qu'aucun test de nullité n'est fait sur l'équipe. Il faudra donc qu'une vue ait une équipe même vide. L'équipe gère les événements WORK_STARTED et WORK_TERMINATED et générera la notification [endOfTasks] pour la vue lorsqu'il n'y aura plus de travailleurs. C'est comme cela que la vue saura que les actions qu'elle a lancées sont terminées. L'événement WORK_INFO n'est pas traité. La classe fille devra redéfinir la méthode pour traiter cet événement ;

Parce que le champ de la ligne 16 est privé, les classes filles doivent passer par leur classe mère pour gérer l'équipe des travailleurs. On trouve donc un certain nombre de méthodes proposées aux classes filles pour intervenir sur l'équipe : lignes 24, 29, 35, 41, 48, 54, 59.

- ligne 64 : parce que la vue est un [IBoss], elle doit implémenter la méthode [notifyEndOfTasks]. C'est la méthode appelée par l'équipe lorsque toutes les actions / tâches ont été accomplies. Le travail à faire ici sera défini par la classe fille.

La classe abstraite [Action]



La classe abstraite [Action] est la suivante :

```

1. package istia.st.avat.android;
2.
3. import istia.st.avat.core.IAction;
4. import istia.st.avat.core.IBoss;
5. import istia.st.avat.core.IFactory;
6. import istia.st.avat.core.ITeam;
7. import istia.st.avat.core.IWorker;
8.
9. public abstract class Action implements IAction {
10.
11.    // ses deux identités
12.    protected String bossId;
13.    protected String workerId;
14.
15.    // la vue qui demande l'action
16.    protected IBoss boss;
17.
18.    // l'équipe de tâches qui travaillent pour l'action

```

```

istia.st.avat.android
├── Action.java
└── ...

```

```

19. private ITeam team;
20.
21. // la factory
22. protected IFactory factory;
23.
24. // l'action en tant que IWorker peut être annulée par le boss
25. public void cancel() {
26.     // l'action annule toutes les tâches qu'elle a lancées
27.     cancelAll();
28. }
29.
30. // on gère certains événements
31. public void notifyEvent(IWorker worker, int eventType, Object event) {
32.     // on passe l'événement à l'équipe de tâches
33.     team.notifyEvent(worker, eventType, event);
34. }
35.
36. // l'action en tant que IBoss peut annuler le travail d'un de ses travailleurs
37. public void cancel(IWorker worker) {
38.     // on délègue à l'équipe
39.     team.cancel(worker);
40. }
41.
42. // l'action en tant que IBoss peut annuler le travail de tous ses travailleurs
43. public void cancelAll() {
44.     // on délègue à l'équipe
45.     team.cancelAll();
46. }
47.
48. // l'action en tant que IBoss peut annuler le travail d'un de ses travailleurs
49. public void cancel(String workerId) {
50.     // on délègue à l'équipe
51.     team.cancel(workerId);
52. }
53.
54. // début du monitoring
55. public void beginMonitoring() {
56.     team.beginMonitoring();
57. }
58.
59. // état verbeux
60. public void setVerbose(boolean verbose) {
61.     team.setVerbose(verbose);
62. }
63.
64. // méthodes de la classe fille
65. // la classe fille fait le travail
66. abstract public void doWork(Object... params);
67.
68. // elle veut être prévenue de la fin des tâches
69. abstract public void notifyEndOfTasks();
70.
71. // getters et setters
72. ...
73. }

```

La classe [Action] est un [IBoss]. En tant que tel, on trouve des champs et des méthodes déjà rencontrées pour la classe [IVue]. On ne s'y attardera pas. Ce sont

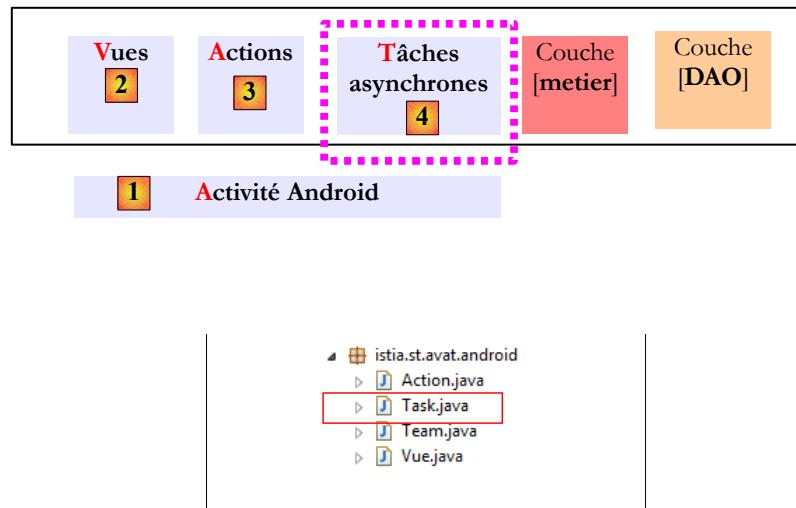
- les champs des lignes 12, 19, 22 ;
- les méthodes des lignes 31, 37, 43, 49, 55, 60, 69

On remarquera que là encore, l'équipe ne peut être nulle. Mais elle peut être vide.

Passons en revue les méthodes liées au fait que [Action] est également un [IWorker].

- ligne 13 : elle a en plus de son identifiant de boss, un identifiant de travailleur. Ils peuvent être ou non identiques ;
- ligne 16 : en tant que [IWorker], l'action a un boss [IBoss]. Ce boss est une vue ;
- ligne 25 : l'action peut être annulée par le boss. En tant que [IBoss], elle peut elle-même annuler le travail de son équipe. C'est ce qu'elle fait en ligne 27 ;
- ligne 66 : en tant que [IWorker], l'action fait un travail. Celui-ci sera défini par la classe fille.

La classe abstraite [Task]



La classe [Task] est la tâche asynchrone qui exécute les méthodes de la couche [métier]. Son code est le suivant :

```
1. package istia.st.avat.android;
2.
3. import istia.st.avat.core.IBoss;
4. import istia.st.avat.core.IFactory;
5. import istia.st.avat.core.ITask;
6. import android.os.AsyncTask;
7.
8. public abstract class Task extends AsyncTask<Object, Object, Void> implements ITask {
9.
10.    // la tâche est identifiée
11.    protected String id;
12.
13.    // la tâche a un boss
14.    protected IBoss boss;
15.
16.    // la factory
17.    protected IFactory factory;
18.
19.    // mode verbose
20.    protected boolean verbose;
21.
22.    // annulation de la tâche
23.    public void cancel() {
24.        // la tâche est annulée
25.        cancel(true);
26.    }
27.
28.    // la tâche fille fait un travail
29.    public void doWork(Object... params) {
30.        // le travail est fait en tâche de fond
31.        this.executeOnExecutor(THREAD_POOL_EXECUTOR, params);
```

```

32. }
33.
34. // getters et setters
35. public void setBoss(IBoss boss) {
36.     this.boss = boss;
37. }
38.
39. public void setWorkerId(String id) {
40.     this.id = id;
41. }
42.
43. public String getWorkerId() {
44.     return id;
45. }
46.
47. public void setFactory(IFactory factory) {
48.     this.factory = factory;
49. }
50. }
51.
52. public void setVerbose(boolean verbose) {
53.     this.verbose=verbose;
54. }
55.
56. }

```

- ligne 8 : la classe [Task] étend la classe Android [AsyncTask]. Cette classe a une méthode qui s'exécute dans un thread différent de celui de l'UI. La classe [Task] est donc asynchrone.

La classe [Task] est un [IWorker]. On retrouve donc les méthodes rencontrées pour la classe [Action]. Dans le modèle AVAT, on est en bout de chaîne. Lorsque la vue lance une annulation des tâches / actions, cette annulation se propage jusqu'aux tâches asynchrones [Task]. C'est alors la méthode *cancel* de la ligne 23 qui est exécutée. La tâche asynchrone [AsyncTask] a une méthode *cancel* qui permet de l'annuler. Son paramètre booléen précise cette annulation. Lorsqu'il est à vrai, l'annulation est faite immédiatement sinon la tâche est autorisée à aller jusqu'au bout de son exécution.

En tant que [IWorker], la [Task] fait un travail. C'est la méthode de la ligne 29 qui assure ce travail. La ligne 31 fait ce travail dans un thread à part de celui de l'UI. La méthode [executeOnExecutor] est une méthode de [AsyncTask]. Pour mieux comprendre la classe [Task], il nous faut parler de sa classe parent [AsynTask<Object, Object, Void>] (ligne 8).

La classe AsynTask<Object, Object, Void>

D'après la documentation Android :

An asynchronous task AsyncTask<Params, Progress, Result> is defined by a computation that runs on a background thread and whose result is published on the UI thread. An asynchronous task is defined by 3 generic types, called Params, Progress and Result, and 4 steps, called onPreExecute, doInBackground, onProgressUpdate and onPostExecute.

- la méthode [doInBackGround] est la méthode exécutée en tâche de fond. Sa signature est la suivante

```
protected abstract Result doInBackground (Params... params)
```

Elle reçoit une suite de paramètres de type *Params*. Pour nous ce type sera *Object*. C'est le premier type générique de notre signature AsyncTask<Object, Object, Void>. Elle rend une donnée de type *Result*. Nous, nous ne rendrons aucun résultat. C'est le troisième type générique de notre signature AsyncTask<Object, Object, Void>;

- la méthode [onProgressUpdate] permet de publier dans le thread de l'UI l'avancement de la tâche. Sa signature est la suivante :

```
protected void onProgressUpdate (Progress... values)
```

Nous n'utiliserons pas cette méthode. Dans la signature AsyncTask<Object, Object, Void>, le type *Progress* est le deuxième type générique, ici *Object* ;

- la méthode [onPreExecute] est exécutée dans le thread de l'UI avant l'exécution de la méthode [doInBackGround]. Sa signature est la suivante :

```
protected void onPreExecute ()
```

C'est dans cette méthode que la [Task] enverra à son boss la notification [WORK_STARTED]. La notification est donc envoyée dans le thread de l'UI ;

- la méthode [onPostExecute] est exécutée dans le thread de l'UI après l'exécution de la méthode [doInBackGround]. Sa signature est la suivante :

```
protected void onPostExecute (Result result)
```

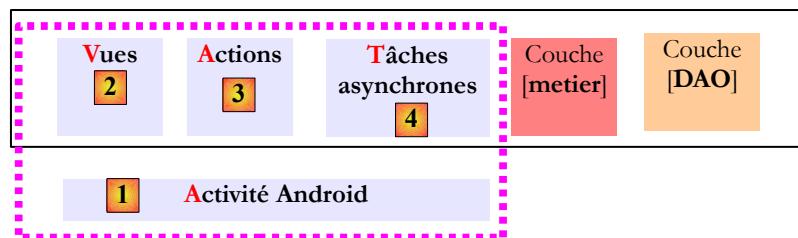
où le type *Result* est le troisième type générique de la signature *AsynTask<Object, Object, Void>*, donc *Void* ici. C'est dans cette méthode que la [Task] enverra à son boss les notifications [WORK_INFO] et [WORK_TERMINATED]. Ces notifications sont donc envoyées dans le thread de l'UI.

Au final, la classe [Team] qui gère les notifications [WORK_STARTED, WORK_INFO, WORK_TERMINATED] les reçoit dans le thread de l'UI. C'est pourquoi les méthodes de cette classe n'ont pas été synchronisées.

Nous avons décrit le modèle AVAT. C'est un modèle ultra-léger d'environ 13 K octets. Voyons maintenant comment nous pouvons l'utiliser.

2.4 Eléments d'implémentation

Reprenons le modèle AVAT :



Nous donnons maintenant quelques indications sur l'implémentation réelle des classes abstraites que nous avons décrites.

2.4.1 L'activité Android

L'activité Android sera de type [FragmentActivity]. Le code ressemblera à ce qui suit :

```

1. public class MainActivity extends FragmentActivity {
2.
3.     // la factory
4.     private Factory factory;
5.
6.     @Override
7.     protected void onCreate(Bundle savedInstanceState) {
8.         super.onCreate(savedInstanceState);
9.         // le sablier
10.        requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
11.        setProgressBarIndeterminateVisibility(false);
12.        // template qui accueille les vues
13.        setContentView(R.layout.main);
14.        // la factory
15.        factory = new Factory(this, new Config());
16.        // on définit l'unique vue
17.        Vue_01 vue = (Vue_01) factory.getObject(Factory.VUE_01, null, "Vue_01");
18.        // on en fait la vue courante

```

```

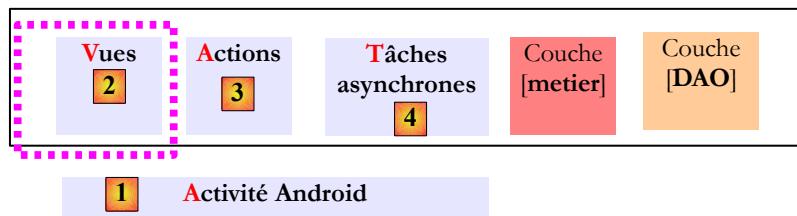
19.     FragmentTransaction fragmentTransaction = getSupportFragmentManager().beginTransaction();
20.     fragmentTransaction.replace(R.id.container, vue);
21.     fragmentTransaction.commit();
22. }
23.
24. }

```

- ligne 4, l'activité est de type [FragmentActivity]. C'est nécessaire puisque nos vues sont de type [Fragment] ;
- lignes 4 et 15 : c'est l'activité qui instanciera la fabrique d'objets auprès de laquelle s'approvisionnent les éléments VAT du modèle pour obtenir des objets ;
- ligne 9-11 : nous utiliserons un indicateur d'attente. Nous le nommerons " sablier " bien que ce n'en soit pas un ;
- ligne 15 : on trouvera dans la factory des éléments de configuration de l'application rassemblés ici dans une classe *Config* ;
- ligne 17 : l'activité demandera à la fabrique les vues [Fragments] qu'elle gère. Contrairement aux objets [Action] et [Task], les objets [Vue] seront des singletons. La fabrique ne les crée qu'en un exemplaire. Celui-ci est créé lors de la première demande. Ensuite lors des demandes suivantes, la fabrique se contente de rendre la référence de l'exemplaire déjà créé ;
- lignes 18-21 : affichage d'une vue.

Dans les exemples d'illustration qui suivent, l'activité se limitera à ces fonctionnalités. Donc peu de choses.

2.4.2 La vue



Une vue pourrait ressembler à ce qui suit :

```

1. public class Vue_01 extends Vue {
2.
3.     // les éléments de l'interface visuelle
4.
5.     private Button btnExecuter;
6.     private Button btnAnnuler;
7.     .....
8.     @Override
9.     public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
   savedInstanceState) {
10.         // on crée la vue du fragment à partir de sa définition XML
11.         return inflater.inflate(R.layout.vue_01, container, false);
12.     }
13.
14.     @Override
15.     public void onActivityCreated(Bundle savedInstanceState) {
16.         // parent
17.         super.onActivityCreated(savedInstanceState);
18.         .....
19.         // bouton Exécuter
20.         btnExecuter = (Button) activity.findViewById(R.id.btn_Executer);
21.         btnExecuter.setOnClickListener(new OnClickListener() {
22.             @Override
23.             public void onClick(View arg0) {
24.                 doExecuter();
25.             }
26.         });

```

```

27.
28.    // bouton Annuler
29.    btnAnnuler = (Button) activity.findViewById(R.id.btn_Annuler);
30.    btnAnnuler.setOnClickListener(new OnClickListener() {
31.        @Override
32.        public void onClick(View arg0) {
33.            // on annule toutes les actions
34.            cancelAll();
35.            // on annule l'attente
36.            cancelWaiting();
37.            // gestion de l'UI
38.            .....
39.        });
40.
41.    }
42.
43.    protected void doExecuter() {
44. .....
45.        // on teste la validité des saisies
46.        if (!isValid()) {
47.            return;
48.        }
49.        // on exécute une action de façon asynchrone
50.        action = (IAction) factory.getObject(Factory.ACTION_01, this, "Action_01");
51.        // on signale au parent le démarrage de l'action
52.        super.notifyEvent(action, WORK_STARTED, null);
53.        // on commence à attendre
54.        beginWaiting();
55.        // on exécute l'action
56.        action.doWork(...);
57.        // début du monitoring
58.        beginMonitoring();
59.    }
60.
61.    private boolean isValid() {
62.        // on vérifie la validité des données saisies
63.        ...
64.    }
65.
66.    @Override
67.    public void notifyEndOfTasks() {
68.        // fin de l'attente
69.        cancelWaiting();
70.        // gestion de l'UI
71.        .....
72.    }
73.
74.    @Override
75.    public void notifyEvent(IWorker worker, int eventType, Object event) {
76.        // on passe l'évt à la classe parent
77.        super.notifyEvent(worker, eventType, event);
78.        // on gère l'évt WORK_INFO
79.        if (eventType == IBoss.WORK_INFO) {
80. ....
81.        }
82.    }
83. }
84.
85. // début de l'attente
86. private void beginWaiting() {
87.     // on met le sablier
88.     activity.setProgressBarIndeterminateVisibility(true);
89.     // état des boutons

```

```

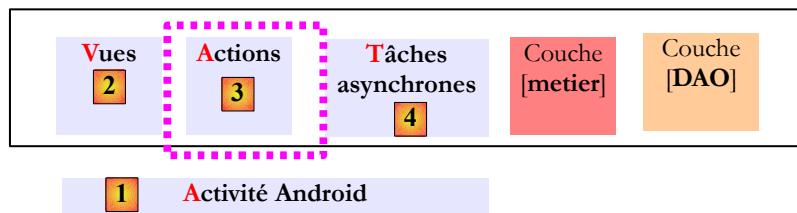
90.     btnExecuter.setVisibility(View.INVISIBLE);
91.     btnAnnuler.setVisibility(View.VISIBLE);
92.
93. }
94.
95. // fin de l'attente
96. protected void cancelWaiting() {
97.     // on cache le sablier
98.     activity.setProgressBarIndeterminateVisibility(false);
99.     // état des boutons
100.    btnExecuter.setVisibility(View.VISIBLE);
101.    btnAnnuler.setVisibility(View.INVISIBLE);
102. }
103.
104. }

```

- ligne 1 : la vue étend la classe [Vue] du modèle AVAT ;
- lignes 4, 5 : on suppose que la vue a deux boutons : [Exécuter] pour lancer une ou plusieurs actions, [Annuler] pour les annuler ;
- ligne 24 : un clic sur le bouton [Exécuter] provoque l'exécution de la méthode ligne 43 ;
- lignes 46-47 : test de la validité des données saisies ;
- ligne 50 : la vue instancie la ou les actions qu'elle va lancer. Elle demande à la fabrique la référence des actions. Elle donne deux informations :
 - **this** : pour indiquer que la vue est leur patron,
 - "**Action_01**" : leur identifiant de travailleur ;
- ligne 52 : la vue indique à sa classe parent [Vue] que l'action passe à l'état démarré ;
- ligne 54 : elle active les éléments de l'UI qui montrent à l'utilisateur qu'une opération potentiellement longue a commencé. Le bouton [Exécuter] peut par exemple être remplacé par le bouton [Annuler] ;
- ligne 56 : elle lance l'action. Celle-ci est normalement asynchrone. La vue récupère donc la main. A partir de ce moment, l'utilisateur peut cliquer sur le bouton [Annuler] ;
- ligne 58 : la vue demande à son parent [Vue] de surveiller la fin des actions lancées. Celui-ci le fera en appelant la méthode [notifyEndOfTasks] de la ligne 67 ;
- ligne 75 : la vue reçoit des notifications des actions lancées. Elle ne gère pas les notifications [WORK_TERMINATED]. Elle les passe directement à sa classe parent ;
- ligne 79 : les notifications WORK_INFO lui sont destinées. Elle les gère ;
- ligne 67 : lorsque toutes les actions lancées auront envoyé la notification [WORK_TERMINATED], le parent [Vue] appellera la méthode [notifyEndOfTasks] de sa classe fille. Celle-ci modifiera l'UI pour indiquer que l'attente est terminée et utilisera les informations qu'elle a reçues pour soit changer l'UI soit passer à une autre vue. Les informations reçues pourront être mises dans une session ;
- ligne 32 : méthode exécutée lorsque l'utilisateur clique sur le bouton [Annuler] ;
- ligne 34 : la vue annule toutes les actions lancées ;
- ligne 36 : elle modifie l'UI pour indiquer que l'attente est terminée ;
- ligne 38 : elle fait autre chose.

Au final, cette vue est peu différente d'une vue classique Android. Les méthodes propres au modèle AVAT sont les méthodes **notifyEvent** (ligne 75) qui permet à la vue de récupérer les informations envoyées par les actions et **notifyEndOfTasks** (ligne 67) qui permet à la vue de savoir que ce qu'elle a demandé a été exécuté.

2.4.3 L'action



Une action pourrait ressembler à ce qui suit :

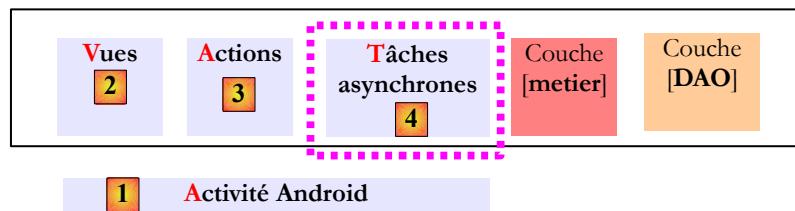
```

1. package istia.st.avat.examples.actions;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5. import java.util.concurrent.RejectedExecutionException;
6.
7. import istia.st.avat.android.Action;
8. import istia.st.avat.android.Task;
9. import istia.st.avat.core.IBoss;
10. import istia.st.avat.core.IWorker;
11. import istia.st.avat.examples.activity.Factory;
12.
13. public class Action_01 extends Action {
14.
15.     ....
16.
17.     @Override
18.     // exécution de l'action
19.     public void doWork(Object... params) {
20.         ....
21.         // on demande la tâche à la Factory
22.         Task task = (Task) factory.getObject(Factory.TASK_01, this,
23.             String.format("[%s]Task_01-%02d", bossId, i));
24.         try {
25.             // on exécute la tâche
26.             task.doWork(...);
27.         } catch (RejectedExecutionException ex) {
28.             // on passe la tâche à l'état TERMINATED
29.             notifyEvent(task, IBoss.WORK_TERMINATED,null);
30.             // elle a été rejetée - on passe l'exception au boss
31.             boss.notifyEvent(this, IBoss.WORK_INFO, ex);
32.         }
33.         ....
34.         // on commence le monitoring de la fin des tâches lancées
35.         beginMonitoring();
36.     }
37.     @Override
38.     public void notifyEndOfTasks() {
39.         // fin des tâches
40.         ....
41.         // on envoie la notification WORK_TERMINATED au boss (la vue)
42.         boss.notifyEvent(this, IBoss.WORK_TERMINATED, ...);
43.     }
44.
45.     // notification d'un événement
46.     @Override
47.     public void notifyEvent(IWorker worker, int eventType, Object event) {
48.         // on passe l'événement au parent
49.         super.notifyEvent(worker, eventType, event);
50.         // on gère l'événement INFO
51.         if (eventType == IBoss.WORK_INFO) {
52.             // on peut gérer localement l'événement INFO
53.             // ou la passer au boss (la vue)
54.             ...
55.         }
56.     }
57.
58. }
```

- ligne 13 : l'action étend la classe [Action] du modèle AVAT ;
- l'action travailleur doit implémenter la méthode [doWork] de la ligne 19 ;

- l'action patron doit elle implémenter les méthodes [notifyEvent] (ligne 47) et [notifyEndOfTasks] (ligne 38) ;
- ligne 19 : l'action a reçu des paramètres de la vue pour s'exécuter. Pour faire le travail demandé elle peut avoir besoin de 0 à N tâches asynchrones. Si 0, l'action demandée n'est pas asynchrone. L'action fait alors le travail elle-même et lorsqu'elle a fini, elle envoie les notifications WORK_INFO et WORK_TERMINATED à son boss (la vue) ;
- si elle a en besoin, l'action va demander à la fabrique des références de tâches (ligne 22). L'action passe deux paramètres à la tâche créée :
 - le boss de la tâche (this),
 - un identifiant ;
- lignes 23-31 : elle va lancer l'exécution de la tâche asynchrone. Celle-ci va s'exécuter dans un thread pris dans un pool de threads. Celui-ci peut être épuisé et alors l'exécution va être refusée avec l'exception de la ligne 26 ;
- ligne 28 : malgré l'exception, la tâche a pu passer dans l'état [WORK_STARTED], c'est même probable. Nous allons voir en effet que la tâche envoie cette notification dans sa méthode [onPreExecute]. Je n'en ai pas eu confirmation dans la documentation mais tout laisse croire que la ligne 25 se poursuit par l'exécution de cette méthode dans le thread de l'UI. Donc la tâche est probablement dans l'état [WORK_STARTED]. On la passe alors dans l'état [WORK_TERMINATED]. Si elle n'était pas dans l'état [WORK_STARTED], c'est une opération nulle qui ne provoque pas de plantage. Il est important que chaque notification [WORK_STARTED] soit équilibrée par une notification [WORK_TERMINATED] sinon la liste des travailleurs ne sera jamais vide et le boss ne recevra jamais la notification [endOfTasks] ;
- ligne 30 : on passe l'exception au boss : les informations remontées par les tâches asynchrones peuvent être également des exceptions. Il est important de signaler à la vue qu'une action n'a pu être exécutée.
- ligne 26 : dans les tests, je n'ai vu apparaître que cette exception. Si d'autres exceptions devaient apparaître, on devrait les gérer également ;
- ligne 34 : une fois toutes les tâches lancées, l'action va se mettre en attente de leur fin ;
- ligne 47 : elle va voir passer les notifications [WORK_STARTED, WORK_INFO, WORK_TERMINATED] des tâches lancées. Elle ne gère elle-même que la seconde et passe les autres à sa classe parent ;
- ligne 38 : l'action reçoit le signal que toutes les tâches sont terminées. Elle va faire ce qu'elle a à faire, peut-être envoyer une notification [WORK_INFO] à son boss puis dans tous les cas, la notification [WORK_TERMINATED].

2.4.4 La tâche asynchrone



Une tâche asynchrone pourrait ressembler à ce qui suit :

```

1. package istia.st.avat.examples.tasks;
2.
3. import istia.st.avat.android.Task;
4. import istia.st.avat.core.IBoss;
5. ...
6.
7. public class Task_01 extends Task {
8.
9.     // info renvoyée par la tâche
10.    private Object info;
11.
12.    @Override
13.    public void onPreExecute(){
14.        // on commence le travail
15.        boss.notifyEvent(this, IBoss.WORK_STARTED, null);
16.    }
17.
18.    @Override
19.    protected Void doInBackground(Object... params) {

```

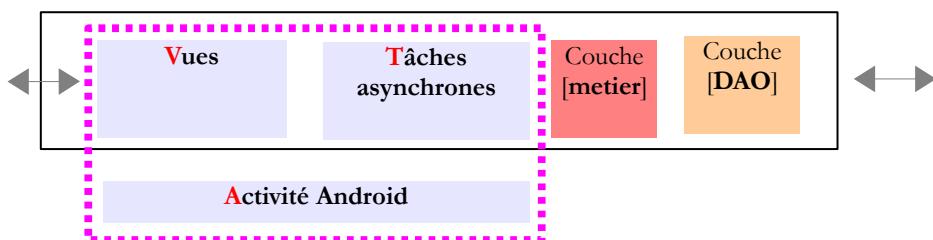
```

20.    // préparation
21.    .....
22.    try {
23.        // exécution de la couche [métier]
24.        .....
25.        info = ... ;
26.    } catch (Exception ex) {
27.        info = ex ;
28.    }
29. }
30.
31. @Override
32. // traitement fin de tâche dans le thred de l'UI
33. protected void onPostExecute(Void result) {
34.     // on passe l'info au boss
35.     boss.notifyEvent(this, IBoss.WORK_INFO, info);
36.     // on a terminé le travail
37.     boss.notifyEvent(this, IBoss.WORK_TERMINATED, null);
38. }
39.
40. }

```

- ligne 7 : la tâche étend la classe [Task] du modèle AVAT. Celle-ci étend la classe Android [AsyncTask<Object, Object, Void>]. On redéfinit trois méthodes de cette classe, [onPreExecute] ligne 13, [doInBackground] ligne 19, [onPostExecute] ligne 33 ;
- ligne 10 : l'information rendue par la tâche ;
- ligne 13 : la méthode exécutée dans le thread de l'UI avant la méthode [doInBackground] de la ligne 19. On l'utilise pour envoyer la notification [WORK_STARTED] au boss (l'action) ;
- ligne 19 : le cœur de la tâche asynchrone. Avec les paramètres que lui a passés l'action, elle exécute une ou plusieurs méthodes de la couche [métier]. Cette exécution est faite dans un thread différent de celui de l'UI. On récupère une information utile (ligne 25) ou une exception (ligne 27), qu'on mémorise dans l'objet de la ligne 10 ;
- ligne 33 : la méthode exécutée dans le thread de l'UI lorsque [doInBackground] est terminée. La tâche envoie à son patron (l'action) l'info qu'elle a mémorisée en ligne 10 ainsi que la notification [WORK_TERMINATED].

Nous avons dit que le modèle AVAT pouvait être rétréci en un modèle AVT :



Nous sommes toujours dans un modèle patron / travailleurs :

- une vue [Vue] est un patron [IBoss]. Son équipe de travailleurs est une équipe de [Task] ;
- la tâche est un travailleur [IWorker]. Son patron est une vue [Vue].

La partie [IBoss] de l'action disparue va être déportée sur la vue et sa partie [IWorker] sur la tâche. Nous utiliserons ce modèle allégé sur le dernier exemple de ce document.

L'essentiel a été dit. Nous pouvons passer aux exemples d'utilisation du modèle.

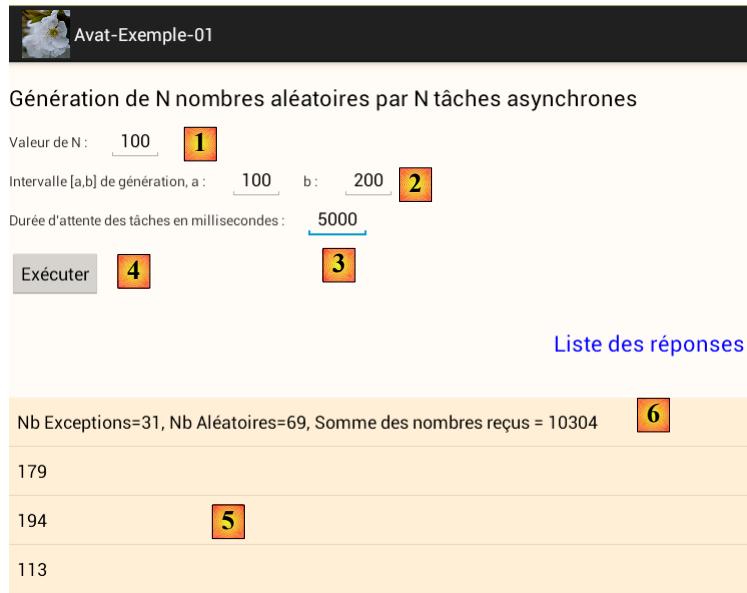
Notes :

- pour ne pas obscurcir l'objectif de ce document qui est de découvrir le modèle AVAT, nous ne détaillerons pas la construction des vues à l'aide de fichiers XML. Nous donnerons simplement le type et le nom des composants présents sur la vue ;
- les applications ont été construites **pour des tablettes**. Aucun effort n'a été fait pour que les vues s'affichent élégamment également sur des smartphones. Dans un cas réel, on aurait bien évidemment fait cet effort.

3 AVAT- Exemple 1

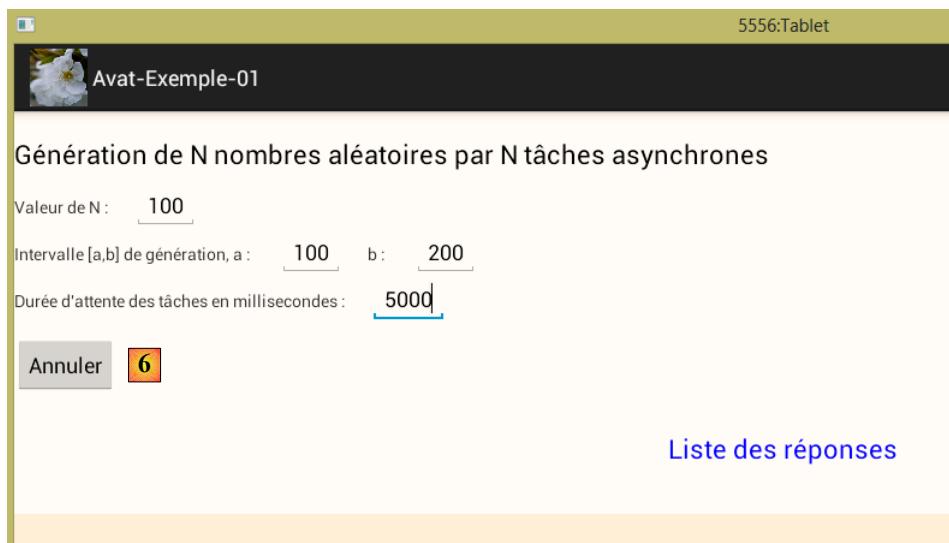
3.1 Le projet

Nous nous proposons de créer une application Android avec l'unique vue suivante :



- l'application lance N [1] tâches asynchrones qui génèrent chacune un nombre aléatoire dans un intervalle [a,b] [2]. Avec une probabilité d'1/3, la tâche peut générer une exception.
- le bouton [4] lance une action unique qui va à son tour lancer les N tâches asynchrones ;
- afin de pouvoir annuler les N tâches lancées, on leur impose un délai d'attente avant de rendre leur réponse, exprimé en millisecondes [3]. Sur l'exemple, le délai de 5 secondes fait que les N tâches sont lancées et toutes attendent 5 secondes avant de faire le travail qui leur est demandé ;
- en [5] remontent les informations produites par les tâches, nombre ou exception ;
- la ligne [6] récapitule ce qui a été reçu. Il faut vérifier qu'on a bien reçu les N réponses des N tâches asynchrones et accessoirement on pourra vérifier la somme des nombres reçus.

Dès que les N tâches ont été lancées, un bouton [Annuler] [6] remplace le bouton [Exécuter]. Il permet d'annuler les tâches lancées :



Outre les résultats affichés sur la vue, on vérifiera les logs. Par exemple pour 2 tâches on a les logs suivants :

```
1. 03-14 02:08:02.847: I/Action_01(7942): eventType=WORK_STARTED, taskId=Action_01, time=02:08:02:849
2. 03-14 02:08:02.847: I/[Action_01]Task_01-00(7942): eventType=WORK_STARTED, taskId=[Action_01]Task_01-00, time=02:08:02:849
3. 03-14 02:08:02.847: I/[Action_01]Task_01-01(7942): eventType=WORK_STARTED, taskId=[Action_01]Task_01-01, time=02:08:02:849
4. 03-14 02:08:07.986: I/[Action_01]Task_01-01(7942): eventType=WORK_INFO, taskId=[Action_01]Task_01-01, time=02:08:07:989
5. 03-14 02:08:07.986: I/Action_01(7942): eventType=WORK_INFO, taskId=Action_01, time=02:08:07:989
6. 03-14 02:08:07.986: I/[Action_01]Task_01-01(7942): eventType=WORK_TERMINATED, taskId=[Action_01]Task_01-01, time=02:08:07:989
7. 03-14 02:08:07.986: I/[Action_01]Task_01-00(7942): eventType=WORK_INFO, taskId=[Action_01]Task_01-00, time=02:08:07:990
8. 03-14 02:08:07.986: I/Action_01(7942): eventType=WORK_INFO, taskId=Action_01, time=02:08:07:990
9. 03-14 02:08:07.986: I/[Action_01]Task_01-00(7942): eventType=WORK_TERMINATED, taskId=[Action_01]Task_01-00, time=02:08:07:990
10. 03-14 02:08:07.986: I/Action_01(7942): eventType=WORK_TERMINATED, taskId=Action_01, time=02:08:07:990
```

- ligne 1 : l'action [Action_01] a été lancée ;
- ligne 2 : la tâche [Action_01]Task_01-00 a été lancée ;
- ligne 3 : la tâche [Action_01]Task_01-01 a été lancée ;
- ligne 4 : la tâche [Action_01]Task_01-01 a généré une information ;
- ligne 5 : l'action [Action_01] a généré une information ;
- ligne 6 : la tâche [Action_01]Task_01-01 est terminée ;
- ligne 7 : la tâche [Action_01]Task_01-00 a généré une information ;
- ligne 8 : l'action [Action_01] a généré une information ;
- ligne 9 : la tâche [Action_01]Task_01-00 est terminée ;
- ligne 10 : l'action [Action_01] est terminée ;

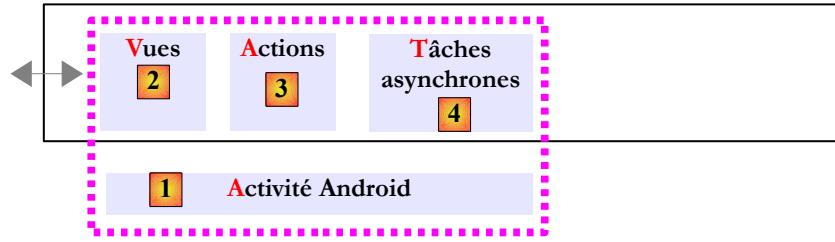
Dans le cas d'annulation de ces deux tâches, on a les logs suivants :

```
1. 03-14 02:13:54.871: I/Action_01(7942): eventType=WORK_STARTED, taskId=Action_01, time=02:13:54:872
2. 03-14 02:13:54.871: I/[Action_01]Task_01-00(7942): eventType=WORK_STARTED, taskId=[Action_01]Task_01-00, time=02:13:54:873
3. 03-14 02:13:54.871: I/[Action_01]Task_01-01(7942): eventType=WORK_STARTED, taskId=[Action_01]Task_01-01, time=02:13:54:873
4. 03-14 02:13:56.280: I/[Action_01]Task_01-01(7942): taskId=[Action_01]Task_01-01 canceled at time=02:13:56:283
5. 03-14 02:13:56.280: I/[Action_01]Task_01-00(7942): taskId=[Action_01]Task_01-00 canceled at time=02:13:56:284
6. 03-14 02:13:56.280: I/Action_01(7942): taskId=Action_01 canceled at time=02:13:56:284
```

- ligne 1 : l'action [Action_01] a été lancée ;
- ligne 2 : la tâche [Action_01]Task_01-00 a été lancée ;
- ligne 3 : la tâche [Action_01]Task_01-01 a été lancée ;
- ligne 4 : la tâche [Action_01]Task_01-01 a été annulée ;
- ligne 5 : la tâche [Action_01]Task_01-00 a été annulée ;
- ligne 6 : l'action [Action_01] a été annulée ;

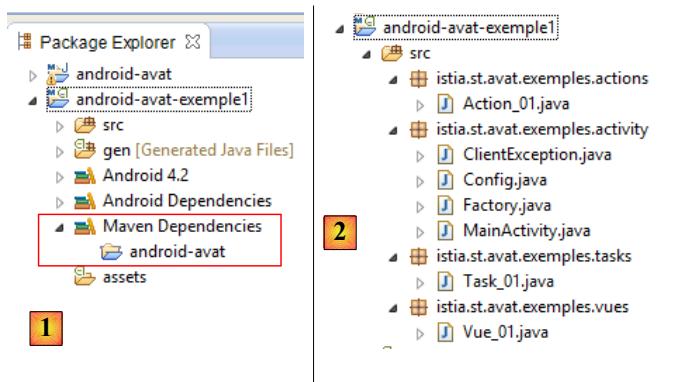
3.2 L'architecture

L'architecture de l'application est la suivante :



Il n'y a pas de couche [métier].

3.3 Le projet Eclipse



- en [1], le projet [android-avat-exemple1] est un projet Maven ayant une dépendance sur le projet [android-avat] ;
- en [2], les sources du projet :
 - le package [istia.st.avat-exemples.vues] rassemble les vues du projet, ici une vue ;
 - le package [istia.st.avat-exemples.actions] rassemble les actions du projet, ici une action ;
 - le package [istia.st.avat-exemples.tasks] rassemble les tâches du projet, ici une tâche ;
 - le package [istia.st.avat-exemples.activity] contient l'activité Android [MainActivity], la fabrique d'objets [Factory], une classe de configuration [Config] et une classe d'exception [ClientException].

3.4 Le manifeste de l'application Android

Le fichier [AndroidManifest.xml] du projet est le suivant :

```

1. <?xml version="1.0" encoding="utf-8"?>
2. <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3.   package="istia.st.avat.exemples"
4.   android:versionCode="1"
5.   android:versionName="1.0" >
6.
7.   <uses-sdk
8.     android:minSdkVersion="11"
9.     android:targetSdkVersion="16" />
10.
11.  <application
12.    android:allowBackup="true"
13.    android:icon="@drawable/fleur"
14.    android:label="@string/app_name"
15.    android:theme="@style/AppTheme" >
16.    <activity
17.      android:name="istia.st.avat.exemples.activity.MainActivity"

```

```

18.        android:label="@string/app_name"
19.        android:windowSoftInputMode="stateHidden">
20.            <intent-filter>
21.                <action android:name="android.intent.action.MAIN" />
22.
23.                <category android:name="android.intent.category.LAUNCHER" />
24.            </intent-filter>
25.        </activity>
26.    </application>
27.
28. </manifest>

```

C'est un fichier classique pour toute personne ayant développé des applications Android. Notons simplement la ligne 19 qui permet de ne pas voir apparaître un clavier logiciel lors de l'affichage de la vue.

3.5 Les dépendances Maven

Le fichier [pom.xml] du projet Android est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
   v4_0_0.xsd">
4.   <modelVersion>4.0.0</modelVersion>
5.   <groupId>istia.st.avat.examples</groupId>
6.   <artifactId>android-avat-exemple1</artifactId>
7.   <version>1.0-SNAPSHOT</version>
8.   <packaging>apk</packaging>
9.
10.  <properties>
11.      <platform.android.version>4.1.1.4</platform.android.version>
12.      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
13.      <android.sdk.path>D:\\\\Programs\\\\devjava\\\\Android\\\\sdk</android.sdk.path>
14.      <spring-android-version>1.0.1.RELEASE</spring-android-version>
15.  </properties>
16.
17.  <dependencies>
18.      <dependency>
19.          <groupId>com.google.android</groupId>
20.          <artifactId>android</artifactId>
21.          <version>${platform.android.version}</version>
22.          <scope>provided</scope>
23.      </dependency>
24.      <dependency>
25.          <groupId>istia.st.android</groupId>
26.          <artifactId>android-avat</artifactId>
27.          <version>1.0-SNAPSHOT</version>
28.          <scope>compile</scope>
29.      </dependency>
30.  </dependencies>
31.
32.  <build>
33.      <plugins>
34.          <plugin>
35.              <groupId>com.jayway.maven.plugins.android.generation2</groupId>
36.              <artifactId>android-maven-plugin</artifactId>
37.              <version>3.1.1</version>
38.              <configuration>
39.                  ...
40.              </configuration>
41.              <extensions>true</extensions>
42.          </plugin>

```

```

43.
44.      <plugin>
45.          <artifactId>maven-compiler-plugin</artifactId>
46.          <version>2.3.2</version>
47.          <configuration>
48.              <source>1.6</source>
49.              <target>1.6</target>
50.          </configuration>
51.      </plugin>
52.  </plugins>
53. </build>
54. </project>

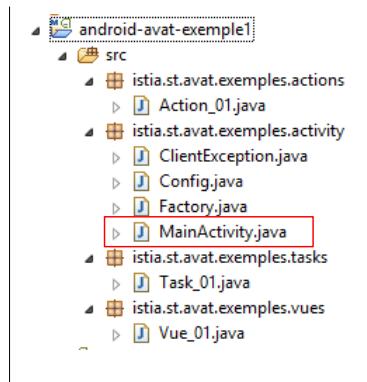
```

Le projet a deux dépendances :

- lignes 18-23 : la dépendance sur Android ;
- lignes 24-29 : la dépendance sur le projet AVAT ;

Lignes 33-52, on trouve la définition des plugins nécessaires à la construction du projet Maven Android. Nous ne nous attarderons pas dessus et nous ne les mentionnerons plus. Le lecteur les trouvera dans les codes source distribués avec ce document.

3.6 L'activité Android



L'activité Android [MainActivity] est la suivante :

```

1. package istia.st.avat.examples.activity;
2.
3. import ...
4.
5. public class MainActivity extends FragmentActivity {
6.
7.     // la factory
8.     private Factory factory;
9.
10.    @Override
11.    protected void onCreate(Bundle savedInstanceState) {
12.        super.onCreate(savedInstanceState);
13.        // le sablier
14.        requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
15.        setProgressBarIndeterminateVisibility(false);
16.        // template qui accueille les vues
17.        setContentView(R.layout.main);
18.        // la factory
19.        factory = new Factory(this, new Config());
20.        // on définit l'unique vue
21.        Vue_01 vue = (Vue_01) factory.getObject(Factory.VUE_01, null, "Vue_01");

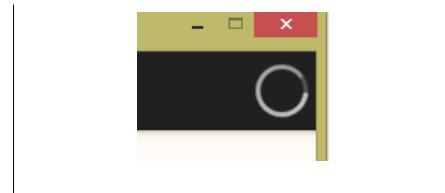
```

```

22.    // on en fait la vue courante
23.    FragmentTransaction fragmentTransaction = getFragmentManager().beginTransaction();
24.    fragmentTransaction.replace(R.id.container, vue);
25.    fragmentTransaction.commit();
26. }
27.
28. }

```

- ligne 5 : l'activité est de type [FragmentActivity] ;
- ligne 8 : la fabrique d'objets du projet. C'est l'activité qui l'instancie ;
- lignes 14-15 : mise en place d'un sablier. Ceci doit être fait avant que l'activité ne définisse son interface visuelle, ligne 17. Le sablier s'installe en haut et à droite de la tablette. Il a la forme suivante :



- ligne 17 : l'interface visuelle est construite à partir du fichier [main.xml]. Celui-ci est le suivant :

```

1. <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.   xmlns:tools="http://schemas.android.com/tools"
3.   android:id="@+id/container"
4.   android:layout_width="match_parent"
5.   android:layout_height="match_parent"
6.   android:background="@color/floral_white"
7.   tools:context=".MainActivity"
8.   tools:ignore="MergeRootFrame" />

```

- ligne 1 : on a un unique composant, un [FrameLayout] avec rien dedans ;
- ligne 3 : on notera son identifiant.

Retour au code de l'activité :

- ligne 19 : la fabrique d'objets est instanciée avec un constructeur à deux paramètres :
 - le 1^{er} paramètre est l'activité elle-même ;
 - le second est une instance de la classe de configuration de l'application ;

La classe [Config] est la suivante :

```

1. public class Config {
2.
3.     // le mode verbeux ou non
4.     private boolean verbose = true;
5.
6.     // getters et setters
7.     public boolean isVerbose() {
8.         return verbose;
9.     }
10. }

```

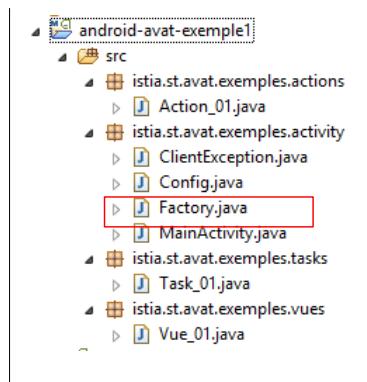
Elle définit le mode verbeux ou non de l'application. En mode débogage, on mettra *verbose* à vrai.

Retour au code de l'activité :

- ligne 21 : une instance de la première vue à afficher est demandée à la fabrique d'objets à l'aide de trois paramètres :
 - n° de l'objet demandé,
 - le boss de l'objet demandé s'il y en a un. Une vue n'a pas de boss,
 - l'identifiant de l'objet demandé ;

- lignes 23-25 : la vue est affichée. Ligne 24, la vue vient remplacer l'objet identifié par *container* dans la vue actuelle. On se rappelle que cette vue était pour l'instant celle d'un [FrameLayout] nommé *container*. Ce [FrameLayout] est alors remplacé par la vue *Vue_01*.

3.7 La fabrique d'objets



La fabrique d'objets est centrale à une application AVAT. Nous y reviendrons à plusieurs reprises. Pour l'instant, montrons le code d'instanciation de la vue "Vue_01" :

```

1. package istia.st.avat.exemples.activity;
2.
3. ...
4.
5. public class Factory implements IFactory {
6.
7.     // constantes
8.     public static final int CONFIG = -1;
9.     public static final int TASK_01 = 1;
10.    public static final int ACTION_01 = 3;
11.    public static final int VUE_01 = 4;
12.
13.    // ----- DEBUT SINGLETONS
14.    // la configuration
15.    private Config config;
16.    private boolean verbose;
17.
18.    // les vues
19.    private Vue vue_01;
20.
21.    // l'activité principale
22.    private Activity activity;
23.
24.    // ----- FIN SINGLETONS
25.
26.    // constructeur
27.    public Factory(Activity activity, Config config) {
28.        // l'activité
29.        this.activity = activity;
30.        // la configuration
31.        this.config = config;
32.        verbose = config.isVerbose();
33.    }
34.
35.    @Override
36.    public Object getObject(int id, Object... params) {
37.        switch (id) {

```

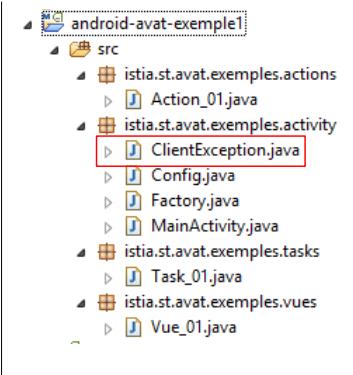
```

38.     case CONFIG:
39.         return config;
40.     case TASK_01:
41.         return getTask_01(params);
42.     case ACTION_01:
43.         return getAction_01(params);
44.     case VUE_01:
45.         return getVue_01(params);
46.     }
47.     return null;
48. }
49.
50. private Object getVue_01(Object... params) {
51.     // la vue est un singleton
52.     if (vue_01 == null) {
53.         // la vue
54.         vue_01 = new Vue_01();
55.         // sa factory
56.         vue_01.setFactory(this);
57.         // son équipe
58.         ITeam team = new Team();
59.         team.setMonitor(vue_01);
60.         vue_01.setTeam(team);
61.         // son activité
62.         vue_01.setActivity(activity);
63.         // son mode verbeux ou non
64.         vue_01.setVerbose(verbose);
65.         // son id
66.         vue_01.setBossId((String) params[0]);
67.     }
68.     // on rend la référence
69.     return vue_01;
70. }
71. ...
72.
73. }

```

- ligne 5 : la fabrique implémente l'interface [IFactory] ;
- lignes 8-11 : les constantes entières qui identifient les objets construits par la fabrique ;
- lignes 13-24 : les singletons de la fabrique. Ici, la classe de configuration (lignes 15-16), l'unique vue de l'application (ligne 19), l'activité Android (ligne 22) ;
- lignes 27-33 : le constructeur de la fabrique. Il reçoit deux paramètres :
 - l'activité Android de l'application,
 - la classe de configuration de l'application ;
- lignes 36-48 : la sélection de l'objet à créer ou recycler sera faite à l'aide d'un switch qui peut être assez imposant. On peut sans doute l'éviter ;
- ligne 50 : le code d'instanciation de la vue " Vue_01 ". On reçoit un paramètre : la référence du boss de la vue qui va être créée (l'activité a passé un pointeur *null*) ;
- ligne 52 : la vue n'est pas régénérée si elle l'a déjà été ;
- ligne 54 : instantiation de la vue ;
- lignes 55-66 : on lui injecte la fabrique (ligne 56), une équipe (lignes 58-60), l'activité Android (ligne 62), le mode verbeux de l'application (ligne 64) et un identifiant (66).

3.8 La classe d'exception



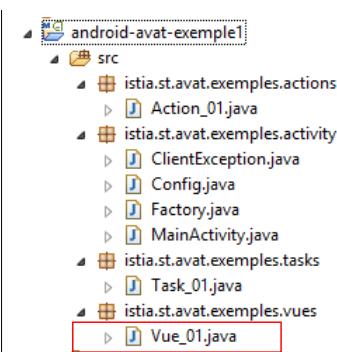
Une application Java définit souvent ses propres exceptions. Ici, ce sera la classe [ClientException] suivante :

```

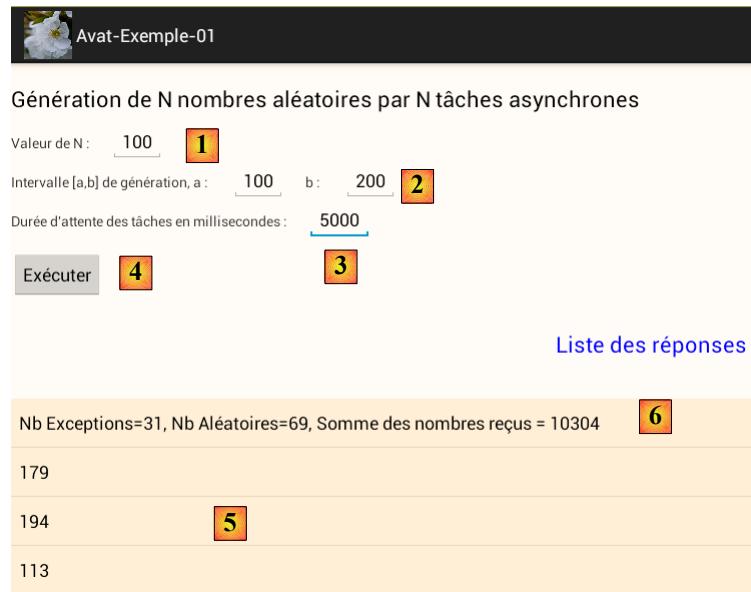
1. package istia.st.avat.exemples.activity;
2.
3. public class ClientException extends RuntimeException {
4.
5.     private static final long serialVersionUID = 1L;
6.
7.     public ClientException() {
8.     }
9.
10.    public ClientException(String detailMessage) {
11.        super(detailMessage);
12.    }
13.
14.    public ClientException(Throwable throwable) {
15.        super(throwable);
16.    }
17.
18.    public ClientException(String detailMessage, Throwable throwable) {
19.        super(detailMessage, throwable);
20.    }
21.
22. }
```

- ligne 3, l'exception dérive de [RuntimeException]. C'est donc une exception non contrôlée par le compilateur. Il n'y a pas d'obligation de la gérer par un *try / catch* ni de la déclarer par les mots clés *throws ClientException*.

3.9 La vue [Vue_01]



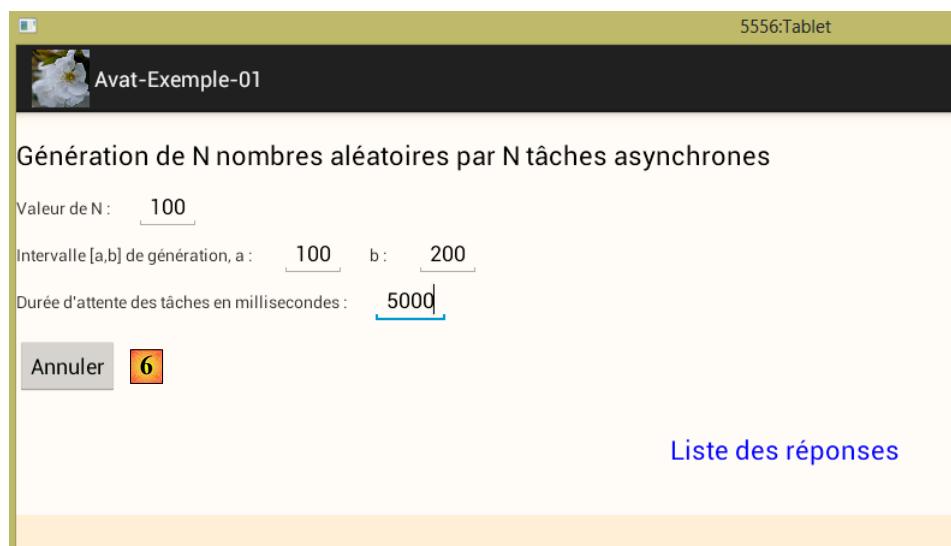
La vue interagit avec l'utilisateur :



Ses composants sont les suivants :

N°	Id	Type	Rôle
1	edtNbAleas	EditText	nombre de nombres aléatoires à générer dans l'intervalle entier [a,b]
2	edtA	EditText	valeur de a
2	edtB	EditText	valeur de b
3	edtSleepTimeEditText	durée d'attente des threads asynchrones	
4	btnExécuter	Button	lance la génération des nombres
5	ListView	lstReponses	liste des nombres générés dans l'ordre inverse de leur génération. On voit d'abord le dernier généré ; récapitulatif de la génération
6			

Lorsque les tâches ont été lancées, l'UI change :



N°	Id	Type	Rôle
6	btnAnnuler	Button	annule la génération des nombres

Le code de la vue est le suivant :

```

1. package istia.st.avat.examples.vues;
2.
3. ...
4.
5. public class Vue_01 extends Vue {
6.
7.     // son identité
8.     private String bossId;
9.
10.    // les éléments de l'interface visuelle
11.    private ListView listRéponses;
12.    private Button btnExecuter;
13.    private Button btnAnnuler;
14.    private EditText edtNbAleas;
15.    private EditText edtA;
16.    private EditText edtB;
17.    private EditText edtSleepTime;
18.    private TextView txtErrorAleas;
19.    private TextView txtErrorIntervalle;
20.    private TextView txtErrorSleepTime;
21.
22.    // liste des réponses à une commande
23.    private List<String> réponses = new ArrayList<String>();
24.    // les saisies
25.    private int nbAleas;
26.    private int a;
27.    private int b;
28.    private int sleepTime;
29.    // les evts
30.    private int nbExceptions = 0;
31.    private int nbAleasReçus = 0;
32.
33.    @Override
34.    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
   savedInstanceState) {
35.        // on crée la vue du fragment à partir de sa définition XML
36.        return inflater.inflate(R.layout.vue_01, container, false);
37.    }
38.
39.    @Override
40.    public void onActivityCreated(Bundle savedInstanceState) {
41.        // parent
42.        super.onActivityCreated(savedInstanceState);
43.
44.        // zones de saisie
45.        edtNbAleas = (EditText) activity.findViewById(R.id.edt_nbaleas);
46.        edtA = (EditText) activity.findViewById(R.id.edt_a);
47.        edtB = (EditText) activity.findViewById(R.id.edt_b);
48.        edtSleepTime = (EditText) activity.findViewById(R.id.edt_sleepTime);
49.
50.        // les messages d'erreur
51.        txtErrorAleas = (TextView) activity.findViewById(R.id.txt_errorNbAleas);
52.        txtErrorIntervalle = (TextView) activity.findViewById(R.id.txt_errorIntervalle);
53.        txtErrorSleepTime = (TextView) activity.findViewById(R.id.txt_errorSleepTime);
54.
55.        // au départ pas de messages d'erreur
56.        txtErrorAleas.setText("");
57.        txtErrorIntervalle.setText("");
58.        txtErrorSleepTime.setText("");
59.
60.        // bouton Exécuter
61.        btnExecuter = (Button) activity.findViewById(R.id.btn_Executer);
62.        btnExecuter.setOnClickListener(new OnClickListener() {

```

```

63.     @Override
64.     public void onClick(View arg0) {
65.         doExecuter();
66.     }
67. });
68.
69. // bouton Annuler
70. btnAnnuler = (Button) activity.findViewById(R.id.btn_Annuler);
71. btnAnnuler.setOnClickListener(new OnClickListener() {
72.     @Override
73.     public void onClick(View arg0) {
74.         // on annule l'action
75.         cancelAll();
76.         // et l'attente
77.         cancelWaiting();
78.     }
79. });
80.
81. // réponses des actions et tâches
82. listRéponses = (ListView) activity.findViewById(R.id.Lst_reponses);
83. // état des boutons
84. btnExecuter.setVisibility(View.VISIBLE);
85. btnAnnuler.setVisibility(View.INVISIBLE);
86. }
87.
88. protected void doExecuter() {
89. ...
90. }
91.
92. // on vérifie la validité des données saisies
93. private boolean isPageValid() {
94. ...
95. }
96.
97. @Override
98. public void notifyEndOfTasks() {
99. ...
100. }
101.
102. @Override
103. public void notifyEvent(IWorker worker, int eventType, Object event) {
104. ...
105. }
106.
107. // début de l'attente
108. private void beginWaiting() {
109. ...
110. }
111.
112. // fin de l'attente
113. protected void cancelWaiting() {
114. ...
115. }
116.
117. // getters et setters
118. ...
119. }

```

On aura dans nos exemples toujours la même structure de vue :

- ligne 40 : la méthode [onActivityCreated] pour déclarer les différents éléments de l'interface graphique ainsi que les gestionnaires d'événements ;
- ligne 88 : la méthode associée au clic du bouton [Exécuter] ;

- ligne 93 : la méthode qui vérifie la validité des données. Elle sera présente mais non détaillée. Le lecteur la trouvera dans les codes source livrés avec ce document ;
- ligne 98 : la méthode [notifyEndOfTasks] exécutée par la classe parent [Vue] lorsque toutes les actions lancées ont été exécutées ;
- ligne 103 : la méthode [notifyEvent] qui voit passer toutes les notifications envoyées par les actions lancées ;
- lignes 108 et 113 : les méthodes qui gèrent le début et la fin de l'attente. Nous les détaillerons ici mais plus dans les exemples qui suivront.

3.9.1 Méthode [doExécuter]

C'est la méthode qui va lancer l'action de génération des nombres. Son code est le suivant :

```

1. protected void doExécuter() {
2.     // on efface les éventuels msg d'erreur précédents
3.     txtErrorAleas.setText("");
4.     txtErrorIntervalle.setText("");
5.     txtErrorSleepTime.setText("");
6.     // on teste la validité des saisies
7.     if (!isValidPage()) {
8.         return;
9.     }
10.    // on efface les réponses précédentes
11.    réponses.clear();
12.    // on exécute Action_01 de façon asynchrone
13.    IAction action = (IAction) factory.getObject(Factory.ACTION_01, this, "Action_01");
14.    // on signale au parent le démarrage de l'action
15.    super.notifyEvent(action, IBoss.WORK_STARTED, null);
16.    // on commence à attendre
17.    beginWaiting();
18.    // raz compteurs
19.    nbAleasReçus = 0;
20.    nbExceptions = 0;
21.    // on exécute l'action (nbAleas,a,b, sleepTime)
22.    action.doWork(Integer.parseInt(edtNbAleas.getText().toString()),
   Integer.parseInt(edtA.getText().toString()),
23.                  Integer.parseInt(edtB.getText().toString()),
   Integer.parseInt(edtSleepTime.getText().toString()));
24.    // début du monitoring
25.    beginMonitoring();
26. }
```

- lignes 3-5 : la vue comporte des composants [TextView] d'affichage d'erreurs de saisie. Ils n'ont pas été présentés ;
- lignes 7-9 : si les données saisies ne sont pas valides, on retourne à l'UI (ligne 8) ;
- ligne 13 : on demande à la fabrique l'action [Action_01]. Dans la fabrique, le code d'instanciation de l'action est le suivant :

```

1. // une action reçoit deux paramètres : IBoss, id
2. private Object getAction_01(Object[] params) {
3.     // instantiation
4.     IAction action = new Action_01();
5.     // initialisation
6.     // le boss
7.     action.setBoss((IBoss) params[0]);
8.     // l'équipe
9.     ITeam team = new Team();
10.    team.setMonitor(action);
11.    action.setTeam(team);
12.    // identités
13.    String id=(String) params[1];
14.    action.setWorkerId(id);
15.    action.setBossId(id);
16.    // la factory
17.    action.setFactory(this);
```

```

18.    // le mode verbeux ou non
19.    action.setVerbose(verbose);
20.    // fin
21.    return action;
22. }

```

- l'action est créée à chaque fois qu'on en demande une référence. Ca peut être discuté. Un singleton pourrait peut-être faire l'affaire. Il faut voir dans quelles conditions il est appelé ;
- ligne 2 : l'action reçoit deux paramètres pour se construire : une référence sur son boss (une vue) et un identifiant ;
- ligne 4 : l'action est instanciée. On y injecte ensuite : son boss (ligne 7), une équipe (lignes 9-11), un identifiant de travailleur et de boss (lignes 13-15), la fabrique (ligne 17), l'état de verbosité de l'application (ligne 19) ;

Retour au code de la méthode [doExécuter] :

- ligne 15 : on passe l'action à l'état démarré. Elle n'est pas encore démarrée mais elle va bientôt l'être. On pourrait déplacer ce code dans l'action ;
- ligne 17 : on modifie l'UI pour signaler l'attente de quelque chose ;
- ligne 22 : l'action [Action_01] est lancée. On lui passe les informations suivantes : le nombre [nbAleas] de nombres aléatoires à générer, l'intervalle [a,b] de génération, la durée [sleepTime] d'attente des threads asynchrones. Un inconvénient d'AVAT est son utilisation assez fréquente d'objets non typés. Le compilateur ne peut alors aider le développeur dans l'écriture de la ligne 22. Le risque est alors de ne pas envoyer le bon nombre de paramètres ou pas dans l'ordre attendu ;
- ligne 25 : une fois l'action lancée, on signale à la classe parent [Vue] qu'elle doit surveiller la fin des actions lancées, ici une seule.

Les méthodes [beginWaiting] et [cancelWaiting] sont les suivantes :

```

1. // début de l'attente
2. private void beginWaiting() {
3.     // on met le sablier
4.     activity.setProgressBarIndeterminateVisibility(true);
5.     // état des boutons
6.     btnExecuter.setVisibility(View.INVISIBLE);
7.     btnAnnuler.setVisibility(View.VISIBLE);
8.
9. }
10.
11. // fin de l'attente
12. protected void cancelWaiting() {
13.     // on cache le sablier
14.     activity.setProgressBarIndeterminateVisibility(false);
15.     // état des boutons
16.     btnExecuter.setVisibility(View.VISIBLE);
17.     btnAnnuler.setVisibility(View.INVISIBLE);
18. }

```

Une fois que la vue a lancé des actions, elle voit passer leurs notifications dans la méthode [notifyEvent] suivante :

```

1. @Override
2. public void notifyEvent(IWorker worker, int eventType, Object event) {
3.     // on passe l'évt à la classe parent
4.     super.notifyEvent(worker, eventType, event);
5.     // on gère l'évt WORK_INFO
6.     if (eventType == IBoss.WORK_INFO) {
7.         // exception ?
8.         if (event instanceof Exception) {
9.             réponses.add(0, ((Exception) event).getMessage());
10.            nbExceptions++;
11.        } else if (event instanceof Integer) {
12.            réponses.add(0, event.toString());
13.            nbAleasReçus++;
14.        }
15.    } else {

```

```

16.      // on gère l'évt WORK_TERMINATED
17.      if (eventType == IBoss.WORK_TERMINATED && event instanceof Integer) {
18.          réponses.add(0, String.format("Nb Exceptions=%d, Nb Aléatoires=%d, Somme des
nombres reçus = %d", nbExceptions,
19.                                      nbAleasReçus, (Integer) event));
20.      }
21.  }
22.  // rafraîchissement des réponses
23.  listRéponses.setAdapter(new ArrayAdapter<String>(activity,
    android.R.layout.simple_list_item_1, android.R.id.text1,
24.    réponses));
25. }

```

- ligne 4 : la notification doit être passée à la classe parent, en fait seulement les notifications [WORK_STARTED] et [WORK_TERMINATED]. A partir de ces deux notifications, la classe parent pourra envoyer le signal [endOfTasks] à sa fille ;
- ligne 6 : la vue générera en général la notification [WORK_INFO]. C'est le cas ici. Les tâches vont remonter deux types d'information :
 - une exception en moyenne une fois sur 3,
 - un nombre aléatoire dans l'intervalle [a,b] ;
Dans les deux cas, la notification est affichée dans la liste des réponses (lignes 9 et 12) et les compteurs respectifs sont incrémentés ;
- ligne 17 : l'action [Action_01] va envoyer la somme des nombres aléatoires générés dans sa notification [WORK_TERMINATED]. On gère ce cas ;
- ligne 23 : la liste des réponses est rafraîchie.

Lorsque les tâches sont terminées, la vue va recevoir la notification [endOfTasks] de sa classe parent. La méthode [notifyEndOfTasks] va être exécutée :

```

1.  @Override
2.  public void notifyEndOfTasks() {
3.      // fin de l'attente
4.      cancelWaiting();
5. }

```

Les résultats ayant déjà été affichés, on se contente de modifier l'UI afin de montrer que l'attente est terminée.

3.9.2 Méthode d'annulation des tâches

Un clic sur le bouton [Annuler] provoque l'exécution du code suivant :

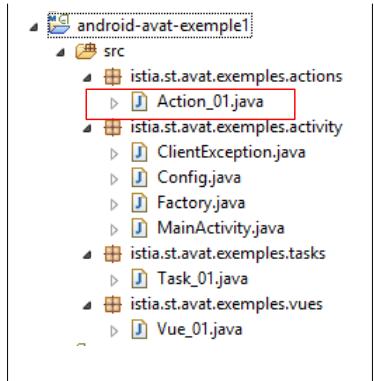
```

1.  btnAnnuler.setOnClickListener(new OnClickListener() {
2.      @Override
3.      public void onClick(View arg0) {
4.          // on annule l'action
5.          cancelAll();
6.          // et l'attente
7.          cancelWaiting();
8.      }
9. });

```

- ligne 5 : toutes les actions lancées sont annulées, ici une seulement ;
- ligne 7 : l'UI est mise à jour pour signaler la fin de l'attente.

3.10 L'action [Action_01]



Le code de l'action [Action_01] est le suivant :

```

1. package istia.st.avat.exemples.actions;
2.
3. ...
4.
5. public class Action_01 extends Action {
6.
7.     // la liste des nombres aléatoires générés par les tâches lancées
8.     List<Integer> aleas = new ArrayList<Integer>();
9.
10.    @Override
11.    // exécution de l'action (aleas, a, b, sleepTime)
12.    public void doWork(Object... params) {
13.        // on crée des tâches asynchrones pour générer les nombres aléatoires
14.        int nbAleas = ((Integer) params[0]).intValue();
15.        for (int i = 0; i < nbAleas; i++) {
16.            // on demande la tâche à la Factory
17.            Task task = (Task) factory.getObject(Factory.TASK_01, this,
String.format("[%s]Task_01-%02d", bossId, i));
18.            try {
19.                // on exécute la tâche
20.                task.doWork(params[1], params[2], params[3]);
21.            } catch (RejectedExecutionException ex) {
22.                // on passe la tâche à l'état TERMINATED
23.                notifyEvent(task, IBoss.WORK_TERMINATED,null);
24.                // elle a été rejetée - on passe l'exception au boss
25.                boss.notifyEvent(this, IBoss.WORK_INFO, ex);
26.            }
27.        }
28.        // on commence le monitoring de la fin des tâches lancées
29.        beginMonitoring();
30.    }
31.
32.    @Override
33.    public void notifyEndOfTasks() {
34.    ...
35.    }
36.
37.    // notification d'un événement
38.    @Override
39.    public void notifyEvent(IWorker worker, int eventType, Object event) {
40.    ...
41.    }
42.
43. }
```

- ligne 12 : l'action fait son travail dans la méthode [doWork]. Elle reçoit quatre paramètres (nbAleas, a, b, sleepTime) ;
- ligne 14 : elle utilise le premier paramètre, le nombre de nombres aléatoires à générer ;

- ligne 15 : elle utilise une tâche asynchrone pour chaque nombre aléatoire ;
- ligne 17 : une tâche est demandée à la fabrique d'objets. On lui passe deux informations :
 - une référence sur le boss, l'action (this),
 - un identifiant ;

Le code d'instanciation de la tâche dans la fabrique est le suivant :

```

1. // une tâche reçoit deux paramètres : IBoss, id
2. private Object getTask_01(Object[] params) {
3.     // instantiation
4.     ITask task = new Task_01();
5.     // initialisation
6.     // le boss
7.     task.setBoss((IBoss) params[0]);
8.     // pas d'équipe
9.     // l'identité
10.    task.setWorkerId((String) params[1]);
11.    // la factory
12.    task.setFactory(this);
13.    // fin
14.    return task;
15. }
```

- la fabrique génère une tâche asynchrone à chaque requête. De nouveau, on pourrait regarder si le singleton est possible et s'il a un intérêt ;
- ligne 4 : la tâche asynchrone est instanciée ;
- lignes 7-12 : on lui injecte son boss (ligne 7), son identifiant (ligne 10) et la fabrique (12) ;
- ligne 8 : la tâche n'est pas un [IBoss]. Elle n'a donc pas d'équipe.

Retour au code de la méthode [doWork] :

- lignes 18-26 : la tâche est lancée. Elle sera exécutée dans un thread asynchrone différent de celui de l'UI ;
- ligne 21 : si le pool de threads disponibles est vide, on aura une exception ;
- ligne 23 : on passe la tâche à l'état TERMINATED. En effet, elle a eu le temps de passer à l'état STARTED ;
- ligne 25 : on remonte l'exception au boss (la vue) ;

L'action est un [IBoss]. Elle gère une équipe de [Task]. Elle voit donc passer des notifications dans [notifyEvent] :

```

1. @Override
2. public void notifyEvent(IWorker worker, int eventType, Object event) {
3.     // on passe l'événement au parent
4.     super.notifyEvent(worker, eventType, event);
5.     // on passe l'événement INFO au boss
6.     if (eventType == IBoss.WORK_INFO) {
7.         boss.notifyEvent(this, eventType, event);
8.         // si l'info est un entier, on le mémorise
9.         if (event instanceof Integer) {
10.             aleas.add((Integer) event);
11.         }
12.     }
13. }
```

- ligne 4 : la notification est passée au parent qui gère les notifications [WORK_STARTED] et [WORK_TERMINATED] ;
- ligne 6 : l'action gère les notifications [WORK_INFO] que lui envoient les tâches qu'elle a lancées ;
- ligne 7 : la notification [WORK_INFO] est remontée telle quelle au boss de l'action (la vue). On notera que l'action se substitue à la tâche (paramètre **this**) ;
- lignes 9-10 : on récupère l'information transportée par la notification [WORK_INFO] et on l'ajoute à la liste des nombres aléatoires déjà générés par les tâches.

Les tâches vont se terminer et l'action va recevoir la notification [endOfTasks] :

```

1. @Override
2. public void notifyEndOfTasks() {
```

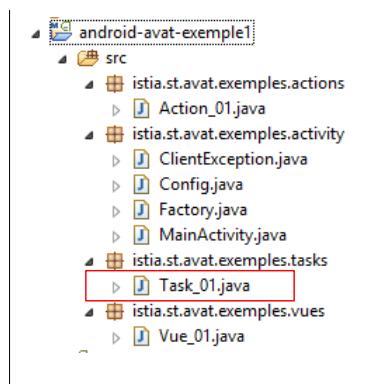
```

3.      // fin des tâches - on fait la somme des nombres reçus
4.      int somme = 0;
5.      for (int alea : aleas) {
6.          somme += alea;
7.      }
8.      // on rend cette somme au boss
9.      boss.notifyEvent(this, IBoss.WORK_TERMINATED, Integer.valueOf(somme));
10. }

```

- lignes 4-7 : la somme des nombres aléatoires mémorisés est calculée ;
- ligne 9 : la notification [WORK_TERMINATED] est envoyée au boss de l'action (la vue) accompagnée de cette somme.

3.11 La tâche asynchrone [Task_01]



Le code de la tâche asynchrone [Task_01] est le suivant :

```

1. package istia.st.avat.exemples.tasks;
2.
3. ...
4.
5. public class Task_01 extends Task {
6.
7.     // info renvoyée par la tâche
8.     private Object info;
9.
10.    @Override
11.    public void onPreExecute(){
12.        // on commence le travail
13.        boss.notifyEvent(this, IBoss.WORK_STARTED, null);
14.    }
15.
16.    @Override
17.    protected Void doInBackground(Object... params) {
18.    ...
19.    }
20.
21.    @Override
22.    // traitement fin de tâche dans le thred de l'UI
23.    protected void onPostExecute(Void result) {
24.        // on passe l'info au boss
25.        boss.notifyEvent(this, IBoss.WORK_INFO, info);
26.        // on a terminé le travail
27.        boss.notifyEvent(this, IBoss.WORK_TERMINATED, null);
28.    }
29.
30. }

```

- ligne 17 : la méthode exécutée en tâche de fond. C'est elle qui fait le travail. Elle exécute normalement une méthode de la couche [métier]. Pas ici ;
- ligne 8 : l'information produite par la tâche. Ici, une exception ou un nombre aléatoire ;
- lignes 11-14 : la méthode [onPreExecute] utilisée pour envoyer la notification [WORK_STARTED] au boss (une action). Elle s'exécute dans le thread de l'UI ;
- lignes 23-28 : la méthode [onPostExecute] utilisée pour envoyer les notifications [WORK_INFO] et [WORK_STARTED] au boss (une action). Elle s'exécute dans le thread de l'UI ;
- ligne 25 : l'information produite par la méthode [doInBackground] est ici envoyée au boss (une action) ;

La méthode [doInBackGround] est la suivante :

```

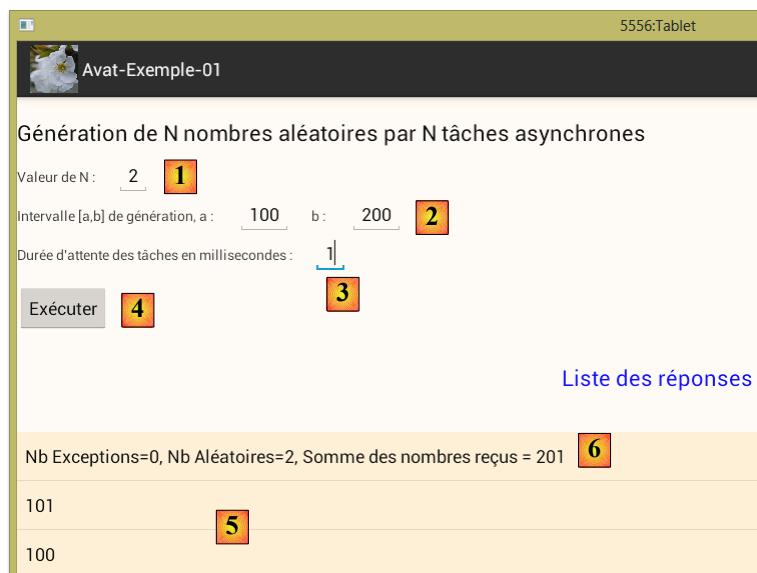
1.  @Override
2.  protected Void doInBackground(Object... params) {
3.      // il faut trois paramètres
4.      if (params.length != 3) {
5.          info = new ClientException("Il faut trois paramètres");
6.          return null;
7.      }
8.      // on récupère les paramètres
9.      int sleepTime = 0;
10.     int a = 0;
11.     int b = 0;
12.     try {
13.         sleepTime = (Integer) params[2];
14.         a = (Integer) params[0];
15.         b = (Integer) params[1];
16.     } catch (Exception ex) {
17.         info = new ClientException(String.format("Paramètres incorrects [%d, %d, %d]", sleepTime, a, b));
18.         return null;
19.     }
20.     // attente
21.     try {
22.         Thread.sleep(sleepTime);
23.     } catch (InterruptedException e) {
24.         info = e;
25.         return null;
26.     }
27.     // on génère une exception aléatoire 1 fois / 3
28.     Random random = new Random();
29.     int i = random.nextInt(3);
30.     if (i == 0) {
31.         info = new ClientException("Exception aléatoire");
32.         return null;
33.     }
34.     // sinon on rend un nombre aléatoire entre deux bornes [a,b]
35.     if (a > b) {
36.         info = new ClientException(String.format("%d doit être >= à %d", b, a));
37.         return null;
38.     }
39.     info = a + random.nextInt(b - a + 1);
40.     return null;
41. }
```

- ligne 2 : on attend les paramètres [a, b, sleepTime] dans l'ordre ;
- lignes 4-7 : on vérifie le nombre de paramètres ;
- lignes 9-19 : on récupère les paramètres en les typant ;
- lignes 21-26 : le thread s'arrête *sleepTime* millisecondes. Il va alors perdre le processeur. Une autre tâche va en bénéficier ;
- lignes 28-29 : on génère un nombre aléatoire entier dans l'intervalle [0-2] ;
- lignes 30-33 : si le nombre généré est 0, l'information produite par la tâche sera une exception. Comme le nombre 0 a une chance sur 3 d'être généré, on génère donc une exception avec une chance sur 3. Si la vue initiale a demandé un grand nombre de nombres aléatoires, on devrait vérifier qu'environ le tiers des réponses sont des exceptions ;

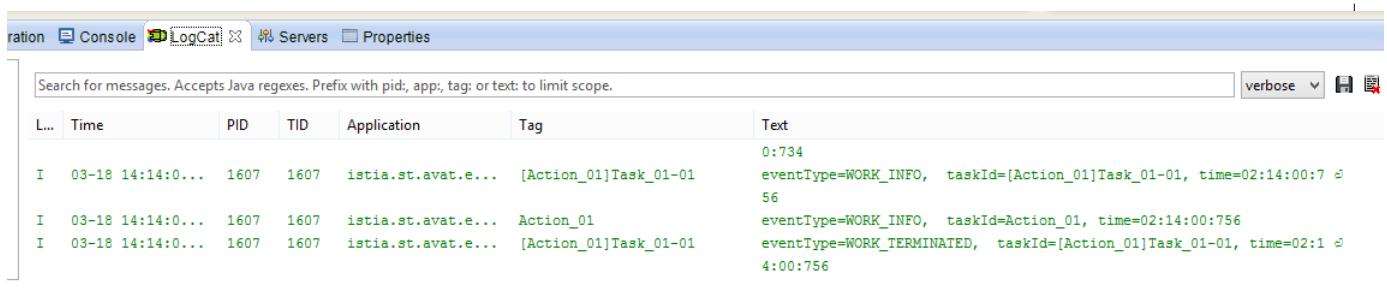
- lignes 35-37 : on vérifie la validité de l'intervalle [a, b] ;
- ligne 39 : on génère un nombre aléatoire dans l'intervalle [a, b].

3.12 Les tests

Le lecteur est invité à tester l'application. Voici quelques conseils :



Pour voir les logs, mettre un petit nombre pour N. Les logs sont trouvés dans l'onglet [LogCat] d'Eclipse :



Nous avons déjà commenté ces logs. Par exemple pour 2 tâches on a les logs suivants :

```

11. 03-14 02:08:02.847: I/Action_01(7942): eventType=WORK_STARTED, taskId=Action_01,
    time=02:08:02:849
12. 03-14 02:08:02.847: I/[Action_01]Task_01-00(7942): eventType=WORK_STARTED,
    taskId=[Action_01]Task_01-00, time=02:08:02:849
13. 03-14 02:08:02.847: I/[Action_01]Task_01-01(7942): eventType=WORK_STARTED,
    taskId=[Action_01]Task_01-01, time=02:08:02:849
14. 03-14 02:08:07.986: I/[Action_01]Task_01-01(7942): eventType=WORK_INFO,
    taskId=[Action_01]Task_01-01, time=02:08:07:986
15. 03-14 02:08:07.986: I/Action_01(7942): eventType=WORK_INFO, taskId=Action_01,
    time=02:08:07:989
16. 03-14 02:08:07.986: I/[Action_01]Task_01-01(7942): eventType=WORK_TERMINATED,
    taskId=[Action_01]Task_01-01, time=02:08:07:989
17. 03-14 02:08:07.986: I/[Action_01]Task_01-00(7942): eventType=WORK_INFO,
    taskId=[Action_01]Task_01-00, time=02:08:07:990

```

```

18. 03-14 02:08:07.986: I/Action_01(7942): eventType=WORK_INFO, taskId=Action_01,
    time=02:08:07:990
19. 03-14 02:08:07.986: I/[Action_01]Task_01-00(7942): eventType=WORK_TERMINATED,
    taskId=[Action_01]Task_01-00, time=02:08:07:990
20. 03-14 02:08:07.986: I/Action_01(7942): eventType=WORK_TERMINATED, taskId=Action_01,
    time=02:08:07:990

```

- ligne 1 : l'action [Action_01] a été lancée ;
- ligne 2 : la tâche [Action_01]Task_01-00 a été lancée ;
- ligne 3 : la tâche [Action_01]Task_01-01 a été lancée ;
- ligne 4 : la tâche [Action_01]Task_01-01 a généré une information ;
- ligne 5 : l'action [Action_01] a généré une information ;
- ligne 6 : la tâche [Action_01]Task_01-01 est terminée ;
- ligne 7 : la tâche [Action_01]Task_01-00 a généré une information ;
- ligne 8 : l'action [Action_01] a généré une information ;
- ligne 9 : la tâche [Action_01]Task_01-00 est terminée ;
- ligne 10 : l'action [Action_01] est terminée ;

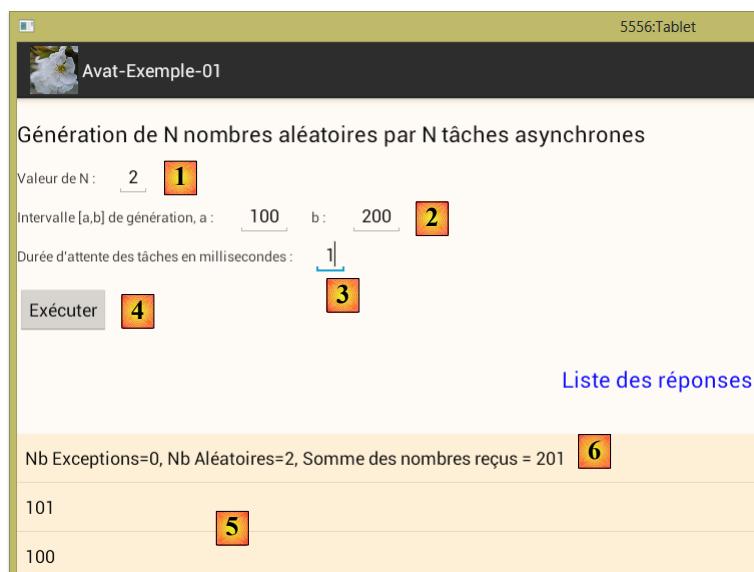
Dans le cas d'annulation de ces deux tâches, on a les logs suivants :

```

1. 03-14 02:13:54.871: I/Action_01(7942): eventType=WORK_STARTED, taskId=Action_01,
    time=02:13:54:872
2. 03-14 02:13:54.871: I/[Action_01]Task_01-00(7942): eventType=WORK_STARTED,
    taskId=[Action_01]Task_01-00, time=02:13:54:873
3. 03-14 02:13:54.871: I/[Action_01]Task_01-01(7942): eventType=WORK_STARTED,
    taskId=[Action_01]Task_01-01, time=02:13:54:873
4. 03-14 02:13:56.280: I/[Action_01]Task_01-01(7942): taskId=[Action_01]Task_01-01 canceled at
    time=02:13:56:283
5. 03-14 02:13:56.280: I/[Action_01]Task_01-00(7942): taskId=[Action_01]Task_01-00 canceled at
    time=02:13:56:284
6. 03-14 02:13:56.280: I/Action_01(7942): taskId=Action_01 canceled at time=02:13:56:284

```

- ligne 1 : l'action [Action_01] a été lancée ;
- ligne 2 : la tâche [Action_01]Task_01-00 a été lancée ;
- ligne 3 : la tâche [Action_01]Task_01-01 a été lancée ;
- ligne 4 : la tâche [Action_01]Task_01-01 a été annulée ;
- ligne 5 : la tâche [Action_01]Task_01-00 a été annulée ;
- ligne 6 : l'action [Action_01] a été annulée ;

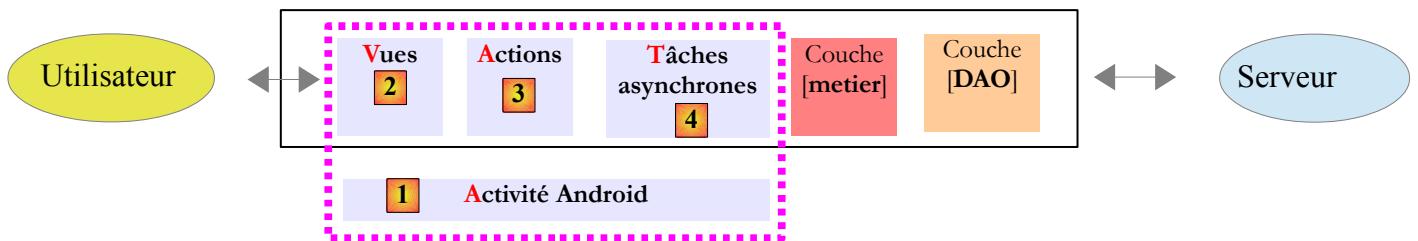


- pour annuler les tâches, on mettra un délai d'attente de plusieurs secondes en [3] ;
- pour épuiser le pool de threads, on mettra 200 en [1].

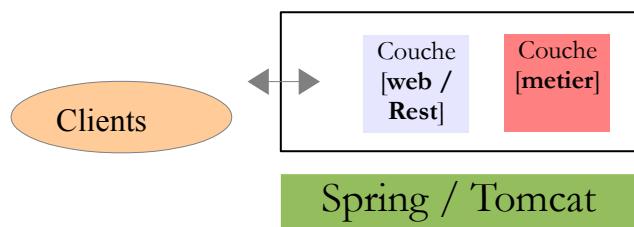
4 AVAT- Exemple 2

4.1 Le projet

Nous nous proposons de créer une application Android faisant la même chose que précédemment mais dans une architecture client / serveur . Le client sera le suivant :



C'est le serveur qui générera les nombres aléatoires affichés par la tablette Android :



La couche [métier] aura une méthode de génération de nombres aléatoires exposée au monde web via un service REST.

La couche AVAT du client va rester celle que l'on vient de décrire. Les tâches asynchrones vont bouger un peu : elles exécutent maintenant une méthode de la couche [métier] du client. Nous allons décrire l'application dans l'ordre suivant :

Le serveur

- sa couche [métier] ;
- son service REST ;

Le client

- sa couche [DAO] ;
- sa couche [métier] ;
- sa tâche asynchrone.

Le reste ne change pas.

4.2 Le manifeste de l'application Android

Le fichier [AndroidManifest.xml] du projet est un peu différent de celui du projet précédent :

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3.   package="istia.st.avat.examples"
4.   android:versionCode="1"
5.   android:versionName="1.0" >
6.
7.   <uses-sdk
        android:minSdkVersion="11"
```

```

9.         android:targetSdkVersion="16" />
10.    <uses-permission android:name="android.permission.INTERNET" />
11.
12.
13.    <application
14.        android:allowBackup="true"
15.        android:icon="@drawable/fleur"
16.        android:label="@string/app_name"
17.        android:theme="@style/AppTheme" >
18.            <activity
19.                android:name="istia.st.avat.examples.activity.MainActivity"
20.                android:label="@string/app_name" >
21.                    <intent-filter>
22.                        <action android:name="android.intent.action.MAIN" />
23.
24.                        <category android:name="android.intent.category.LAUNCHER" />
25.                    </intent-filter>
26.            </activity>
27.        </application>
28.
29. </manifest>

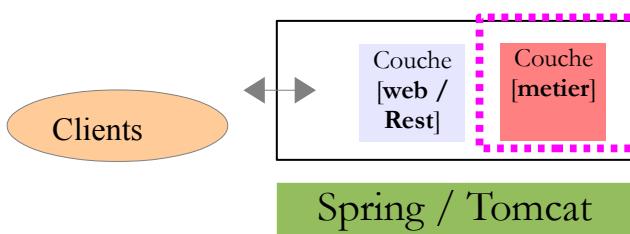
```

La ligne importante est la ligne 11. C'est elle qui permet au client Android d'ouvrir des connexions réseau. Si on l'oublie, ça ne marche pas mais les messages d'erreur ne sont pas toujours explicites quant à la cause de l'erreur.

4.3 Le serveur

4.3.1 La couche [métier]

C'est elle qui génère les nombres aléatoires.



4.3.1.1 Le projet Eclipse

Le projet Eclipse de la couche [métier] est le suivant :



- en [1], le projet est un projet [Maven] ;
- en [2] :
 - [IMetier] est l'interface de la couche [métier] ;
 - [Metier] l'implémentation de cette interface ;

- [ServerException] une classe d'exception.

4.3.1.2 Les dépendances Maven

Le fichier [pom.xml] est le suivant :

```

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
   4.0.0.xsd">
2.   <modelVersion>4.0.0</modelVersion>
3.   <groupId>istia.st.avat.server</groupId>
4.   <artifactId>server-metier</artifactId>
5.   <version>0.0.1-SNAPSHOT</version>
6.   <build>
7.     <sourceDirectory>src</sourceDirectory>
8.     <plugins>
9.       <plugin>
10.         <artifactId>maven-compiler-plugin</artifactId>
11.         <version>3.0</version>
12.         <configuration>
13.           <source/>
14.           <target/>
15.         </configuration>
16.       </plugin>
17.     </plugins>
18.   </build>
19. </project>
```

Il n'y a aucune dépendance.

4.3.1.3 L'interface [IMetier]

Le code de l'interface est le suivant :

```

1. package istia.st.avat.server;
2.
3. public interface IMetier {
4.
5.   // génération d'un nombre aléatoire
6.   public abstract int randomNumber(int a, int b);
7.
8. }
```

- ligne 6 : la méthode qui génère un nombre aléatoire entre [a,b]

4.3.1.4 L'implémentation [Metier]

Le code de la classe [Metier] est le suivant :

```

1. package istia.st.avat.server;
2.
3. import java.io.Serializable;
4. import java.util.Random;
5.
6. public class Metier implements Serializable, IMetier {
7.
8.   private static final long serialVersionUID = 1L;
9. }
```

```

10. public Metier() {
11. }
12.
13. // génération d'un nombre aléatoire
14. /* (non-Javadoc)
15.  * @see istia.st.avat.server.IMetier#randomNumber(int, int)
16. */
17. public int randomNumber(int a, int b) {
18.     // vérification des paramètres
19.     if (a > b) {
20.         throw new ServerException(String.format("%d doit être >= à %d", b, a));
21.     }
22.     // on génère une exception aléatoire 1 fois / 3
23.     Random random = new Random();
24.     int i = random.nextInt(3);
25.     if (i == 0) {
26.         throw new ServerException("Exception aléatoire");
27.     }
28.     // on rend un nombre aléatoire entre a et b
29.     return a + random.nextInt(b - a + 1);
30. }
31. }
```

On y trouve ce qui précédemment était dans la tâche [Task_01]. Nous ne commentons pas la classe : elle rend 2 fois sur 3 un nombre aléatoire et 1 fois sur 3 lance une exception.

4.3.1.5 La classe d'exception

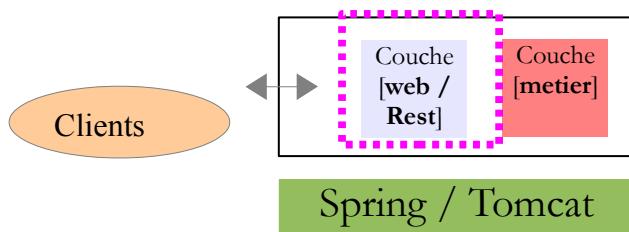
La classe d'exception est la suivante :

```

1. package istia.st.avat.server;
2.
3. import java.io.Serializable;
4.
5. public class ServerException extends RuntimeException implements Serializable{
6.
7.     private static final long serialVersionUID = 1L;
8.
9.     public ServerException() {
10.    }
11.
12.    public ServerException(String message) {
13.        super(message);
14.    }
15.
16.    public ServerException(Throwable cause) {
17.        super(cause);
18.    }
19.
20.    public ServerException(String message, Throwable cause) {
21.        super(message, cause);
22.    }
23.
24.    public ServerException(String message, Throwable cause, boolean enableSuppression,
25.        boolean writableStackTrace) {
26.        super(message, cause, enableSuppression, writableStackTrace);
27.    }
28. }
```

- ligne 5 : comme la classe [ClientException] chez le client, la classe [ServerException] dérive de la classe [RuntimeException].

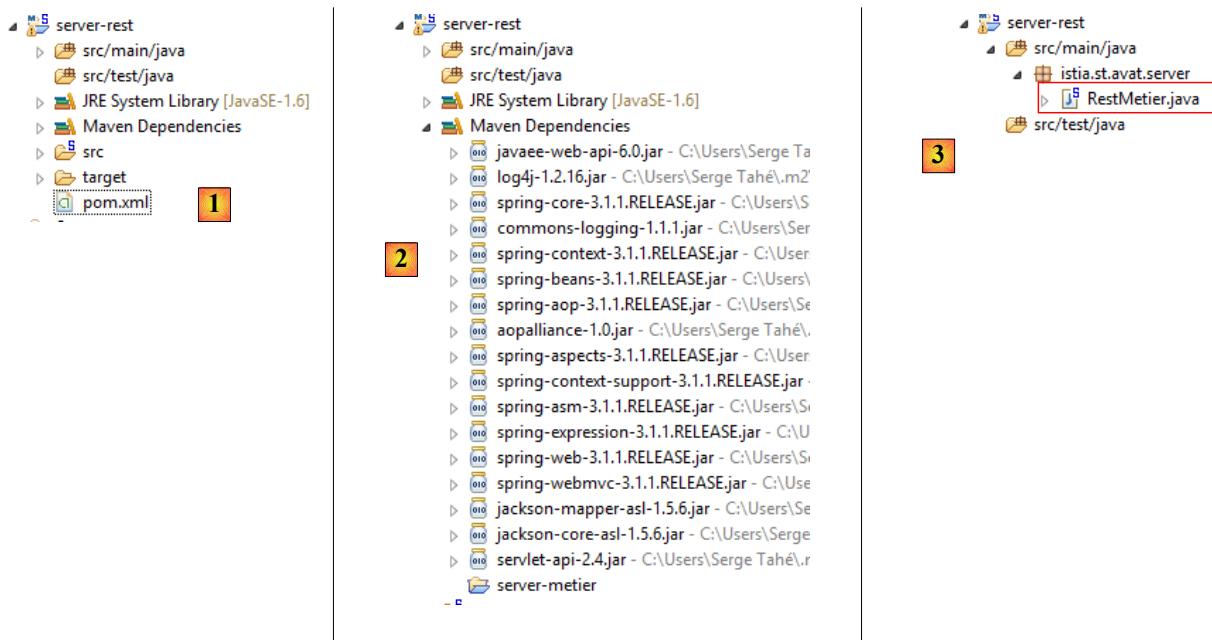
4.3.2 Le service REST



Le service REST est implémenté par SpringMVC. Un service REST (RepresEntational State Transfer) est un service HTTP répondant aux demandes GET, POST, PUT, DELETE d'un client HTTP. Sa définition formelle indique pour chacune de ces méthodes, les objets que le client doit envoyer et celui qu'il reçoit. Dans les exemples de ce document, nous n'utilisons que la méthode GET alors que dans certains cas, la définition formelle de REST indique qu'on devrait utiliser une méthode PUT. Nous appellerons REST notre service parce qu'il est implémenté par un service de Spring qu'on a l'habitude d'appeler service REST et non parce qu'il respecte la définition formelle des services REST.

4.3.2.1 Le projet Eclipse

Le projet Eclipse de la couche [web] est le suivant :



- en [1], le projet dans son ensemble ;
- en [2], les dépendances Maven ;
- en [3], le projet ne contient qu'une classe.

4.3.2.2 Les dépendances Maven

Le fichier [pom.xml] du projet Maven est le suivant :

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```

<groupId>istia.st.avat.server</groupId>
<artifactId>server-rest</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<name>server-rest</name>

<properties>
    <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <releaseCandidate>1</releaseCandidate>
    <spring.version>3.1.1.RELEASE</spring.version>
    <jackson.mapper.version>1.5.6</jackson.mapper.version>
</properties>

<repositories>
    <repository>
        <id>com.springsource.repository.bundles.release</id>
        <name>SpringSource Enterprise Bundle Repository - Release</name>
        <url>http://repository.springsource.com/maven/bundles/release</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.codehaus.jackson</groupId>
        <artifactId>jackson-mapper-asl</artifactId>
        <version>${jackson.mapper.version}</version>
    </dependency>
    <dependency>
        <groupId>istia.st.avat.server</groupId>
        <artifactId>server-metier</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </dependency>
</dependencies>

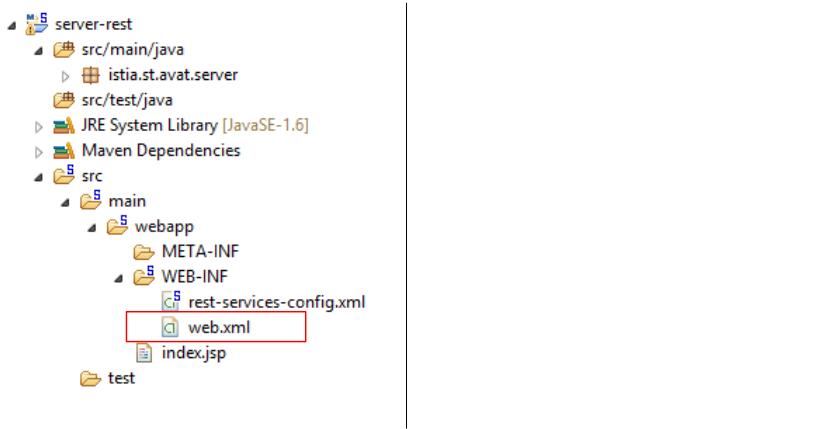
<build>
    <plugins>
        ...
    </plugins>
</build>

```

- le service REST est implémenté à l'aide du framework SpringMVC. On trouve donc un certain nombre de dépendances sur ce framework (lignes 29-48) ;
- lignes 49-53 : le serveur REST va échanger avec ses clients des objets au format JSON (JavaScript Object Notation). C'est une représentation texte des objets. Un objet JAVA peut être sérialisé en un texte JSON. Inversement, ce dernier peut être déserialisé pour créer un objet JAVA. Côté serveur, la bibliothèque Jackson est utilisée pour faire ces opérations ;
- lignes 54-58 : le serveur REST a une dépendance sur la couche [métier] que nous avons décrite.

4.3.2.3 Configuration du service REST

Un service REST est une application web et à ce titre est configurée par un fichier [web.xml] classique :



Le fichier [web.xml] est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
   app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
3.   <display-name>REST for AVAT</display-name>
4.   <servlet>
5.     <servlet-name>restservices</servlet-name>
6.     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
7.     <init-param>
8.       <param-name>contextConfigLocation</param-name>
9.       <param-value>/WEB-INF/rest-services-config.xml</param-value>
10.    </init-param>
11.    <load-on-startup>1</load-on-startup>
12.  </servlet>
13.  <servlet-mapping>
14.    <servlet-name>restservices</servlet-name>
15.    <url-pattern>/</url-pattern>
16.  </servlet-mapping>
17. </web-app>

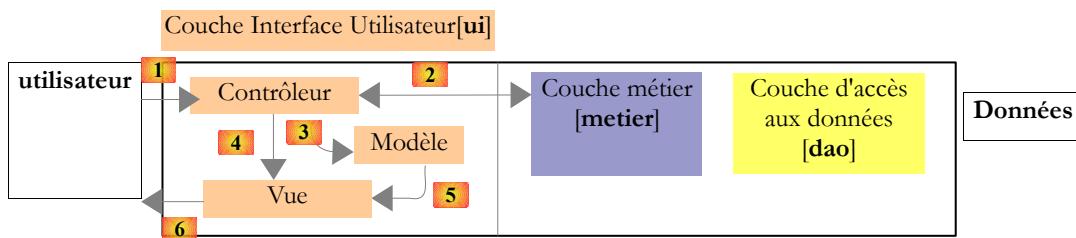
```

- lignes 4-12 : définition d'une servlet, une classe capable de traiter les requêtes HTTP des clients web ;
- ligne 5 : son nom *restservices*. On peut l'appeler comme on veut ;
- ligne 6 : la servlet qui gère les requêtes HTTP des clients web. C'est la classe [DispatcherServlet] de SpringMVC ;
- lignes 7-10 : indiquent le fichier de configuration de la servlet, ici [WEB-INF/rest-services-config.xml] ;
- ligne 11 : la servlet sera chargée au démarrage du serveur, ici un serveur Tomcat 7 ;
- lignes 13-16 : les URL (ligne 15) traitées par la servlet *restservices* (ligne 14).

Au final, ce fichier indique que toutes les URL (ligne 15) seront gérées par la servlet *restservices* (ligne 14) définie lignes 4-12.

4.3.2.4 SpringMVC

SpringMVC est un framework MVC (Modèle-Vue-Contrôleur). Rappelons les grands principes de ce *design pattern* :



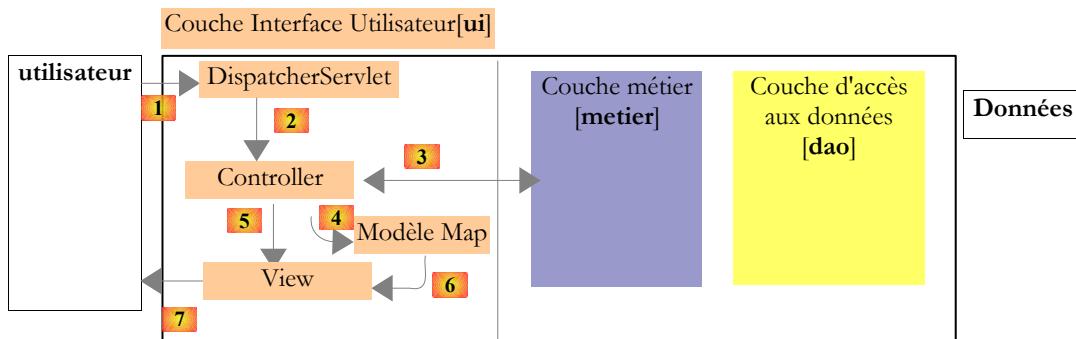
Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

1. le client fait une demande au contrôleur. Celui-ci voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le **C** de MVC ;
2. le contrôleur **C** traite cette demande. Pour ce faire, il peut avoir besoin de l'aide de la couche métier. Une fois la demande du client traitée, celle-ci peut appeler diverses réponses. Un exemple classique est :
 - une page d'erreurs si la demande n'a pu être traitée correctement,
 - une page de confirmation sinon ;
3. le contrôleur choisit la réponse (= vue) à envoyer au client. Choisir la réponse à envoyer au client nécessite plusieurs étapes :
 - choisir l'objet qui va générer la réponse. C'est ce qu'on appelle la vue **V**, le **V** de MVC. Ce choix dépend en général du résultat de l'exécution de l'action demandée par l'utilisateur,
 - lui fournir les données dont il a besoin pour générer cette réponse. En effet, celle-ci contient le plus souvent des informations calculées par le contrôleur. Ces informations forment ce qu'on appelle le modèle **M** de la vue, le **M** de MVC ;

L'étape 3 consiste donc en le choix d'une vue **V** et en la construction du modèle **M** nécessaire à celle-ci.

4. le contrôleur **C** demande à la vue choisie de s'afficher ;
5. la vue **V** utilise le modèle **M** préparé par le contrôleur **C** pour initialiser les parties dynamiques de la réponse qu'elle doit envoyer au client ;
6. la réponse est envoyée au client. Ce peut être un flux HTML, PDF, Excel, ...

Spring MVC implémente cette architecture de la façon suivante :



1. le client fait une demande au contrôleur. Celui-ci voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le **C** de MVC. Ici le contrôleur est assuré par une servlet générique :

org.springframework.web.servlet.DispatcherServlet
2. le contrôleur principal [DispatcherServlet] fait exécuter l'action demandée par l'utilisateur par une classe implementant l'interface :

org.springframework.web.servlet.mvc.Controller

A cause du nom de l'interface, nous appellerons une telle classe un contrôleur secondaire pour le distinguer du contrôleur principal [**DispatcherServlet**] ou simplement contrôleur lorsqu'il n'y a pas d'ambiguïté. Le schéma ci-dessus s'est contenté de représenter un contrôleur particulier. Il y a en général plusieurs contrôleurs, un par action.

3. le contrôleur [**Controller**] traite une demande particulière de l'utilisateur. Pour ce faire, il peut avoir besoin de l'aide de la couche métier. Une fois la demande du client traitée, celle-ci peut appeler diverses réponses. Un exemple classique est :
 - une page d'erreurs si la demande n'a pu être traitée correctement
 - une page de confirmation sinon
4. le contrôleur choisit la réponse (= vue) à envoyer au client. Choisir la réponse à envoyer au client nécessite plusieurs étapes :
 - choisir l'objet qui va générer la réponse. C'est ce qu'on appelle la vue **V**, le **V** de MVC. Ce choix dépend en général du résultat de l'exécution de l'action demandée par l'utilisateur.
 - lui fournir les données dont il a besoin pour générer cette réponse. En effet, celle-ci contient le plus souvent des informations calculées par la couche métier ou le contrôleur lui-même. Ces informations forment ce qu'on appelle le modèle **M** de la vue, le **M** de MVC. Spring MVC fournit ce modèle sous la forme d'un dictionnaire de type **java.util.Map**.

L'étape 4 consiste donc en le choix d'une vue **V** et la construction du modèle **M** nécessaire à celle-ci.

1. le contrôleur **DispatcherServlet** demande à la vue choisie de s'afficher. Il s'agit d'une classe implementant l'interface **org.springframework.web.servlet.View**
Spring MVC propose différentes implémentations de cette interface pour générer des flux HTML, Excel, PDF, ... Le schéma ci-dessus s'est contenté de représenter une vue particulière. Il y a en général plusieurs vues.
5. le générateur de vue **View** utilise le modèle **Map** préparé par le contrôleur **Controller** pour initialiser les parties dynamiques de la réponse qu'il doit envoyer au client.
6. la réponse est envoyée au client. La forme exacte de celle-ci dépend du générateur de vue. Ce peut être un flux HTML, PDF, Excel, ...

Dans notre cas, la classe [DispatcherServlet] est configurée par le fichier [rest-services-config.xml] suivant :

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xmlns:util="http://www.springframework.org/schema/util"
5.         xmlns:mvc="http://www.springframework.org/schema/mvc"
6.         xmlns:context="http://www.springframework.org/schema/context"
7.         xsi:schemaLocation="http://www.springframework.org/schema/beans
8.                             http://www.springframework.org/schema/spring-beans-3.0.xsd
9.                             http://www.springframework.org/schema/util
10.                            http://www.springframework.org/schema/mvc
11.                            http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
12.                            http://www.springframework.org/schema/context
13.                            http://www.springframework.org/schema/context/spring-context-3.0.xsd
14.                            http://www.springframework.org/schema/util
15.                            http://www.springframework.org/schema/util/spring-util-3.0.xsd">
16.
17.
18.    <!-- couche métier -->
19.    <bean id="metier" class="istia.st.avat.server.Metier" />
20.
21.    <!-- Recherche des annotations SpringMVC dans les classes du package nommé -->
22.    <context:component-scan base-package="istia.st.avat.server" />
23.
24.    <!-- L'unique View utilisée. Elle génère du JSON -->
25.    <bean id="jsonMessageConverter"
26.          class="org.springframework.web.servlet.view.json.MappingJacksonJsonView">
27.        <property name="contentType" value="text/plain" />
28.      </bean>
29.
30.      <!-- Le convertisseur JSON -->
31.      <bean id="jsonMessageConverter"
32.          class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter" />
33.
34.      <!-- la liste des convertisseurs de messages. Il n'y en qu'un : le JSON ci-dessus -->
35.      <bean
36.          class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
37.        <property name="messageConverters">
```

```

32.      <util:list id="beanList">
33.          <ref bean="jsonMessageConverter" />
34.      </util:list>
35.  </property>
36. </bean>
37.
38.
39.</beans>

```

- ligne 13 : le bean de la couche [métier]. Il sera instancié par Spring ;
- ligne 16 : il est possible de configurer Spring à l'aide d'annotations dans le code Java. C'est ce qui a été fait ici. La ligne 16 dit à Spring d'exploiter les annotations qu'il trouvera dans le package [*istia.st.avat.server*] ;

Il y trouvera trois genres d'annotations :

```

1. @Controller
2. public class RestMetier {
3.
4.     // couche métier
5.     @Autowired
6.     private IMetier metier;
7.     @Autowired
8.     private View view;
9.
10.    // nombre aléatoire
11.    @RequestMapping(value = "/random/{a}/{b}", method = RequestMethod.GET)
12.    public ModelAndView randomNumber(@PathVariable("a") int a,
13.                                       @PathVariable("b") int b) {
14. ...
15. }

```

- ligne 1 : l'annotation **@Controller** fait de la classe [RestMetier] un contrôleur SpringMVC, c-à-d une classe capable de traiter des requêtes HTTP. Celles-ci sont définies par l'annotation **@RequestMapping** qu'on voit en ligne 11 ;
- lignes 5 et 7 : l'annotation **@Autowired** tague des champs qui sont des références sur des beans définis dans le fichier de configuration de SpringMVC. Dans notre cas, ce sont les beans des lignes 2 et 8 du fichier [rest-services-config.xml] ci-dessous. Les champs des lignes ainsi taguées sont automatiquement initialisés par Spring.

Revenons au code du fichier [rest-services-config.xml] :

```

1. <!-- couche métier -->
2.   <bean id="metier" class="istia.st.avat.server.Metier" />
3.
4.   <!-- Recherche des annotations SpringMVC dans les classes du package nommé -->
5.   <context:component-scan base-package="istia.st.avat.server" />
6.
7.   <!-- L'unique View utilisée. Elle génère du JSON -->
8.   <bean
9.       class="org.springframework.web.servlet.view.json.MappingJacksonJsonView">
10.      <property name="contentType" value="text/plain" />
11.   </bean>
12.
13.   <!-- Le convertisseur JSON -->
14.   <bean id="jsonMessageConverter"
15.       class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"
16.   />
17.   <!-- la liste des convertisseurs de messages. Il n'y en qu'un : le JSON ci-dessus
18.   -->
19.   <bean
20.       class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
21.         <property name="messageConverters">
22.             <util:list id="beanList">

```

```

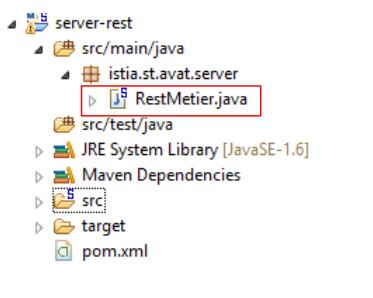
22.          <ref bean="jsonMessageConverter" />
23.      </util:list>
24.  </property>
25. </bean>

```

- lignes 8-10 : les vues générées par l'unique contrôleur seront du type [*MappingJacksonJsonView*] (ligne 9). Cette vue va transformer son modèle en un unique objet JSON qu'elle va transmettre à son client. La ligne 10 configure un entête HTTP envoyé par le serveur à son client : elle dit que le serveur envoie du texte, ici l'objet JSON ;
- lignes 14-16 : un « convertisseur de messages ». Je ne sais pas bien ce que signifie cette expression. Il semble que cela définisse une classe capable de créer un objet à partir d'une requête ou d'une réponse HTTP et inversement. Ici la classe [*MappingJacksonHttpMessageConverter*] va créer un objet JSON à partir du modèle de la vue. Inversement, elle est capable de produire un objet Java à partir de données JSON postées par le client ;
- lignes 18-25 : il peut y avoir plusieurs « convertisseurs de messages » pour différents cas de figure. Entre les lignes 21-23, on met les beans chargés de « convertir des messages ». On y met donc le bean défini ligne 14. C'est le seul.

4.3.2.5 Le serveur REST

Maintenons que nous avons vu comment le service REST était configuré, examinons son code :



Le code de la classe [RestMetier] est le suivant :

```

1. package istia.st.avat.server;
2.
3.
4. import java.util.HashMap;
5. import java.util.Map;
6. ...
7.
8. @Controller
9. public class RestMetier {
10.
11.     // couche métier
12.     @Autowired
13.     private IMetier metier;
14.     @Autowired
15.     private View view;
16.
17.     // nombre aléatoire
18.     @RequestMapping(value = "/random/{a}/{b}", method = RequestMethod.GET)
19.     public ModelAndView randomNumber(@PathVariable("a") int a,
20.                                     @PathVariable("b") int b) {
21.
22.         // on utilise la couche métier
23.         int n = 0;
24.         try {
25.             n = metier.randomNumber(a, b);
26.             // suivi
27.             return new ModelAndView(view, "data", String.valueOf(n));
28.         } catch (Exception e) {
29.             // suivi
30.             return createResponseError("103", e.getMessage());

```

```

31.     }
32.   }
33.
34.   // crée une réponse d'erreur
35.   private ModelAndView createResponseError(String numero, String message) {
36.     Map<String, Object> modèle = new HashMap<String, Object>();
37.     modèle.put("error", numero);
38.     modèle.put("message", message);
39.     return new ModelAndView(view, modèle);
40.   }
41. }
```

- ligne 19 : la méthode qui génère le nombre aléatoire. Lorsqu'elle s'exécute, les champs des lignes 13 et 15 ont été initialisés par SpringMVC. Par ailleurs, si elle s'exécute, c'est parce que le serveur web a reçu une requête HTTP GET pour l'URL de la ligne 18 ;
- ligne 18 : l'URL traitée est de la forme `/random/{a}/{b}` où {x} représente une variable. Les deux variables {a} et {b} sont affectées aux paramètres de la méthode lignes 19 et 20. Cela se fait via l'annotation `@PathVariable("x")`. On notera que {a} et {b} sont des composantes d'une URL et sont donc de type `String`. La conversion de `String` vers le type des paramètres peut échouer. SpringMVC lance alors une exception. Résumons : si avec un navigateur je demande l'URL `/random/100/200`, la méthode `random` de la ligne 19 s'exécutera avec les paramètres entiers 100 et 200 ;
- ligne 25 : on demande à la couche [métier] un nombre aléatoire dans l'intervalle [a,b]. On se souvient que la méthode `[metier].random` peut lancer des exceptions. On les gère ;
- ligne 27 : le contrôleur MVC rend une vue (View) avec son modèle (Map). Ici la `View` sera celle de la ligne 15, une `JacksonView` définie dans le fichier de configuration de Spring, donc une vue qui transforme son modèle en objet JSON. Le modèle ici sera un dictionnaire (Map) avec la clé "data" et la valeur associée le nombre aléatoire, par exemple "140". La chaîne JSON correspondant à ce modèle est la suivante :

```
{"data":"140"}
```

C'est donc cette chaîne que recevra le client.

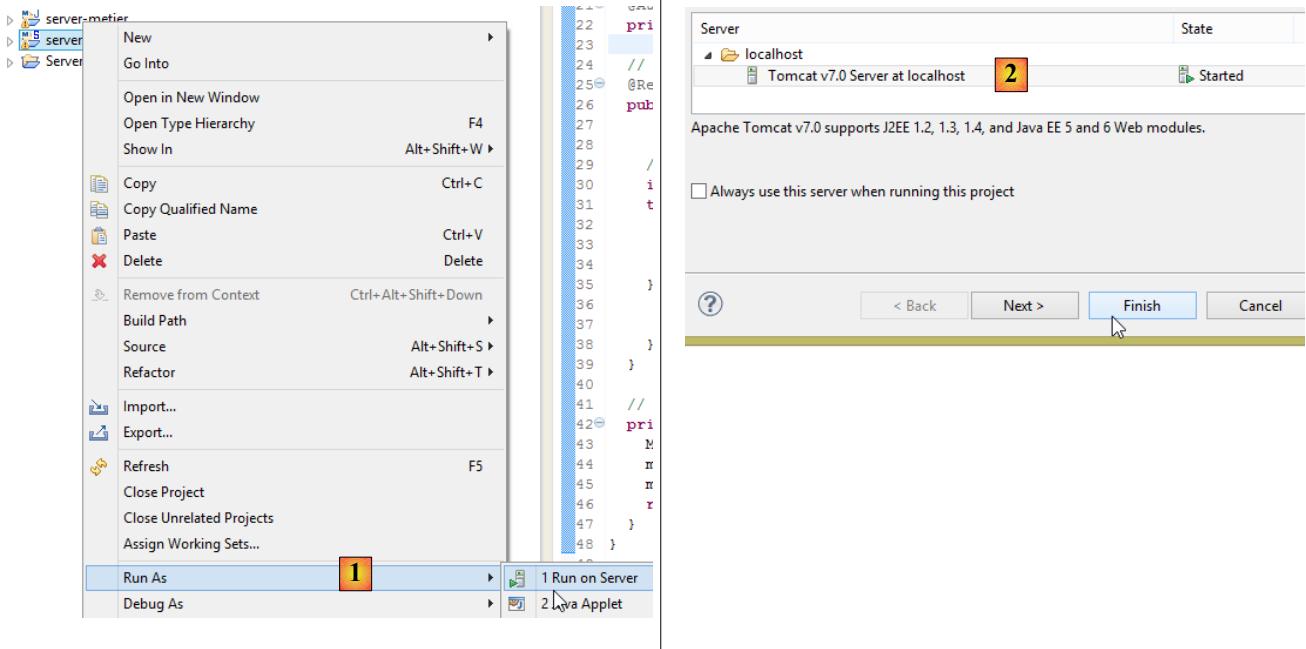
- lignes 35-39 : la vue envoyée lorsque se produit une exception ;
- lignes 36-38 : on construit un dictionnaire avec deux clés :
 - "error" : un n° d'erreur. Ici ce sera 103,
 - "message" : le message de l'erreur ;

La chaîne JSON envoyée au client aura la forme

```
{"error":"103", "message":"...”}
```

Ceux qui ne connaissent pas le format JSON trouveront page 168, paragraphe 11.1, un exemple de lecture / écriture de JSON avec la bibliothèque Jackson.

4.3.2.6 Déploiement et test du service REST



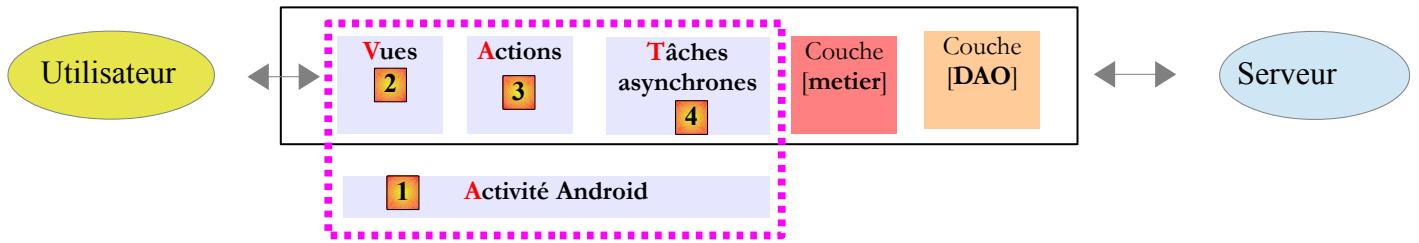
- en [1, 2] : le service REST est exécuté sur le serveur Tomcat ;

- en [3], le fichier [index.jsp] de l'application web a été exécuté ;
- en [4], on demande l'URL du service de génération du nombre aléatoire. Ici, on demande un nombre entre 100 et 200 ;
- en [5], la réponse JSON ;

- en [6] : en rafraîchissant la page un certain nombre de fois, on récupère l'exception aléatoire générée par la couche [métier], de nouveau au format JSON.

4.4 Le client Android

Le client Android a l'architecture suivante :



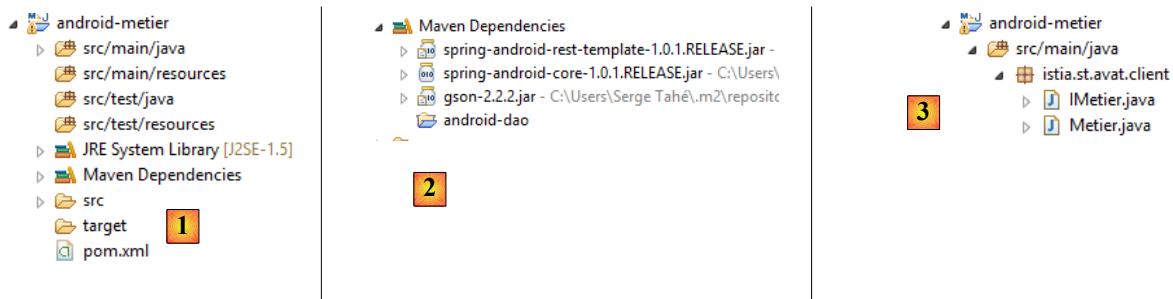
Le client a trois composantes :

1. la couche [AVAT] que nous avons étudiée dans l'exemple précédent ;
2. la couche [métier] qui va présenter une interface analogue à celle de la couche [métier] du serveur ;
3. la couche [DAO] qui s'adresse au service REST que nous avons étudié précédemment.

4.4.1 La couche [metier]

4.4.1.1 Le projet Eclipse

Le projet Eclipse est le suivant :



- en [1], le projet Eclipse de la couche [métier] ;
- en [2], les dépendances Maven du projet ;
- en [3], l'implémentation de la couche [métier] ;

4.4.1.2 Les dépendances Maven

Le fichier [pom.xml] du projet Maven est le suivant :

```

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
   4.0.0.xsd">
3.   <modelVersion>4.0.0</modelVersion>
4.   <groupId>istia.st.avat.client</groupId>
5.   <artifactId>android-metier</artifactId>
6.   <version>1.0-SNAPSHOT</version>
7.   <properties>
8.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
9.   </properties>
10.
11.  <dependencies>
12.    <dependency>
13.      <groupId>istia.st.avat.client</groupId>
14.      <artifactId>android-dao</artifactId>

```

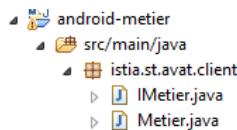
```

15.      <version>1.0-SNAPSHOT</version>
16.    </dependency>
17.  </dependencies>
18.
19. </project>

```

Le projet de la couche [métier] a une unique dépendance (lignes 12-15) sur la couche [DAO] que nous n'avons pas encore écrite.

4.4.1.3 Implémentation de la couche [métier]



Rappelons l'interface de la couche [métier] du serveur :

```

1. package istia.st.avat.server;
2.
3. public interface IMetier {
4.
5.     // génération d'un nombre aléatoire
6.     public abstract int randomNumber(int a, int b);
7.
8. }

```

Celle du client sera analogue :

```

1. package istia.st.avat.client;
2.
3. import istia.st.avat.client.dao.IDao;
4.
5. public interface IMetier {
6.
7.     // génération d'un nombre aléatoire
8.     public abstract int randomNumber(String urlServiceRest,int a, int b);
9.
10.    public abstract void setDao(IDao dao);
11. }

```

- ligne 8 : la méthode de génération du nombre aléatoire. Un nouveau paramètre s'ajoute : celle de l'URL du service REST qui délivre ce nombre ;
- ligne 10 : la couche [métier] a besoin de la couche [DAO]. C'est cette dernière qui assure les échanges avec le service REST.

L'implémentation [Metier] du client est la suivante :

```

1. package istia.st.avat.client;
2.
3. import istia.st.avat.client.dao.ClientException;
4. import istia.st.avat.client.dao.IDao;
5.
6. import java.util.HashMap;
7. import java.util.Map;
8.
9. import com.google.gson.Gson;
10.
11. public class Metier implements IMetier {
12.

```

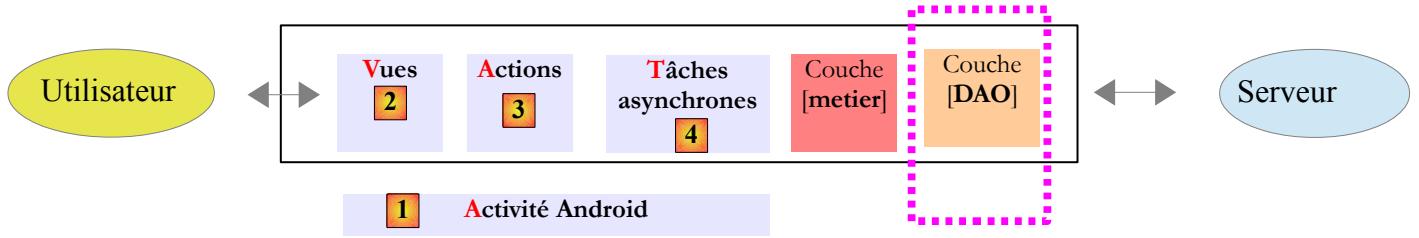
```

13. // couche [dao]
14. private IDao dao;
15. // mapper JSON
16. private Gson gson = new Gson();
17.
18. // génération d'un nombre aléatoire
19. public int randomNumber(String urlServiceRest, int a, int b) {
20.     // adresse du service REST
21.     String urlService = String.format("http://%s/random/{a}/{b}", urlServiceRest);
22.     // paramètres service REST
23.     Map<String, String> paramètres = new HashMap<String, String>();
24.     paramètres.put("a", String.valueOf(a));
25.     paramètres.put("b", String.valueOf(b));
26.     // exécution service
27.     String réponse = dao.executeRestService("get", urlService, null, paramètres);
28.     // exploitation réponse
29.     try {
30.         return Integer.parseInt(gson.fromJson(réponse, String.class));
31.     } catch (Exception ex) {
32.         throw new ClientException(String.format("Réponse incorrecte du serveur: %s",
réponse), ex);
33.     }
34. }
35. // setters
36.
37. public void setDao(IDao dao) {
38.     this.dao = dao;
39. }
40.
41. }

```

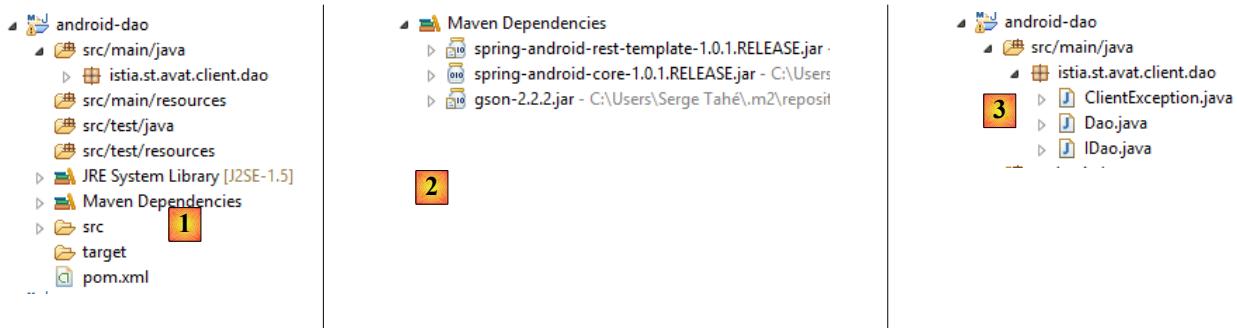
- lignes 3-4 : la couche [métier] s'appuie sur la couche [DAO] ;
 - ligne 9 : elle utilise une bibliothèque JSON de Google appelée **Gson**. Ceci nous est imposé par la bibliothèque utilisée [spring-android-rest-template]. Cette bibliothèque de Spring va s'interfacer avec le serveur REST SpringMVC. On trouvera page 169, paragraphe 11.2, un exemple de lecture / écriture de JSON avec la bibliothèque Gson ;
 - ligne 14 : une référence sur la couche [DAO]. Elle sera instanciée par la fabrique d'objets du modèle AVAT ;
 - ligne 16 : l'objet Gson qui va nous permettre de sérialiser / déserialiser des objets JSON ;
 - ligne 19 : la méthode de génération du nombre aléatoire. Le 1^{er} paramètre est l'adresse du service REST, les deux autres les bornes de l'intervalle [a,b] dans lequel on tire le nombre aléatoire ;
 - ligne 21 : on construit l'URL complète du service REST demandé. On notera bien la syntaxe des variables a et b dans l'URL ;
 - lignes 23-25 : un dictionnaire dont les clés sont les variables de l'URL demandée ;
 - ligne 27 : on exécute la méthode `[dao].executeRestService` de la couche [DAO] avec les paramètres suivants :
 1. la méthode " get " ou " post " de la requête HTTP à émettre,
 2. l'URL complète du service REST à exécuter,
 3. un dictionnaire des données transmises par une opération HTTP POST. Donc *null* ici, puisqu'on fait une opération HTTP GET,
 4. sous la forme d'un dictionnaire, les valeurs des variables de l'URL, ici les variables {a} et {b} ;
- On ne gère pas d'exception. On verra prochainement que la couche [DAO] lance des exceptions non contrôlées. Elles vont remonter toutes seules jusqu'à la couche [Présentation] assurée par le modèle AVAT ;
- lignes 29-33 : on a reçu une chaîne JSON du serveur. On attend un entier aléatoire dans l'intervalle [a,b]. On essaie donc de transformer la chaîne en entier. Si on y arrive, on rend ce nombre (ligne 30) comme résultat de la méthode sinon on lance une exception (ligne 32).

4.4.2 La couche [DAO]



4.4.2.1 Le projet Eclipse

Le projet Eclipse est le suivant :



- en [1], le projet Eclipse de la couche [DAO] ;
- en [2], les dépendances Maven du projet ;
- en [3], l'implémentation de la couche [DAO] ;

4.4.2.2 Les dépendances Maven

Le fichier [pom.xml] du projet Maven est le suivant :

```

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4. 4.0.0.xsd">
5.   <modelVersion>4.0.0</modelVersion>
6.   <groupId>istia.st.avat.client</groupId>
7.   <artifactId>android-dao</artifactId>
8.   <version>1.0-SNAPSHOT</version>
9.   <properties>
10.    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
11.    <spring-android-version>1.0.1.RELEASE</spring-android-version>
12.    <com.google.code.gson-version>2.2.2</com.google.code.gson-version>
13.  </properties>
14.  <repositories>
15.    <repository>
16.      <id>springsource-repo</id>
17.      <name>Spring Spring Repository</name>
18.      <url>http://repo.springsource.org/release</url>
19.    </repository>
20.  </repositories>
21.  <dependencies>
```

```

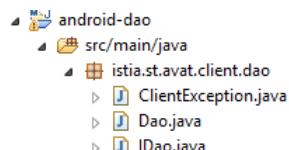
23.      <dependency>
24.          <groupId>org.springframework.android</groupId>
25.          <artifactId>spring-android-rest-template</artifactId>
26.          <version>${spring-android-version}</version>
27.      </dependency>
28.      <!-- Gson JSON Processor -->
29.      <dependency>
30.          <groupId>com.google.code.gson</groupId>
31.          <artifactId>gson</artifactId>
32.          <version>${com.google.code.gson-version}</version>
33.      </dependency>
34.  </dependencies>
35. </project>

```

Le projet de la couche [DAO] a deux dépendances :

- la communication avec le serveur REST va être assurée par un client REST fourni par la bibliothèque [spring-android-rest-template] (lignes 23-27) ;
- l'échange d'objets JSON entre le client et le serveur REST nous oblige à avoir une bibliothèque de sérialisation / désérialisation d'objets JSON. Ce sera la bibliothèque Gson de Google (lignes 29-33).

4.4.2.3 *Implémentation de la couche [DAO]*



L'interface de la couche [DAO] est la suivante :

```

1. package istia.st.avat.client.dao;
2.
3. import java.util.Map;
4.
5. public interface IDao {
6.     public String executeRestService(String method, String urlService, Object request,
7.         Map<String, String> paramètres);
8.     public void setTimeout(int millis);
9. }

```

- ligne 6 : la méthode [executeRestService] dont nous avons parlé précédemment ;
- ligne 8 : une méthode pour fixer un temps de réponse maximal de la part du serveur REST. Passé ce délai, la couche [DAO] lance une exception [ClientException]. Ce temp est fixé en millisecondes.

L'implémentation est la suivante :

```

1. package istia.st.avat.client.dao;
2.
3. ...
4.
5. import org.springframework.http.converter.StringHttpMessageConverter;
6. import org.springframework.http.converter.json.GsonHttpMessageConverter;
7. import org.springframework.web.client.RestTemplate;
8.
9. public class Dao implements IDao {
10.
11.     // client REST

```

```

12.     private RestTemplate restTemplate;
13.     // mapper JSON
14.     private Gson gson = new Gson();
15.     // délai d'attente maximal
16.     private int timeout;
17.
18.     // constructeur
19.     public Dao() {
20.         // on crée un objet [RestTemplate]
21.         restTemplate = new RestTemplate();
22.         // on le configure
23.         restTemplate.getMessageConverters().add(new GsonHttpMessageConverter());
24.         restTemplate.getMessageConverters().add(new StringHttpMessageConverter());
25.     }
26.
27.     // exécution de l'appel au service REST
28.     public String executeRestService(String method, String urlService, Object request,
29.         Map<String, String> paramètres) {
30.
31.         // on vérifie que le serveur distant répond assez vite
32.         // une exception est lancée sinon
33.         checkResponsiveness(urlService);
34.         // vérification méthode HTTP
35.         method = method.toLowerCase();
36.         if (!method.equals("get") && !method.equals("post")) {
37.             throw new ClientException("L'argument [method] doit avoir la valeur post ou get
38.             dans [%s]");
39.         }
40.         try {
41.             // exécution service
42.             String réponse = null;
43.             if (method.equals("get")) {
44.                 réponse = restTemplate.getForObject(urlService, String.class, paramètres);
45.             } else {
46.                 réponse = restTemplate.postForObject(urlService, request, String.class,
47.                     paramètres);
48.             }
49.             // résultat
50.             Map<String, Object> map = gson.fromJson(réponse, new TypeToken<Map<String,
51.                 Object>>() {
52.                     .getType());
53.                     // erreur ?
54.                     String erreur = (String) map.get("error");
55.                     if (erreur != null) {
56.                         throw new ClientException(String.format("Le serveur a renvoyé l'erreur
57.                         suivante : %s",
58.                             (String) map.get("message")));
59.                     }
60.                     // data ?
61.                     Object data = map.get("data");
62.                     if (data == null) {
63.                         throw new ClientException(String.format("Réponse incorrecte du serveur : %s",
64.                             réponse));
65.                     }
66.             }
67.             // c'est bon - on rend le résultat
68.             return gson.toJson(data);
69.         } catch (Exception ex) {
70.             throw new ClientException(String.format("Une erreur s'est produite :
71.             %s",ex.getMessage()));
72.         }
73.     }
74.
75.     private void checkResponsiveness(String urlService) {

```

```

68. ...
69. }
70.
71. // délai d'attente maximal
72. public void setTimeout(int millis) {
73.     this.timeout = millis;
74. }
75.
76. }

```

- lignes 5-7 : la couche [DAO] s'appuie sur un client REST fourni par la bibliothèque [spring-android-framework]. Le principal élément de cette bibliothèque est la classe [RestTemplate] de la ligne 7. Le cœur du framework Spring n'a pas encore été porté sur Android. Alors qu'habituellement, le champ de la ligne 7 aurait été initialisé par Spring, ici il le sera par le constructeur ;
- ligne 21 : l'objet [RestTemplate] est instancié ;
- lignes 23-24 : il est configuré. On lui donne ici à la main la liste de " convertisseurs de message " qu'on avait définie dans le fichier de configuration du serveur REST. Dans ce dernier, on n'avait défini que le convertisseur [JacksonHttpMessageConverter]. Ici pour le remplacer on utilise le convertisseur [GsonHttpMessageConverter] qui utilise la bibliothèque Gson. On définit également le convertisseur [StringHttpMessageConverter]. On n'en avait pas eu besoin pour le serveur. Ici, le client plante sans ce convertisseur ;
- ligne 28 : la méthode [executeRestService] reçoit les paramètres suivants :
 1. la méthode " get " ou " post " de la requête HTTP à émettre,
 2. l'URL complète du service REST à exécuter,
 3. un dictionnaire des données transmises par une opération HTTP POST ;
 4. sous la forme d'un dictionnaire, les valeurs des variables de l'URL ;
- ligne 32 : on vérifie que le serveur REST répond au bout de *timeout* millisecondes. Si ce n'est pas le cas, la méthode [checkResponsiveness] lance une exception ;
- lignes 34-37 : vérification de la méthode HTTP ;
- lignes 38-45 : exécution du service REST par l'objet [RestTemplate] de Spring (ligne 12). Cette classe a de nombreuses méthodes pour dialoguer avec un service REST. Celles utilisées ici vont rendre comme résultat, la chaîne JSON renvoyée par le serveur. On se rappelle que celui-ci peut renvoyer deux types de chaînes JSON :

```

1. {"data":"131"}
2. {"error":"103","message":"Exception aléatoire"}

```

Ligne 1, la chaîne contenant le nombre aléatoire, ligne 2, la chaîne contenant un message d'erreur.

- ligne 47 : on transforme la chaîne JSON reçue en dictionnaire ;
- ligne 50 : on regarde si dans ce dictionnaire il y a la clé "error". Si oui, on lance une [ClientException] (lignes 51-54) ;
- ligne 56-59 : on regarde si dans ce dictionnaire il y a la clé "data". Si non, on lance une [ClientException] (lignes 57-59) ;
- ligne 61 : on retourne le nombre aléatoire.

La méthode [checkResponsiveness] est la suivante :

```

1. private void checkResponsiveness(String urlService) {
2.     // on crée l'URI du service distant
3.     String url = urlService.replace("{", "").replace("}", "");
4.     URI service = null;
5.     try {
6.         service = new URI(url);
7.     } catch (URISyntaxException ex) {
8.         throw new ClientException(String.format("Format d'URL incorrect [%s]", urlService), ex);
9.     }
10.    // on se connecte au service
11.    Socket client = null;
12.    try {
13.        // avec attente maximale définie par configuration
14.        client = new Socket();
15.        client.connect(new InetSocketAddress(service.getHost(), service.getPort()),
16.                      timeout);
16.    } catch (IOException e) {
17.        // le serveur n'a pas répondu

```

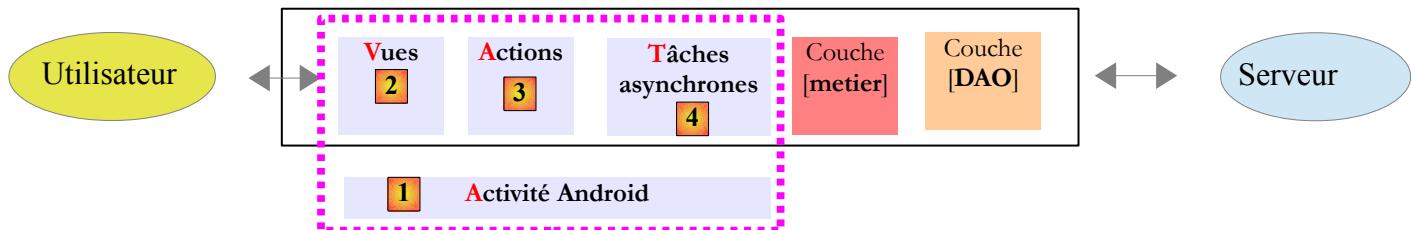
```

18.         throw new ClientException(String.format("Le service distant n'a pas répondu
    assez vite : %s", e.getMessage()));
19.     } finally {
20.         // on libère les ressources
21.         if (client != null) {
22.             try {
23.                 client.close();
24.             } catch (IOException ex) {
25.                 Logger.getLogger(Dao.class.getName()).log(Level.SEVERE, null, ex);
26.             }
27.         }
28.     }

```

- ligne 1 : le paramètre [urlService] est de la forme [http://localhost:8080/server-rest/random/{a}/{b}];
- ligne 3 : l'URL précédente devient [http://localhost:8080/server-rest/random/a/b];
- lignes 5-9 : à partir de cette URL, on essaie de construire un objet URI. Si on n'y arrive pas c'est que l'URL est incorrecte ;
- lignes 14-15 : on se connecte à la machine et au port définis par l'URI qui vient d'être construite et on donne un temps maximum de *timeout* millisecondes pour obtenir la réponse (ligne 15) ;
- ligne 18 : si pour une raison ou une autre (absence du serveur ou délai d'attente dépassé), la connexion échoue alors on lance une exception.

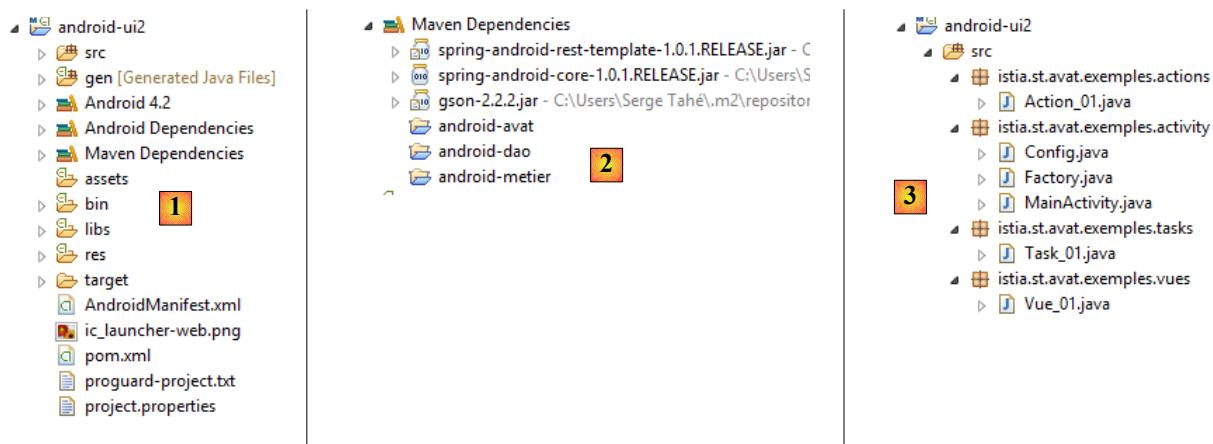
4.4.3 La couche [AVAT]



La couche [AVAT] change peu vis à vis de la version précédente :

- la vue demande une information supplémentaire, l'URL du service REST ;
- la tâche asynchrone [Task_01] évolue.

4.4.3.1 Le projet Eclipse



- en [1], le projet Eclipse de la couche [AVAT] ;
- en [2], les dépendances Maven du projet ;

- en [3], l'implémentation de la couche [AVAT] ;

4.4.3.2 Les dépendances Maven

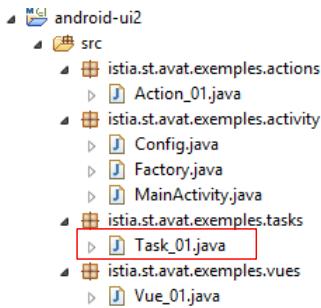
Le fichier [pom.xml] du projet est le suivant :

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
   v4_0_0.xsd">
4.   <modelVersion>4.0.0</modelVersion>
5.   <groupId>istia.st.avat.examples</groupId>
6.   <artifactId>android-ui2</artifactId>
7.   <version>1.0-SNAPSHOT</version>
8.   <packaging>apk</packaging>
9.
10.  <properties>
11.    <platform.android.version>4.1.1.4</platform.android.version>
12.    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
13.    <android.sdk.path>D:\\Programs\\devjava\\Android\\sdk</android.sdk.path>
14.    <spring-android-version>1.0.1.RELEASE</spring-android-version>
15.  </properties>
16.
17.  <dependencies>
18.    <dependency>
19.      <groupId>com.google.android</groupId>
20.      <artifactId>android</artifactId>
21.      <version>${platform.android.version}</version>
22.      <scope>provided</scope>
23.    </dependency>
24.    <dependency>
25.      <groupId>istia.st.android</groupId>
26.      <artifactId>android-avat</artifactId>
27.      <version>1.0-SNAPSHOT</version>
28.      <scope>compile</scope>
29.    </dependency>
30.    <dependency>
31.      <groupId>istia.st.avat.client</groupId>
32.      <artifactId>android-metier</artifactId>
33.      <version>1.0-SNAPSHOT</version>
34.    </dependency>
35.  </dependencies>
36.
37.  <build>
38.    <plugins>
39.      ...
40.    </plugins>
41.  </build>
42. </project>
```

- lignes 18-23 : dépendance sur la plateforme Android ;
- lignes 24-29 : dépendance sur le modèle [AVAT] ;
- lignes 30-34 : dépendance sur la couche [métier] du client ;

4.4.3.3 La tâche [Task_01]



La classe [Task_01] devient la suivante :

```

1. package istia.st.avat.examples.tasks;
2.
3. ...
4.
5. public class Task_01 extends Task {
6.
7.     // info renvoyée par la tâche
8.     private Object info;
9.
10.    @Override
11.    public void onPreExecute(){
12.    ...
13.    }
14.
15.    @Override
16.    // on exécute la tâche (urlServiceRest, a, b, sleepTime)
17.    protected Void doInBackground(Object... params) {
18.        // il faut quatre paramètres
19.        if (params.length != 4) {
20.            info = new ClientException("Il faut quatre paramètres");
21.            return null;
22.        }
23.        // on récupère les paramètres
24.        String urlServiceRest = null;
25.        int a = 0;
26.        int b = 0;
27.        int sleepTime = 0;
28.
29.        try {
30.            urlServiceRest = (String) params[0];
31.            a = (Integer) params[1];
32.            b = (Integer) params[2];
33.            sleepTime = (Integer) params[3];
34.        } catch (Exception ex) {
35.            info = new ClientException(String.format("Paramètres incorrects [%s, %d, %d, %d]", urlServiceRest, a, b, sleepTime));
36.            return null;
37.        }
38.        // vérification des bornes [a,b]
39.        if (a > b) {
40.            info = new ClientException(String.format("%d doit être >= à %d", b, a));
41.            return null;
42.        }
43.        // attente
44.        try {
45.            Thread.sleep(sleepTime);
46.        } catch (InterruptedException e) {

```

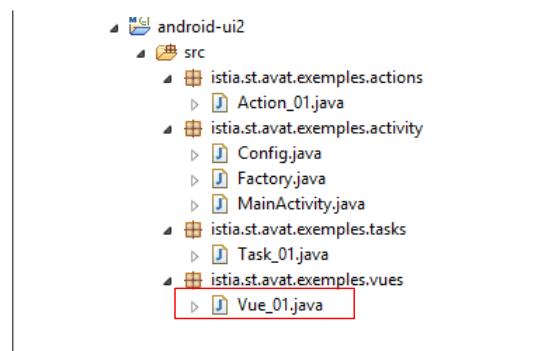
```

47.         info = e;
48.         return null;
49.     }
50.     // on demande le nombre à la couche métier
51.     IMetier metier = (IMetier) factory.getObject(Factory.METIER, (Object[]) null);
52.     try {
53.         info = metier.randomNumber(urlServiceRest, a, b);
54.     } catch (Exception ex) {
55.         info = ex;
56.     }
57.     // fin
58.     return null;
59. }
60.
61. @Override
62. // traitement fin de tâche dans le thread de l'UI
63. protected void onPostExecute(Void result) {
64. ...
65. }
66.
67. }

```

- ligne 17 : dans la version précédente, la méthode [doWork] recevait trois paramètres. Elle en reçoit maintenant un de plus : l'URL du service REST qui délivre les nombres aléatoires ;
- ligne 51 : le nombre aléatoire est délivré par la couche [métier] du client. La tâche demande à la fabrique d'objets une référence sur cette couche. Celle-ci est un singleton. La fabrique ne la crée qu'en un unique exemplaire ;
- ligne 53 : le nombre aléatoire est demandé à la couche [métier].

4.4.3.4 La vue [Vue_01]



La vue [Vue_01] évolue pour demander une information supplémentaire : l'URL du service REST :



Avat-Exemple-02

Génération de N nombres aléatoires par N tâches asynchrones

URL du service REST: [1]

Valeur de N :

Intervalle [a,b] de génération, a : b :

Durée d'attente des tâches en millisecondes :

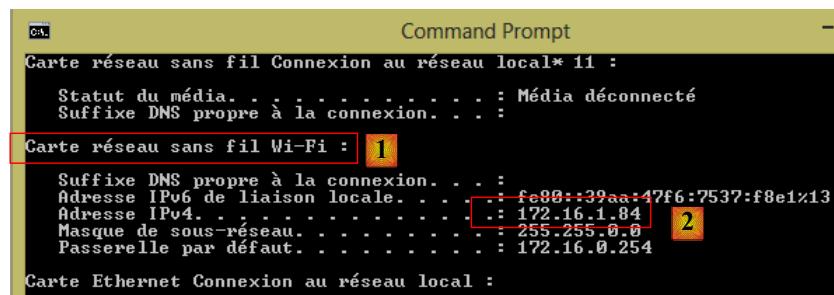
Exécuter

Liste des réponses

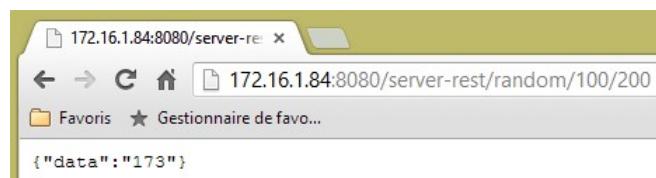
Nb Exceptions=7, Nb Aléatoires=13, Somme des nombres reçus = 2085
183
171
116

Pour mettre la bonne URL en [1], on peut procéder ainsi :

- taper la commande [ipconfig] dans une fenêtre [DOS] :



- prendre l'adresse IP [2] de la carte Wifi [1] ;
- déployer le serveur REST sous Eclipse et demander l'URL du service REST avec un navigateur. Utiliser l'adresse IP précédente ;



- l'URL du service REST à taper sur la tablette est alors [172.16.1.84:8080/server-rest]. Il faudra peut-être désactiver le pare-feu du PC voire inhiber un éventuel logiciel antivirus qui pourrait bloquer les connexions entrantes. C'est par exemple le cas de McAfee.

4.4.3.5 La configuration de l'application

La classe [Config] qui configure l'application est la suivante :

- package istia.st.avat.examples.activity;
- .

```

3. public class Config {
4.
5.     // le mode verbeux ou non
6.     private boolean verbose = true;
7.     // le temps d'attente de la réponse du serveur REST en millisecondes
8.     private int timeout=1000;
9.
10.    // getters et setters
11.    public boolean isVerbose() {
12.        return verbose;
13.    }
14.
15.    public int getTimeout() {
16.        return timeout;
17.    }
18.
19. }

```

4.4.3.6 La fabrique d'objets

Il y a davantage d'objets dans cette application que dans la précédente. La fabrique d'objets est la suivante :

```

1. package istia.st.avat.exemples.activity;
2.
3. ...
4.
5. public class Factory implements IFactory {
6.
7.     // constantes
8.     public static final int CONFIG = -1;
9.     public static final int METIER = -2;
10.    public static final int TASK_01 = 1;
11.    public static final int ACTION_01 = 3;
12.    public static final int VUE_01 = 4;
13.
14.    // ----- DEBUT SINGLETONS
15.    // la configuration
16.    private Config config;
17.    private boolean verbose;
18.    private int timeout;
19.
20.    // la couche [métier]
21.    private IMetier metier;
22.
23.    // les vues
24.    private Vue vue_01;
25.
26.    // l'activité principale
27.    private Activity activity;
28.
29.    // ----- FIN SINGLETONS
30.
31.    // constructeur
32.    public Factory(Activity activity, Config config) {
33.        // l'activité
34.        this.activity = activity;
35.        // la configuration
36.        this.config = config;
37.        // le mode verbeux ou non
38.        verbose = config.isVerbose();
39.        // le temps d'attente de la réponse du serveur REST
40.        timeout = config.getTimeout();

```

```

41. }
42.
43. @Override
44. public Object getObject(int id, Object... params) {
45.     switch (id) {
46.         case METIER:
47.             return getMetier();
48.         case CONFIG:
49.             return config;
50.         case TASK_01:
51.             return getTask_01(params);
52.         case ACTION_01:
53.             return getAction_01(params);
54.         case VUE_01:
55.             return getVue_01(params);
56.     }
57.     return null;
58. }
59.
60. private Object getMetier() {
61.     // metier est un singleton
62.     if (metier == null) {
63.         // couche [dao]
64.         IDao dao = new Dao();
65.         dao.setTimeout(timeout);
66.         // couche [metier]
67.         metier = new Metier();
68.         metier.setDao(dao);
69.     }
70.     // fin
71.     return metier;
72. }
73.
74. private Object getVue_01(Object... params) {
75. ...
76. }
77.
78. // une tâche reçoit deux paramètres : IBoss, id
79. private Object getTask_01(Object[] params) {
80. ...
81. }
82.
83. // une action reçoit deux paramètres : IBoss, id
84. private Object getAction_01(Object[] params) {
85. ...
86. }
87.
88. }

```

- lignes 60-72 : instanciation de la couche [métier] ;
- ligne 64 : on instancie la couche [DAO] ;
- ligne 65 : on lui injecte son *timeout* ;
- ligne 67 : on instancie la couche [métier] ;
- ligne 68 : on lui injecte une référence sur la couche [DAO] ;
- ligne 62 : la couche [métier] est un singleton. Une fois créé, il est recyclé.

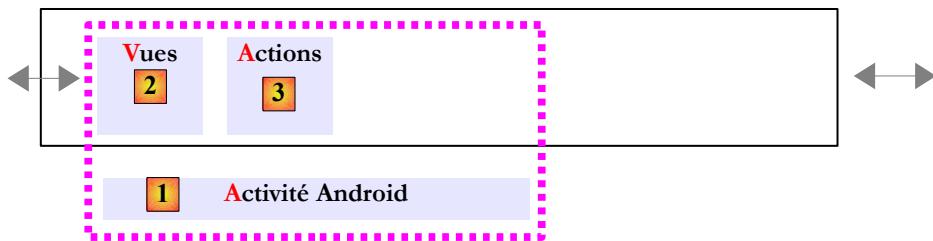
5 AVAT- Exemple 3

5.1 Le projet

Nous nous proposons maintenant de montrer qu'il y a une vie en-dehors des tâches asynchrones et qu'on peut même alors, continuer à utiliser le modèle AVAT. C'est un peu baroque mais ça marche.

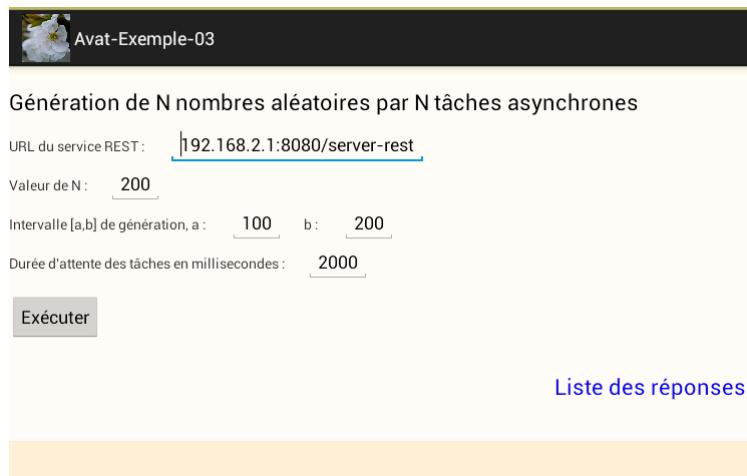
L'essentiel ayant été dit sur le modèle AVAT, nous commençons une série d'exemples courts et rapidement expliqués. Nous ne détaillons que des concepts que nous n'aurions pas encore vus. Le reste est disponible dans les codes source livrés avec ce document. Nous terminerons par une application plus complexe de client d'un service web de gestion de rendez-vous.

L'application de ce nouvel exemple a l'architecture suivante :



La couche [AVAT] est seule et sans tâches asynchrones.

L'application aura la même vue que dans l'exemple précédent :



A l'exécution, l'action [Action_01] se contentera de renvoyer à la vue les informations que celle-ci lui aura données :

Avat-Exemple-03

Génération de N nombres aléatoires par N tâches asynchrones

URL du service REST :

Valeur de N :

Intervalle [a,b] de génération, a : b :

Durée d'attente des tâches en millisecondes :

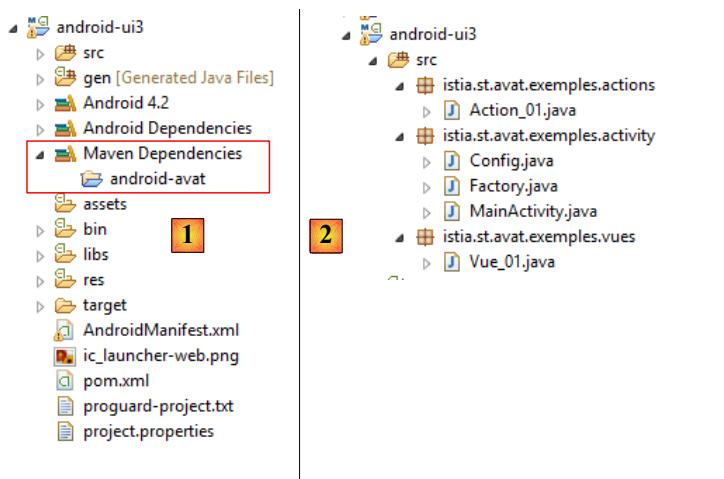
Exécuter

Liste des réponses

192.168.2.1:8080/server-rest
200
100
200

5.2 Le projet Eclipse

Le projet Eclipse de cet exemple est le suivant :



- en [1], le projet [android-ui3] est un projet Maven ayant une dépendance sur le projet [android-avat] ;
- en [2], les sources du projet :
 - le package [istia.st.avat-exemples.vues] rassemble les vues du projet, ici une vue ;
 - le package [istia.st.avat-exemples.actions] rassemble les actions du projet, ici une action ;
 - le package [istia.st.avat-exemples.activity] contient l'activité Android [MainActivity], la fabrique d'objets [Factory] et une classe de configuration [Config].

On notera que l'application n'a pas de tâches.

5.3 L'action [Action_01]

L'action [Action_01] se contente de renvoyer à la vue les informations que celles-ci lui a transmises. Son code est le suivant :

```

1. package istia.st.avat.exemples.actions;
2.
3. ...
4.
5. public class Action_01 extends Action {
6.
7.     // la liste des nombres aléatoires générés par les tâches lancées
8.     List<Integer> aleas = new ArrayList<Integer>();
9.
10.    @Override
11.    // on exécute l'action (urlServiceRest,nbAleas,a,b, sleepTime)
12.    public void doWork(Object... params) {
13.        // on renvoie les paramètres après un délai d'attente
14.        try {
15.            Thread.sleep(1000);
16.            // on renvoie les paramètres
17.            boss.notifyEvent(this, IBoss.WORK_INFO, params);
18.        } catch (InterruptedException e) {
19.            // on renvoie l'exception
20.            boss.notifyEvent(this, IBoss.WORK_INFO, e);
21.        }
22.        // fin
23.        boss.notifyEvent(this, IBoss.WORK_TERMINATED, (Object[])null);
24.    }
25.
26.    @Override
27.    public void notifyEndOfTasks() {
28.    }
29.
30. }
```

- ligne 11 : la méthode [doWork] reçoit quatre paramètres de la vue ;
- lignes 15 : elle commence par s'assoupir 1 seconde pour simuler un travail ;
- ligne 17 : ensuite elle renvoie à son boss (la vue) les paramètres qu'elle a reçus dans le même ordre ;
- ligne 20 : s'il se produit une exception, celle-ci est également envoyée au boss ;
- ligne 23 : l'action signale à son boss qu'elle a terminé son travail ;
- ligne 27 : l'action en tant que [IBoss] doit implémenter la méthode [notifyEndOfTasks]. Ici l'action n'ayant pas de tâches, cette méthode ne sera jamais appelée.

5.4 La vue [Vue_01]

La vue [Vue_01] est la suivante :

```

1. package istia.st.avat.exemples.vues;
2.
3. ....
4.
5. public class Vue_01 extends Vue {
6.
7. ....
8.     @Override
9.     public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
   savedInstanceState) {
10. ...
11. }
12.
13.     @Override
14.     public void onActivityCreated(Bundle savedInstanceState) {
15. ...
16. }
17. }
```

```

18.     protected void doExecuter() {
19. ...
20.         // on exécute l'action (urlServiceRest,nbAleas,a,b, sleepTime)
21.         action.doWork(edtUrlServiceRest.getText().toString(),
22.             Integer.parseInt(edtNbAleas.getText().toString()),
23.             Integer.parseInt(edtA.getText().toString()),
24.             Integer.parseInt(edtB.getText().toString()),
25.             Integer.parseInt(edtSleepTime.getText().toString())));
26. ...
27.     }
28. ...
29. ...
30. ...
31. ...
32. ...
33. ...
34. ...
35. ...
36. ...
37. ...
38. ...
39. ...
40. ...
41. ...
42. ...
43. ...
44. ...
45. ...
46. ...
47. ...
48. ...
49. ...
50. ...
51. ...
52. ...
53. ...
54. ...
55. ...
56. ...
57. ...
58. ...
59. ...
60. ...
61. ...
62. ...
63. ...
64. ...
65. ...
66. }

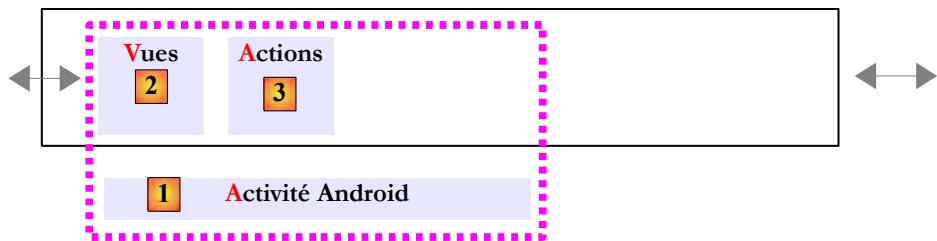
```

- lignes 20-21 : l'action [Action_01] est appelée avec les paramètres [urlServiceRest,nbAleas,a,b, sleepTime] ;
- ligne 34 : ces paramètres vont lui être renvoyées dans une notification [WORK_INFO] ;
- lignes 45-48 : ils sont alors placés dans la liste des réponses.

6 AVAT- Exemple 4

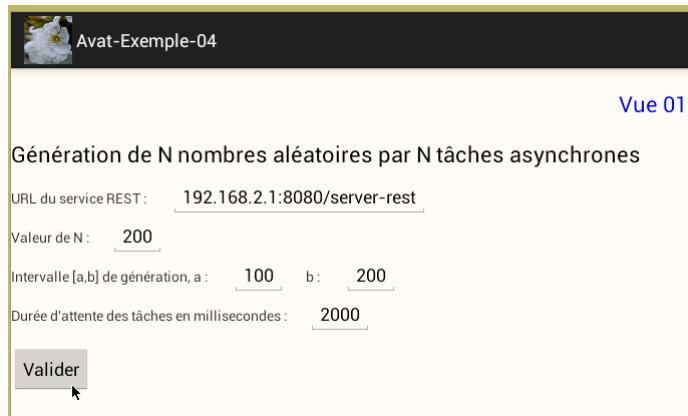
6.1 Le projet

L'application de ce nouvel exemple a elle également l'architecture suivante :

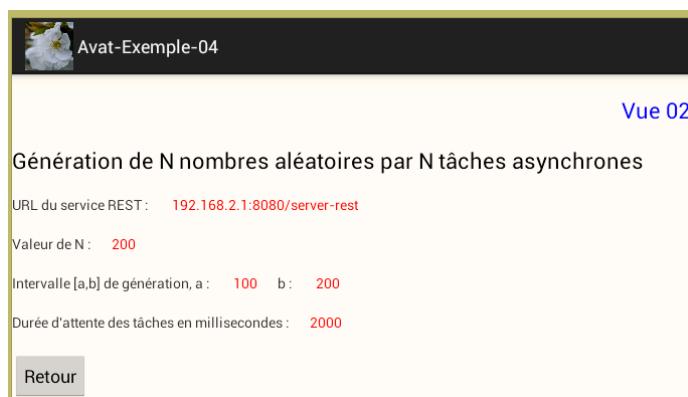


La couche [AVAT] est seule et sans tâches asynchrones.

L'application aura deux vues. La première est la même que dans l'exemple précédent :



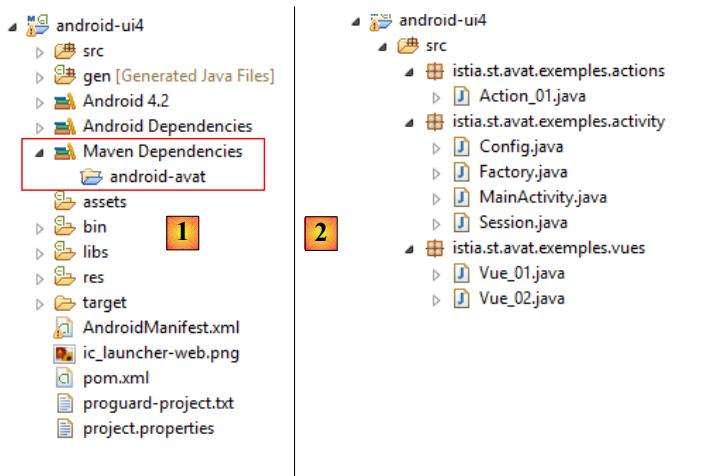
A l'exécution, l'action [Action_01] se contentera de renvoyer à la vue les informations que celles-ci lui aura données. La vue [Vue_01] mettra alors ces informations dans une session et demandera à l'activité Android d'afficher une seconde vue [Vue_02]. Celle-ci affichera alors les informations trouvées dans la session. La vue [Vue_02] est la suivante :



Elle réaffiche en rouge les données qu'elle a trouvées dans la session. Le bouton [Retour] ramène à la vue [Vue_01].

6.2 Le projet Eclipse

Le projet Eclipse de cet exemple est le suivant :



- en [1], le projet [android-ui4] est un projet Maven ayant une dépendance sur le projet [android-avat] ;
- en [2], les sources du projet :
 - le package [istia.st.avat-exemples.vues] rassemble les vues du projet, ici deux vues ;
 - le package [istia.st.avat-exemples.actions] rassemble les actions du projet, ici une action ;
 - le package [istia.st.avat-exemples.activity] contient l'activité Android [MainActivity], la fabrique d'objets [Factory], une classe de configuration [Config] et une implémentation de l'interface [ISession].

6.3 La session

Rappelons la définition de l'interface [ISession] du modèle AVAT :

```
1. package istia.st.avat.core;
2.
3. public interface ISession {
4.     // ajoute un élément dans la session
5.     void add(String id, Object value);
6.     // enlève un élément de la session
7.     void remove(String id);
8.     // recherche d'un élément
9.     Object get(String id);
10.    // vide la session
11.    void clear();
12. }
```

La classe [Session] l'implémente de la façon suivante :

```
1. package istia.st.avat.exemples.activity;
2.
3. import istia.st.avat.core.ISession;
4.
5. import java.util.HashMap;
6. import java.util.Map;
7.
8. public class Session implements ISession {
9.
10.    // data
11.    Map<String, Object> session = new HashMap<String, Object>();
```

```

12.
13.     public void add(String id, Object value) {
14.         session.put(id, value);
15.
16.     }
17.
18.     public void remove(String id) {
19.         session.remove(id);
20.
21.     }
22.
23.     public Object get(String id) {
24.         return session.get(id);
25.     }
26.
27.     public void clear() {
28.         session.clear();
29.
30.     }
31.
32. }
```

6.4 L'activité Android

Jusqu'à maintenant, l'activité Android n'affichait qu'une vue. Nous la modifions pour qu'elle puisse afficher n'importe quelle vue. Son code devient le suivant :

```

1. package istia.st.avat.examples.activity;
2.
3. ...
4.
5. public class MainActivity extends FragmentActivity {
6.
7.     // la factory
8.     private Factory factory;
9.     // les vues
10.    private Vue[] vues;
11.
12.    @Override
13.    protected void onCreate(Bundle savedInstanceState) {
14.        super.onCreate(savedInstanceState);
15.        // le sablier
16.        requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
17.        setProgressBarIndeterminateVisibility(false);
18.        // template qui accueille les vues
19.        setContentView(R.layout.main);
20.        // la factory
21.        factory = new Factory(this, new Config());
22.        // on définit les deux vues
23.        Vue_01 vue_01 = (Vue_01) factory.getObject(Factory.VUE_01, null, "Vue_01");
24.        Vue_02 vue_02 = (Vue_02) factory.getObject(Factory.VUE_02, null, "Vue_02");
25.        // on stocke leurs références dans un tableau
26.        vues = new Vue[] { vue_01, vue_02 };
27.        // on fait de vue_01 la vue courante
28.        showVue(1);
29.    }
30.
31.    // affichage d'une vue
32.    public void showVue(int i) {
33.        // i=1 --> vue_01, i=2 --> vue_02
34.        // affichage vue
```

```

35.     FragmentTransaction fragmentTransaction = getFragmentManager().beginTransaction();
36.     fragmentTransaction.replace(R.id.container, vues[i - 1]);
37.     fragmentTransaction.commit();
38. }
39.
40. }
```

Nous ne commentons que ce qui change :

- lignes 23-24 : les deux vues sont instanciées par la fabrique ;
- ligne 26 : leurs références sont stockées dans un tableau défini ligne 10 ;
- ligne 28 : la première vue est affichée via une méthode interne [showVue]. Les vues, qui ont accès à l'activité, demanderont le changement de vue par cette méthode.

6.5 L'action [Action_01]

L'action [Action_01] se contente de renvoyer à la vue les informations que celles-ci lui a transmises. Son code est le suivant :

```

1. package istia.st.avat.examples.actions;
2.
3. import istia.st.avat.android.Action;
4. import istia.st.avat.core.IBoss;
5.
6. public class Action_01 extends Action {
7.
8.     @Override
9.     // on exécute l'action (urlServiceRest,nbAleas,a,b, sleepTime)
10.    public void doWork(Object... params) {
11.        // on renvoie les paramètres reçus
12.        boss.notifyEvent(this, IBoss.WORK_TERMINATED, params);
13.    }
14.
15.    @Override
16.    public void notifyEndOfTasks() {
17.    }
18.
19. }
```

- ligne 11 : les paramètres reçus sont immédiatement renvoyés à la vue dans la notification [WORK_TERMINATED].

6.6 La vue [Vue_01]

La vue [Vue_01] est la suivante :

```

1. @Override
2.     public void notifyEndOfTasks() {
3.         // on affiche la vue 2
4.         ((MainActivity) activity).showVue(2);
5.     }
6.
7.     @Override
8.     public void notifyEvent(IWorker worker, int eventType, Object event) {
9.         // parent
10.        super.notifyEvent(worker, eventType, event);
11.        // on gère l'évt WORK_TERMINATED
12.        if (eventType == IBoss.WORK_TERMINATED) {
13.            // on met les résultats dans la session (urlServiceRest, nbAleas, a, b,
14.            // sleepTime)
```

```

15.         ISession session = (ISession) factory.getObject(Factory.SESSION, (Object[])
16.             null);
17.         Object[] results = (Object[]) event;
18.         session.add("urlServiceRest", results[0]);
19.         session.add("nbAleas", results[1]);
20.         session.add("a", results[2]);
21.         session.add("b", results[3]);
22.         session.add("sleepTime", results[4]);
23.     }

```

- ligne 12 : la vue gère la notification [WORK_TERMINATED] ;
- ligne 15 : une session est demandée à la fabrique. C'est un singleton ;
- lignes 16-21 : les cinq informations renvoyées par [Action_01] sont mémorisées dans la session ;
- lignes 2-5 : lorsqu'arrive la notification [endOfTasks], la vue [Vue_02] est affichée. On rappelle que l'activité Android est injectée dans chacune des vues par la fabrique dans le champ *activity*.

6.7 La vue [Vue_02]

La vue [Vue_02] est la suivante :

```

1. package istia.st.avat.examples.vues;
2.
3. ...
4.
5. public class Vue_02 extends Vue {
6.
7.     // les éléments de l'interface visuelle
8.     private Button btnRetour;
9.     private TextView edtNbAleas;
10.    private TextView edtA;
11.    private TextView edtB;
12.    private TextView edtUrlServiceRest;
13.    private TextView edtSleepTime;
14.
15.    // les saisies
16.    private int nbAleas;
17.    private int a;
18.    private int b;
19.    private String urlServiceRest;
20.    private int sleepTime;
21.
22.    @Override
23.    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
24.        savedInstanceState) {
25.        // on crée la vue du fragment à partir de sa définition XML
26.        return inflater.inflate(R.layout.vue_02, container, false);
27.    }
28.
29.    @Override
30.    public void onActivityCreated(Bundle savedInstanceState) {
31.        // parent
32.        super.onActivityCreated(savedInstanceState);
33.
34.        edtUrlServiceRest = (TextView) activity.findViewById(R.id.edt_urlServiceRest);
35.        edtNbAleas = (TextView) activity.findViewById(R.id.edt_nbaleas);
36.        edtA = (TextView) activity.findViewById(R.id.edt_a);
37.        edtB = (TextView) activity.findViewById(R.id.edt_b);
38.        edtSleepTime = (TextView) activity.findViewById(R.id.edt_sleepTime);
39.

```

```

40.    // bouton Retour
41.    btnRetour = (Button) activity.findViewById(R.id.btn_Retour);
42.    btnRetour.setOnClickListener(new OnClickListener() {
43.        @Override
44.        public void onClick(View arg0) {
45.            doRetour();
46.        }
47.    });
48.
49.    }
50.
51.    @Override
52.    public void onResume() {
53.        // parent d'abord
54.        super.onResume();
55.        // on récupère les résultats dans la session (urlServiceRest, nbAleas, a, b,
56.        // sleepTime)
57.        ISession session = (ISession) factory.getObject(Factory.SESSION, (Object[]) null);
58.        urlServiceRest = (String) session.get("urlServiceRest");
59.        nbAleas = (Integer) session.get("nbAleas");
60.        a = (Integer) session.get("a");
61.        b = (Integer) session.get("b");
62.        sleepTime = (Integer) session.get("sleepTime");
63.        // on met à jour la page avec ces éléments
64.        edtUrlServiceRest.setText(urlServiceRest);
65.        edtNbAleas.setText(String.valueOf(nbAleas));
66.        edtA.setText(String.valueOf(a));
67.        edtB.setText(String.valueOf(b));
68.        edtSleepTime.setText(String.valueOf(sleepTime));
69.    }
70.
71.    protected void doRetour() {
72.        // on affiche la vue 1
73.        ((MainActivity) activity).showVue(1);
74.    }
75.
76.    @Override
77.    public void notifyEndOfTasks() {
78.    }
79.
80. }

```

- ligne 52 : tout se passe dans la méthode [onResume] exécutée juste avant que l'interface ne devienne visible ;
- ligne 57 : on récupère le singleton [ISession] auprès de la fabrique ;
- lignes 58-62 : on y récupère les cinq informations que la vue [Vue_01] y a placées ;
- lignes 64-68 : elles sont ensuite affichées sur la page ;
- ligne 71 : la méthode exécutée sur un clic sur le bouton [Retour] ;
- ligne 73 : on affiche la vue [Vue_01].

6.8 La fabrique

La fabrique instancie une nouvelle vue ainsi que la session :

```

1. package istia.st.avat.examples.activity;
2.
3. ...
4.
5. public class Factory implements IFactory {
6.
7.     // constantes
8.     public static final int CONFIG = -1;

```

```

9.  public static final int ACTION_01 = 3;
10. public static final int VUE_01 = 4;
11. public static final int VUE_02 = 5;
12. public static final int SESSION = 10;
13. // ----- DEBUT SINGLETONS
14. // la configuration
15. private Config config;
16. private boolean verbose;
17.
18. // les vues
19. private Vue_01 vue_01;
20. private Vue_02 vue_02;
21.
22. // l'activité principale
23. private Activity activity;
24.
25. // la session
26. private Session session;
27.
28. // ----- FIN SINGLETONS
29.
30. // constructeur
31. public Factory(Activity activity, Config config) {
32.     // l'activité
33.     this.activity = activity;
34.     // la configuration
35.     this.config = config;
36.     // le mode verbeux ou non
37.     verbose = config.isVerbose();
38.     // la session
39.     session = new Session();
40. }
41.
42. @Override
43. public Object getObject(int id, Object... params) {
44.     switch (id) {
45.         case CONFIG:
46.             return config;
47.         case SESSION:
48.             return session;
49.         case ACTION_01:
50.             return getAction_01(params);
51.         case VUE_01:
52.             return getVue_01(params);
53.         case VUE_02:
54.             return getVue_02(params);
55.     }
56.     return null;
57. }
58.
59. // initialisation d'une vue
60. private void init(Vue vue, Object... params) {
61.     // sa factory
62.     vue.setFactory(this);
63.     // son équipe
64.     ITeam team = new Team();
65.     team.setMonitor(vue);
66.     vue.setTeam(team);
67.     // son activité
68.     vue.setActivity(activity);
69.     // son mode verbeux ou non
70.     vue.setVerbose(verbose);
71.     // son id

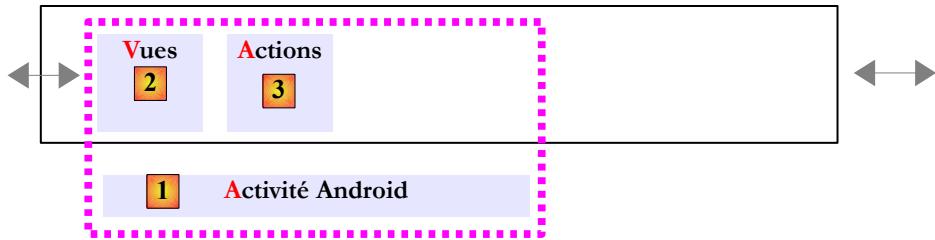
```

```
72.     vue.setBossId((String) params[1]);
73. }
74.
75. private Object getVue_02(Object[] params) {
76.     // la vue est un singleton
77.     if (vue_02 == null) {
78.         // la vue
79.         vue_02 = new Vue_02();
80.         // initialisation
81.         init(vue_02, params);
82.     }
83.     // on rend la référence
84.     return vue_02;
85. }
86.
87. private Object getVue_01(Object... params) {
88.     // la vue est un singleton
89.     if (vue_01 == null) {
90.         // la vue
91.         vue_01 = new Vue_01();
92.         // initialisation
93.         init(vue_01, params);
94.     }
95.     // on rend la référence
96.     return vue_01;
97. }
98. ....
99. }
```

7 AVAT- Exemple 5

7.1 Le projet

L'application de ce nouvel exemple a elle également l'architecture suivante :

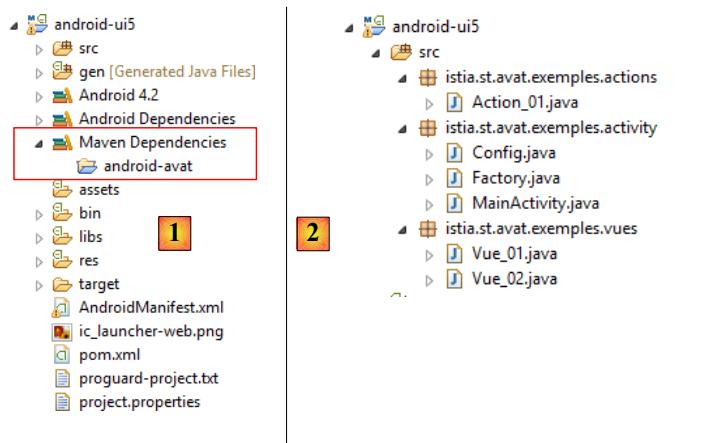


La couche [AVAT] est seule et sans tâches asynchrones.

L'application a les mêmes deux vues que précédemment et fait la même chose. La différence réside dans la façon dont les informations sont passées d'une vue à l'autre.

7.2 Le projet Eclipse

Le projet Eclipse de cet exemple est le suivant :



- en [1], le projet [android-ui5] est un projet Maven ayant une dépendance sur le projet [android-avat] ;
- en [2], les sources du projet :
 - le package [istia.st.avat-exemples.vues] rassemble les vues du projet, ici deux vues ;
 - le package [istia.st.avat-exemples.actions] rassemble les actions du projet, ici une action ;
 - le package [istia.st.avat-exemples.activity] contient l'activité Android [MainActivity], la fabrique d'objets [Factory], une classe de configuration [Config].

7.3 La vue [Vue_01]

La vue [Vue_01] est la suivante :

```
1. protected void doValider() {
2.     ...
```

```

3.      // on récupère une référence sur [Vue_02]
4.      vue_02 = (Vue_02) factory.getObject(Factory.VUE_02);
5.      // on exécute Action_01 de façon asynchrone
6.      // le boss sera la vue 2
7.      IAction action = (IAction) factory.getObject(Factory.ACTION_01, vue_02,
   "Action_01");
8.      // on signale à la vue [Vue_02] le démarrage de l'action
9.      vue_02.notifyEvent(action, WORK_STARTED, null);
10.     // on exécute l'action (urlServiceRest,nbAleas,a,b, sleepTime)
11.     action.doWork(edtUrlServiceRest.getText().toString(),
   Integer.parseInt(edtNbAleas.getText().toString()),
12.           Integer.parseInt(edtA.getText().toString()),
   Integer.parseInt(edtB.getText().toString()),
13.           Integer.parseInt(edtSleepTime.getText().toString()));
14.     // vue_02 va monitorer la fin des tâches
15.     vue_02.beginMonitoring();
16.     // on affiche la vue 2
17.     ((MainActivity) activity).showVue(2);
18. }
19.
20. @Override
21. public void notifyEndOfTasks() {
22. }
```

- ligne 4 : on récupère une référence sur la vue [Vue_02] auprès de la fabrique (c'est un singleton) ;
- ligne 7 : on crée une action [Action_01] dont le boss est [Vue_02] et non plus [Vue_01] ;
- ligne 9 : c'est désormais [Vue_02] le boss. On lui indique qu'une action a démarré ;
- lignes 11-13 : l'action [Action_01] est exécutée ;
- ligne 15 : on demande à [Vue_02] de monitorer la fin des tâches
- ligne 12 : on fait afficher [Vue_02]. C'est cette vue qui va récupérer les notifications envoyées par l'action.

7.4 La vue [Vue_02]

La vue [Vue_02] est la suivante :

```

1. @Override
2. public void onResume(){
3.     // parent
4.     super.onResume();
5.     // on affiche les résultats mémorisés
6.     edtUrlServiceRest.setText(urlServiceRest);
7.     edtNbAleas.setText(String.valueOf(nbAleas));
8.     edtA.setText(String.valueOf(a));
9.     edtB.setText(String.valueOf(b));
10.    edtSleepTime.setText(String.valueOf(sleepTime));
11. }
12.
13. @Override
14. public void notifyEndOfTasks() {
15.     Toast.makeText(activity, "Action terminée", Toast.LENGTH_SHORT).show();
16. }
17.
18. @Override
19. public void notifyEvent(IWorker worker, int eventType, Object event) {
20.     // parent
21.     super.notifyEvent(worker, eventType, event);
22.     // on gère l'évt WORK_TERMINATED
23.     if (eventType == IBoss.WORK_TERMINATED) {
24.         // on mémorise les éléments reçus
25.         Object[] results = (Object[]) event;
26.         urlServiceRest = (String) results[0];
27.         nbAleas = (Integer) results[1];
```

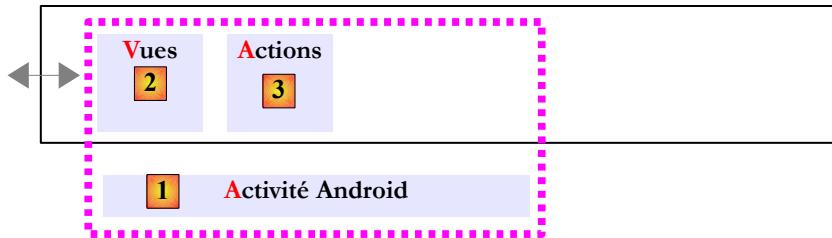
```
28.         a = (Integer) results[2];
29.         b = (Integer) results[3];
30.         sleepTime = (Integer) results[4];
31.     }
32. }
```

- ligne 19 : la vue gère les notifications envoyées par l'action ;
- ligne 21 : la notification est d'abord passée à la classe parent ;
- ligne 23 : on gère la notification [WORK_TERMINATED] ;
- lignes 24-20 : on mémorise les informations transportées par la notification ;
- ligne 14 : la méthode exécutée lorsque l'action [Action_01] est terminée ;
- ligne 15 : on affiche un message à l'écran. On ne peut pas encore faire ce qui est fait dans [onResume] (lignes 2-11). En effet, si on le fait, on récupère des pointeurs *null* pour les éléments de l'interface visuelle. Cela montre que la méthode [onActivityCreated] n'a pas encore été exécutée lorsqu'arrive la notification [endOfTasks]. C'est normal. La vue [Vue_02] ne sera affichée que lorsque la méthode *[Vue_01].doValider()* sera terminée. Celle-ci est totalement synchrone car il n'y a pas de tâche asynchrone ici. Donc la notification [endOfTasks] arrive avant que [Vue_02] ne soit affichée.

8 AVAT- Exemple 6

8.1 Le projet

L'application de ce nouvel exemple a elle également l'architecture suivante :



La couche [AVAT] est seule et sans tâches asynchrones.

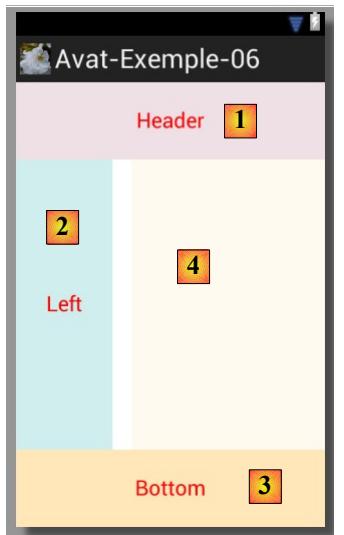
L'application est strictement identique à la précédente si ce n'est qu'on change l'apparence des vues :



Chacune des deux vues est structurée de la même façon :

- en [1], un entête ;
- en [2], une colonne de gauche qui pourrait contenir des liens ;
- en [3], un bas de page ;
- en [4], un contenu.

Ceci est obtenu en modifiant la vue de base [main.xml] de l'activité ;



Le code XML de la vue est le suivant :

```
1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.   xmlns:tools="http://schemas.android.com/tools"
3.   android:layout_width="match_parent"
4.   android:layout_height="match_parent"
5.   android:gravity="center"
6.   android:orientation="vertical" >
7.
8.   <LinearLayout
9.     android:id="@+id/header"
10.    android:layout_width="match_parent"
11.    android:layout_height="100dp"
12.    android:layout_weight="0.1"
13.    android:background="@color/LavenderBlushh2" >
14.
15.     <TextView
16.       android:id="@+id/textViewHeader"
17.       android:layout_width="match_parent"
18.       android:layout_height="wrap_content"
19.       android:layout_gravity="center"
20.       android:gravity="center_horizontal"
21.       android:text="@string/txt_header"
22.       android:textAppearance="?android:attr/textAppearanceLarge"
23.       android:textColor="@color/red" />
24.
25.   </LinearLayout>
26.
27.   <LinearLayout
28.     android:layout_width="match_parent"
29.     android:layout_height="fill_parent"
30.     android:layout_weight="0.8"
31.     android:orientation="horizontal" >
32.
33.     <LinearLayout
34.       android:id="@+id/left"
35.       android:layout_width="100dp"
36.       android:layout_height="match_parent"
37.       android:background="@color/Lightcyan2" >
38.
39.       <TextView
```

```

40.        android:id="@+id/txt_left"
41.        android:layout_width="fill_parent"
42.        android:layout_height="fill_parent"
43.        android:gravity="center_vertical/center_horizontal"
44.        android:text="@string/txt_left"
45.        android:textAppearance="?android:attr/textAppearanceLarge"
46.        android:textColor="@color/red" />
47.
48.    </LinearLayout>
49.
50.    <FrameLayout
51.        android:id="@+id/container"
52.        android:layout_width="match_parent"
53.        android:layout_height="match_parent"
54.        android:layout_marginLeft="20dp"
55.        android:background="@color/floral_white"
56.        tools:context=".MainActivity"
57.        tools:ignore="MergeRootFrame" >
58.    </FrameLayout>
59.  </LinearLayout>
60.
61.  <LinearLayout
62.      android:id="@+id/bottom"
63.      android:layout_width="match_parent"
64.      android:layout_height="100dp"
65.      android:layout_weight="0.1"
66.      android:background="@color/wheat1" >
67.
68.    <TextView
69.        android:id="@+id/textViewBottom"
70.        android:layout_width="fill_parent"
71.        android:layout_height="fill_parent"
72.        android:gravity="center_vertical/center_horizontal"
73.        android:text="@string/txt_bottom"
74.        android:textAppearance="?android:attr/textAppearanceLarge"
75.        android:textColor="@color/red" />
76.
77.  </LinearLayout>
78.
79. </LinearLayout>

```

- l'en-tête [1] est obtenu avec les lignes 8-25 ;
- la bande gauche [2] est obtenue avec les lignes 33-48 ;
- le bas de page [3] est obtenu avec les lignes 61-77 ;
- le contenu [4] est obtenu avec les lignes 50-58. On notera le nom [*container*] de ce conteneur.

Maintenant rappelons le code dans l'activité [MainActivity] qui affiche une vue :

```

1.  // affichage d'une vue
2.  public void showVue(int i) {
3.      // i=1 --> vue_01, i=2 --> vue_02
4.      // affichage vue
5.      FragmentTransaction fragmentTransaction = getFragmentManager().beginTransaction();
6.      fragmentTransaction.replace(R.id.container, vues[i - 1]);
7.      fragmentTransaction.commit();
8.  }

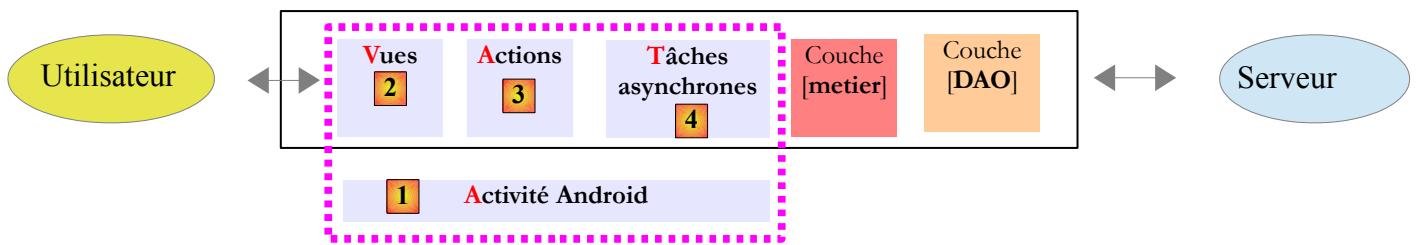
```

- ligne 6 : le conteneur d'id [*container*] va être remplacé par la vue. Le reste (en-tête, bande gauche, bas de page) est conservé.

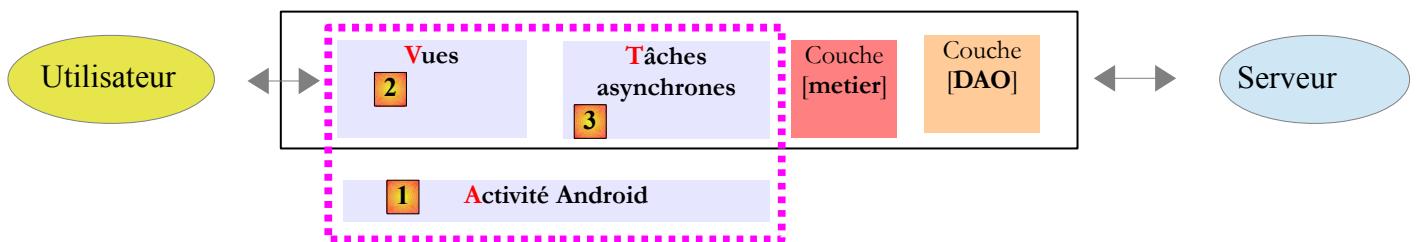
On a là une méthode analogue à celle des templates des facelets de JSF2 (Java Server Faces).

9 AVAT- Exemple 7

Revenons à l'architecture la plus générale des clients Android présentés jusqu'à maintenant, celle de l'exemple 2 :



La couche [Actions] est facultative **dans tous les cas**. On peut déporter la logique qu'on y trouve dans la vue. La vue [Vue] et l'action [Action] sont en effet tous les deux des boss [IBoss]. A ce titre, la vue peut employer des travailleurs [IWorker] quelconques dont les tâches asynchrones [Task]. Nous allons étudier un client Android plus complexe que les précédents avec cette architecture simplifiée :



Nous appellerons cette architecture AVT (Activité-Vues-Tâches).

9.1 Le projet

Dans le document "Introduction aux frameworks JSF2, Primefaces et Primefaces mobile" [<http://tahe.developpez.com/java/primefaces/>], on étudie une application de prise de rendez-vous pour des médecins. On y développe trois types d'interfaces :

- une interface web JSF2 ;
- une interface web Primefaces ;
- une interface web mobile pour smartphones avec Primefaces Mobile ;

Par la suite nous référencerons ce document [pfm] (Primefaces Mobile). Le projet sera ici de créer, sur tablette Android, une interface analogue aux précédentes.

L'interface JSF2 était la suivante :

[La page d'accueil](#)

Les Médecins Associés

[Français](#) [Anglais](#)

Réservations

Médecin	Jour (jj/mm/aaaa)
Mme Marie PELISSIER	23/05/2012

[Agenda](#)

ISTIA, université d'Angers

A partir de cette première page, l'utilisateur (Secrétariat, Médecin) va engager un certain nombre d'actions. Nous les présentons ci-dessous. La vue de gauche présente la vue à partir de laquelle l'utilisateur fait une **demande**, la vue de droite la **réponse** envoyée par le serveur.

Les Médecins Associés

[Français](#) [Anglais](#)

Réservations

Médecin	Jour (jj/mm/aaaa)
Mme Marie PELISSIER	23/05/2012

[Agenda](#)

ISTIA, université d'Angers

Les Médecins Associés

[Français](#) [Anglais](#)

Agenda de Mme Marie PELISSIER

Accueil		
Créneau horaire	Client	
8:0 - 8:20		Réserver
8:20 - 8:40		Réserver
8:40 - 9:0		Réserver
9:0 - 9:20		Réserver
9:20 - 9:40		Réserver

L'utilisateur a sélectionné un médecin et a saisi un jour de RV

On obtient la liste (vue partielle ici) des RV du médecin sélectionné pour le jour indiqué.

Agenda de Mme Marie PELISSIER

Créneau horaire	Client	
8:0 - 8:20		Réserver
8:20 - 8:40		Réserver
8:40 - 9:0		Réserver

On réserve

Les Médecins Associés

Français Anglais

Prise de rendez-vous de Mme Marie

Client

Mr Jules MARTIN

[Valider](#) [Annuler](#)

ISTIA, université d'Angers

On obtient une page à renseigner

Prise de rendez-vous de Mme Marie

Client

Melle Brigitte BISTROU

[Valider](#) [Annuler](#)

On la renseigne

Agenda de Mme Marie PELISSIER le 23 mai 2012

Créneau horaire	Client	
8:0 - 8:20	Melle Brigitte BISTROU	Supprimer
8:20 - 8:40		Réserver

Le nouveau RV apparaît dans la liste

Agenda de Mme Marie PELISSIER

Créneau horaire	Client	
8:0 - 8:20	Melle Brigitte BISTROU	Supprimer
8:20 - 8:40		Réserver

L'utilisateur peut supprimer un RV

Agenda de Mme Marie PELISSIER

Créneau horaire	Client	
8:0 - 8:20		Réserver
8:20 - 8:40		Réserver

Le RV supprimé a disparu de la liste des RV

Agenda de Mme Marie PI

Accueil	Créneau horaire	Client
8:00 - 8:20		Réservé
8:20 - 8:40		Réserver

Une fois l'opération de prise de RV ou d'annulation de RV faite, l'utilisateur peut revenir à la page d'accueil

Réervations

Médecin

Mme Marie PELISSIER

Jour (jj/mm/aaaa)

23/05/2012

[Agenda](#)

Il la retrouve dans son dernier état

Les Médecins Associés

[Français](#) [Anglais](#)

Réervations

Médecin

Mme Marie PELISSIER

[Agenda](#)

ISTIA, université d'Angers

Associated Doctors

[French](#) [English](#)

Reservations

Doctor

Mme Marie PELISSIER

[Agenda](#)

Date (dd/mm/yyyy)

23/05/2012

ISTIA, Angers university

On peut changer la langue

La langue a été changée

Enfin, on peut également obtenir une page d'erreurs :

Associated Doctors

[French](#) [English](#)

An error occurred

[Welcome Page](#)

Cause

javax.ejb.EJBException

Error message

Exception [EclipseLink-4002] (Eclipse Persistence Services - 2.3.2.v20111125-r10461):
org.eclipse.persistence.exceptions.DatabaseException Internal Exception:
com.mysql.jdbc.exceptions.jdbc4.MySQLNonTransientConnectionException: No operations allowed after connection

L'application Android va présenter des écrans analogues. Nous ne gérerons pas l'internationalisation de l'application mais nous faciliterons celle-ci en déportant les textes affichés dans des fichiers.

Il y aura cinq écrans :

Ecran de configuration

The screenshot shows a software interface for managing appointments. At the top, there's a logo of a white flower and the text "Avat-Les Médecins associés". Below this, a pink header bar contains the text "Les Médecins associés". The main area has a light blue sidebar on the left with the following menu items: "Configuration" (which is bolded), "Accueil", "Agenda", and "Ajout". The main content area is titled "Gestion des rendez-vous". It contains a form field for "URL du service REST" with the value "172.16.1.84:8080/server-rdvmedecins-rest" and a "Valider" button.

Ecran de choix du médecin du rendez-vous

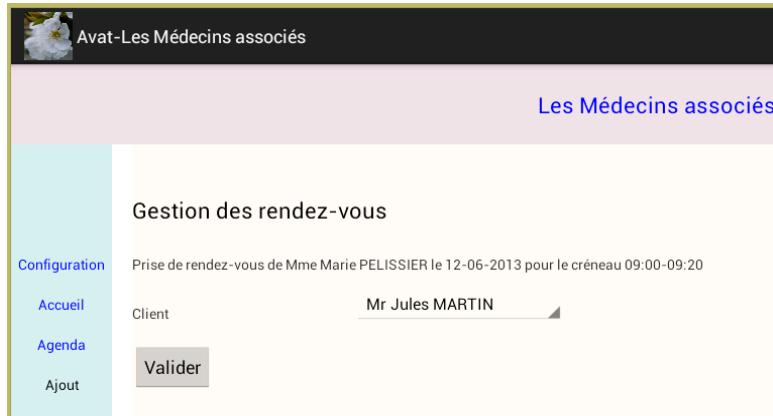
This screenshot shows the same software interface as the previous one, but with different input fields. The "Médecin" field is populated with "Mme Marie PELISSIER" and the "Jour du rendez-vous" field is populated with "12-06-2013". The "Valider" button is present at the bottom.

Ecran de choix du créneau horaire du rendez-vous

This screenshot shows the software interface again, but now displaying a list of time slots for the appointment on June 12, 2013. The list is as follows:

Slot	Action
08:00-08:20	Ajouter
08:20-08:40	Ajouter
08:40-09:00	Ajouter
09:00-09:20	Ajouter
09:20-09:40	Ajouter
09:40-10:00	Ajouter
10:00-10:20	Ajouter

Ecran de choix du client du rendez-vous



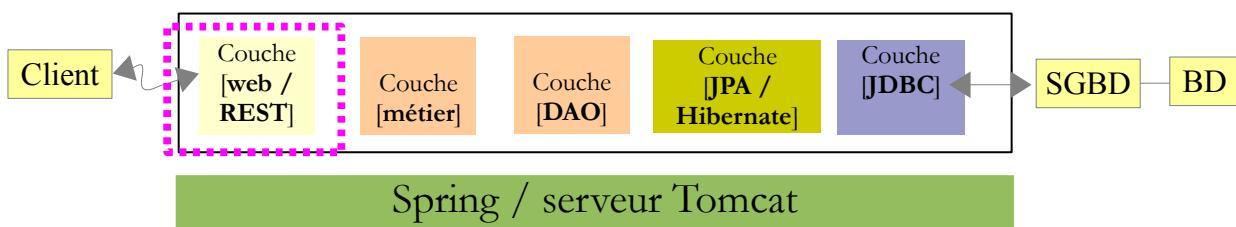
Après la prise de rendez-vous :



9.2 L'architecture du projet

On aura une architecture client / serveur analogue à celle de l'exemple 2 de ce document :

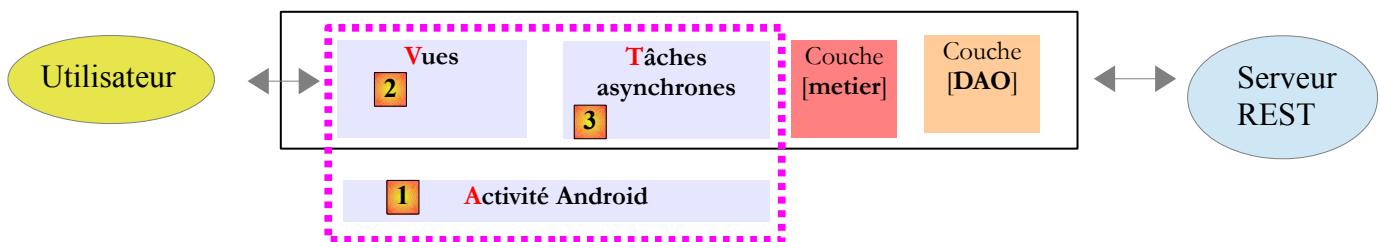
Le serveur



- nous allons réutiliser le travail fait dans [pfm] : les couches [métier] et [DAO], les entités JPA, la base de données MySQL ;
- notre travail consistera à exposer au monde web, via un service REST, l'interface de la couche [métier].

Le client Android

L'architecture de celui-ci a déjà été présentée :

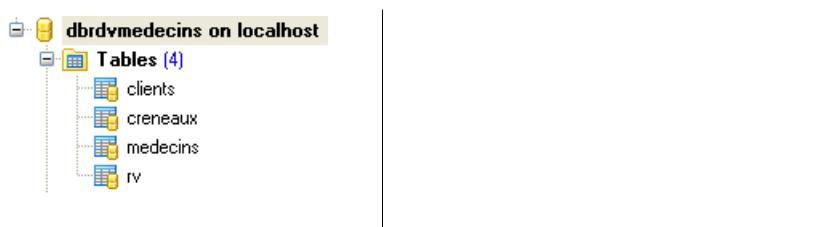


Pour développer ce client, nous nous appuierons sur ce qui a été fait dans l'exemple 2 de ce document.

9.3 La base de données

Elle ne joue pas un rôle fondamental dans ce document. Nous la donnons à titre d'information.

On l'appellera [dbrdvmmedecins2]. C'est une base de données MySQL5 avec quatre tables :



9.3.1 La table [MEDECINS]

Elle contient des informations sur les médecins gérés par l'application [RdvMedecins].

Fields					Indices	Foreign Keys	Data	Description	DDL
Field Name	Field Type	Size	Precision	Not Null					
ID	BIGINT	20	0	✓					
VERSION	INTEGER	11	0	✓					
TITRE	VARCHAR	5	0	✓					
NOM	VARCHAR	30	0	✓					
PRENOM	VARCHAR	30	0	✓					

ID	VERSION	TITRE	NOM	PRENOM
1	1	Mme	PELISSIER	Marie
2	1	Mr	BROMARD	Jacques
3	1	Mr	JANDOT	Philippe
4	1	Melle	JACQUEMOT	Justine

- ID : n° identifiant le médecin - clé primaire de la table
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- NOM : le nom du médecin
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

9.3.2 La table [CLIENTS]

Les clients des différents médecins sont enregistrés dans la table [CLIENTS] :

Fields	Indices	Foreign Keys	Data	Description	DDL
Field Name	Field Type	Size	Precision	Not Null	
ID	BIGINT	20	0	<input checked="" type="checkbox"/>	
VERSION	INTEGER	11	0	<input checked="" type="checkbox"/>	
TITRE	VARCHAR	5	0	<input checked="" type="checkbox"/>	
NOM	VARCHAR	30	0	<input checked="" type="checkbox"/>	
PRENOM	VARCHAR	30	0	<input checked="" type="checkbox"/>	

ID	VERSION	TITRE	NOM	PRENOM
1	1	Mr	MARTIN	Jules
2	1	Mme	GERMAN	Christine
3	1	Mr	JACQUARD	Jules
4	1	Melle	BISTRON	Brigitte

- ID : n° identifiant le client - clé primaire de la table
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- NOM : le nom du client
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

9.3.3 La table [CRENEAUX]

Elle liste les créneaux horaires où les RV sont possibles :

Fields	Indices	Foreign Keys	Data	Description	DDL
Field Name	Field Type	Size	Precision	Not Null	Default
ID	BIGINT	20	0	<input checked="" type="checkbox"/>	Null
VERSION	INTEGER	11	0	<input checked="" type="checkbox"/>	
HDEBUT	INTEGER	11	0	<input checked="" type="checkbox"/>	
MDEBUT	INTEGER	11	0	<input checked="" type="checkbox"/>	
HFIN	INTEGER	11	0	<input checked="" type="checkbox"/>	
MFIN	INTEGER	11	0	<input checked="" type="checkbox"/>	
ID_MEDECIN	BIGINT	20	0	<input checked="" type="checkbox"/>	

ID	VERSION	ID_MEDECIN	HDEBUT	MDEBUT	HFIN	MFIN
1	1	1	8	0	8	20
2	1	1	8	20	8	40
3	1	1	8	40	9	0
4	1	1	9	0	9	20
5	1	1	9	20	9	40
6	1	1	9	40	10	0
7	1	1	10	0	10	20
8	1	1	10	20	10	40
9	1	1	10	40	11	0
10	1	1	11	0	11	20
11	1	1	11	20	11	40
12	1	1	11	40	12	0
13	1	1	14	0	14	20
14	1	1	14	20	14	40
15	1	14	40	15	0	
1						
16	1	1	15	0	15	20
17	1	1	15	20	15	40
18	1	1	15	40	16	0
19	1	1	16	0	16	20
20	1	1	16	20	16	40
21	1	1	16	40	17	0
22	1	1	17	0	17	20
23	1	1	17	20	17	40
24	1	1	17	40	18	0
25	1	2	8	0	8	20
26	1	2	8	20	8	40
27	1	2	8	40	9	0
28	1	2	9	0	9	20
29	1	2	9	20	9	40
30	1	2	9	40	10	0
31	1	2	10	0	10	20

- ID : n° identifiant le créneau horaire - clé primaire de la table (ligne 8)
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- ID_MEDECIN : n° identifiant le médecin auquel appartient ce créneau – clé étrangère sur la colonne MEDECINS(ID).
- HDEBUT : heure début créneau
- MDEBUT : minutes début créneau
- HFIN : heure fin créneau
- MFIN : minutes fin créneau

La seconde ligne de la table [CRENEAUX] (cf [1] ci-dessus) indique, par exemple, que le créneau n° 2 commence à 8 h 20 et se termine à 8 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER).

9.3.4 La table [RV]

Elle liste les RV pris pour chaque médecin :

The screenshot shows two parts of a database interface. On the left, a table structure for the 'RV' table is displayed with columns: Field Name, Field Type, Size, Precision, Not Null, and Default. The columns are: ID (BIGINT, 20, 0, checked, Null), JOUR (DATE, 10, 0, checked), ID_CLIENT (BIGINT, 20, 0, checked), and ID_CRENEAU (BIGINT, 20, 0, checked). On the right, a table named 'CRENEAUX' is shown with columns: ID, JOUR, ID_CLIENT, and ID_CRENEAU. The data rows are: (1, 22/08/2006, 2, 1), (3, 23/08/2006, 4, 20), (4, 10/09/2006, 2, 10), (6, 23/08/2006, 3, 7), and (9, 23/08/2006, 2, 10). Row 3 is highlighted with a red border, and row 1 is highlighted with a yellow border.

- ID : n° identifiant le RV de façon unique – clé primaire
- JOUR : jour du RV
- ID_CRENEAU : créneau horaire du RV - clé étrangère sur le champ [ID] de la table [CRENEAUX] – fixe à la fois le créneau horaire et le médecin concerné.
- ID_CLIENT : n° du client pour qui est faite la réservation – clé étrangère sur le champ [ID] de la table [CLIENTS]

Cette table a une contrainte d'unicité sur les valeurs des colonnes jointes (JOUR, ID_CRENEAU) :

```
ALTER TABLE RV ADD CONSTRAINT UNQ1_RV UNIQUE (JOUR, ID_CRENEAU);
```

Si une ligne de la table[RV] a la valeur (JOUR1, ID_CRENEAU1) pour les colonnes (JOUR, ID_CRENEAU), cette valeur ne peut se retrouver nulle part ailleurs. Sinon, cela signifierait que deux RV ont été pris au même moment pour le même médecin. D'un point de vue programmation Java, le pilote JDBC de la base lance une *SQLException* lorsque ce cas se produit.

La ligne d'*id* égal à 3 (cf [1] ci-dessus) signifie qu'un RV a été pris pour le créneau n° 20 et le client n° 4 le 23/08/2006. La table [CRENEAUX] nous apprend que le créneau n° 20 correspond au créneau horaire 16 h 20 - 16 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER). La table [CLIENTS] nous apprend que le client n° 4 est Melle Brigitte BISTRÔU.

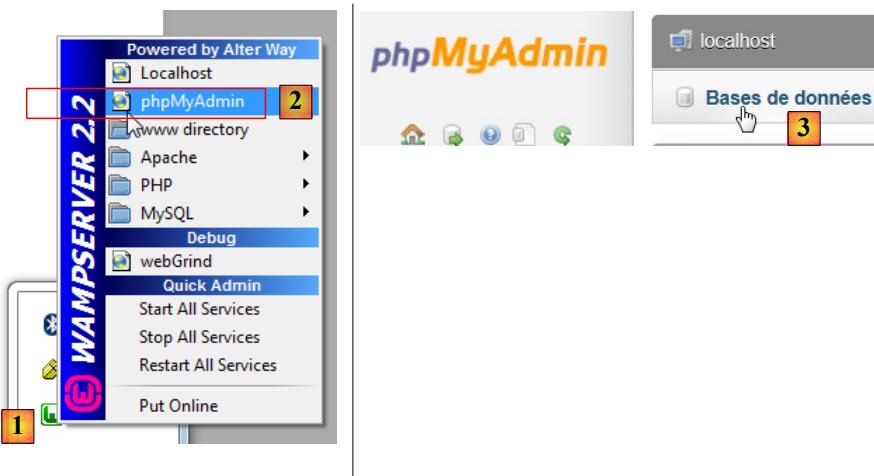
9.3.5 Génération de la base

Pour créer les tables et les remplir on pourra utiliser le script [dbrdvmmedecins2.sql] qu'on trouvera sur le site des exemples.

The screenshot shows the file system view in the Eclipse IDE. The path is: 1213 > eclipse > dvp > avat-02 > android-avat-exemple7. The 'Nom' column lists various Java packages and files, and the 'Modifié le' column shows their last modification date. A folder named 'database' is highlighted with a red border.

Nom	Modifié le
android-rdvmmedecins-dao	13/03/2013 17:44
android-rdvmmedecins-entities	13/03/2013 17:44
android-rdvmmedecins-metier	13/03/2013 17:44
android-rdvmmedecins-ui3	13/03/2013 17:44
database	20/03/2013 09:20
server-rdvmmedecins-dao	13/03/2013 17:44
server-rdvmmedecins-entities	13/03/2013 17:44
server-rdvmmedecins-jsf	13/03/2013 17:44
server-rdvmmedecins-metier	13/03/2013 17:44
server-rdvmmedecins-rest	13/03/2013 17:44

Avec [WampServer] (cf paragraphe 11.9, page 186), on pourra procéder comme suit :



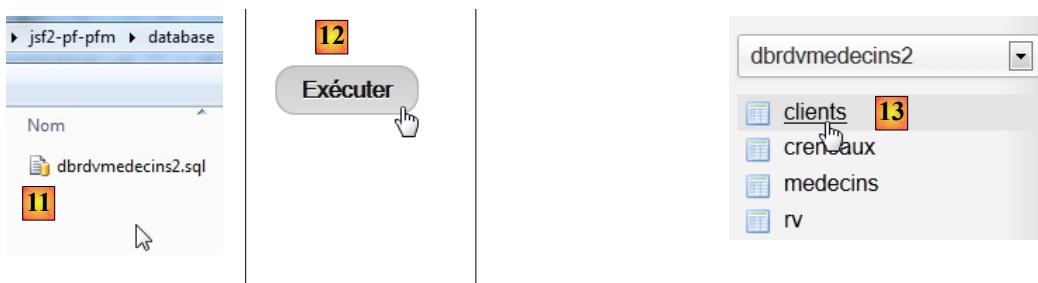
- en [1], on clique sur l'icône de [WampServer] et on choisit l'option [PhpMyAdmin] [2],
- en [3], dans la fenêtre qui s'est ouverte, on sélectionne le lien [Bases de données],



- en [2], on crée une base de données dont on a donné le nom [4] et l'encodage [5],
- en [7], la base a été créée. On clique sur son lien,



- en [8], on importe un fichier SQL,
- qu'on désigne dans le système de fichiers avec le bouton [9],



- en [11], on sélectionne le script SQL et en [12] on l'exécute,

- en [13], les quatre tables de la base ont été créées. On suit l'un des liens,

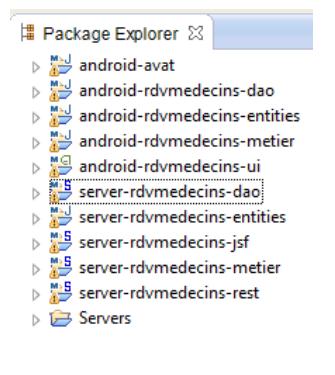
	ID	TITRE	NOM	VERSION	PRENOM
<input type="checkbox"/> Modifier <input type="checkbox"/> Éditer en place <input type="checkbox"/> Copier <input type="checkbox"/> Effacer	1	Mr	MARTIN	1	Jules
<input type="checkbox"/> Modifier <input type="checkbox"/> Éditer en place <input type="checkbox"/> Copier <input type="checkbox"/> Effacer	2	Mme	GERMAN	14	Christine
<input type="checkbox"/> Modifier <input type="checkbox"/> Éditer en place <input type="checkbox"/> Copier <input type="checkbox"/> Effacer	3	Mr	JACQUARD	1	Jules
<input type="checkbox"/> Modifier <input type="checkbox"/> Éditer en place <input type="checkbox"/> Copier <input type="checkbox"/> Effacer	4	Melle	BISTROU	1	Brigitte

- en [14], le contenu de la table.

Par la suite, nous ne reviendrons plus sur cette base. Mais le lecteur est invité à suivre son évolution au fil des tests surtout lorsque ça ne marche pas.

9.4 Les projets Eclipse de l'application

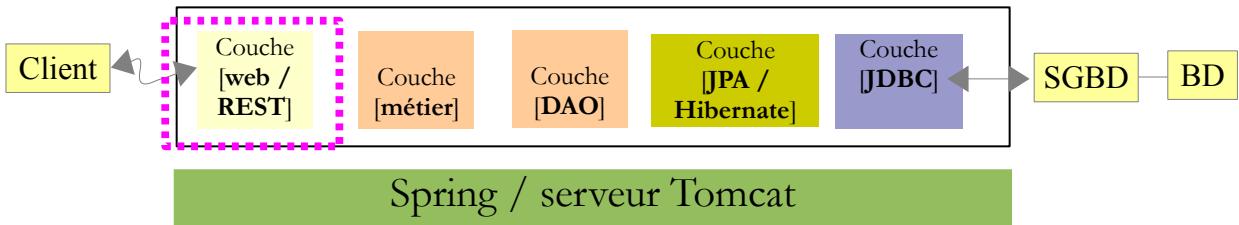
L'application regroupe plusieurs projets Eclipse, tous des projets Maven :



- [android-avat] : le modèle AVAT. Fait ;
- [server-rdvmmedecins-entities] : les entités JPA du serveur. Fait. Elles sont tirées du document [pfm] ;
- [server-rdvmmedecins-dao] : la couche [DAO] du serveur. Fait. C'est celle du document [pfm] ;
- [server-rdvmmedecins-metier] : la couche [métier] du serveur. Fait. C'est celle du document [pfm] ;
- [server-rdvmmedecins-jsf] : la couche [JSF2] du serveur. Fait. C'est celle du document [pfm]. Elle sert de point de repère pour notre application. On essaie de reproduire son fonctionnement ;
- [server-rdvmmedecins-rest] : la couche [REST] du serveur. A faire;
- [android-rdvmmedecins-entities] : les entités du client. Ce sont celles du serveur débarrassées de leurs annotations JPA ;
- [android-rdvmmedecins-dao] : la couche [DAO] du client Android. Ce sera celle de l'exemple 2 ;
- [android-rdvmmedecins-metier] : la couche [métier] du client Android. A faire ;
- [android-rdvmmedecins-ui] : la couche [présentation] du client Android. A faire ;

9.5 La couche [métier] du serveur

Revenons à l'architecture du serveur :



Nous devons écrire la couche [REST] qui expose au monde web la couche [métier]. Celle-ci présente l'interface [IMetier] suivante :

```

1. package server.rdvmedecins.metier.service;
2.
3. import java.util.Date;
4. import java.util.List;
5.
6. import server.rdvmedecins.entities.jpa.Client;
7. import server.rdvmedecins.entities.jpa.Creneau;
8. import server.rdvmedecins.entities.jpa.Medecin;
9. import server.rdvmedecins.entities.jpa.Rv;
10. import server.rdvmedecins.entities.metier.AgendaMedecinJour;
11.
12. public interface IMetier {
13.
14.     // couche dao
15.     // liste des clients
16.     public List<Client> getAllClients();
17.
18.     // liste des Médecins
19.     public List<Medecin> getAllMedecins();
20.
21.     // liste des créneaux horaires d'un médecin
22.     public List<Creneau> getAllCreneaux(Medecin medecin);
23.
24.     // liste des Rv d'un médecin, un jour donné
25.     public List<Rv> getRvMedecinJour(Medecin medecin, Date jour);
26.
27.     // trouver un client identifié par son id
28.     public Client getClientById(Long id);
29.
30.     // trouver un médecin identifié par son id
31.     public Medecin getMedecinById(Long id);
32.
33.     // trouver un Rv identifié par son id
34.     public Rv getRvById(Long id);
35.
36.     // trouver un créneau horaire identifié par son id
37.     public Creneau getCreneauById(Long id);
38.
39.     // ajouter un RV
40.     public Rv ajouterRv(Date jour, Creneau creneau, Client client);
41.
42.     // supprimer un RV
43.     public void supprimerRv(Rv rv);
44.
45.     // metier
46.     public AgendaMedecinJour getAgendaMedecinJour(Medecin medecin, Date jour);
47.
48. }

```

- les méthodes des lignes 16 à 43 déléguent leur exécution à la couche [DAO]. Ce sont des méthodes d'accès aux données du SGBD ;

- ligne 46 : la seule méthode appartenant véritablement à la couche [métier].

Nous allons exposer ces méthodes via un service REST en changeant parfois leur signature afin de faciliter les échanges avec les clients. On suivra la règle suivante : plutôt que d'échanger un objet JPA avec une clé primaire, on échangera simplement la clé primaire et on ira chercher l'objet en base.

Les objets paramètres ou résultat des méthodes de la couche [métier] sont les suivants :

La classe d'exception [RdvMedecinsException] : elle est lancée par les méthodes de la couche [métier]

```

1. public class RdvMedecinsException extends RuntimeException implements Serializable {
2.
3.     private static final long serialVersionUID = 1L;
4.     // champs privés
5.     private int code = 0;
6.
7.     // constructeurs
8.     public RdvMedecinsException() {
9.         super();
10.    }
11.
12.    public RdvMedecinsException(String message) {
13.        super(message);
14.    }
15.
16.    public RdvMedecinsException(String message, Throwable cause) {
17.        super(message, cause);
18.    }
19.
20.    public RdvMedecinsException(Throwable cause) {
21.        super(cause);
22.    }
23.
24.    public RdvMedecinsException(String message, int code) {
25.        super(message);
26.        setCode(code);
27.    }
28.
29.    public RdvMedecinsException(Throwable cause, int code) {
30.        super(cause);
31.        setCode(code);
32.    }
33.
34.    public RdvMedecinsException(String message, Throwable cause, int code) {
35.        super(message, cause);
36.        setCode(code);
37.    }
38.
39.    // getters - setters
40. ...
41. }
```

L'entité JPA [Personne] : représente un médecin ou un client. On ne reproduit pas les annotations JPA afin d'alléger le code.

```

1. public class Personne implements Serializable {
2.     private static final long serialVersionUID = 1L;
3.     // identifiant de la personne
4.     private Long id;
5.     // son titre Mr, Mme, Melle
6.     private String titre;
7.     // son nom
8.     private String nom;
```

```

9.     // la version de l'entité en base
10.    private int version;
11.    // le prénom de la personne
12.    private String prenom;
13. ....
14. @Override
15.    public String toString() {
16.        return String.format("[%s,%s,%s,%s,%s]", id, version, titre, prenom, nom);
17.    }
18.
19. }
```

La classe [Medecin] : représente un médecin

```

1. public class Medecin extends Personne implements Serializable {
2.     private static final long serialVersionUID = 1L;
3. ...
4.
5. }
```

La classe [Client] : représente un client

```

6. public class Client extends Personne implements Serializable {
7.     private static final long serialVersionUID = 1L;
8. ...
9.
10. }
```

La classe [Creneau] : représente un créneau horaire.

```

1. public class Creneau implements Serializable {
2.
3.     private static final long serialVersionUID = 1L;
4.     // identifiant
5.     private Long id;
6.     // minutes début créneau
7.     private int mdebut;
8.     // heure de fin de créneau
9.     private int hfin;
10.    // heure de début du créneau
11.    private int hdebut;
12.    // heure de fin du créneau
13.    private int mfin;
14.    // version de l'entité JPA en base
15.    private int version;
16.    // le médecin propriétaire du créneau
17.    private Medecin medecin;
18.
19. ...
20. @Override
21. public String toString() {
22.     return String.format("Creneau [%s, %s, %s:%s, %s:%s,%s]", id, version, hdebut, mdebut,
23.     hfin, mfin, medecin);
24. }
```

La classe [Rv] : représente un rendez-vous.

```

1. public class Rv implements Serializable {
2.
3.     private static final long serialVersionUID = 1L;
4.     // l'identifiant du rendez-vous
5.     private Long id;
```

```

6.    // la date du rdv
7.    private Date jour;
8.    // le créneau horaire du rdv
9.    private Creneau creneau;
10.   // le client du rdv
11.   private Client client;
12. ...
13.   @Override
14.   public String toString() {
15.       return String.format("Rv[%s, %s, %s]", id, creneau, client);
16.   }
17. }
```

La classe [AgendaMedecinJour] : l'agenda d'un médecin pour un jour donné.

```

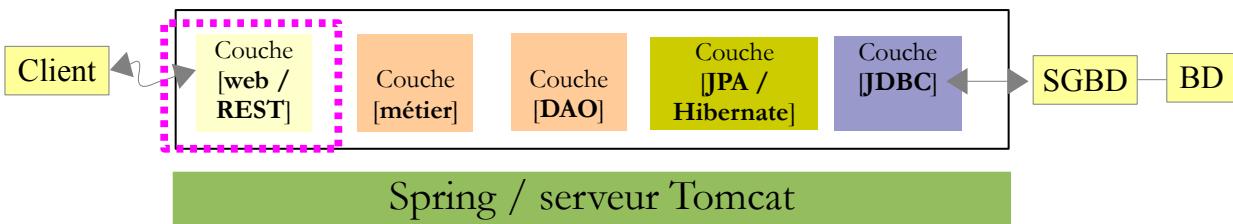
1.  public class AgendaMedecinJour implements Serializable {
2.
3.    private static final long serialVersionUID = 1L;
4.    // le médecin
5.    private Medecin medecin;
6.    // le jour dans l'agenda
7.    private Date jour;
8.    // les créneaux horaires pour ce jour
9.    private CreneauMedecinJour[] creneauxMedecinJour;
10. ...
11.   public String toString() {
12.       StringBuffer str = new StringBuffer("");
13.       for (CreneauMedecinJour cr : creneauxMedecinJour) {
14.           str.append(" ");
15.           str.append(cr.toString());
16.       }
17.       return String.format("Agenda[%s,%s,%s]", medecin, new
SimpleDateFormat("dd/MM/yyyy").format(jour), str.toString());
18.   }
19. }
```

La classe [CreneauMedecinJour] : représente l'occupation d'un créneau horaire.

```

1.  public class CreneauMedecinJour implements Serializable {
2.
3.    private static final long serialVersionUID = 1L;
4.    // le créneau horaire
5.    private Creneau creneau;
6.    // le rdv pour ce créneau ou null si pas de rdv
7.    private Rv rv;
8. ...
9.   @Override
10.  public String toString() {
11.      return String.format("[%s %s]", creneau, rv);
12.  }
13. }
```

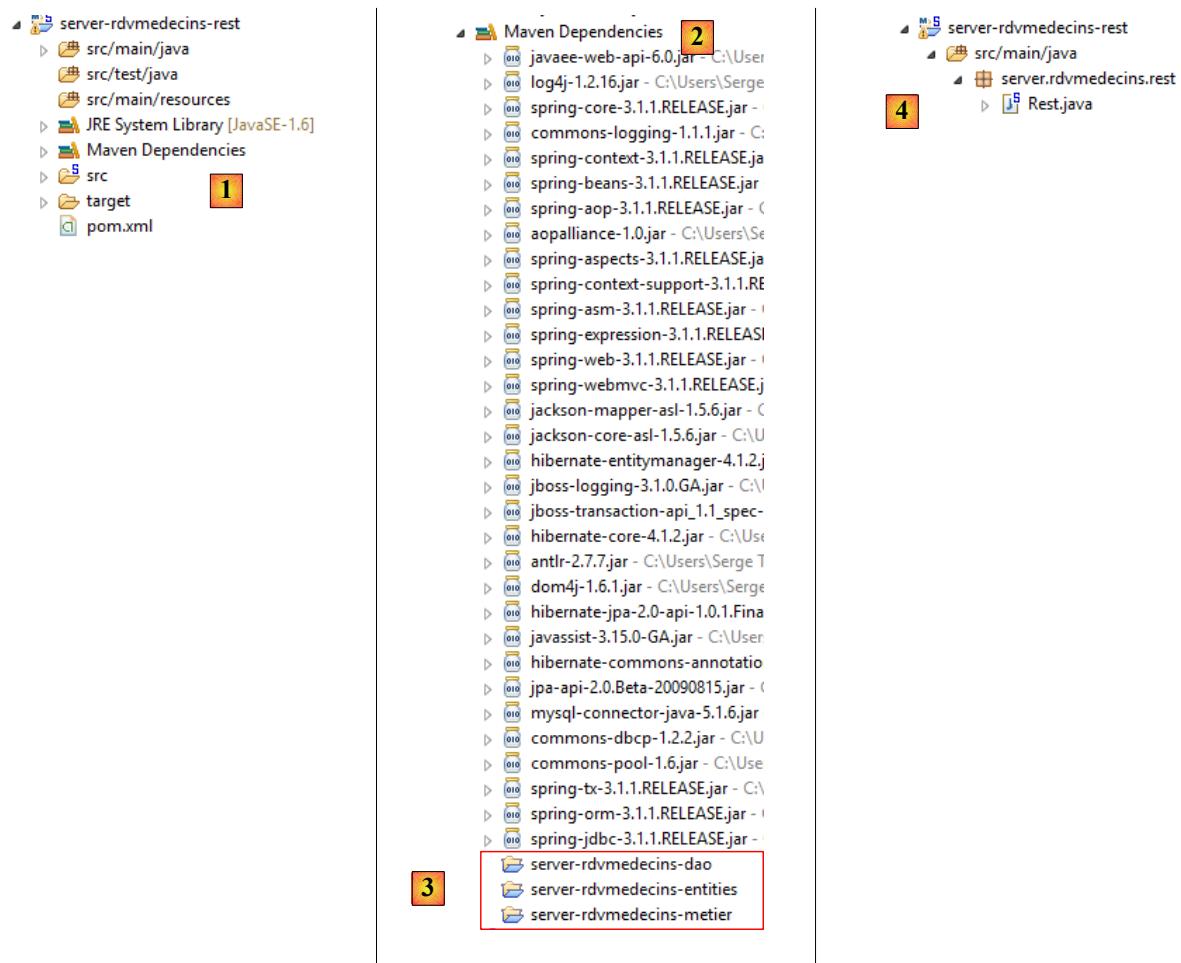
9.6 Le serveur REST



Le service REST est implémenté par SpringMVC. Un service REST (RepresEntational State Transfer) est un service HTTP répondant aux demandes GET, POST, PUT, DELETE d'un client HTTP. Sa définition formelle indique pour chacune de ces méthodes, les objets que le client doit envoyer et celui qu'il reçoit. Dans les exemples de ce document, nous n'utilisons que la méthode GET alors que dans certains cas, la définition formelle de REST indique qu'on devrait utiliser une méthode PUT ou DELETE. Nous appelons REST notre service parce qu'il est implémenté par un service de Spring qu'on a l'habitude d'appeler service REST et non parce qu'il respecte la définition formelle des services REST.

9.6.1 Le projet Eclipse

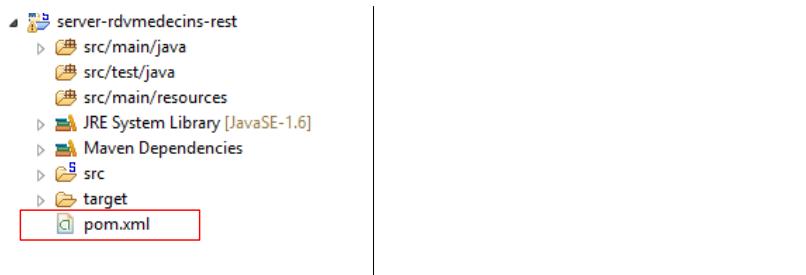
Le projet Eclipse du serveur REST est le suivant :



- en [1], le projet dans son ensemble. C'est un projet Maven de type web ;
- en [2], les dépendances Maven du projet. Elles sont très nombreuses. Le gros des archives est fourni par Hibernate qui assure la persistance des données en base et Spring qui assure l'intégration des couches ainsi que la gestion des transactions ;
- en [3], le projet présente des dépendances sur les couches [DAO] et [métier] du serveur ainsi que sur ses entités ;

- en [4], la couche REST est assurée par une classe.

9.6.2 Les dépendances Maven



Le fichier [pom.xml] est le suivant :

```

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.   4.0.0.xsd">
5.     <modelVersion>4.0.0</modelVersion>
6.     <groupId>server.rdvmedecins</groupId>
7.     <artifactId>server-rdvmdecins-rest</artifactId>
8.     <version>1.0-SNAPSHOT</version>
9.     <packaging>war</packaging>
10.    <name>server-rdvmdecins-rest</name>
11.
12.    <properties>
13.      <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
14.      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.      <releaseCandidate>1</releaseCandidate>
16.      <spring.version>3.1.1.RELEASE</spring.version>
17.      <jackson.mapper.version>1.5.6</jackson.mapper.version>
18.    </properties>
19.
20.    <repositories>
21.      <repository>
22.        <id>com.springsource.repository.bundles.release</id>
23.        <name>SpringSource Enterprise Bundle Repository - Release</name>
24.        <url>http://repository.springsource.com/maven/bundles/release</url>
25.      </repository>
26.    </repositories>
27.
28.    <dependencies>
29.      <dependency>
30.        <groupId>org.springframework</groupId>
31.        <artifactId>spring-core</artifactId>
32.        <version>${spring.version}</version>
33.      </dependency>
34.      <dependency>
35.        <groupId>org.springframework</groupId>
36.        <artifactId>spring-beans</artifactId>
37.        <version>${spring.version}</version>
38.      </dependency>
39.      <dependency>
40.        <groupId>org.springframework</groupId>
41.        <artifactId>spring-web</artifactId>
42.        <version>${spring.version}</version>

```

```

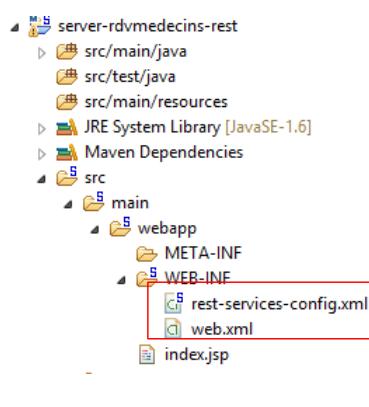
43.      </dependency>
44.      <dependency>
45.          <groupId>org.springframework</groupId>
46.          <artifactId>spring-webmvc</artifactId>
47.          <version>${spring.version}</version>
48.      </dependency>
49.      <dependency>
50.          <groupId>org.codehaus.jackson</groupId>
51.          <artifactId>jackson-mapper-asl</artifactId>
52.          <version>${jackson.mapper.version}</version>
53.      </dependency>
54.      <dependency>
55.          <groupId>javax.servlet</groupId>
56.          <artifactId> servlet-api</artifactId>
57.          <version>2.4</version>
58.      </dependency>
59.      <dependency>
60.          <groupId>server.rdvmedecins</groupId>
61.          <artifactId>server-rdvmedecins-metier</artifactId>
62.          <version>1.0-SNAPSHOT</version>
63.      </dependency>
64.  </dependencies>
65.
66.  <build>
67.      <plugins>
68.          ...
69.      </plugins>
70.  </build>
71.
72. </project>

```

- lignes 29-48 : les dépendances sur Spring ;
- lignes 49-53 : la dépendance sur la bibliothèque JSON Jackson ;
- lignes 54-58 : la dépendance sur l'API servlet ;
- lignes 59-63 : la dépendance sur la couche [métier] du serveur.

9.6.3 Configuration de la couche [REST]

Le service REST est configuré de façon analogue à celui de l'exemple 2.



9.6.3.1 Le fichier [web.xml]

Le fichier [web.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
```

```

2. <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
   app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/
   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
3.   <display-name>REST for Medecins</display-name>
4.   <servlet>
5.     <servlet-name>restservices</servlet-name>
6.     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
7.     <init-param>
8.       <param-name>contextConfigLocation</param-name>
9.       <param-value>/WEB-INF/rest-services-config.xml</param-value>
10.    </init-param>
11.    <load-on-startup>1</load-on-startup>
12.  </servlet>
13.  <servlet-mapping>
14.    <servlet-name>restservices</servlet-name>
15.    <url-pattern>/</url-pattern>
16.  </servlet-mapping>
17. </web-app>

```

Il est identique à celui de l'exemple 2 (page 60, paragraphe 4.3.2.3).

9.6.3.2 Le fichier [rest-services-config.xml]

Le fichier [rest-services-config.xml] est le suivant :

```

1. <beans xmlns="http://www.springframework.org/schema/beans"
2.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:oxm="http://www.springframework.org/schema/oxm"
3.   xmlns:util="http://www.springframework.org/schema/util"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
4.   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:tx="http://www.springframework.org/schema/tx"
5.   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
6.           http://www.springframework.org/schema/mvc
   http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
7.           http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd
8.           http://www.springframework.org/schema/oxm
9.           http://www.springframework.org/schema/oxm/spring-oxm-3.0.xsd
10.          http://www.springframework.org/schema/util
11.          http://www.springframework.org/schema/util/spring-util-3.0.xsd
12.          http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx-
13. 3.0.xsd">
14.
15.
16.  <!-- couches applicatives -->
17.  <bean id="dao" class="server.rdvmedecins.dao.DaoJpa" />
18.  <bean id="metier" class="server.rdvmedecins.metier.service.Metier">
19.    <property name="dao" ref="dao" />
20.  </bean>
21.
22.  <!-- EntityManagerFactory -->
23.  <bean id="entityManagerFactory"
24.    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
25.    <property name="dataSource" ref="dataSource" />
26.    <property name="jpaVendorAdapter">
27.      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
28.        <property name="databasePlatform"
   value="org.hibernate.dialect.MySQL5InnoDBDialect" />

```

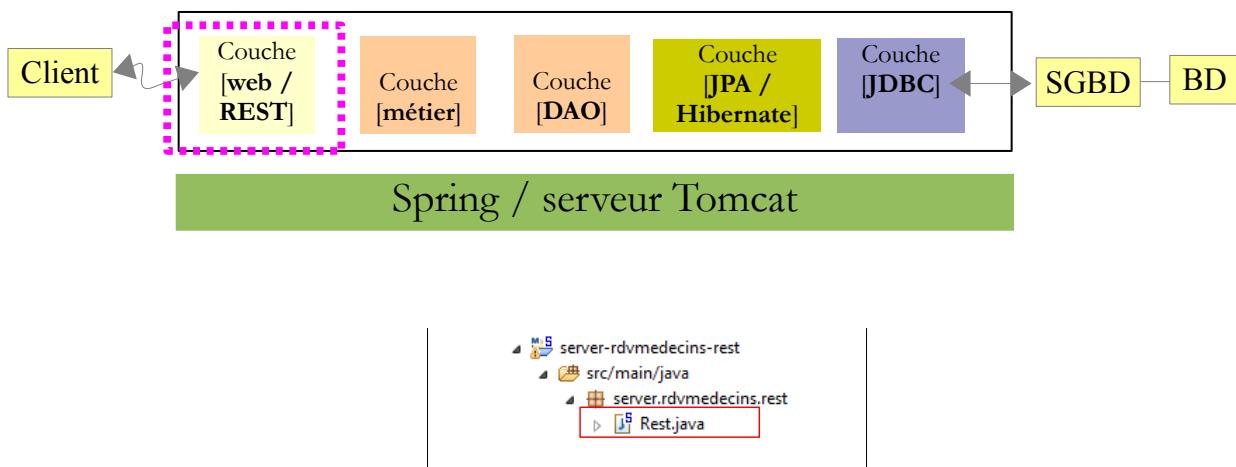
```

29.          <!-- <property name="showSql" value="true" /> <property name="generateDdl"
30.              value="true" /> -->
31.      </bean>
32.  </property>
33. </bean>
34.
35. <!-- la source de données DBCP -->
36. <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
37.     destroy-method="close">
38.     <property name="driverClassName" value="com.mysql.jdbc.Driver" />
39.     <property name="url" value="jdbc:mysql://localhost:3306/dbrdvmedecins2" />
40.     <property name="username" value="root" />
41.     <property name="password" value="" />
42. </bean>
43.
44. <!-- le gestionnaire de transactions -->
45. <tx:annotation-driven transaction-manager="txManager" />
46. <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
47.     <property name="entityManagerFactory" ref="entityManagerFactory" />
48. </bean>
49.
50. <!-- traduction des exceptions -->
51. <bean
52.     class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"
53. />
54. <!-- persistence -->
55. <bean
56.     class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
57.
58. <!--
59.          Spring MVC
60.          -->
61. <!-- Recherche des annotations SpringMVC dans les classes du package nommé -->
62. <context:component-scan base-package="server.rdvmedecins.rest" />
63.
64. <!-- L'unique View utilisée. Elle génère du JSON -->
65. <bean
66.     class="org.springframework.web.servlet.view.json.MappingJacksonJsonView">
67.     <property name="contentType" value="text/plain" />
68. </bean>
69.
70. <!-- Le convertisseur JSON -->
71. <bean id="jsonMessageConverter"
72.     class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter" />
73.
74. <!-- la liste des convertisseurs de messages. Il n'y en qu'un : le JSON
75.     ci-dessus -->
76. <bean
77.     class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
78.     <property name="messageConverters">
79.         <util:list id="beanList">
80.             <ref bean="jsonMessageConverter" />
81.         </util:list>
82.     </property>
83. </bean>
83. </beans>
```

- lignes 16-56 : configurent les couches [métier], [DAO], [JPA] du serveur. Cette configuration est expliquée dans le document [pfm] ;
- ligne 36-42 : définissent la source de données, la base MySQL. Pour ses propres tests, le lecteur sera peut être amené à changer ces informations ;

- lignes 58-82 : configurent le service REST. La configuration est analogue à ce qu'elle était dans l'exemple 2 (page 62, paragraphe 4.3.2.4).

9.6.4 Implémentation de la couche [REST]



La couche REST est implémentée par la classe [Rest] suivante :

```

1. package server.rdvmedecins.rest;
2.
3. ...
4.
5. @Controller
6. public class Rest {
7.
8.     // couche métier
9.     @Autowired
10.    private IMetier metier;
11.    @Autowired
12.    private View view;
13.
14.    // liste des clients
15.    @RequestMapping(value = "/getAllClients", method = RequestMethod.GET)
16.    public ModelAndView getAllClients() {
17.        ...
18.    }
19.
20.    // liste des médecins
21.    @RequestMapping(value = "/getAllMedecins", method = RequestMethod.GET)
22.    public ModelAndView getAllMedecins() {
23.        ...
24.    }
25.
26.    // liste des créneaux horaires d'un médecin
27.    @RequestMapping(value = "/getAllCreneaux/{idMedecin}", method = RequestMethod.GET)
28.    public ModelAndView getAllCreneaux(@PathVariable("idMedecin") long idMedecin) {
29.        ...
30.    }
31.
32.    // liste des Rv d'un médecin, un jour donné
33.    @RequestMapping(value = "/getRvMedecinJour/{idMedecin}/{jour}", method =
34.        RequestMethod.GET)
34.    public ModelAndView getRvMedecinJour(@PathVariable("idMedecin") long idMedecin,
            @PathVariable("jour") String jour) {

```

```

35. ...
36. }
37.
38. // trouver un client identifié par son id
39. @RequestMapping(value = "/getClientById/{idClient}", method = RequestMethod.GET)
40. public ModelAndView getClientById(@PathVariable("idClient") long idClient) {
41. ...
42. }
43.
44. // trouver un médecin identifié par son id
45. @RequestMapping(value = "/getMedecinById/{idMedecin}", method = RequestMethod.GET)
46. public ModelAndView getMedecinById(@PathVariable("idMedecin") long idMedecin) {
47. ...
48. }
49.
50. // trouver un Rv identifié par son id
51. @RequestMapping(value = "/getRvById/{idRv}", method = RequestMethod.GET)
52. public ModelAndView getRvById(@PathVariable("idRv") long idRv) {
53. ...
54. }
55.
56. // trouver un créneau horaire identifié par son id
57. @RequestMapping(value = "/getCreneauById/{idCreneau}", method = RequestMethod.GET)
58. public ModelAndView getCreneauById(@PathVariable("idCreneau") long idCreneau) {
59. ...
60. }
61.
62. // ajouter un RV
63. @RequestMapping(value = "/ajouterRv/{jour}/{idCreneau}/{idClient}", method =
   RequestMethod.GET)
64. public ModelAndView ajouterRv(@PathVariable("jour") String jour,
   @PathVariable("idCreneau") long idCreneau,
   @PathVariable("idClient") long idClient) {
65. ...
66. }
67. }
68.
69. // supprimer un RV
70. @RequestMapping(value = "/supprimerRv/{idRv}", method = RequestMethod.GET)
71. public ModelAndView supprimerRv(@PathVariable("idRv") long idRv) {
72. ...
73. }
74.
75. // l'agenda d'un medecin
76. @RequestMapping(value = "/getAgendaMedecinJour/{idMedecin}/{jour}", method =
   RequestMethod.GET)
77. public ModelAndView getAgendaMedecinJour(@PathVariable("idMedecin") long idMedecin,
   @PathVariable("jour") String jour) {
78. ...
79. }
80.
81. // crée une réponse d'erreur
82. private ModelAndView createResponseError(String numero, String message) {
83. ...
84. }
85.
86. }

```

La classe suit les principes utilisés dans l'exemple 2 (page 64, paragraphe 4.3.2.5).

- ligne 15 : l'URL permettant d'obtenir la liste des clients ;
- ligne 21 : l'URL permettant d'obtenir la liste des médecins ;
- ligne 27 : l'URL permettant d'obtenir la liste des créneaux horaires d'un médecin. Le paramètre est l'identifiant du médecin ;

- ligne 33 : l'URL permettant d'obtenir la liste des rendez-vous d'un médecin pour un jour donné. Les paramètres sont :
 - l'identifiant du médecin,
 - le jour ;
- ligne 39 : l'URL permettant d'obtenir un client identifié par son numéro ;
- ligne 46 : idem pour un médecin ;
- ligne 52 : idem pour un rendez-vous ;
- ligne 58 : idem pour un créneau horaire ;
- ligne 63 : l'URL pour ajouter un rendez-vous. Les paramètres passés dans l'URL sont :
 - le jour du rendez-vous,
 - le n° du créneau horaire du rendez-vous,
 - le n° du client pour qui le rendez-vous est pris ;
- ligne 70 : l'URL pour supprimer un rendez-vous identifié par son n° ;
- ligne 76 : l'URL pour obtenir l'agenda d'un médecin pour un jour donné. Les paramètres de l'URL sont :
 - le n° du médecin,
 - le jour ;
- ligne 82 : une méthode privée qui génère une réponse en cas d'exception.

Voyons maintenant le code de ces méthodes.

Note : pour obtenir les copies d'écran ci-dessous, le SGBD doit être lancé.

9.6.4.1 *Liste des clients*

Le code est le suivant :

```

1.  // liste des clients
2.  @RequestMapping(value = "/getAllClients", method = RequestMethod.GET)
3.  public ModelAndView getAllClients() {
4.
5.      // on utilise la couche métier
6.      try {
7.          List<Client> clients = metier.getAllClients();
8.          // réponse
9.          return new ModelAndView(view, "data", clients);
10.     } catch (Exception e) {
11.         // réponse
12.         return createResponseError("103", e.getMessage());
13.     }
14. }
```

- ligne 2 : l'URL n'a pas de paramètres ;
- ligne 7 : la liste des clients est demandée à la couche [métier] ;
- ligne 9 : la réponse renvoyée au client. Le paramétrage du serveur REST étant le même que celui de l'exemple 2, le client recevra une réponse JSON contenant la liste des clients (voir copie d'écran ci-dessous) ;
- ligne 12 : la réponse envoyée en cas d'exception.

La méthode [createResponseError] est la méthode chargée d'envoyer la réponse au client en cas d'exception. Elle a deux paramètres :

- le n° de l'erreur,
- le message de l'erreur ;

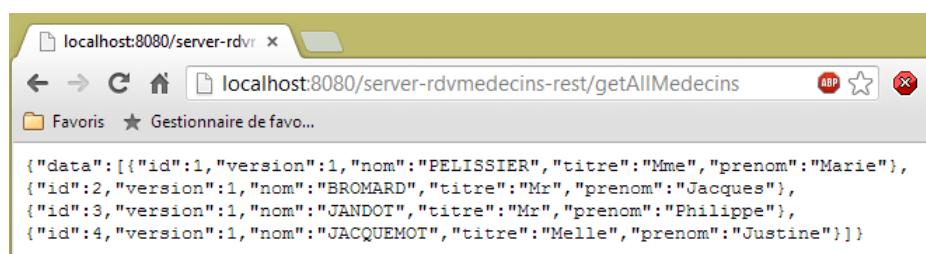
Son code est le suivant :

```

1.  // crée une réponse d'erreur
2.  private ModelAndView createResponseError(String numero, String message) {
3.      Map<String, Object> modèle = new HashMap<String, Object>();
4.      modèle.put("error", numero);
5.      modèle.put("message", message);
6.      return new ModelAndView(view, modèle);
7. }
```

Le client recevra une chaîne JSON de la forme :

```
{"error":"54","message":"..."}  
Voici un exemple d'exécution :
```

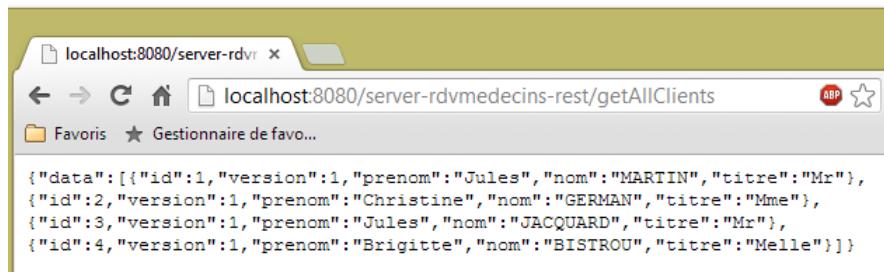


9.6.4.2 Liste des médecins

Le code est analogue :

```
1. // liste des médecins
2. @RequestMapping(value = "/getAllMedecins", method = RequestMethod.GET)
3. public ModelAndView getAllMedecins() {
4.
5.     // on utilise la couche métier
6.     try {
7.         List<Medecin> medecins = metier.getAllMedecins();
8.         // réponse
9.         return new ModelAndView(view, "data", medecins);
10.    } catch (Exception e) {
11.        // réponse
12.        return createResponseError("104", e.getMessage());
13.    }
14. }
```

Voici un exemple d'exécution :



9.6.4.3 Liste des créneaux d'un médecin

Le code est le suivant :

```
1. // liste des créneaux horaires d'un médecin
2. @RequestMapping(value = "/getAllCreneaux/{idMedecin}", method = RequestMethod.GET)
3. public ModelAndView getAllCreneaux(@PathVariable("idMedecin") long idMedecin) {
4.
5.     // on utilise la couche métier
6.     try {
7.         Medecin medecin = metier.getMedecinById(idMedecin);
8.         List<Creneau> creneaux = metier.getAllCreneaux(medecin);
```

```

9.      // réponse
10.     return new ModelAndView(view, "data", creneaux);
11. } catch (Exception e) {
12.     // réponse
13.     return createResponseError("105", e.getMessage());
14. }
15. }

```

Voici un exemple d'exécution (vue partielle) :



9.6.4.4 Liste des rendez-vous d'un médecin

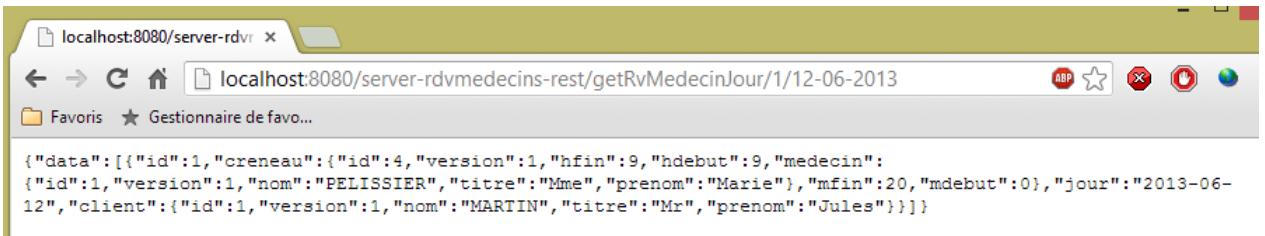
Le code est le suivant :

```

1.  // liste des Rv d'un médecin, un jour donné
2.  @RequestMapping(value = "/getRvMedecinJour/{idMedecin}/{jour}", method =
RequestMethod.GET)
3.  public ModelAndView getRvMedecinJour(@PathVariable("idMedecin") long idMedecin,
@PathVariable("jour") String jour) {
4.
5.      // on utilise la couche métier
6.      try {
7.          Date jourRv = new SimpleDateFormat("dd-MM-yy").parse(jour);
8.          Medecin medecin = metier.getMedecinById(idMedecin);
9.          List<Rv> rv = metier.getRvMedecinJour(medecin, jourRv);
10.         // réponse
11.         return new ModelAndView(view, "data", rv);
12.     } catch (Exception e) {
13.         // réponse
14.         return createResponseError("106", e.getMessage());
15.     }
16. }

```

Voici un exemple d'exécution :



9.6.4.5 Obtenir un client identifié par son n°

Le code est le suivant :

```
1. // trouver un client identifié par son id
2. @RequestMapping(value = "/getClientById/{idClient}", method = RequestMethod.GET)
3. public ModelAndView getClientById(@PathVariable("idClient") long idClient) {
4.
5.     // on utilise la couche métier
6.     try {
7.         // réponse
8.         return new ModelAndView(view, "data", metier.getClientById(idClient));
9.     } catch (Exception e) {
10.        // réponse
11.        return createResponseError("107", e.getMessage());
12.    }
13. }
```

Voici un exemple d'exécution :



9.6.4.6 Obtenir un médecin identifié par son n°

Le code est le suivant :

```
1. // trouver un médecin identifié par son id
2. @RequestMapping(value = "/getMedecinById/{idMedecin}", method = RequestMethod.GET)
3. public ModelAndView getMedecinById(@PathVariable("idMedecin") long idMedecin) {
4.
5.     // on utilise la couche métier
6.     try {
7.         // réponse
8.         return new ModelAndView(view, "data", metier.getMedecinById(idMedecin));
9.     } catch (Exception e) {
10.        // réponse
11.        return createResponseError("108", e.getMessage());
12.    }
13. }
```

Voici un exemple d'exécution :



9.6.4.7 Obtenir un rendez-vous identifié par son n°

Le code est le suivant :

```

1. // trouver un Rv identifié par son id
2. @RequestMapping(value = "/getRvById/{idRv}", method = RequestMethod.GET)
3. public ModelAndView getRvById(@PathVariable("idRv") long idRv) {
4.
5.     // on utilise la couche métier
6.     try {
7.         // réponse
8.         return new ModelAndView(view, "data", metier.getRvById(idRv));
9.     } catch (Exception e) {
10.        // réponse
11.        return createResponseError("109", e.getMessage());
12.    }
13. }
```

Voici un exemple d'exécution :



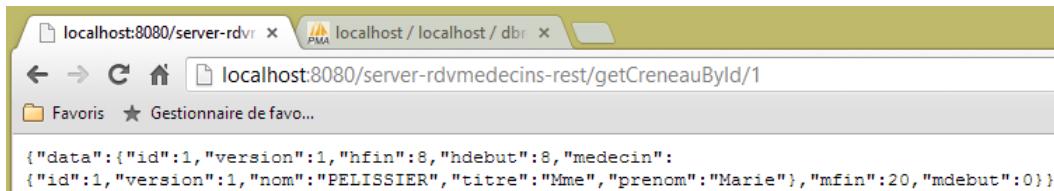
9.6.4.8 Obtenir un créneau horaire identifié par son n°

Le code est le suivant :

```

1. // trouver un créneau horaire identifié par son id
2. @RequestMapping(value = "/getCreneauById/{idCreneau}", method = RequestMethod.GET)
3. public ModelAndView getCreneauById(@PathVariable("idCreneau") long idCreneau) {
4.
5.     // on utilise la couche métier
6.     try {
7.         // réponse
8.         return new ModelAndView(view, "data", metier.getCreneauById(idCreneau));
9.     } catch (Exception e) {
10.        // réponse
11.        return createResponseError("110", e.getMessage());
12.    }
13. }
```

Voici un exemple d'exécution :



9.6.4.9 Ajouter un rendez-vous

Le code est le suivant :

```

1. // ajouter un RV
2. @RequestMapping(value = "/ajouterRv/{jour}/{idCreneau}/{idClient}", method =
   RequestMethod.GET)
3. public ModelAndView ajouterRv(@PathVariable("jour") String jour,
   @PathVariable("idCreneau") long idCreneau,
   @PathVariable("idClient") long idClient) {
4.
5.
6.    // on utilise la couche métier
7.    try {
8.        Date jourRv = new SimpleDateFormat("dd-MM-yyyy").parse(jour);
9.        Client client = metier.getClientById(idClient);
10.       Creneau creneau = metier.getCreneauById(idCreneau);
11.       Rv rv = metier.ajouterRv(jourRv, creneau, client);
12.       // réponse
13.       return new ModelAndView(view, "data", rv);
14.    } catch (Exception e) {
15.        // réponse
16.        return createResponseError("111", e.getMessage());
17.    }
18. }
```

Voici un exemple d'exécution :



On notera :

- le n° (2) du rendez-vous créé ;
- le format JSON du jour. Cela nous posera ultérieurement des problèmes car le client Gson ne s'attend pas à recevoir une date sous cette forme. Il doit être possible de fixer la forme JSON d'une date. Cela n'a pas été fait ici.

9.6.4.10 Supprimer un rendez-vous

Le code est le suivant :

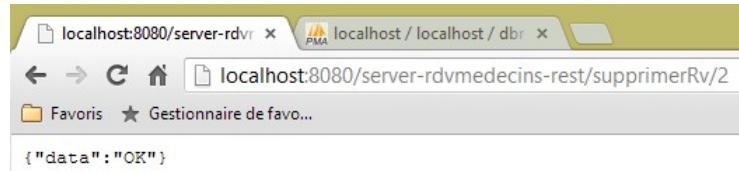
```

1. // supprimer un RV
2. @RequestMapping(value = "/supprimerRv/{idRv}", method = RequestMethod.GET)
3. public ModelAndView supprimerRv(@PathVariable("idRv") long idRv) {
4.
5.    // on utilise la couche métier
```

```

6.     try {
7.         Rv rv = metier.getRvById(idRv);
8.         metier.supprimerRv(rv);
9.         // réponse
10.        return new ModelAndView(view, "data", "OK");
11.    } catch (Exception e) {
12.        // réponse
13.        return createResponseError("112", e.getMessage());
14.    }
15. }
```

Voici un exemple d'exécution :



9.6.4.11 Obtenir l'agenda d'un médecin

Le code est le suivant :

```

1. // l'agenda d'un medecin
2. @RequestMapping(value = "/getAgendaMedecinJour/{idMedecin}/{jour}", method =
RequestMethod.GET)
3. public ModelAndView getAgendaMedecinJour(@PathVariable("idMedecin") long idMedecin,
@PathVariable("jour") String jour) {
4.
5.     // on utilise la couche métier
6.     try {
7.         Date jourRv = new SimpleDateFormat("dd-MM-yyyy").parse(jour);
8.         Medecin medecin = metier.getMedecinById(idMedecin);
9.         AgendaMedecinJour agenda = metier.getAgendaMedecinJour(medecin, jourRv);
10.        System.out.println(String.format("jourRv=%s, jour=%s", jourRv, agenda.getJour()));
11.        // réponse
12.        return new ModelAndView(view, "data", agenda);
13.    } catch (Exception e) {
14.        // réponse
15.        return createResponseError("113", e.getMessage());
16.    }
17. }
```

Voici un exemple d'exécution :

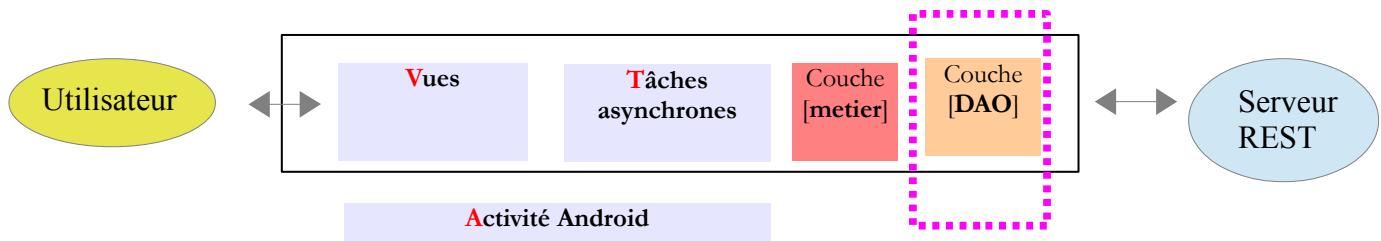
```

{
  "data": [
    {
      "jour": 1370988000000,
      "creneauxMedecinJour": [
        {
          "creneau": {
            "id": 1,
            "version": 1,
            "hfin": 8,
            "hdebut": 8,
            "medecin": {
              "id": 1,
              "version": 1,
              "nom": "PELISSIER",
              "titre": "Mme",
              "prenom": "Marie"
            },
            "mfin": 20,
            "mdebut": 0,
            "rv": null
          }
        }
      ]
    }
  ]
}

```

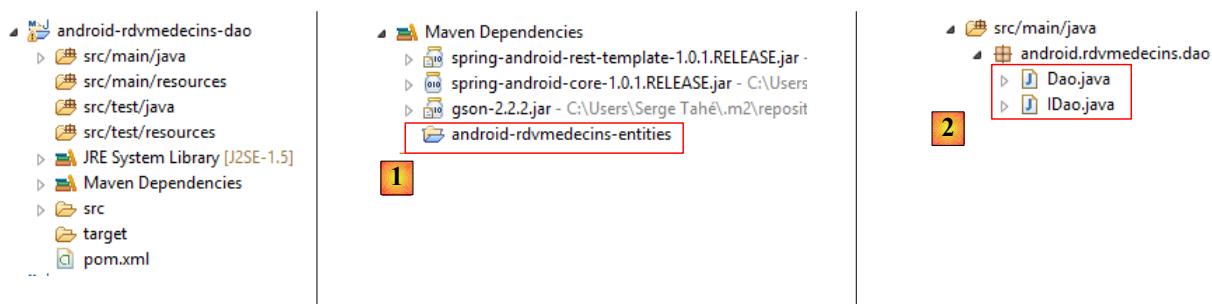
- en [1], le format JSON du jour de l'agenda ;
- en [2], le format JSON du jour d'un rendez-vous. Ce ne sont pas les mêmes. Il faudra gérer ces différences côté client.

9.7 La couche [DAO] du client Android



9.7.1 Le projet Eclipse

Le projet Eclipse de la couche [DAO] du client Android est le suivant :



- en [1], le projet Maven de la couche [DAO] a une dépendance sur le projet [android-rdvmedecins-entities]. Ce projet rassemble les entités décrites page 111, paragraphe 9.5 ;
- en [2], la couche [DAO] du client Android.

L'interface [IDao] est la suivante :

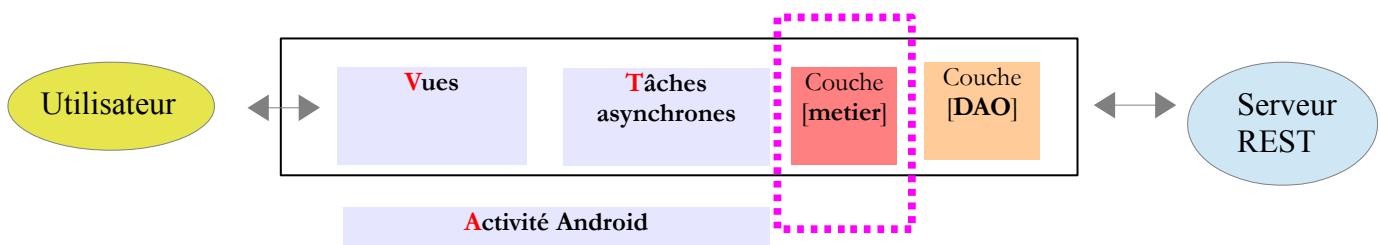
```

1. package android.rdvmedecins.dao;
2. import java.util.Map;
3.
4. public interface IDao {
5.     public String executeRestService(String method, String urlService, Object request,
6.         Map<String, String> paramètres);
7.     public void setTimeout(int millis);

```

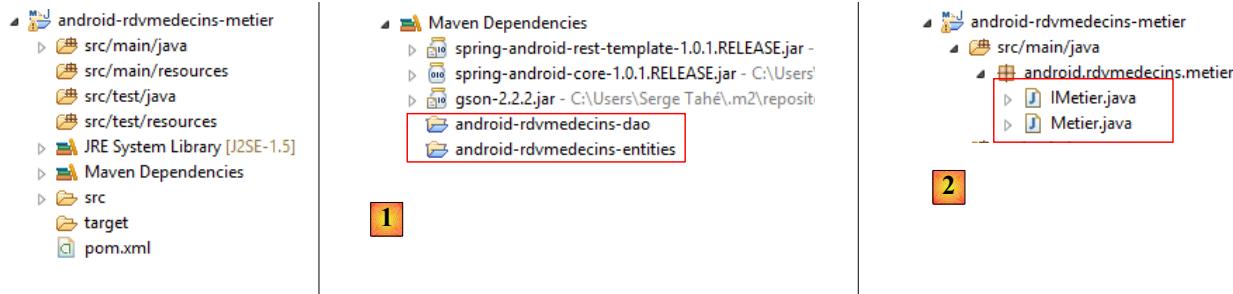
Cette interface et son implémentation sont identiques à ce qu'elles étaient dans la couche [DAO] de l'exemple 2 page 69, paragraphe 4.4.2.

9.8 La couche [métier] du client Android



9.8.1 Le projet Eclipse

Le projet Eclipse de la couche [métier] du client Android est le suivant :



- en [1], le projet Maven de la couche [métier] a une unique dépendance, celle sur la couche [DAO] du client. Les autres apparaissent par cascade ;
- en [2], la couche [métier] du client Android.

9.8.2 Implémentation

L'interface [IMetier] est la suivante :

```

1. package android.rdvmedecins.metier;
2.
3. import java.util.List;
4.
5. import android.rdvmedecins.dao.IDao;
6. import android.rdvmedecins.entities.dao.Client;
7. import android.rdvmedecins.entities.dao.Creneau;

```

```

8. import android.rdvmedecins.entities.dao.Medecin;
9. import android.rdvmedecins.entities.dao.Rv;
10. import android.rdvmedecins.entities.metier.AgendaMedecinJour;
11.
12. public interface IMetier {
13.
14.     // @RequestMapping(value = "/getAllClients", method = RequestMethod.GET)
15.     public abstract List<Client> getAllClients();
16.
17.     // @RequestMapping(value = "/getAllMedecins", method = RequestMethod.GET)
18.     public abstract List<Medecin> getAllMedecins();
19.
20.     // @RequestMapping(value = "/getAllCreneaux/{idMedecin}", method =
21.     // RequestMethod.GET)
22.     public abstract List<Creneau> getAllCreneaux(Long idMedecin);
23.
24.     // @RequestMapping(value = "/getRvMedecinJour/{idMedecin}/{jour}", method =
25.     // RequestMethod.GET)
26.     public abstract List<Rv> getRvMedecinJour(Long idMedecin, String jour);
27.
28.     // @RequestMapping(value = "/getClientById/{idClient}", method = RequestMethod.GET)
29.     public abstract Client getClientById(Long idClient);
30.
31.     // @RequestMapping(value = "/getMedecinById/{idMedecin}", method = RequestMethod.GET)
32.     public abstract Medecin getMedecinById(Long idMedecin);
33.
34.     // @RequestMapping(value = "/getRvById/{idRv}", method = RequestMethod.GET)
35.     public abstract Rv getRvById(Long idRv);
36.
37.     // @RequestMapping(value = "/getCreneauById/{idCreneau}", method = RequestMethod.GET)
38.     public abstract Creneau getCreneauById(Long idCreneau);
39.
40.     // @RequestMapping(value = "/ajouterRv/{jour}/{idCreneau}/{idClient}", method =
41.     RequestMethod.GET)
41.     public abstract Rv ajouterRv(String jour, Long idCreneau, Long idClient);
42.
43.     // @RequestMapping(value = "/supprimerRv/{idRv}", method = RequestMethod.GET)
44.     public abstract void supprimerRv(Long idRv);
45.
46.     // @RequestMapping(value = "/getAgendaMedecinJour/{idMedecin}/{jour}", method =
47.     RequestMethod.GET)
47.     public abstract AgendaMedecinJour getAgendaMedecinJour(Long idMedecin, String jour);
48.
49.     // setters
50.     public abstract void setDao(IDao dao);
51.
52.     public abstract void setUrlServiceRest(String urlServiceRest);
53.
54. }

```

- l'interface [IMetier] du client Android correspond aux méthodes exposées par le service REST du serveur. On rappelle ainsi pour chaque méthode, l'URL du service REST visée.

L'implémentation [Metier] de cette interface est la suivante :

```

1. package android.rdvmedecins.metier;
2.
3. ...
4.
5. public class Metier implements IMetier {
6.
7.     // couche [dao]
8.     private IDao dao;

```

```

9.    // mapper JSON
10.   private Gson gson = new Gson();
11.   // url du service REST
12.   private String urlServiceRest;
13.
14.   // @RequestMapping(value = "/getAllClients", method = RequestMethod.GET)
15.   public List<Client> getAllClients() {
16.       // adresse du service REST
17.       String urlService = String.format("http://%s/getAllClients", urlServiceRest);
18.       // exécution service
19.       String réponse = dao.executeRestService("get", urlService, null, new HashMap<String,
      String>());
20.       // exploitation réponse
21.       try {
22.           return gson.fromJson(réponse, new TypeToken<List<Client>>() {
23.               }.getType());
24.       } catch (Exception ex) {
25.           throw new RdvMedecinsException(String.format("Réponse incorrecte du serveur: %s",
      réponse), ex, 20);
26.       }
27.   }
28.
29.   // @RequestMapping(value = "/getAllMedecins", method = RequestMethod.GET)
30.   public List<Medecin> getAllMedecins() {
31. ...
32. }
33. ...
34.
35.   // setters
36. ...
37. }

```

- ligne 8 : une référence sur la couche [DAO]. Sera instanciée par la fabrique du modèle AVT ;
- ligne 10 : la bibliothèque JSON utilisée côté client Android est la bibliothèque Gson de Google ;
- ligne 12 : l'URL du service REST interrogé par le client Android. Est initialisée lorsque l'utilisateur donne cette information ;
- ligne 15 : la méthode d'obtention de la liste des clients ;
- ligne 17 : construction de l'URL complète de la méthode REST à appeler ;
- ligne 19 : on demande à la couche [DAO] d'appeler cette méthode. On rappelle les quatre paramètres de la méthode [executeRestService] de la couche [DAO] :
 1. la méthode "get" ou "post" de la requête HTTP à émettre,
 2. l'URL complète de la méthode REST à exécuter,
 3. un dictionnaire des données transmises par une opération HTTP POST. Donc *null* ici, puisqu'on fait une opération HTTP GET,
 4. sous la forme d'un dictionnaire, les valeurs des variables de l'URL. Ici il n'y a pas de variables dans l'URL. On fournit alors un dictionnaire vide. Il ne peut être *null* ;

On ne gère pas d'exception. La couche [DAO] lance des exceptions non contrôlées. Elles vont remonter toutes seules jusqu'à la couche [Présentation] assurée par le modèle AVAT ;

- ligne 22 : on reçoit de la couche [DAO] une chaîne JSON représentant une liste de clients. Cette chaîne est alors utilisée pour créer un objet de type *List<Client>* (ligne 15) ;
- ligne 25 : au cas où la chaîne n'a pu être transformée en un objet de type *List<Client>* une exception est lancée.

Toutes les méthodes de la couche [métier] suivent la même logique :

```

1. public T méthode(T1 param1, T2 param2, ...) {
2.     // adresse du service REST
3.     String urlService = String.format("http://%s/unCheminAvecOuPasDeVariables",
      urlServiceRest);
4.     // paramètres du service REST
5.     Map<String, String> paramètres = new HashMap<String, String>();
6.     paramètres.put("clé1", objet1);
7.     paramètres.put("clé2", objet2);
8.     // exécution service REST

```

```

9.     String réponse = dao.executeRestService("get", urlService, null, paramètres);
10.    // exploitation réponse du service REST
11.    try {
12.        return gson.fromJson(réponse, new TypeToken<T>() {
13.            .getType());
14.    } catch (Exception ex) {
15.        throw new RdvMedecinsException(String.format("Réponse incorrecte du serveur:
16. %s", réponse), ex, 20);
17.    }

```

Prenons par exemple la méthode [getAllCreneaux]

```

1.   // @RequestMapping(value = "/getAllCreneaux/{idMedecin}", method =
2.   // RequestMethod.GET)
3.   public List<Creneau> getAllCreneaux(Long idMedecin) {
4.       // adresse du service REST
5.       String urlService = String.format("http://%s/getAllCreneaux/{idMedecin}",
urlServiceRest);
6.       // paramètres du service REST
7.       Map<String, String> paramètres = new HashMap<String, String>();
8.       paramètres.put("idMedecin", String.valueOf(idMedecin));
9.       // exécution service REST
10.      String réponse = dao.executeRestService("get", urlService, null, paramètres);
11.      // exploitation de la réponse du service REST
12.      try {
13.          return gson.fromJson(réponse, new TypeToken<List<Creneau>>() {
14.              .getType());
15.      } catch (Exception ex) {
16.          throw new RdvMedecinsException(String.format("Réponse incorrecte du serveur:
17. %s", réponse), ex, 22);
17.      }
18.  }

```

- ligne 1 : l'URL interrogée. Elle a une variable, le n° [idMedecin] du médecin ;
- ligne 5 : l'URL complète est construite ;
- lignes 7-8 : un dictionnaire est construit pour affecter des valeurs aux variables de l'URL, ici une seule variable ;
- ligne 10 : la requête au service REST est faite par la couche [DAO] ;
- ligne 13 : on rend la réponse, ici un objet de type *List<Creneau>* ;
- ligne 16 : si la chaîne JSON n'a pu être convertie en un objet de type *List<Creneau>* une exception est lancée.

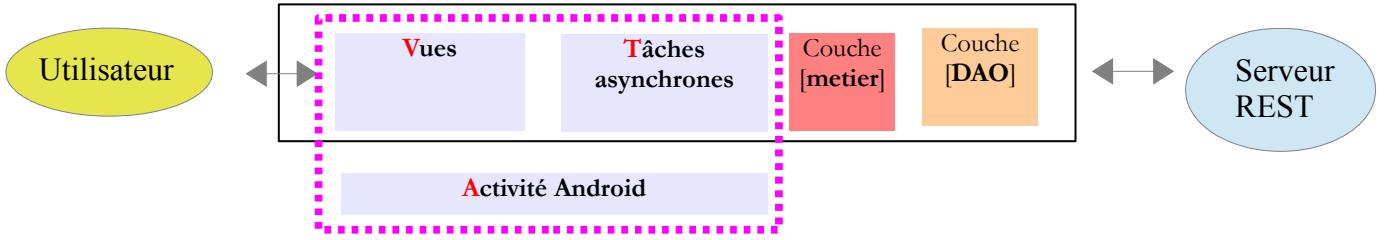
Nous n'allons pas décrire la totalité des méthodes. Le lecteur intéressé les trouvera dans les codes source qui accompagnent ce document. Certaines méthodes ont été plus difficiles à écrire que d'autres à cause de la représentation JSON de l'objet de type [java.util.Date] envoyée par la bibliothèque Jackson du serveur. Elle n'a pas été reconnue par la bibliothèque Gson du client Android et on avait alors une exception. Dans ces cas, il a fallu procéder en trois temps :

- déserialiser la chaîne JSON reçue en un objet *Map<String, Object>* ;
- remplacer dans ce dictionnaire les jours par le bon format ;
- resérialiser le nouveau dictionnaire en l'objet T que doit rendre la méthode.

C'est assez lourd et présente peu d'intérêt mais c'est un bon exercice de sérialisation / désérialisation d'un objet JSON.

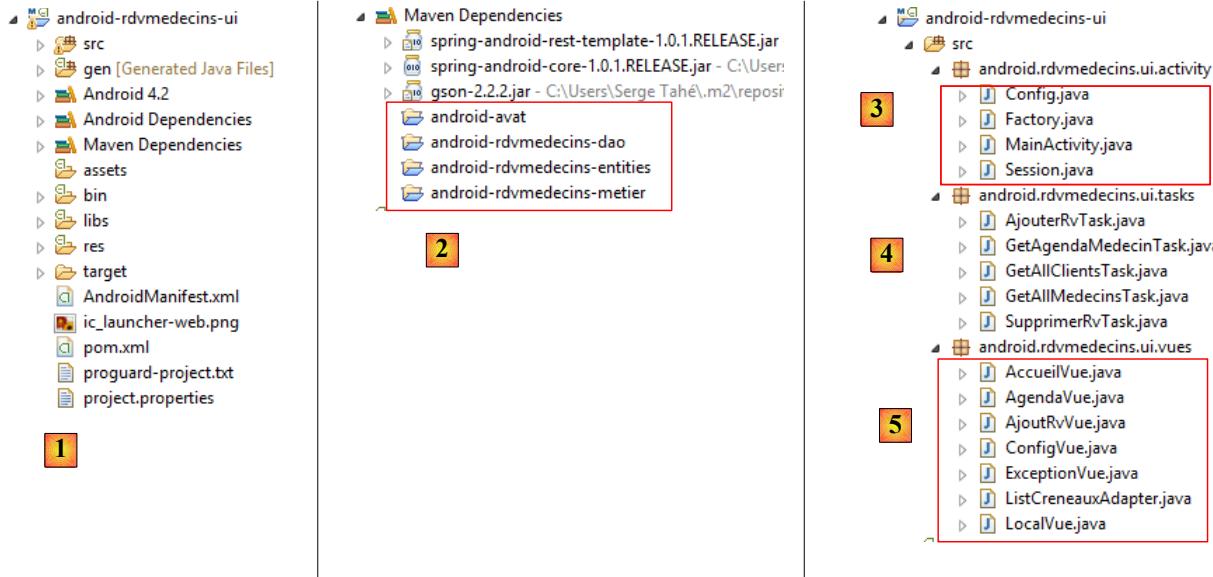
9.9 La couche [AVT]

On aborde ici la couche AVT (Activité – Vues – Tâches) du client Android.



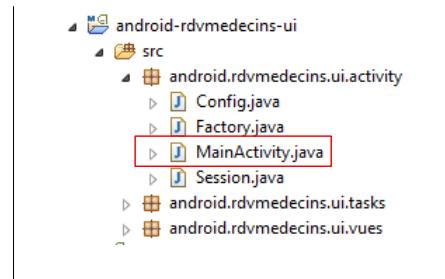
9.9.1 Le projet Eclipse

Le projet Eclipse est le suivant :



- en [1], un projet Maven de type Android ;
- en [2] les dépendances Maven. Il y en a deux :
 - celle sur le modèle [AVT] ;
 - celle sur la couche [métier] que nous venons d'écrire ;
 Les autres dépendances découlent des deux précédentes.
- en [3], les codes source :
 - en [3] : l'activité, la fabrique, la configuration et la session ;
 - en [4] : les tâches asynchrones ;
 - en [5] : les vues.

9.9.2 L'activité Android



La classe [MainActivity] est la suivante :

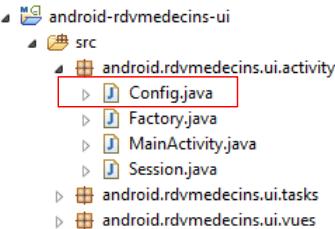
```
1. package android.rdvmedecins.ui.activity;
2.
3. ...
4.
5. public class MainActivity extends FragmentActivity {
6.
7.     // la factory
8.     private Factory factory;
9.     // les vues
10.    private Vue configVue;
11.
12.    @Override
13.    protected void onCreate(Bundle savedInstanceState) {
14.        super.onCreate(savedInstanceState);
15.        // le sablier
16.        requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
17.        setProgressBarIndeterminateVisibility(false);
18.        // template qui accueille les vues
19.        setContentView(R.layout.main);
20.        // la factory
21.        factory = new Factory(this, new Config());
22.        // on définit la première vue
23.        configVue = (Vue) factory.getObject(Factory.CONFIG_VUE, null, "ConfigVue");
24.        // on fait de configVue la vue courante
25.        showVue(configVue);
26.    }
27.
28.    // affichage d'une vue
29.    public void showVue(Vue vue) {
30.        // affichage vue
31.        FragmentTransaction fragmentTransaction = getFragmentManager().beginTransaction();
32.        fragmentTransaction.replace(R.id.container, vue);
33.        fragmentTransaction.commit();
34.    }
35.
36. }
```

Elle a les fonctionnalités suivantes :

- ligne 21 : créer la fabrique d'objets en lui passant la classe de configuration de l'application ;
- lignes 23-25 : afficher la première vue. Celle-ci est la suivante :



9.9.3 La classe de configuration



La classe [Config] est la suivante :

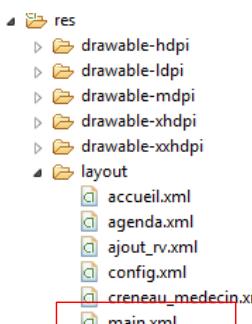
```

1. package android.rdvmedecins.ui.activity;
2.
3. public class Config {
4.
5.     // le mode verbeux ou non de l'application
6.     private boolean verbose = true;
7.     // temps d'attente en millisecondes de la réponse du service REST
8.     private int timeout=1000;
9.
10.    // getters et setters
11.    ...
12.
13. }
```

Elle est identique à ce qu'elle était dans l'exemple 2.

9.9.4 Le patron des vues

Toutes les vues auront le format précédent imposé par le fichier [main.xml] de l'activité :



Le fichier [main.xml] est le suivant :

```

1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     xmlns:tools="http://schemas.android.com/tools"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:gravity="center"
6.     android:orientation="vertical" >
7.
8.     <LinearLayout
9.         android:id="@+id/header"
10.        android:layout_width="match_parent"
11.        android:layout_height="100dp"
12.        android:layout_weight="0.1"
```

```

13.        android:background="@color/LavenderBlush2" >
14.
15.        <TextView
16.            android:id="@+id/textViewHeader"
17.            android:layout_width="match_parent"
18.            android:layout_height="wrap_content"
19.            android:layout_gravity="center"
20.            android:gravity="center_horizontal"
21.            android:text="@string/txt_header"
22.            android:textAppearance="?android:attr/textAppearanceLarge"
23.            android:textColor="@color/blue" />
24.    </LinearLayout>
25.
26.    <LinearLayout
27.        android:layout_width="match_parent"
28.        android:layout_height="fill_parent"
29.        android:layout_weight="0.8"
30.        android:orientation="horizontal" >
31.
32.        <LinearLayout
33.            android:id="@+id/left"
34.            android:layout_width="100dp"
35.            android:layout_height="match_parent"
36.            android:background="@color/Lightcyan2"
37.            android:orientation="vertical" >
38.
39.            <TextView
40.                android:id="@+id/Lnk_Config"
41.                android:layout_width="fill_parent"
42.                android:layout_height="40dp"
43.                android:layout_marginTop="100dp"
44.                android:gravity="center_vertical|center_horizontal"
45.                android:text="@string/Lnk_config"
46.                android:textColor="@color/blue" />
47.
48.            <TextView
49.                android:id="@+id/Lnk_Accueil"
50.                android:layout_width="fill_parent"
51.                android:layout_height="40dp"
52.                android:gravity="center_vertical|center_horizontal"
53.                android:text="@string/Lnk_accueil"
54.                android:textColor="@color/blue" />
55.
56.            <TextView
57.                android:id="@+id/Lnk_Agenda"
58.                android:layout_width="fill_parent"
59.                android:layout_height="40dp"
60.                android:gravity="center_vertical|center_horizontal"
61.                android:text="@string/Lnk_agenda"
62.                android:textColor="@color/blue" />
63.
64.            <TextView
65.                android:id="@+id/Lnk_Ajout"
66.                android:layout_width="fill_parent"
67.                android:layout_height="40dp"
68.                android:gravity="center_vertical|center_horizontal"
69.                android:text="@string/Lnk_ajout"
70.                android:textColor="@color/blue" />
71.        </LinearLayout>
72.
73.        <FrameLayout
74.            android:id="@+id/container"
75.            android:layout_width="match_parent"

```

```

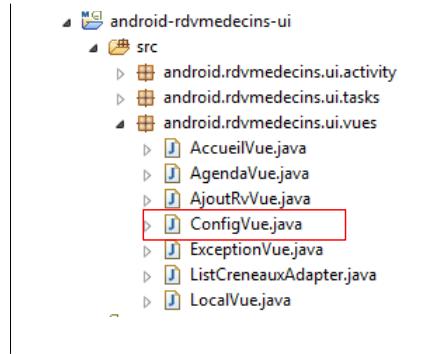
76.         android:layout_height="match_parent"
77.         android:layout_marginLeft="20dp"
78.         android:background="@color/floral_white"
79.         tools:context=".MainActivity"
80.         tools:ignore="MergeRootFrame" >
81.     
```

`</FrameLayout>`
`</LinearLayout>`
`83.`
`84. <LinearLayout`
`85. android:id="@+id/bottom"`
`86. android:layout_width="match_parent"`
`87. android:layout_height="100dp"`
`88. android:layout_weight="0.1"`
`89. android:background="@color/wheat1" >`
`90.`
`91. <TextView`
`92. android:id="@+id/textViewBottom"`
`93. android:layout_width="fill_parent"`
`94. android:layout_height="fill_parent"`
`95. android:gravity="center_vertical/center_horizontal"`
`96. android:text="@string/txt_bottom"`
`97. android:textColor="@color/blue" />`
`98.`
`</LinearLayout>`
`99.`
`100. </LinearLayout>`

et correspond à la vue suivante :



9.9.5 La vue [Config]



C'est la première vue affichée. Elle est composée des éléments suivants :



Nº	Id	Type	Rôle
1	edtUrlServiceRest	EditText	URL du service REST
2	btnValider	Button	tente une connexion au serveur REST
2	btnAnnuler	Button	annule la connexion

Le code de la vue est le suivant :

```

1. package android.rdvmedecins.ui.vues;
2.
3. ...
4.
5. public class ConfigVue extends LocalVue {
6.
7.     // les éléments de l'interface visuelle
8.     private Button btnValider;
9.     private Button btnAnnuler;
10.    private EditText edtUrlServiceRest;
11.    private TextView txtErrorUrlServiceRest;
12.
13.    // ses résultats
14.    private List<Object> results = new ArrayList<Object>();
15.
16.    // les saisies
17.    private String urlServiceRest;
18.
19.    @Override

```

```

20.    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
21.        savedInstanceState) {
22.            // on crée la vue du fragment à partir de sa définition XML
23.            return inflater.inflate(R.layout.config, container, false);
24.
25.    @Override
26.    public void onActivityCreated(Bundle savedInstanceState) {
27. ....
28.        // bouton Annuler
29.        btnAnnuler = (Button) activity.findViewById(R.id.btn_Annuler);
30.        btnAnnuler.setOnClickListener(new OnClickListener() {
31.            @Override
32.            public void onClick(View arg0) {
33.                // on annule les tâches en cours
34.                cancelAll();
35.                // on annule l'attente
36.                cancelWaiting();
37.            }
38.        });
39.        // état des boutons
40.        btnValider.setVisibility(View.VISIBLE);
41.        btnAnnuler.setVisibility(View.INVISIBLE);
42.
43.    }
44.
45.    @Override
46.    public void onResume() {
47.        // parent
48.        super.onResume();
49.        // on gère les liens de navigation
50.        activateLinks(0);
51.    }
52.
53.    // validation de la page
54.    protected void doValidator() {
55. ....
56.        // on demande la liste des médecins
57.        ITask getAllMedecinsTask = (ITask) factory.getObject(Factory.GETALLMEDECINS_TASK,
58.            this, "GetAllMedecinsTask");
59.        // on exécute la tâche (urlServiceRest)
60.        getAllMedecinsTask.doWork(edtUrlServiceRest.getText().toString());
61.        // on demande la liste des clients
62.        ITask getAllClientsTask = (ITask) factory.getObject(Factory.GETALLCLIENTS_TASK, this,
63.            "GetAllClientsTask");
64.        // on exécute la tâche (urlServiceRest)
65.        getAllClientsTask.doWork(edtUrlServiceRest.getText().toString());
66.        // début de l'attente
67.        beginWaiting();
68.        // début monitoring
69.        beginMonitoring();
70.    }
71.    private boolean isPageValid() {
72. ....
73.    }
74.    @Override
75.    public void notifyEndOfTasks() {
76.        // fin de l'attente
77.        cancelWaiting();
78.        // on affiche le résultat
79.        Toast.makeText(activity, "Travail terminé", Toast.LENGTH_LONG).show();

```

```

80.     boolean erreur = false;
81.     for (Object result : results) {
82.         // on traite les résultats des tâches exécutées
83.         if (result instanceof Map<?, ?>) {
84.             // on a un dictionnaire - son unique entrée est une liste - on la met dans la
85.             session
86.             @SuppressWarnings("unchecked")
87.             Map<String, Object> map = (Map<String, Object>) result;
88.             String clé = map.keySet().iterator().next();
89.             // on met la liste dans la session
90.             session.add(clé, map.get(clé));
91.         } else if (result instanceof Exception) {
92.             // on affiche l'exception
93.             showException((Exception) result);
94.             // on ne continue pas
95.             erreur = true;
96.             break;
97.         }
98.     } // si pas d'erreur, on passe à la vue suivante
99.     if (!erreur) {
100.         // on définit la vue suivante
101.         Vue accueilVue = (Vue) factory.getObject(Factory.ACCEUIL_VUE, null,
102.             "AccueilVue");
103.         // on fait de configVue la vue courante
104.         mainActivity.showVue(accueilVue);
105.     }
106. }
107.
108. @Override
109. // on reçoit des notifications de l'action
110. public void notifyEvent(IWorker worker, int eventType, Object event) {
111.     // on passe l'info au parent
112.     super.notifyEvent(worker, eventType, event);
113.     // on traite le cas WORK_INFO
114.     if (eventType == IBoss.WORK_INFO) {
115.         results.add(event);
116.     }
117. }
118.
119. // début de l'attente
120. private void beginWaiting() {
121. ...
122. }
123.
124. // fin de l'attente
125. protected void cancelWaiting() {
126. ...
127. }
128.
129. }

```

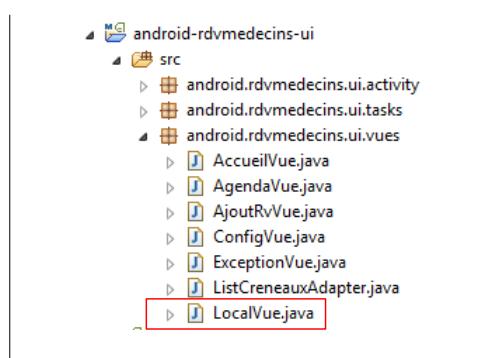
- ligne 5 : la vue dérive de la vue [LocalVue]. Cette vue sera le parent de toutes les vues de l'application. On y rassemble tout ce qui peut être factorisé entre vues ;
- lignes 30-38 : la gestion du clic sur le bouton [Annuler]. C'est celui désormais classique de tous les exemples vus dans ce document. Nous n'y reviendrons plus ;
- lignes 46-51 : la méthode [onResume] est exécutée juste avant l'affichage de la vue. On l'utilisera dans chaque vue pour deux choses :
 - récupérer des informations dans la session afin de les afficher ;
 - activer certains liens du bandeau gauche.

- ligne 50 : la méthode `activateLinks(int i)` appartient à la classe parent [LocalVue]. Elle fait en sorte que les liens du bandeau gauche de n°*i* soient cliquables et de couleur bleue, les autres étant désactivés et de couleur noire. Cela permet à l'utilisateur de revenir en arrière dans l'assistant de prise de rendez-vous. `activateLinks(0)` va ici désactiver tous les liens.
- ligne 57 : on instancie la tâche qui va demander la liste des médecins au serveur REST. Cette liste restera en session ;
- ligne 61 : la tâche est lancée ;
- lignes 60-63 : on lance une seconde tâche, cette fois pour obtenir la liste des clients qui sera mémorisée elle aussi en session ;
- ligne 65 : début de l'attente ;
- ligne 67 : on surveille la fin de ces deux tâches ;
- ligne 110 : la vue [Config] va voir passer les notifications des deux tâches lancées ;
- lignes 114-116 : dans le cas d'une notification [WORK_INFO], on mémorise l'information dans la liste d'objets de la ligne 14. Pour chacune des deux tâches, l'information enregistrée est un dictionnaire :
 - avec la clé " `client` " et une valeur de type `List<Client>` pour la liste des clients,
 - avec la clé " `medecin` " et une valeur de type `List<Medecin>` pour la liste des médecins ;
- ligne 75 : on est prévenu de la fin des deux tâches ;
- ligne 77 : on annule l'attente ;
- ligne 81 : on exploite les deux informations stockées dans la liste des résultats ;
- ligne 83 : on regarde si le résultat obtenu est bien un dictionnaire. Si oui, il est mis dans la session (lignes 84-89). Sinon on regarde si le résultat est une exception (ligne 90). Si oui, elle est affichée (ligne 92) par une méthode de la classe [LocalVue] ;
- ligne 99 : s'il n'y a pas eu d'erreurs, la liste des clients et celle des médecins sont maintenant en session. C'est là qu'iront les chercher les vues qui en ont besoin ;
- lignes 101-103 : s'il n'y a pas eu d'erreur, la vue suivante de l'assistant est affichée ;

On notera les points suivants :

- on trouve désormais un peu de logique dans la vue (lignes 56-63). La vue doit " savoir " qu'elle doit appeler deux tâches asynchrones et dans quel ordre. Auparavant, cette connaissance était dans l'action. On s'écarte un peu de la notion de séparation des tâches (separation of concern) mais peu. Il y aura toujours très peu de logique dans la vue, celle-ci étant concentrée dans la couche [métier] ;
- lignes 56-63, on appelle deux tâches asynchrones qui vont s'exécuter en parallèle. On aurait pu écrire une méthode dans la couche [métier] pour récupérer les deux listes, médecins et clients. Une tâche asynchrone aurait alors suffi pour l'exécuter et on aurait mieux atteint l'objectif de séparation des tâches. Mais dans ce cas les deux listes auraient été obtenues l'une après l'autre et non pas en parallèle.

9.9.6 La vue [LocalVue]



La vue [LocalVue] est parente de toutes les vues de l'application. On y a rassemblé tout ce qui pouvait être factorisé entre vues. Son code est le suivant :

```

1. package android.rdvmedecins.ui.vues;
2.
3. ...
4.
5. public abstract class LocalVue extends Vue {
6.     // encapsule des méthodes et données communes aux classes filles
7.     protected ISession session;

```

```

8.    // la main activity
9.    protected MainActivity mainActivity;
10.   // navigation
11.   protected TextView lnkConfig;
12.   protected TextView lnkAccueil;
13.   protected TextView lnkAgenda;
14.   protected TextView lnkAjout;
15.   protected TextView[] links;
16.
17.   // constructeur
18.   public LocalVue() {
19.   }
20.
21.   @Override
22.   public void onActivityCreated(Bundle savedInstanceState) {
23.       // parent
24.       super.onActivityCreated(savedInstanceState);
25.       // la session
26.       session = (ISession) factory.getObject(Factory.SESSION, (Object[]) null);
27.       // la main activity
28.       mainActivity = (MainActivity) activity;
29.       // les liens de la navigation
30.       lnkConfig = (TextView) activity.findViewById(R.id.Lnk_Config);
31.       lnkAccueil = (TextView) activity.findViewById(R.id.Lnk_Accueil);
32.       lnkAgenda = (TextView) activity.findViewById(R.id.Lnk_Agenda);
33.       lnkAjout = (TextView) activity.findViewById(R.id.Lnk_Ajout);
34.       // tableau des liens
35.       links = new TextView[] { lnkConfig, lnkAccueil, lnkAgenda, lnkAjout };
36.   }
37.
38.   // navigation
39.   protected void navigateToView(View v) {
40. ...
41.   }
42.
43.   // gestion des liens
44.   protected void activateLinks(int idLink) {
45. ...
46.   }
47.
48.   // affichage exception
49.   protected void showException(Exception ex) {
50.       // on affiche l'exception dans une boîte de dialogue
51.       DialogFragment dialog = new ExceptionVue(ex);
52.       dialog.show(getFragmentManager(), "exception");
53.   }
54.
55.   // gestion du sablier
56.   protected void showHourGlass() {
57.       activity.setProgressBarIndeterminateVisibility(true);
58.   }
59.
60.   protected void hideHourGlass() {
61.       activity.setProgressBarIndeterminateVisibility(false);
62.   }
63.
64.   @Override
65.   abstract public void notifyEndOfTasks();
66.
67. }
```

- ligne 5 : [LocalVue] étend la vue abstraite [Vue] qui implémente l'interface [IVue] ;
- ligne 65 : la méthode abstraite [notifyEndOfTasks] est implémentée par les classes filles ;

- ligne 56 : pour afficher le sablier ;
- ligne 60 : pour le cacher ;
- ligne 49 : la méthode d'affichage des exceptions qui remontent jusqu'aux vues. On utilise une boîte de dialogue telle que la suivante :



- ligne 51 : on utilise la classe [ExceptionVue] suivante :

```

1. package android.rdvmedecins.ui.vues;
2.
3. ...
4.
5.
6. public class ExceptionVue extends DialogFragment {
7.
8.     // data
9.     private Exception ex;
10.
11.    // constructeurs
12.    public ExceptionVue() {
13.
14.    }
15.
16.    public ExceptionVue(Exception ex) {
17.        this.ex = ex;
18.    }
19.
20.    @Override
21.    public Dialog onCreateDialog(Bundle savedInstanceState) {
22.        // on construit le message
23.        String message = ex.getMessage();
24.        Throwable cause = ex.getCause();
25.        while (cause != null) {
26.            message += String.format("\n[%s, %s]", cause.getClass().getCanonicalName(),
27.            cause.getMessage());
28.            cause = cause.getCause();
29.        }
30.        // on construit la boîte de dialogue
31.        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
32.        builder.setTitle(R.string.dialog_show_exception).setMessage(message)
33.            .setPositiveButton(R.string.dialog_show_exception_close, new
34.                DialogInterface.OnClickListener() {
35.                    public void onClick(DialogInterface dialog, int id) {
36.                        // rien
37.                    });
38.        // on la rend
39.        return builder.create();
40.    }

```

- ligne 6 : la classe étend [DialogFragment] qui permet d'afficher des boîtes de dialogue ;
- lignes 16-18 : le constructeur reçoit en paramètre l'exception à afficher ;
- ligne 21 : la méthode [onCreateDialog] est exécutée à la création du dialogue avant son affichage ;

- lignes 23-28 : on construit le message que l'on va afficher dans la boîte de dialogue. C'est la concaténation des messages d'erreur de toutes les exceptions contenues dans l'exception passée au constructeur ;
- ligne 30 : on construit une boîte de dialogue prédéfinie [AlertDialog] ;
- ligne 31 :
 - setTitle** : fixe le titre de la boîte de dialogue [1] ;
 - setMessage** : fixe le message que la boîte de dialogue va afficher [2] ;
 - setPositiveButton** : fixe la nature du bouton qui va permettre de fermer la boîte de dialogue [3] ;



- ligne 38 : on doit rendre la boîte de dialogue qui vient d'être créée.

Revenons au code de [showException] :

```

1.  // affichage exception
2.  protected void showException(Exception ex) {
3.    // on affiche l'exception dans une boîte de dialogue
4.    DialogFragment dialog = new ExceptionVue(ex);
5.    dialog.show(getFragmentManager(), "exception");
6. }
```

- ligne 4 : la boîte de dialogue est instanciée mais pas créée ;
- ligne 5 : la boîte est créée et affichée.

Revenons au code de [LocalVue] :

```

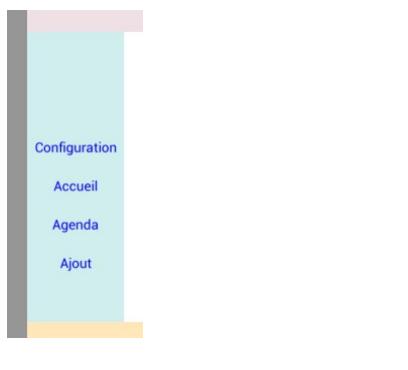
1. public abstract class LocalVue extends Vue {
2.   // encapsule des méthodes et données communes aux classes filles
3.   protected ISession session;
4.   // la main activity
5.   protected MainActivity mainActivity;
6.   // navigation
7.   protected TextView lnkConfig;
8.   protected TextView lnkAccueil;
9.   protected TextView lnkAgenda;
10.  protected TextView lnkAjout;
11.  protected TextView[] links;
12.
13.  // constructeur
14.  public LocalVue() {
15.  }
16.
17.  @Override
18.  public void onActivityCreated(Bundle savedInstanceState) {
19.    // parent
20.    super.onActivityCreated(savedInstanceState);
21.    // la session
22.    session = (ISession) factory.getObject(Factory.SESSION, (Object[]) null);
23.    // la main activity
24.    mainActivity = (MainActivity) activity;
25.    // les liens de la navigation
26.    lnkConfig = (TextView) activity.findViewById(R.id.Lnk_Config);
27.    lnkAccueil = (TextView) activity.findViewById(R.id.Lnk_Accueil);
28.    lnkAgenda = (TextView) activity.findViewById(R.id.Lnk_Agenda);
29.    lnkAjout = (TextView) activity.findViewById(R.id.Lnk_Ajout);
30.    // tableau des liens
```

```

31.     links = new TextView[] { lnkConfig, lnkAccueil, lnkAgenda, lnkAjout };
32. }

```

- ligne 3 : la session qui sert aux échanges d'informations entre vues ;
- ligne 5 : une référence sur l'activité Android qui sous-tend les vues ;
- lignes 7-10 : les quatre liens du bandeau gauche des vues :



- ligne 11 : un tableau contenant ces quatre liens ;
- ligne 22 : récupère une référence sur la session (singleton) ;
- ligne 24 : récupère la nature exacte de l'activité. *activity* est une référence injectée dans la classe abstraite [Vue] dont dérive [LocalVue] et est de type [Activity]. Afin d'éviter des transtypages dans le code des vues, on crée un champ *mainActivity* avec le bon type ;
- lignes 26-29 : on récupère les références des quatre liens ;
- ligne 31 : on les stocke dans le tableau.

L'activation / désactivation des liens se fait avec la méthode suivante :

```

1. // gestion des liens
2. protected void activateLinks(int idLink) {
3.     // on active tous les liens qui précèdent idLink
4.     int i = 0;
5.     while (i < idLink) {
6.         links[i].setTextColor(getResources().getColor(R.color.blue));
7.         links[i].setOnClickListener(new OnClickListener() {
8.
9.             @Override
10.            public void onClick(View v) {
11.                navigateToView(v);
12.
13.            }
14.        });
15.        i++;
16.    }
17.    // on désactive le lien lnk et les suivants
18.    while (i < links.length) {
19.        links[i].setTextColor(getResources().getColor(R.color.black));
20.        links[i].setOnClickListener(null);
21.        i++;
22.    }
23. }

```

- ligne 2 : le paramètre de la méthode est un n° de lien. La méthode active tous les liens qui ont un n° <*idLink*> et désactivent tous les autres ;
- ligne 6 : le lien activé sera de couleur bleue ;
- lignes 7-14 : le clic sur ce lien est géré ;
- ligne 19 : un lien désactivé sera de couleur noire ;
- ligne 20 : le clic sur le lien n'est pas géré ;
- ligne 11 : un clic sur un lien actif va provoquer l'exécution de la méthode [navigateToView] qui va afficher la vue correspondant au lien :

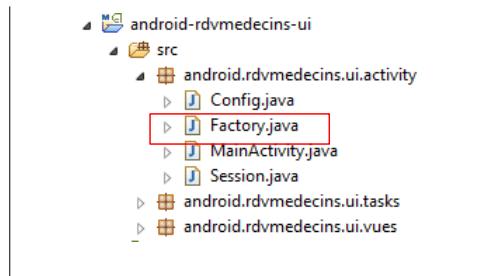
```

1. // navigation
2.     protected void navigateToView(View v) {
3.         // libellé du lien
4.         String libellé = ((TextView) v).getText().toString();
5.         // affichage de la vue correspondant au lien
6.         if (libellé.equals(getResources().getString(R.string.Lnk_config))) {
7.             mainActivity.showVue((Vue) factory.getObject(Factory.CONFIG_VUE, (Object[])
8.             null));
9.         }
10.        if (libellé.equals(getResources().getString(R.string.Lnk_accueil))) {
11.            mainActivity.showVue((Vue) factory.getObject(Factory.ACCEUIL_VUE, (Object[])
12.            null));
13.        }
14.        if (libellé.equals(getResources().getString(R.string.Lnk_agenda))) {
15.            mainActivity.showVue((Vue) factory.getObject(Factory.AGENDA_VUE, (Object[])
16.            null));
17.        }
18.    }

```

- ligne 2 : le paramètre de la méthode est un lien de type [TextView] ;
- ligne 4 : on récupère le libellé de ce lien ;
- lignes 6-17 : on compare ce libellé à ceux des quatres liens ;
- ligne 7 : la classe [MainActivity] a une classe [showVue] permettant d'afficher un objet [Vue]. Cet objet est demandé à la fabrique (singleton).

9.9.7 La fabrique



La fabrique crée tous les objets du client Android :

```

1. public class Factory implements IFactory {
2.
3.     // constantes
4.     public static final int CONFIG = -1;
5.     public static final int SESSION = -2;
6.     public static final int METIER = -3;
7.     public static final int CONFIG_VUE = 10;
8.     public static final int ACCUEIL_VUE = 11;
9.     public static final int AGENDA_VUE = 12;
10.    public static final int AJOUT_RV_VUE = 13;
11.    public static final int GETALLMEDECINS_TASK = 30;
12.    public static final int GETALLCLIENTS_TASK = 31;
13.    public static final int GETAGENDAMEDECIN_TASK = 32;
14.    public static final int AJOUTER_RV_TASK = 33;
15.    public static final int SUPPRIMER_RV_TASK = 34;

```

```

16.
17. // ----- DEBUT SINGLETONS
18. // la configuration
19. private Config config;
20. private boolean verbose;
21. private int timeout;
22.
23. // la couche métier
24. private IMetier metier;
25.
26. // la session
27. private ISession session;
28.
29. // les vues
30. private Vue configVue;
31. private Vue accueilVue;
32. private Vue agendaVue;
33. private Vue ajoutRvVue;
34.
35. // l'activité principale
36. private Activity activity;
37.
38. // ----- FIN SINGLETONS
39.
40. // constructeur
41. public Factory(Activity activity, Config config) {
42.     // l'activité
43.     this.activity = activity;
44.     // la configuration
45.     this.config = config;
46.     // le mode verbeux ou non
47.     verbose = config.isVerbose();
48.     // le temps d'attente de la réponse du service REST en millisecondes
49.     timeout = config.getTimeout();
50. }
51.
52. @Override
53. public Object getObject(int id, Object... params) {
54.     switch (id) {
55.         case CONFIG:
56.             return config;
57.         case SESSION:
58.             return getSession();
59.         case METIER:
60.             return getMetier(params);
61.         case CONFIG_VUE:
62.             return getConfigVue(params);
63.         case ACCUEIL_VUE:
64.             return getAccueilVue(params);
65.         case AGENDA_VUE:
66.             return getAgendaVue(params);
67.         case AJOUT_RV_VUE:
68.             return getAjoutRvVue(params);
69.         case GETALLMEDECINS_TASK:
70.             return getAllMedecinsTask(params);
71.         case GETALLCLIENTS_TASK:
72.             return getAllClientsTask(params);
73.         case GETAGENDAMEDECIN_TASK:
74.             return getAgendaMedecinTask(params);
75.         case AJOUTER_RV_TASK:
76.             return getAjouterRvTask(params);
77.         case SUPPRIMER_RV_TASK:
78.             return getSupprimerRvTask(params);

```

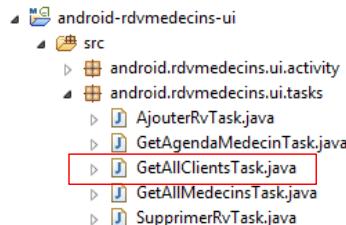
```

79.      }
80.      return null;
81.    }
82. ....

```

Nous donnons ce code seulement pour lister les objets gérés par la fabrique. Leur code de fabrication est analogue à celui des fabriques déjà présentées.

9.9.8 La tâche [GetAllClientsTask]



Cette tâche fournit la liste des médecins :

```

1. package android.rdvmedecins.ui.tasks;
2.
3. ...
4.
5. public class GetAllClientsTask extends Task {
6.
7.     // résultat de la tâche
8.     private Object info;
9.
10.    @Override
11.    // traitement début de tâche dans le thread de l'UI
12.    protected void onPreExecute() {
13.        // on passe la tâche à l'état démarré
14.        boss.notifyEvent(this, IBoss.WORK_STARTED, null);
15.    }
16.
17.    @Override
18.    protected Void doInBackground(Object... params) {
19.        // on demande une référence sur la couche [métier]
20.        IMetier metier = (IMetier) factory.getObject(Factory.METIER, params[0]);
21.        try {
22.            // on demande la liste des clients à la couche métier
23.            List<Client> clients = metier.getAllClients();
24.            // on la met dans un dictionnaire
25.            Map<String, Object> result=new HashMap<String, Object>();
26.            result.put("clients",clients);
27.            info=result;
28.        } catch (Exception ex) {
29.            // on note l'exception
30.            info = ex;
31.        }
32.        // fin
33.        return null;
34.    }
35.
36.    @Override
37.    // traitement fin de tâche dans le thread de l'UI
38.    protected void onPostExecute(Void result) {

```

```

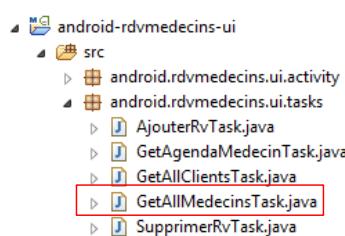
39.     // on a terminé le travail
40.     boss.notifyEvent(this, IBoss.WORK_INFO, info);
41.     boss.notifyEvent(this, IBoss.WORK_TERMINATED, null);
42. }
43.
44. }

```

Toutes les tâches du client Android sont construites sur le même modèle :

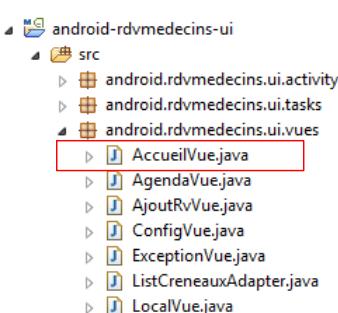
- lignes 12-15 : la méthode [onPreExecute] est utilisée pour envoyer au boss (la vue) la notification [WORK_STARTED] ;
- lignes 38-42 : la méthode [onPostExecute] est utilisée pour envoyer au boss (la vue) la notification [WORK_INFO] avec le résultat de la tâche et la notification [WORK_TERMINATED] ;
- ligne 20 : on demande à la fabrique une référence sur la couche [métier] ;
- lignes 21-31 : on exécute la méthode adéquate de la couche [métier] ;
- lignes 25-27 : l'information produite par la tâche est un dictionnaire avec une unique entrée de clé "**clients**" et de valeur la liste des clients de type **List<Client>**.

9.9.9 La tâche [GetAllMedecinsTask]

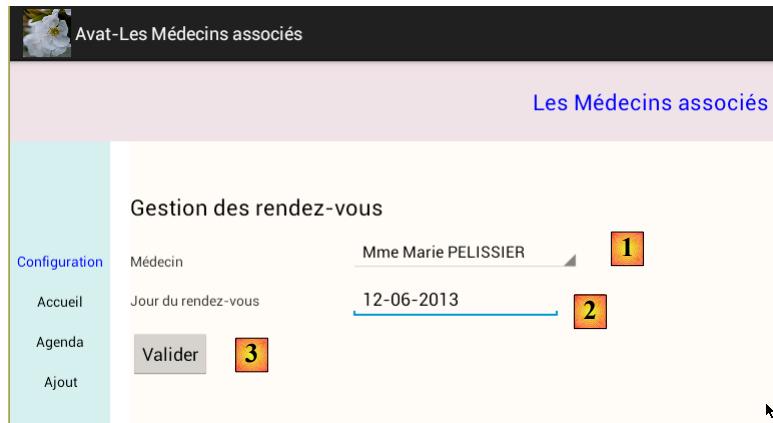


Cette tâche fournit la liste des médecins sous la forme d'un dictionnaire avec une unique entrée de clé "**medecins**" et de valeur la liste des clients de type **List<Medecin>**, ceci de façon similaire à [GetAllClientsTask].

9.9.10 La vue [AccueilVue]



C'est la vue suivante :



N°	Id	Type	Rôle
1	spinnerMedecins	Spinner	liste déroulante des médecins
2	edtJourRv	EditText	la date du rendez-vous au format JJ-MM-AAAA
3	btnValider	Button	affiche l'agenda du médecin choisi
3	btnAnnuler	Button	annule la demande

Le code de la vue est le suivant :

```

1. public void onResume() {
2.     // parent
3.     super.onResume();
4.     // on récupère les médecins en session
5.     medecins = (List<Medecin>) session.get("medecins");
6.     // on construit le tableau affiché par le spinner
7.     String[] arrayMedecins = new String[medecins.size()];
8.     int i = 0;
9.     for (Medecin medecin : medecins) {
10.         arrayMedecins[i] = String.format("%s %s %s", medecin.getTitre(),
11.             medecin.getPrenom(), medecin.getNom());
12.         i++;
13.     }
14.     ArrayAdapter<String> dataAdapter = new ArrayAdapter<String>(activity,
15.         android.R.layout.simple_spinner_item,
16.         arrayMedecins);
17.     dataAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
18.     spinnerMedecins.setAdapter(dataAdapter);
19.     // on gère les liens de navigation
20.     activateLinks(1);
}

```

- ligne 1 : avant de s'afficher la vue doit mettre à jour son modèle avec des informations trouvées dans la session ;
- ligne 5 : la liste des médecins est récupérée en session. La session est injectée dans les vues et les tâches par la fabrique lors de leur création ;
- lignes 8-12 : un tableau [arrayMedecin] de chaînes de la forme [Mr Paul Marand] est construit ;
- lignes 14-17 : ce tableau sera le contenu de la liste déroulante [spinnerMedecins] ;
- ligne 19 : le 1^{er} lien du bandeau gauche est activé. Il permet à l'utilisateur de revenir à la vue [ConfigVue] précédente.

```

1. // validation de la page
2. protected void doValider() {
3.     // on teste la validité des saisies
4.     if (!isValidPage()) {
5.         return;
6.     }

```

```

7.      // on nettoie les résultats
8.      results.clear();
9.      // on demande l'agenda du médecin
10.     ITask getAgendaMedecinTask = (ITask) factory
11.         .getObject(Factory.GETAGENDAMEDECIN_TASK, this, "AgendaMedecinJourTask");
12.     // on exécute la tâche (idMedecin, jourRv)
13.     getAgendaMedecinTask.doWork(idMedecin, jourRv);
14.     // début de l'attente
15.     beginWaiting();
16.     // début monitoring
17.     beginMonitoring();
18. }

```

- ligne 2 : la méthode exécutée lors d'un clic sur le bouton [Valider] ;
- ligne 8 : les résultats des tâches asynchrones sont stockées dans une liste d'objets [results]. Avant chaque nouvelle exécution, on nettoie cette liste ;
- lignes 10-13 : avec les informations saisies (médecin et jour), on lance la tâche asynchrone [GetAgendaMedecinTask] qui crée l'agenda du médecin ;
- ligne 15 : début attente visuelle ;
- ligne 17 : début attente de la fin des tâches.

```

1.   @Override
2.   // on reçoit des notifications de l'action
3.   public void notifyEvent(IWorker worker, int eventType, Object event) {
4.       // on passe l'info au parent
5.       super.notifyEvent(worker, eventType, event);
6.       // on traite le cas WORK_INFO
7.       if (eventType == IBoss.WORK_INFO) {
8.           results.add(event);
9.       }
10.  }

```

- ligne 3 : on traite les notifications envoyées par la tâche lancée ;
- ligne 5 : la notification est d'abord passée au parent ;
- lignes 7-8 : l'information transportée par la notification [WORK_INFO] est mémorisée ;

```

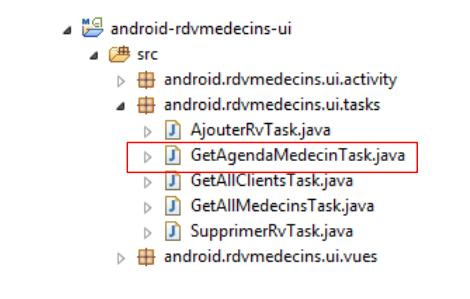
1. @Override
2. public void notifyEndOfTasks() {
3.     // fin de l'attente
4.     cancelWaiting();
5.     // on gère le résultat
6.     boolean erreur = false;
7.     for (Object result : results) {
8.         // on traite les résultats des tâches exécutées
9.         if (result instanceof AgendaMedecinJour) {
10.             // on met l'agenda dans la session
11.             session.add("agenda", result);
12.         } else if (result instanceof Exception) {
13.             // on affiche l'exception
14.             showException((Exception) result);
15.             // erreur
16.             erreur = true;
17.             break;
18.         }
19.     }
20.     // si pas d'erreur, on passe à la vue suivante
21.     if (!erreur) {
22.         Vue agendaVue = (Vue) factory.getObject(Factory.AGENDA_VUE, null, "AgendaVue");
23.         mainActivity.showVue(agendaVue);
24.     }
25. }

```

- ligne 2 : la méthode exécutée à la fin de la tâche ;

- ligne 4 : fin de l'attente visuelle ;
- ligne 7 : on scanne la liste des résultats. On n'en attend qu'un donc on peut se passer de la boucle ;
- lignes 9-11 : si le résultat est l'agenda attendu, il est mémorisé dans la session ;
- lignes 12-14 : si le résultat est une exception, celle-ci est affichée ;
- lignes 21-24 : s'il n'y pas eu d'erreur, on affiche la vue suivante [AgendaVue].

9.9.11 La tâche [GetAgendaMedecinTask]



La tâche rend l'agenda du médecin. Son code est le suivant :

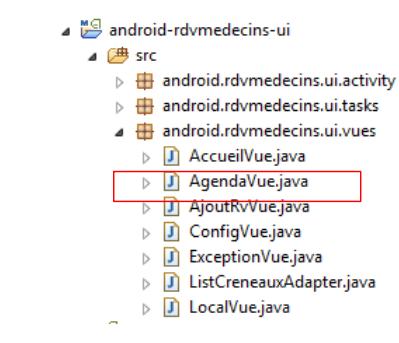
```

1. 1. @Override
2. 2.     // on exécute la tâche (idMedecin, jourRv)
3. 3.     protected Void doInBackground(Object... params) {
4. 4.         // on demande une référence sur la couche [métier]
5. 5.         IMetier metier = (IMetier) factory.getObject(Factory.METIER, (Object[])null);
6. 6.         try {
7. 7.             // on demande l'agenda du médecin pour le jour indiqué
8. 8.             Long idMedecin = (Long) params[0];
9. 9.             String jourRv = (String) params[1];
10. 10.            AgendaMedecinJour agenda = metier.getAgendaMedecinJour(idMedecin, jourRv);
11. 11.            // on mémorise l'agenda
12. 12.            info = agenda;
13. 13.        } catch (Exception ex) {
14. 14.            // on mémorise l'exception
15. 15.            info = ex;
16. 16.        }
17. 17.        // fin
18. 18.        return null;
19. 19.    }

```

- ligne 3 : la vue a passé les paramètres (N° du médecin, Jour du rendez-vous) ;
- ligne 12 : l'information rendue est de type [AgendaMedecinJour].

9.9.12 La vue [AgendaVue] - 1



C'est la vue suivante :



Nº	Id	Type	Rôle
1	lstCreneaux	ListView	liste des créneaux horaires des médecins
2	txtTitre2	TextView	une ligne d'information
3		TextView	lien pour ajouter un rendez-vous
4		TextView	lien pour supprimer un rendez-vous

Le code de la vue est le suivant :

```
1. public class AgendaVue extends LocalVue {  
2.  
3.     // les éléments de l'interface visuelle  
4.     private TextView txtTitre2;  
5.     private ListView lstCreneaux;  
6.     private Button btnAnnuler;  
7.  
8.     // l'agenda  
9.     private AgendaMedecinJour agenda;  
10.  
11.    // les résultats des tâches lancées par la vue  
12.    private List<Object> results = new ArrayList<Object>();  
13.  
14.    // mémoires  
15.    private int firstPosition;  
16.    private int top;  
17. ....  
18.    @Override  
19.    public void onResume() {  
20.        // parent  
21.        super.onResume();  
22.        // on récupère l'agenda en session  
23.        agenda = (AgendaMedecinJour) session.get("agenda");  
24.        // on génère le titre de la page  
25.        Medecin medecin = agenda.getMedecin();  
26.        String text = String.format("Rendez-vous de %s %s %s le %s", medecin.getTitre(),  
               medecin.getPrenom(),  
               medecin.getNom(), new SimpleDateFormat("dd-MM-yyyy",  
               Locale.FRANCE).format(agenda.getJour()));  
27.        txtTitre2.setText(text);  
28.    }
```

```

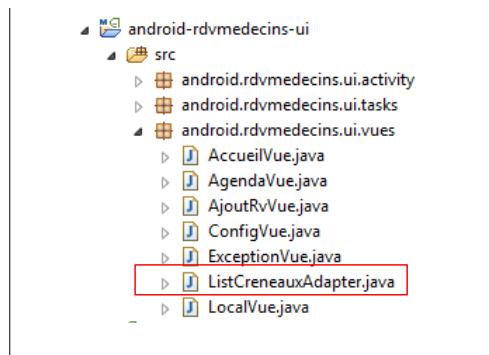
29.    // on génère la liste des créneaux
30.    ArrayAdapter<CreneauMedecinJour> adapter = new ListCreneauxAdapter(mainActivity,
   R.layout.creneau_medecin,
31.        agenda.getCreneauxMedecinJour(), this);
32.    lstCreneaux.setAdapter(adapter);
33.    // on se positionne au bon endroit du ListView
34.    lstCreneaux.setSelectionFromTop(firstPosition, top);
35.    // on gère les liens de navigation
36.    activateLinks(2);
37.    // état du bouton Annuler
38.    btnAnnuler.setVisibility(View.INVISIBLE);
39. }
40. ...

```

- ligne 19 : méthode exécutée juste avant l'affichage de la vue ;
- ligne 23 : on récupère l'agenda du médecin en session ;
- lignes 25-28 : on génère la ligne d'information [2] ;
- lignes 30-32 : on initialise le ListView [1] via un adaptateur propriétaire [ListCreneauxAdapter] auquel on passe les paramètres suivants :
 1. l'activité Android en cours,
 2. le fichier XML définissant le contenu de chaque élément du [ListView],
 3. le tableau des créneaux horaires du médecin,
 4. la vue elle-même ;
- ligne 34 : on positionne le [ListView] au bon endroit. Lorsqu'on supprime un rendez-vous, le [ListView] est régénéré totalement à partir de la base de données afin de prendre en compte les éventuelles modifications apportées en base par d'autres utilisateurs. On veut alors réafficher le [ListView] dans la position où il était lorsque l'utilisateur a cliqué le lien [Supprimer]. Nous expliquerons comment faire ;
- ligne 36 : les deux premiers liens du bandeau gauche sont activés permettant à l'utilisateur de revenir à l'une des deux premières vues de l'assistant.

Pour comprendre le reste de la vue, il nous faut d'abord expliquer le fonctionnement du [ListView] [1] ;

9.9.13 L'adaptateur [ListCreneauxAdapter]



La classe [ListCreneauxAdapter] sert à définir une ligne du [ListView] :



On voit ci-dessus, que selon le créneau a un rendez-vous ou non, l'affichage n'est pas le même. Le code de la classe [ListCreneauxAdapter] est le suivant :

```

1. package android.rdvmedecins.ui.vues;
2.
3. ...
4.
5. public class ListCreneauxAdapter extends ArrayAdapter<CreneauMedecinJour> {
6.
7.     // le tableau des arduinos
8.     private CreneauMedecinJour[] creneauxMedecinJour;
9.     // le contexte d'exécution
10.    private Context context;
11.    // l'id du layout d'affichage d'une ligne de la liste des créneaux
12.    private int layoutResourceId;
13.    // listener des clics
14.    private AgendaVue vue;
15.
16.    // constructeur
17.    public ListCreneauxAdapter(Context context, int layoutResourceId, CreneauMedecinJour[]
    creneauxMedecinJour,
        AgendaVue vue) {
18.        super(context, layoutResourceId, creneauxMedecinJour);
19.        // on mémorise les infos
20.        this.creneauxMedecinJour = creneauxMedecinJour;
21.        this.context = context;
22.        this.layoutResourceId = layoutResourceId;
23.        this.vue = vue;
24.        // on trie le tableau des créneaux dans l'ordre horaire
25.        Arrays.sort(creneauxMedecinJour, new MyComparator());
26.    }
27.
28.
29.    @Override
30.    public View getView(final int position, View convertView, ViewGroup parent) {
31.        ...
32.    }
33.
34.    // tri du tableau des créneaux
35.    class MyComparator implements Comparator<CreneauMedecinJour> {
36.        ...
37.    }
38.}
```

- ligne 5 : la classe [ListCreneauxAdapter] doit étendre un adaptateur prédefini pour les [ListView], ici la classe [ArrayAdapter] qui comme son nom l'indique alimente le [ListView] avec un tableau d'objets, ici de type [CreneauMedecinJour]. Rappelons le code de cette entité :

```

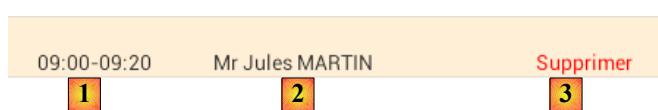
1. public class CreneauMedecinJour implements Serializable {
2.
3.     private static final long serialVersionUID = 1L;
4.     // champs
5.     private Creneau creneau;
6.     private Rv rv;
7. ...
8. }
```

- la classe [CreneauMedecinJour] contient un créneau horaire (ligne 5) et un éventuel rendez-vous (ligne 6) ou *null* si pas de rendez-vous ;

Retour au code de la classe [ListCreneauxAdapter] :

- ligne 17 : le constructeur reçoit quatre paramètres :
 - l'activité Android en cours,
 - le fichier XML définissant le contenu de chaque élément du [ListView],
 - le tableau des créneaux horaires du médecin,
 - la vue elle-même ;
- ligne 26 : le tableau des créneaux horaires est trié dans l'ordre croissant des horaires ;

La méthode [getView] est chargée de générer la vue correspondant à une ligne du [ListView]. Celle-ci comprend trois éléments :



Nº	Id	Type	Rôle
1	txtCreneau	TextView	créneau horaire
2	txtClient	TextView	le client
3	btnValider	TextView	lien pour ajouter / supprimer un rendez-vous

Le code de la méthode [getView] est le suivant :

```

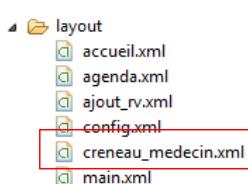
1. @Override
2.     public View getView(final int position, View convertView, ViewGroup parent) {
3.         // on se positionne sur le bon créneau
4.         CreneauMedecinJour creneauMedecin = creneauxMedecinJour[position];
5.         // on crée la ligne
6.         View row = ((Activity) context).getLayoutInflater().inflate(layoutResourceId,
    parent, false);
7.         // le créneau horaire
8.         TextView txtCreneau = (TextView) row.findViewById(R.id.txt_Creneau);
9.         txtCreneau.setText(String.format("%02d:%02d-%02d:%02d",
    creneauMedecin.getCreneau().getHdebut(), creneauMedecin
10.            .getCreneau().getMdebut(), creneauMedecin.getCreneau().getHfin(),
    creneauMedecin.getCreneau().getMfin()));
11.        // le client
12.        TextView txtClient = (TextView) row.findViewById(R.id.txt_Client);
13.        String text;
14.        if (creneauMedecin.getRv() != null) {
15.            Client client = creneauMedecin.getRv().getClient();
16.            text = String.format("%s %s %s", client.getTitre(), client.getPrenom(),
    client.getNom());
17.        } else {
18.            text = "";
```

```

19.    }
20.    txtClient.setText(text);
21.    // le lien
22.    final TextView btnValider = (TextView) row.findViewById(R.id.btn_Valider);
23.    if (creneauMedecin.getRv() == null) {
24.        // ajouter
25.        btnValider.setText(R.string.btn_ajouter);
26.        btnValider.setTextColor(context.getResources().getColor(R.color.blue));
27.    } else {
28.        // supprimer
29.        btnValider.setText(R.string.btn_supprimer);
30.        btnValider.setTextColor(context.getResources().getColor(R.color.red));
31.    }
32.    // listener du lien
33.    btnValider.setOnClickListener(new OnClickListener() {
34.
35.        @Override
36.        public void onClick(View v) {
37.            // on passe les infos à la vue de l'agenda
38.            vue.doValider(position, btnValider.getText().toString());
39.        }
40.    });
41.    // on rend la ligne
42.    return row;
43. }

```

- ligne 1 : **position** est le n° de ligne qu'on va générer dans le [ListView]. C'est également le n° du créneau dans le tableau [creneauxMedecinJour]. On ignore les deux autres paramètres ;
- ligne 4 : on récupère le créneau horaire à afficher dans la ligne du [ListView] ;
- ligne 6 : la ligne est construite à partir de sa définition XML



Le code de [creneau_medecin.xml] est le suivant :

```

1. <?xml version="1.0" encoding="utf-8"?>
2. <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:id="@+id/RelativeLayout1"
4.     android:layout_width="match_parent"
5.     android:layout_height="match_parent"
6.     android:background="@color/wheat" >
7.
8.     <TextView
9.         android:id="@+id/txt_Creneau"
10.        android:layout_width="100dp"
11.        android:layout_height="wrap_content"
12.        android:layout_marginTop="20dp"
13.        android:layout_marginLeft="20dp"
14.        android:text="@string/txt_dummy" />
15.
16.     <TextView
17.         android:id="@+id/txt_Client"
18.         android:layout_width="200dp"
19.         android:layout_height="wrap_content"
20.         android:layout_alignBaseline="@+id/txt_Creneau"
21.         android:layout_marginLeft="20dp"

```

```

22.         android:layout_toRightOf="@+id/txt_Creneau"
23.         android:text="@string/txt_dummy" />
24.
25.     <TextView
26.         android:id="@+id/btn_Validator"
27.         android:layout_width="wrap_content"
28.         android:layout_height="wrap_content"
29.         android:layout_alignBaseline="@+id/txt_Client"
30.         android:layout_marginLeft="20dp"
31.         android:layout_toRightOf="@+id/txt_Client"
32.         android:text="@string/btn_valider"
33.         android:textColor="@color/blue" />
34.
35. </RelativeLayout>

```



- lignes 8-10 : le créneau horaire [1] est construit ;
- lignes 12-20 : l'identité du client [2] est construite ;
- ligne 23 : si le créneau n'a pas de rendez-vous ;
- lignes 25-26 : on construit le lien [Ajouter] de couleur bleue ;
- lignes 29-30 : sinon on construit le lien [Annuler] de couleur rouge ;
- lignes 33-40 : quelque soit la nature lien [Ajouter / Supprimer] c'est la méthode [doValider] de la vue qui gèrera le clic sur le lien. La méthode recevra deux arguments :
 - le n° du créneau qui a été cliqué,
 - le libellé du lien qui a été cliqué ;
- ligne 42 : on rend la ligne qu'on vient de construire.

On notera que c'est la méthode [doValider] de la vue qui gère les liens. On y vient.

9.9.14 La vue [AgendaVue] - 2

La méthode [doValider] est la suivante :

```

1. // validation de la page
2. protected void doValider(int position, String texte) {
3.     // on met la position dans la session
4.     session.add("position", position);
5.     // on note la position du scroll pour y revenir
6.     // lire [http://stackoverflow.com/questions/3014089/maintain-save-restore-scroll-
    position-when-returning-to-a-listview]
7.     // position du 1er élément visible complètement ou non
8.     firstPosition = lstCreneaux.getFirstVisiblePosition();
9.     // offset Y de cet élément par rapport au haut du ListView
10.    // mesure la hauteur de la partie éventuellement cachée
11.    View v = lstCreneaux.getChildAt(0);
12.    top = (v == null) ? 0 : v.getTop();
13.    // selon le texte, on ne fait pas la même chose
14.    if (texte.equals(getResources().getString(R.string.btn_ajouter))) {
15.        doAjouter(position);
16.    } else {
17.        doSupprimer(position);
18.    }
19. }

```

- ligne 2 : on reçoit le n° du créneau dans lequel l'utilisateur a cliqué un lien ainsi que le libellé de ce lien ;
- ligne 4 : le n° du créneau est mis en session ;

- lignes 5-12 : on note des informations sur la position du créneau cliqué afin de pouvoir le réafficher ultérieurement à l'endroit où il était lorsqu'il a été cliqué. On note deux informations (firstPosition, top). Ces deux informations sont utilisées dans la méthode [onResume] :

```
// on se positionne au bon endroit du ListView
lstCreneaux.setSelectionFromTop(firstPosition, top);
```

L'explication du mécanisme mis en oeuvre est assez complexe. Il faudrait faire un dessin. En ligne 6, on trouve l'URL qui donne cette explication ;

- lignes 14-18 : selon le libellé du lien cliqué on ajoute (ligne 15) ou on supprime (ligne 17) un rendez-vous.

La méthode [doAjouter] est la suivante :

```
1.    // ajout d'un Rdv
2.    private void doAjouter(int position) {
3.        // on passe à la vue suivante
4.        Vue ajoutRvVue = (Vue) factory.getObject(Factory.AJOUT_RV_VUE, null, "AjoutRvVue");
5.        mainActivity.showVue(ajoutRvVue);
6.    }
```

- lignes 4-5 : on fait afficher la vue suivante [AjoutRvVue].

La méthode [doSupprimer] est la suivante :

```
1.    private void doSupprimer(int position) {
2.        // on nettoie les résultats
3.        results.clear();
4.        // on supprime le Rdv
5.        ITask supprimerRvTask = (ITask) factory.getObject(Factory.SUPPRIMER_RV_TASK, this,
"SupprimerRvTask");
6.        // on exécute la tâche (idRv)
7.        supprimerRvTask.doWork(agenda.getCreneauxMedecinJour()[position].getRv().getId());
8.        // début de l'attente
9.        beginWaiting();
10.       // début monitoring
11.       beginMonitoring();
12.    }
```

- lignes 5-7 : on lance une tâche asynchrone pour supprimer le rendez-vous en base. Parce qu'on a lancé une tâche asynchrone, il faut gérer ses notifications.

La méthode [notifyEvent] est la suivante :

```
1.    @Override
2.    // on reçoit des notifications de l'action
3.    public void notifyEvent(IWorker worker, int eventType, Object event) {
4.        // on passe l'info au parent
5.        super.notifyEvent(worker, eventType, event);
6.        // on traite le cas WORK_INFO
7.        if (eventType == IBoss.WORK_INFO) {
8.            results.add(event);
9.        }
10.    }
```

- lignes 7-9 : on gère la notification [WORK_INFO]. On se contente de mettre dans une liste les informations reçues. Ici, il n'y en aura qu'une, la chaîne "OK" ou une exception.

La méthode [notifyEndOfTasks] est appelée lorsque la tâche est terminée ;

```
1.    @Override
2.    public void notifyEndOfTasks() {
3.        // fin de l'attente
4.        cancelWaiting();
```

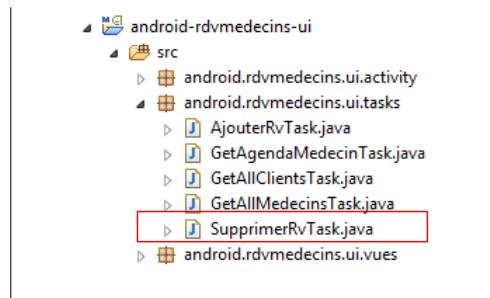
```

5.      // on gère l'unique résultat
6.      Object result = results.get(0);
7.      // on traite les résultats des tâches exécutées
8.      if (result instanceof String) {
9.          // on nettoie la liste des résultats
10.         results.clear();
11.         // la suppression s'est bien passée - on régénère l'agenda
12.         ITask getAgendaMedecinTask = (ITask)
factory.getObject(Factory.GETAGENDAMEDECIN_TASK, this,
13.                 "AgendaMedecinJourTask");
14.         // on exécute la tâche (idMedecin, jourRv)
15.         getAgendaMedecinTask.doWork(agenda.getMedecin().getId(),
16.             new SimpleDateFormat("dd-MM-yyyy",
Locale.FRANCE).format(agenda.getJour())));
17.         // début de l'attente
18.         beginWaiting();
19.         // début monitoring
20.         beginMonitoring();
21.     } else {
22.         if (result instanceof AgendaMedecinJour) {
23.             // on met l'agenda dans la session
24.             session.add("agenda", result);
25.             // on régénère la liste des créneaux
26.             AgendaMedecinJour agenda = (AgendaMedecinJour) result;
27.             ArrayAdapter<CreneauMedecinJour> adapter = new
ListCreneauxAdapter(mainActivity, R.layout.creneau_medecin,
28.                     agenda.getCreneauxMedecinJour(), this);
29.             lstCreneaux.setAdapter(adapter);
30.             // on se positionne au bon endroit du ListView
31.             lstCreneaux.setSelectionFromTop(firstPosition, top);
32.         } else if (result instanceof Exception) {
33.             // on affiche l'exception
34.             showException((Exception) result);
35.         }
36.     }
37. }

```

- ligne 6 : on récupère le résultat de la suppression, la chaîne " OK " si tout s'est bien passé, une exception sinon ;
- ligne 8 : on teste le 1er cas ;
- lignes 11-16 : si la suppression s'est bien passée, on lance une nouvelle tâche asynchrone pour régénérer l'agenda du médecin à partir de la base. On a pris soin auparavant de nettoyer la liste des résultats. La nouvelle tâche va s'exécuter et produire une information de type [AgendaMedecinJour]. Puis la méthode [notifyEndOfTasks] va être de nouveau appelée avec cette nouvelle information ;
- ligne 22 : traite cette information ;
- ligne 24 : elle est mise en session ;
- lignes 26-31 : le [ListView] est régénéré à partir de ce nouvel agenda ;
- lignes 32-34 : affichent une éventuelle exception issue de ces deux tâches.

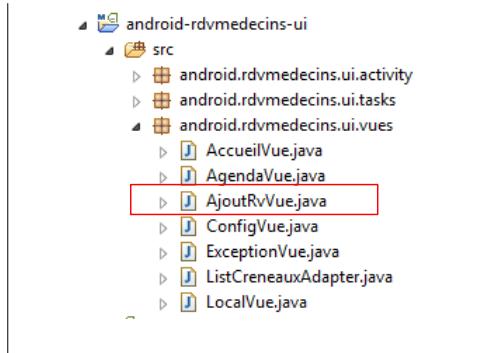
9.9.15 La tâche [SupprimerRvTask]



La tâche [SupprimerRvTask] supprime un rendez-vous de la façon suivante :

```
1. @Override
2.     // on exécute la tâche (idRv)
3.     protected Void doInBackground(Object... params) {
4.         // on récupère le paramètre, l'id du Rdv
5.         Long idRv = (Long) params[0];
6.         // on demande une référence sur la couche [métier]
7.         IMetier metier = (IMetier) factory.getObject(Factory.METIER, (Object[]) null);
8.         try {
9.             // on supprime le RV
10.            metier.supprimerRv(idRv);
11.            // on mémorise l'info
12.            info = "OK";
13.        } catch (Exception ex) {
14.            // on mémorise l'exception
15.            info = ex;
16.        }
17.    // fin
18.    return null;
19. }
```

9.9.16 La vue [AjoutRvVue]



C'est la vue suivante :



Nº	Id	Type	Rôle
1	spinnerClients	Spinner	liste déroulante des clients
2	txtTitre2	TextView	une ligne d'information

3 btnValider	Button	bouton pour valider le rendez-vous
3 btnAnnuler	Button	bouton pour annuler la demande de validation

Le code de la vue est le suivant :

```

1.  @SuppressLint("unchecked")
2.  @Override
3.  public void onResume() {
4.      // parent
5.      super.onResume();
6.      // on récupère les clients en session
7.      clients = (List<Client>) session.get("clients");
8.      // on construit le tableau affiché par le spinner
9.      String[] arrayClients = new String[clients.size()];
10.     int i = 0;
11.     for (Client client : clients) {
12.         arrayClients[i] = String.format("%s %s %s", client.getTitre(),
13.             client.getPrenom(), client.getNom());
14.         i++;
15.     }
16.     // on associe les clients au spinner
17.     ArrayAdapter<String> dataAdapter = new ArrayAdapter<String>(activity,
18.         android.R.layout.simple_spinner_item,
19.         arrayClients);
20.     dataAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
21.     spinnerClients.setAdapter(dataAdapter);
22.     // on construit le titre 2 de la page
23.     // on récupère le n° du créneau à réserver en session
24.     int position = (Integer) session.get("position");
25.     // on récupère l'agenda du médecin dans la session
26.     AgendaMedecinJour agenda = (AgendaMedecinJour) session.get("agenda");
27.     medecin = agenda.getMedecin();
28.     Creneau creneau = agenda.getCreneauxMedecinJour()[position].getCreneau();
29.     idCreneau = creneau.getId();
30.     jour = new SimpleDateFormat("dd-MM-yyyy", Locale.FRANCE).format(agenda.getJour());
31.     String titre2 = String.format(Locale.FRANCE,
32.         "Prise de rendez-vous de %s %s %s le %s pour le créneau %02d:%02d-%02d:%02d",
33.         medecin.getTitre(),
34.         medecin.getPrenom(), medecin.getNom(), jour, creneau.getHdebut(),
35.         creneau.getMdebut(), creneau.getHfin(),
36.         creneau.getMfin());
37.     txtTitre2.setText(titre2);
38.     // on gère les liens de navigation
39.     activateLinks(3);
40. }
```

- ligne 1 : la méthode [onResume] est appelée juste avant l'affichage de la vue. Elle va aller chercher en session le modèle qu'elle doit afficher ;
- ligne 7 : la liste des clients est cherchée en session ;
- lignes 8-19 : le contenu de la liste déroulante des clients est construit avec des lignes de la forme [Melle Brigitte Bistrou] ;
- lignes 20-33 : la ligne d'information [2] est construite à partir d'informations trouvées également en session ;
- ligne 35 : active les trois premiers liens du bandeau gauche. Permet à l'utilisateur de revenir sur les trois précédentes vues.

La méthode exécutée lors du clic sur le bouton [Valider] est la suivante :

```

1. // validation de la page
2. protected void doValider() {
3.     // on nettoie les résultats
4.     results.clear();
5.     // on récupère l'id du client choisi
6.     Long idClient = clients.get(spinnerClients.getSelectedItemPosition()).getId();
7.     // on ajoute le RV
```

```

8.     ITask ajouterRvTask = (ITask) factory.getObject(Factory.AJOUTER_RV_TASK, this,
    "AjouterRvTask");
9.     // on exécute la tâche (jour, idCreneau, idClient)
10.    ajouterRvTask.doWork(jour, idCreneau, idClient);
11.    // début de l'attente
12.    beginWaiting();
13.    // début monitoring
14.    beginMonitoring();
15. }

```

- lignes 5-10 : exécutent la tâche asynchrone [AjouterRvTask] qui va ajouter le rendez-vous en base.

Parce qu'elle a lancé une tâche asynchrone, la vue voit passer des notifications :

```

1.  @Override
2.  // on reçoit des notifications de l'action
3.  public void notifyEvent(IWorker worker, int eventType, Object event) {
4.      // on passe l'info au parent
5.      super.notifyEvent(worker, eventType, event);
6.      // on traite le cas WORK_INFO
7.      if (eventType == IBoss.WORK_INFO) {
8.          results.add(event);
9.      }
10. }

```

- lignes 7-9 : on se contente d'emmageriser les résultats (ici un seul) dans une liste.

La vue va être prévenue de la fin des tâches (une seule ici) :

```

1.  @Override
2.  public void notifyEndOfTasks() {
3.      // fin de l'attente
4.      cancelWaiting();
5.      // on gère le résultat unique
6.      Object result = results.get(0);
7.      if (result instanceof AgendaMedecinJour) {
8.          // on met l'agenda dans la session
9.          session.add("agenda", result);
10.         // on passe à la vue suivante
11.         Vue agendaVue = (Vue) factory.getObject(Factory.AGENDA_VUE, null, "AgendaVue");
12.         mainActivity.showVue(agendaVue);
13.     } else if (result instanceof Rv) {
14.         // on nettoie la liste des résultats
15.         results.clear();
16.         // on régénère l'agenda
17.         ITask getAgendaMedecinTask = (ITask)
    factory.getObject(Factory.GETAGENDAMEDECIN_TASK, this,
        "AgendaMedecinJourTask");
18.         // on exécute la tâche (idMedecin, jourRv)
19.         getAgendaMedecinTask.doWork(medecin.getId(), jour);
20.         // début de l'attente
21.         beginWaiting();
22.         // début monitoring
23.         beginMonitoring();
24.     } else if (result instanceof Exception) {
25.         // on affiche l'exception
26.         showException((Exception) result);
27.     }
28. }
29. }

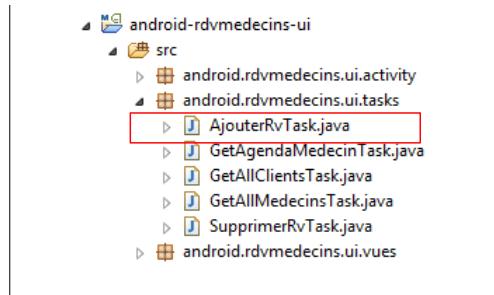
```

La tâche [AjouterRvTask] rend une information de type [Rv], le rendez-vous ajouté. Ce cas est traité ligne 13 :

- ligne 13 : résultat de la tâche [AjouterRvTask] ;

- lignes 16-20 : on lance la tâche qui régénère l'agenda du médecin à partir des informations en base. Cette tâche va rendre une information de type [AgendaMedecinJour]. Ce cas est traité ligne 7 ;
- ligne 7 : on a reçu l'agenda du médecin ;
- ligne 9 : on le met dans la session pour la vue suivante ;
- lignes 11-12 : on affiche la vue [AgendaVue] ;
- lignes 25-27 : pour le cas où l'une des deux tâches précédentes renvoie une exception.

9.9.17 La tâche [AjouterRvTask]



La tâche [AjouterRvTask] ajoute un rendez-vous de la façon suivante :

```

1.  @Override
2.      // on exécute la tâche (jour, idCreneau, idClient)
3.      protected Void doInBackground(Object... params) {
4.          // on récupère les paramètres
5.          String jour = (String) params[0];
6.          Long idCreneau = (Long) params[1];
7.          Long idClient = (Long) params[2];
8.          // on demande une référence sur la couche [métier]
9.          IMetier metier = (IMetier) factory.getObject(Factory.METIER, (Object[]) null);
10.         try {
11.             // on ajoute le RV
12.             Rv rv = metier.ajouterRv(jour, idCreneau, idClient);
13.             // on mémorise l'info
14.             info = rv;
15.         } catch (Exception ex) {
16.             // on mémorise l'exception
17.             info = ex;
18.         }
19.         // fin
20.         return null;
21.     }

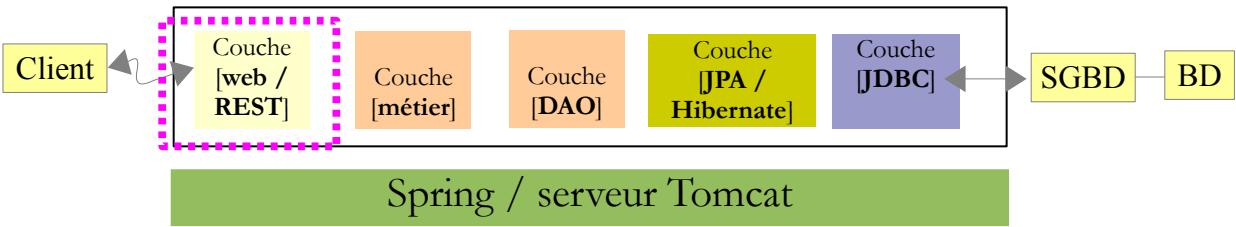
```

Ceci termine notre revue de l'exemple 7.

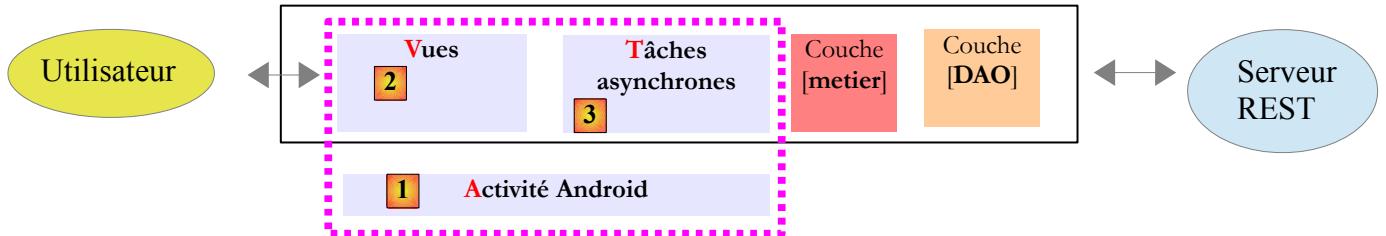
9.10 Conclusion de l'exemple 7

Rappelons l'architecture client / serveur de cet exemple :

Le serveur J2EE

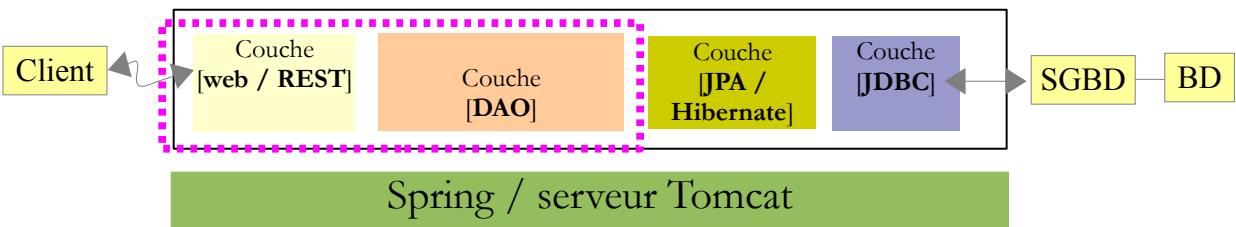


Le client Android



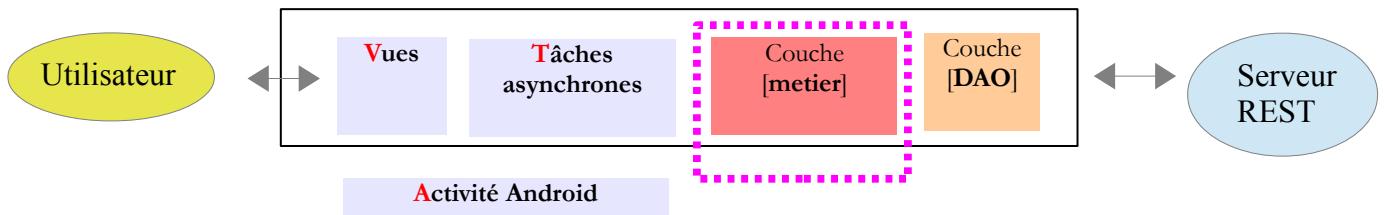
Dans le document [pfm], une application web mobile avait été construite avec les mêmes fonctionnalités que celle qui vient d'être construite. Pour l'instant l'application web mobile l'emporte sur l'application native Android. En effet, elle fonctionne sur n'importe quel smartphone ce qui n'est pas le cas de notre application Android. Pour rendre celle-ci plus attractive, on pourrait utiliser des éléments du smartphone / tablette (caméra, sonnerie, micro, ...) que l'application web mobile ne pourrait utiliser. Mais ça peut relever du gadget pour un client de gestion par exemple. Plus intéressant, on peut déporter la couche [métier] du serveur sur le client Android. L'architecture devient alors la suivante :

Le serveur J2EE



Le service REST expose désormais l'interface de la couche [DAO].

Le client Android



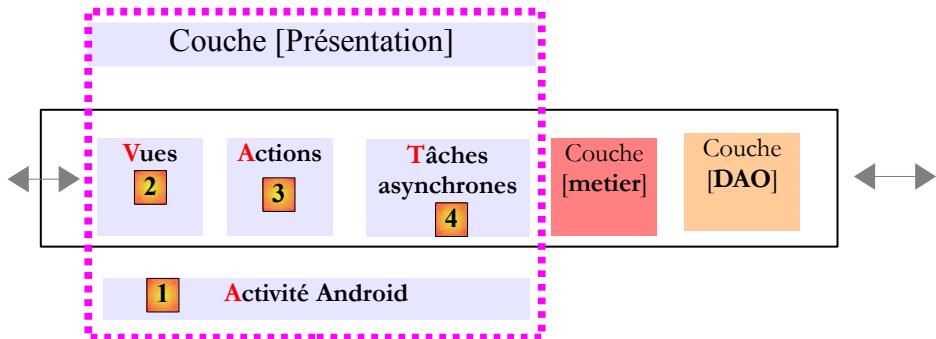
La nouvelle couche [métier] du client Android va désormais avoir deux fonctionnalités :

- s'interfacer avec le service REST de la couche [DAO] du serveur ;
- embarquer la couche [métier] du serveur.

Le métier est désormais déporté sur les clients. On allège ainsi le serveur et on gagne probablement en performances. C'est plus difficile à atteindre avec une application web mobile.

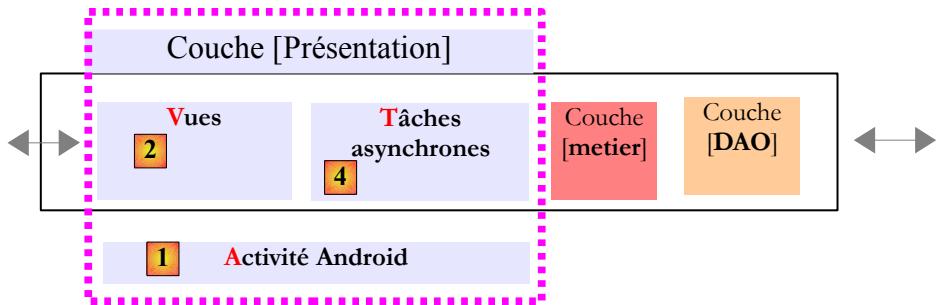
10 Conclusion générale

Nous avons introduit un modèle appelé AVAT (Activité – Vues – Actions - Tâches) pour gérer les tâches asynchrones que pouvait être amené à gérer un client Android. L'architecture la plus générale du client est alors la suivante :



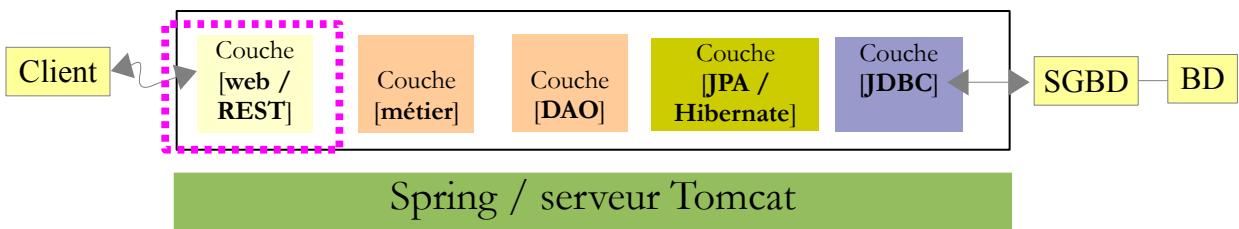
Cette architecture a été utilisée dans les exemples 1 à 6.

Puis nous avons introduit une architecture simplifiée AVT (Activité – Vues – Tâches) :



L'exemple 7 a été construit avec cette architecture simplifiée. Elle peut toujours être utilisée en lieu et place de l'architecture AVAT.

Nous avons montré deux applications client / serveur (exemples 2 et 7) où le serveur était un service REST. Le serveur de l'exemple 7 était ainsi le suivant :



Aux lecteurs désormais de valider ou non ce modèle asynchrone AVAT, de le modifier, de l'enrichir, ...

11 Annexes

11.1 La bibliothèque Jackson

La bibliothèque JSON appelée " Jackson " permet de construire :

- la chaîne JSON d'un objet : `new ObjectMapper().writeValueAsString(object)` ;
- un objet à partir d'un chaîne JSON : `new ObjectMapper().readValue(jsonString, Object.class)`.

Les deux méthodes sont susceptibles de lancer une `IOException`. Voici un exemple.

Soit la classe [Personne] suivante :

```
1. package istia.st.jsondemo;
2.
3. public class Personne {
4.     // data
5.     private String nom;
6.     private String prenom;
7.     private int age;
8.
9.     // constructeurs
10.    public Personne(){
11.
12.    }
13.
14.    public Personne(String nom, String prénom, int âge){
15.        this.nom=nom;
16.        this.prenom=prénom;
17.        this.age=âge;
18.    }
19.
20.    // signature
21.    public String toString(){
22.        return String.format("Personne[%s, %s, %d]", nom, prenom, age);
23.    }
24.
25.    // getters et setters
26.    ...
27. }
```

et la classe principale suivante :

```
1. package istia.st.json;
2.
3. import java.io.IOException;
4. import java.util.HashMap;
5. import java.util.Map;
6.
7. import org.codehaus.jackson.map.ObjectMapper;
8. import org.codehaus.jackson.type.TypeReference;
9.
10. public class App {
11.
12.     static ObjectMapper mapper = new ObjectMapper();
13.
14.     public static void main(String[] args) throws IOException {
15.         // création d'une personne
16.         Personne paul = new Personne("Denis", "Paul", 40);
17.         // affichage Json
```

```

18.     String json = mapper.writeValueAsString(paul);
19.     System.out.println("Json=" + json);
20.     // instantiation Personne à partir du Json
21.     Personne p = mapper.readValue(json, Personne.class);
22.     // affichage personne
23.     System.out.println("Personne=" + p);
24.     // un tableau
25.     Personne virginie = new Personne("Radot", "Virginie", 20);
26.     Personne[] personnes = new Personne[] { paul, virginie };
27.     // affichage Json
28.     json = mapper.writeValueAsString(personnes);
29.     System.out.println("Json personnes=" + json);
30.     // dictionnaire
31.     Map<String, Personne> hpersonnes = new HashMap<String, Personne>();
32.     hpersonnes.put("1", paul);
33.     hpersonnes.put("2", virginie);
34.     // affichage Json
35.     json = mapper.writeValueAsString(hpersonnes);
36.     System.out.println("Json hpersonnes=" + json);
37.     // relecture du Json
38.     Map<String, Personne> map = mapper.readValue(json, new TypeReference<Map<String,
Object>>() {
39.         });
40.     System.out.println("Personne=" + map.get("1"));
41. }
42. }
```

Les affichages écran sont les suivants :

```

1. Json={"prenom":"Paul","age":40,"nom":"Denis"}
2. Personne=Personne[Denis, Paul, 40]
3. Json personnes=[{"prenom":"Paul","age":40,"nom":"Denis"}, {"prenom":"Virginie","age":20,"nom":"Radot"}]
4. Json hpersonnes={"2":{"prenom":"Virginie","age":20,"nom":"Radot"},"1": {"prenom":"Paul","age":40,"nom":"Denis"}}
5. Personne={prenom=Paul, age=40, nom=Denis}
```

De l'exemple on retiendra :

- l'objet [ObjectMapper] nécessaire aux transformations JSON / *Object* : ligne 12 ;
- la transformation JSON --> [Personne] : ligne 21 ;
- la transformation [Personne] --> JSON : ligne 18 ;
- l'exception [IOException] lancée par les deux méthodes : ligne 14 ;
- ligne 38 : la classe [TypeReference] permet d'obtenir l'objet *Class* d'un type Java complexe ;

11.2 La bibliothèque Gson

La bibliothèque Gson de Google est analogue mais légèrement différente. Le programme précédent s'écrit comme suit :

```

1. package istia.st.json;
2.
3. import java.util.HashMap;
4. import java.util.Map;
5.
6. import com.google.gson.Gson;
7. import com.google.gson.reflect.TypeToken;
8.
9. public class App {
10.
11.     static Gson gson = new Gson();
12.
13.     public static void main(String[] args){
```

```

14. // création d'une personne
15. Personne paul = new Personne("Denis", "Paul", 40);
16. // affichage Json
17. String json = gson.toJson(paul);
18. System.out.println("Json=" + json);
19. // instantiation Personne à partir du Json
20. Personne p = gson.fromJson(json, Personne.class);
21. // affichage personne
22. System.out.println("Personne=" + p);
23. // un tableau
24. Personne virginie=new Personne("Radot", "Virginie", 20);
25. Personne[] personnes=new Personne[]{paul,virginie};
26. // affichage Json
27. json=gson.toJson(personnes);
28. System.out.println("Json personnes="+json);
29. // dictionnaire
30. Map<String,Personne> hpersonnes=new HashMap<String,Personne>();
31. hpersonnes.put("1",paul);
32. hpersonnes.put("2",virginie);
33. // affichage Json
34. json=gson.toJson(hpersonnes);
35. System.out.println("Json hpersonnes="+json);
36. // relecture du Json
37. Map<String, Personne> map = gson.fromJson(json, new TypeToken<Map<String, Personne>>()
{
38.     }.getType());
39.     System.out.println("Personne=" +map.get("1"));
40. }
41. }
```

- les exceptions lancées par la bibliothèque Gson ne sont pas contrôlées par le compilateur. On n'a donc pas de *try / catch* ou de *throws Exception* ;
- ligne 17 : la méthode *toJson* permet d'obtenir la représentation JSON d'un objet ;
- ligne 20 : la méthode *fromJson* permet de créer un objet à partir de sa représentation JSON ;
- ligne 37 : on notera la classe *TypeToken* qui permet d'obtenir l'objet *Class* d'un type Java complexe.

L'affichage écran obtenu est le suivant :

```

1. Json={"nom":"Denis","prenom":"Paul","age":40}
2. Personne=Personne[Denis, Paul, 40]
3. Json personnes=[{"nom":"Denis","prenom":"Paul","age":40},
   {"nom":"Radot","prenom":"Virginie","age":20}]
4. Json hpersonnes={"2":{"nom":"Radot","prenom":"Virginie","age":20}, "1":
   {"nom":"Denis","prenom":"Paul","age":40}}
5. Personne=Personne[Denis, Paul, 40]
```

11.3 Installation de l'IDE Eclipse

Plutôt qu'installer un Eclipse brut, nous allons installer **SpringSource Tool Suite** [<http://www.springsource.com/developer/sts>], un Eclipse pré-équipé avec de nombreux plugins liés au framework Spring et également avec une configuration Maven pré-installée.

1

The screenshot shows the official website for SpringSource Tool Suite (STS). The header includes the URL "www.springsource.com/developer/sts", a search bar, and navigation links for "Les plus visités", "Débuter avec Firefox", and "À la une". A sidebar on the left lists "Developers Tools & Communities" such as Spring, Groovy & Grails, SpringSource Tool Suite (highlighted with a red box), tc Server Developer Edition, and WaveMaker. The main content area features a heading "SpringSource Tool Suite — The Best Development Tool for Enterprise Java", a description of the tool's capabilities, and a logo for "springsource TOOLSUITE". A testimonial from Charlie Babcock of InformationWeek is displayed, followed by a "Download STS" button and a "Contact Request: Support" link.

- aller sur le site de **SpringSource Tool Suite** (STS) [1], pour télécharger la version courante de STS [2A] [2B],

2B

The screenshot shows the download page for STS 2.9.2. It offers two download paths: "Windows" leading to "springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32-installer.exe" [2B] and "Windows" leading to "springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32.zip".

3A

The screenshot shows the contents of the "sts-2.9.2.RELEASE" directory. It includes subfolders like "configuration", "dropins", "features", "p2", "plugins", "readme", ".eclipseproduct", "artifacts.xml", "epl-v10.html", "notice.html", "SPRINGSOURCE-EULA.txt", and files "STS.exe" [4] and "STS.ini".

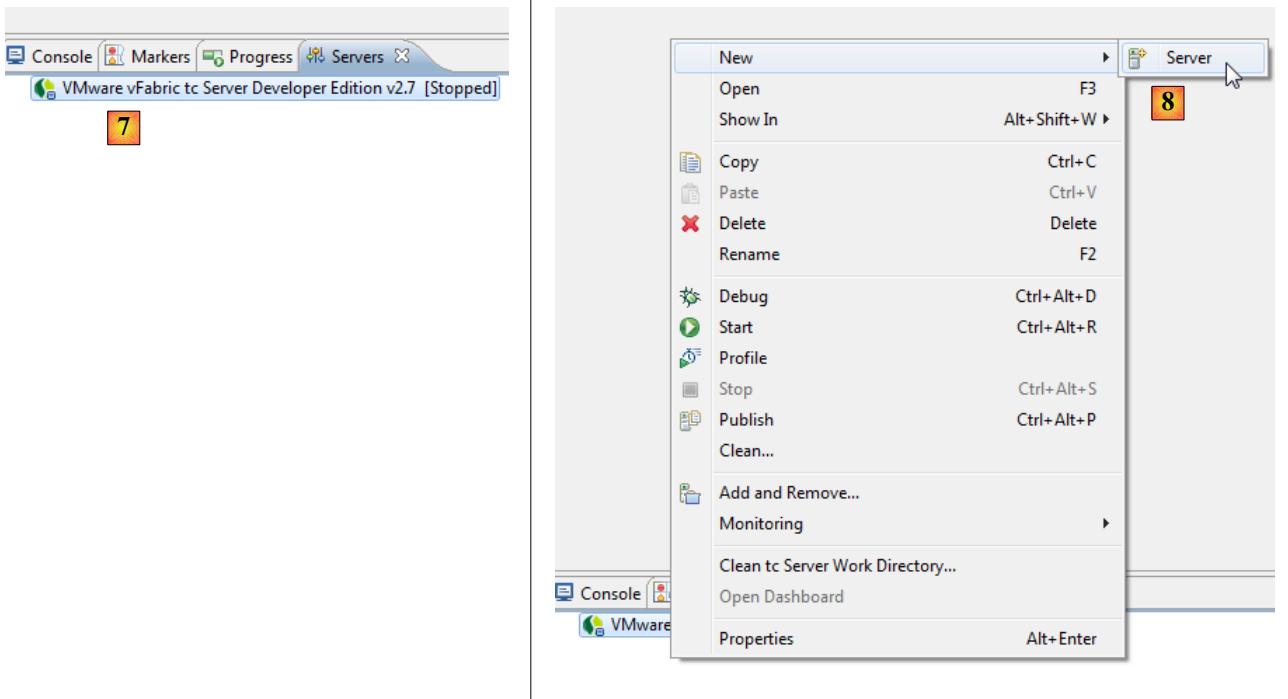
5

The screenshot shows the Eclipse IDE interface with the title "Java EE - Eclipse". The "Project Explorer" view is open, showing a single folder icon [5].

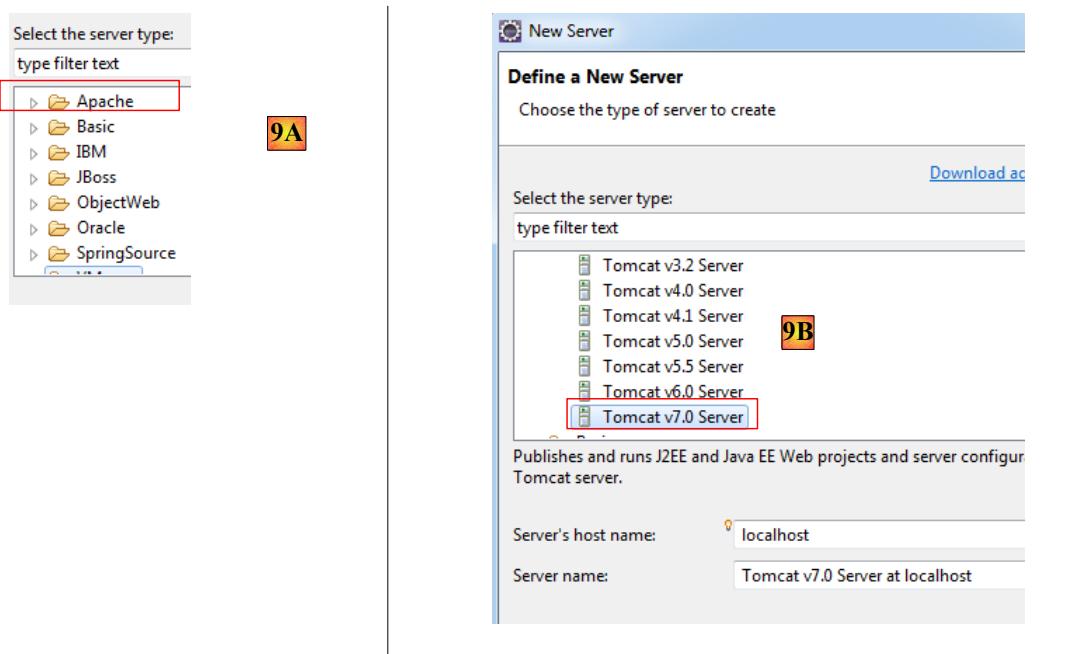
6

The screenshot shows the "Window" menu in the Eclipse IDE. The "Servers" option [6] is highlighted with a red box.

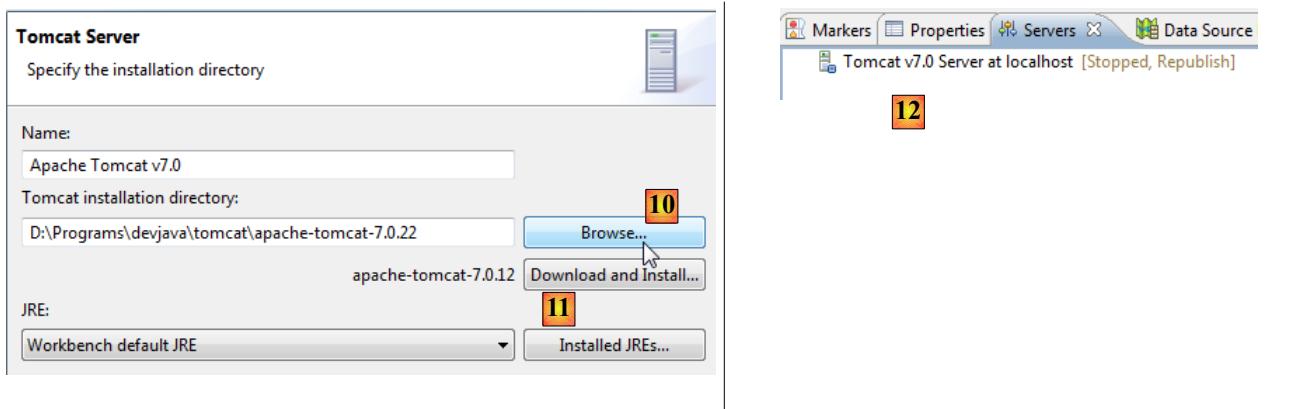
- le fichier téléchargé est un installateur qui crée l'arborescence de fichiers [3A] [3B]. En [4], on lance l'exécutable,
- en [5], la fenêtre de travail de l'IDE après avoir fermé la fenêtre de bienvenue. En [6], on fait afficher la fenêtre des serveurs d'applications,



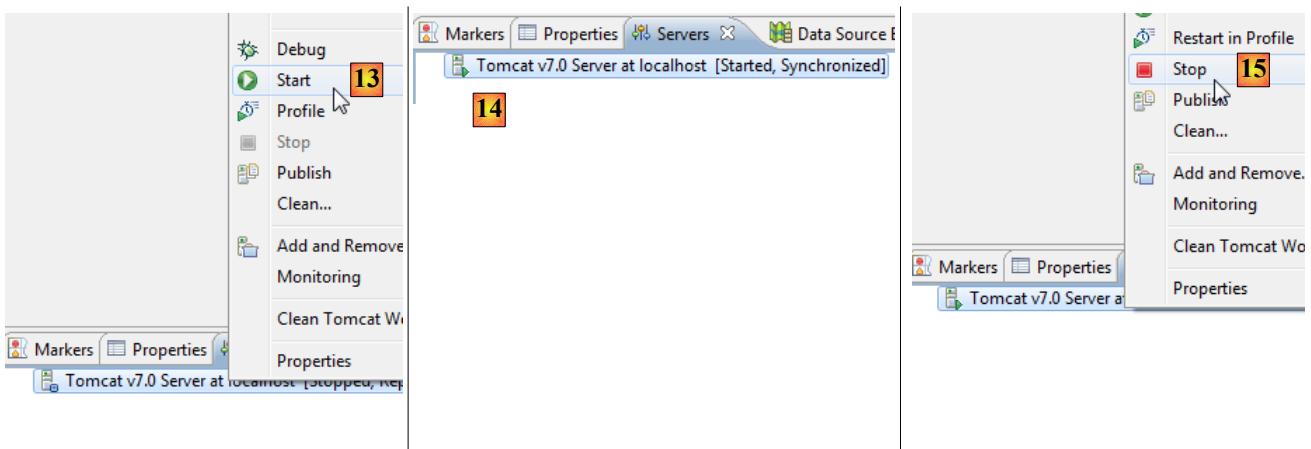
- en [7], la fenêtre des serveurs. Un serveur est enregistré. C'est un serveur VMware que nous n'utiliserons pas. En [8], on appelle l'assistant d'ajout d'un nouveau serveur,



- en [9A], divers serveurs nous sont proposés. Nous choisissons d'installer un serveur Tomcat 7 d'Apache [9B],



- en [10], nous désignons le dossier d'installation du serveur Tomcat 7 installé. Si on n'a pas de serveur Tomcat, utiliser le bouton [11],
- en [12], le serveur Tomcat apparaît dans la fenêtre [Servers],



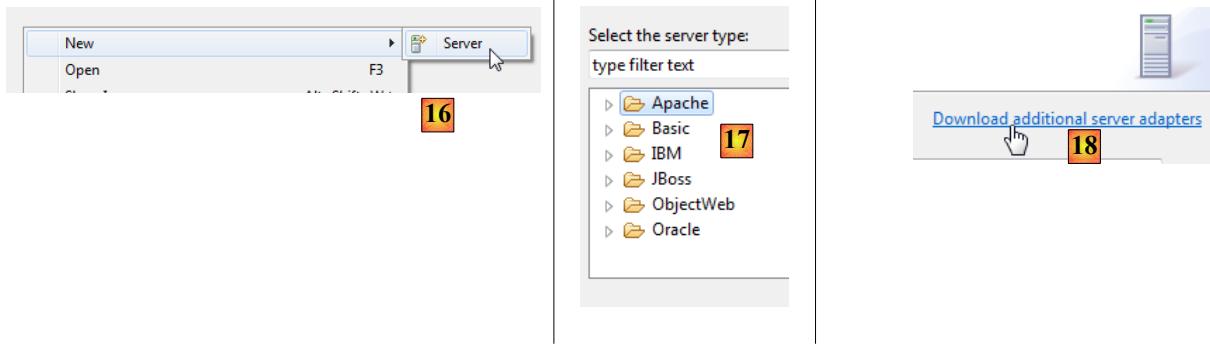
- en [13], nous lançons le serveur,
- en [14], il est lancé,
- en [15], on l'arrête.

Dans la fenêtre [Console], on obtient les logs suivants de Tomcat si tout va bien :

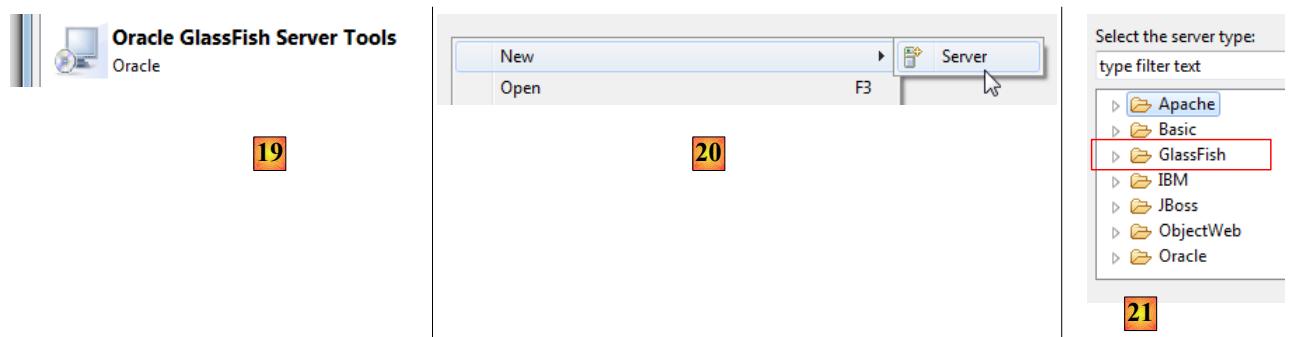
```

1. mai 26, 2012 8:56:51 AM org.apache.catalina.core.AprLifecycleListener init
2. Infos: The APR based Apache Tomcat Native library which allows optimal performance in production
environments was not found on the java.library.path: ...
3. mai 26, 2012 8:56:55 AM org.apache.coyote.AbstractProtocol init
4. Infos: Initializing ProtocolHandler ["http-bio-8080"]
5. mai 26, 2012 8:56:55 AM org.apache.coyote.AbstractProtocol init
6. Infos: Initializing ProtocolHandler ["ajp-bio-8009"]
7. mai 26, 2012 8:56:55 AM org.apache.catalina.startup.Catalina load
8. Infos: Initialization processed in 4527 ms
9. mai 26, 2012 8:56:55 AM org.apache.catalina.core.StandardService startInternal
10. Infos: Démarrage du service Catalina
11. mai 26, 2012 8:56:55 AM org.apache.catalina.core.StandardEngine startInternal
12. Infos: Starting Servlet Engine: Apache Tomcat/7.0.22
13. mai 26, 2012 8:56:57 AM org.apache.catalina.util.SessionIdGenerator createSecureRandom
14. Infos: Creation of SecureRandom instance for session ID generation using [SHA1PRNG] took [218]
milliseconds.
15. mai 26, 2012 8:56:57 AM org.apache.coyote.AbstractProtocol start
16. Infos: Starting ProtocolHandler ["http-bio-8080"]
17. mai 26, 2012 8:56:57 AM org.apache.coyote.AbstractProtocol start
18. Infos: Starting ProtocolHandler ["ajp-bio-8009"]
19. mai 26, 2012 8:56:57 AM org.apache.catalina.startup.Catalina start
20. Infos: Server startup in 2252 ms

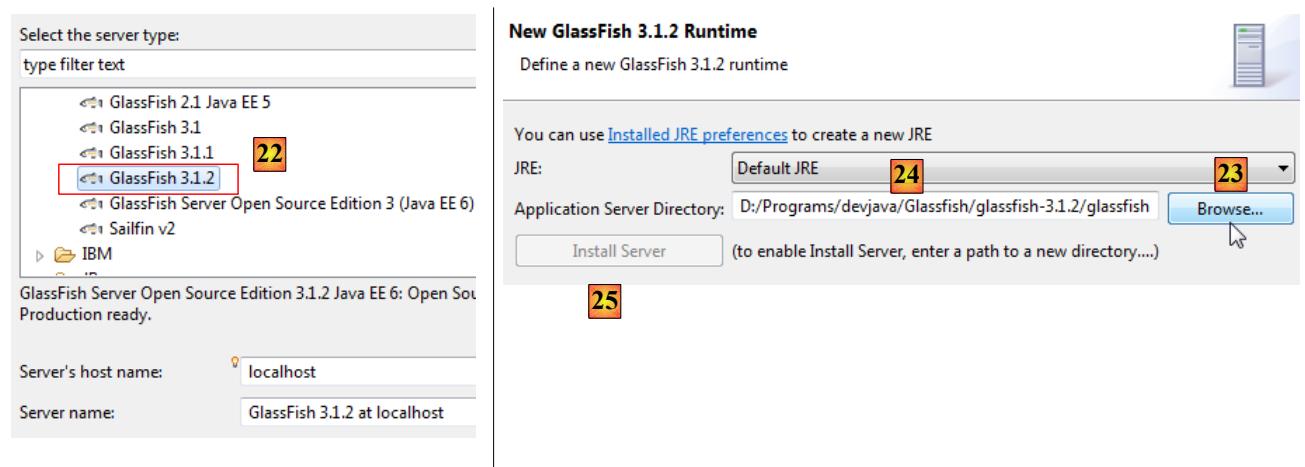
```



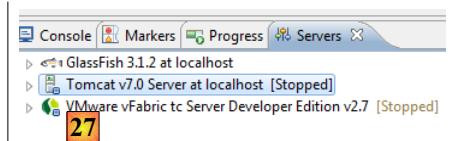
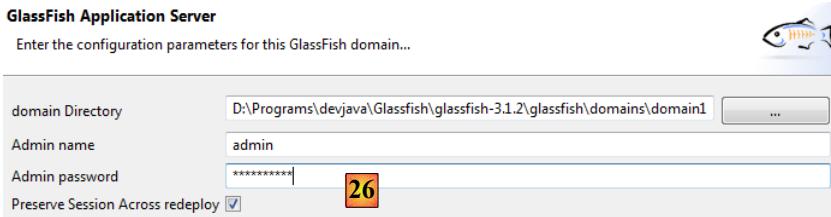
- toujours dans la fenêtre [Servers], on ajoute un nouveau serveur [16],
- en [17], le serveur Glassfish n'est pas proposé,
- dans ce cas, on utilise le lien [18],



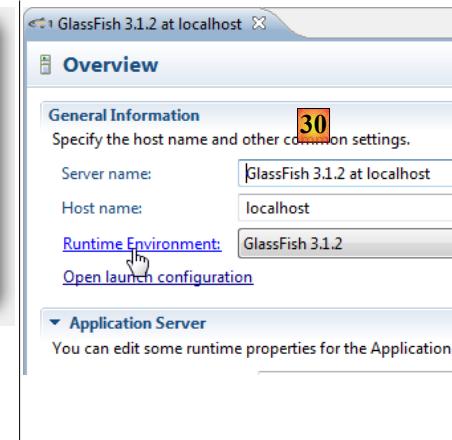
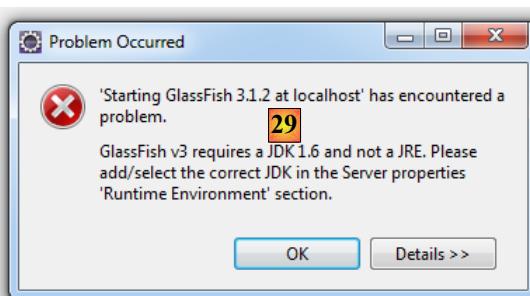
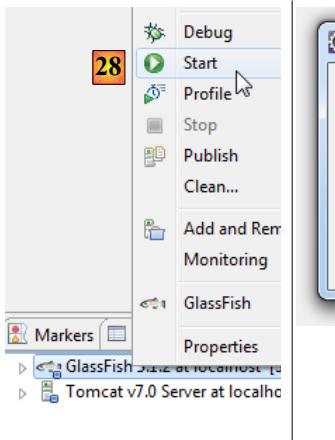
- en [19], on choisit d'ajouter un adaptateur pour le serveur Glassfish,
- celui est téléchargé et de retour dans la fenêtre [Servers], on ajoute un nouveau serveur,
- cette fois-ci en [21], les serveurs Glassfish sont proposés,



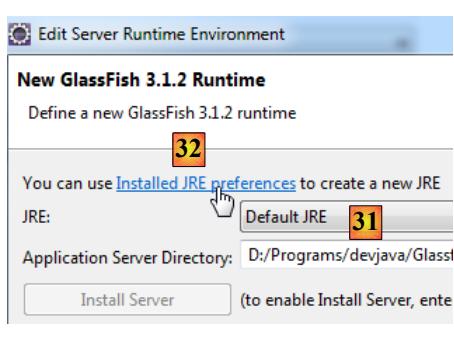
- en [22], on choisit le serveur Glassfish 3.1.2 téléchargé avec Netbeans,
- en [23], on désigne le dossier d'installation de Glassfish 3.1.2 (faire attention au chemin indiqué en [24]),
- si on n'a pas de serveur Glassfish, utiliser le bouton [25],



- en [26], l'assistant nous demande le mot de passe de l'administrateur du serveur Glassfish. Pour une première installation, c'est normalement *adminadmin*,
- lorsque l'assistant est terminé, dans la fenêtre [Servers], le serveur Glassfish apparaît [27],



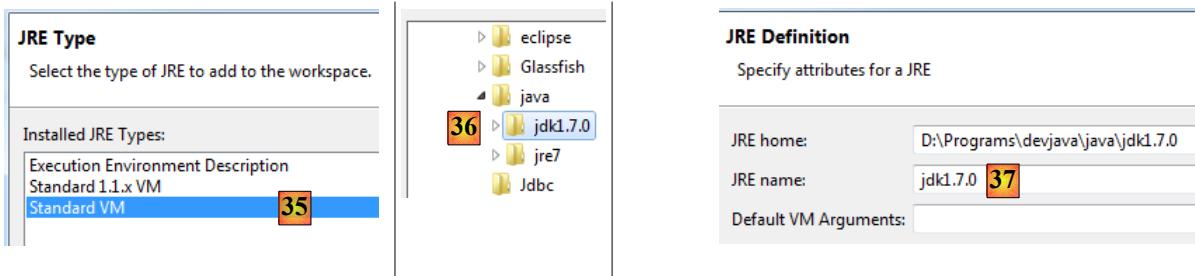
- en [28], on le lance,
- en [29] un problème peut survenir. Cela dépend des informations données à l'installation. Glassfish veut un JDK (Java Development Kit) et non pas un JRE (Java Runtime Environment),
- pour avoir accès aux propriétés du serveur Glassfish, on double-clique dessus dans la fenêtre [Servers],
- on obtient la fenêtre [20] dans laquelle on suit le lien [Runtime Environment],



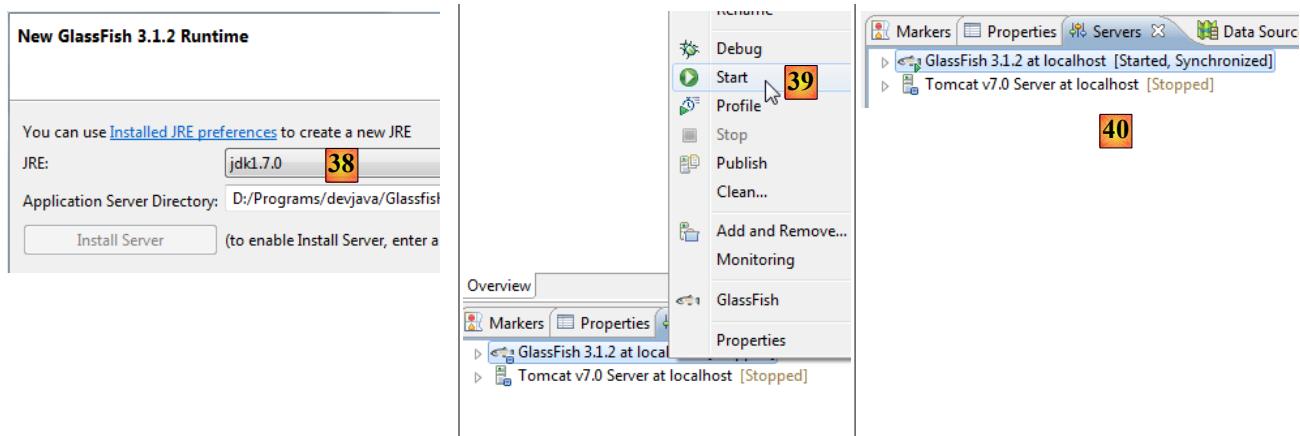
Installed JREs:			
Name	Location	Type	
jre7	D:\Programs\devjava\java\jre7	Standard VM	Add... Edit... Duplicate...

A red box labeled 33 highlights the 'jre7' entry in the table, and a red box labeled 34 highlights the 'Edit...' button.

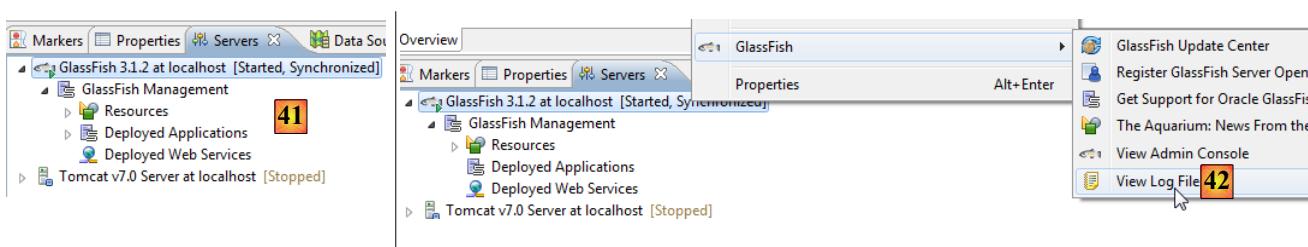
- en [31], on va remplacer le JRE utilisé par défaut par un JDK. Pour cela, on utilise le lien [32],
- en [33], les JRE installés sur la machine,
- en [34], on en ajoute un autre,



- en [35], on choisit [Standard VM] (Virtual Machine),
- en [36], on sélectionne d'un JDK (>=1.6),
- en [37], le nom donné au nouveau JRE,



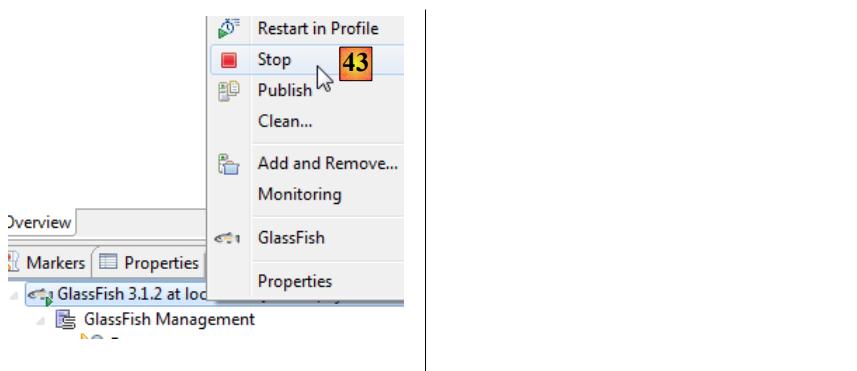
- revenu au choix du JRE pour Glassfish, nous choisissons le nouveau JRE déclaré [38],
- une fois l'assistant de configuration terminé, on relance [Glassfish] [39],
- en [40], il est lancé,



- en [41], l'arborescence du serveur,
- en [42], on accède aux logs de Glassfish :

- ...
- Infos: Running GlassFish Version: GlassFish Server Open Source Edition 3.1.2 (build 23)*
-
- Infos: Grizzly Framework 1.9.46 started in: 125ms - bound to [0.0.0.0:7676]*
- Infos: Grizzly Framework 1.9.46 started in: 125ms - bound to [0.0.0.0:3700]*
- Infos: Grizzly Framework 1.9.46 started in: 188ms - bound to [0.0.0.0:8080]*
- Infos: Grizzly Framework 1.9.46 started in: 141ms - bound to [0.0.0.0:4848]*
- Infos: Grizzly Framework 1.9.46 started in: 141ms - bound to [0.0.0.0:8181]*
- Infos: Registered org.glassfish.ha.store.adapter.cache.ShardBackingStoreProxy for persistence-type = replicated in BackingStoreFactoryRegistry*
-
- Infos: GlassFish Server Open Source Edition 3.1.2 (23) heure de démarrage : Felix (13 790 ms), services de démarrage (2 028 ms), total (15 818 ms)*
-

13. Infos: JMX005: JMXStartupService had Started JMXConnector on JMXService URL
 service:jmx:rmi://Gportpers3.ad.univ-angers.fr:8686/jndi/rmi://Gportpers3.ad.univ-angers.fr:8686/jmxrmi



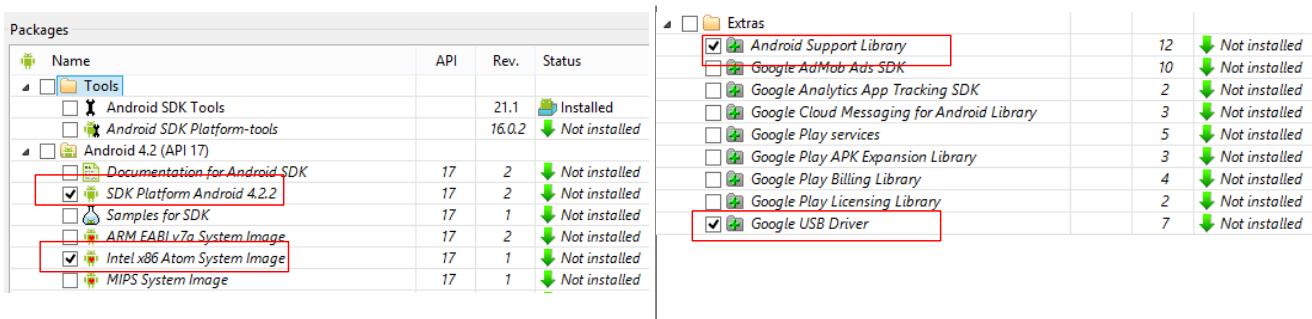
- en [43], nous arrêtons le serveur Glassfish.

11.4 Installation du SDK Manager d'Android

The screenshot shows the 'Get the Android SDK' page. On the left, there's a sidebar with 'Get the Android SDK' and 'The Android SDK provides you the API libraries and developer tools necessary to build, test, and debug apps for Android.' Below it is a red box labeled '1'. On the right, there's a section titled '^ USE AN EXISTING IDE' with a red box labeled '2'. It contains text about setting up a new SDK if you have an existing IDE, and a blue button labeled 'Download the SDK Tools for Windows' with a hand cursor icon.

- en [1], pourquoi on a besoin du SDK d'Android ;
- en [2], on est dans le cadre d'un IDE déjà installé ;

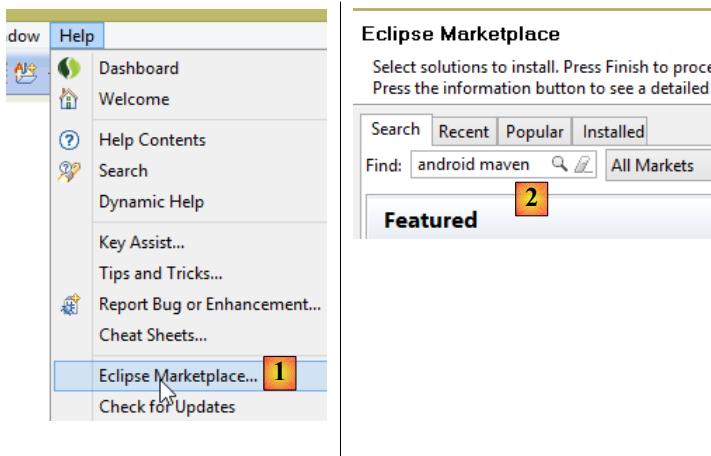
On trouvera le SDK Manager d'Android à l'adresse [<http://developer.android.com/sdk/index.html>] (mars 2013). Faire l'installation du SDK. Une fois installé, le lancer. L'assistant suivant s'affiche :



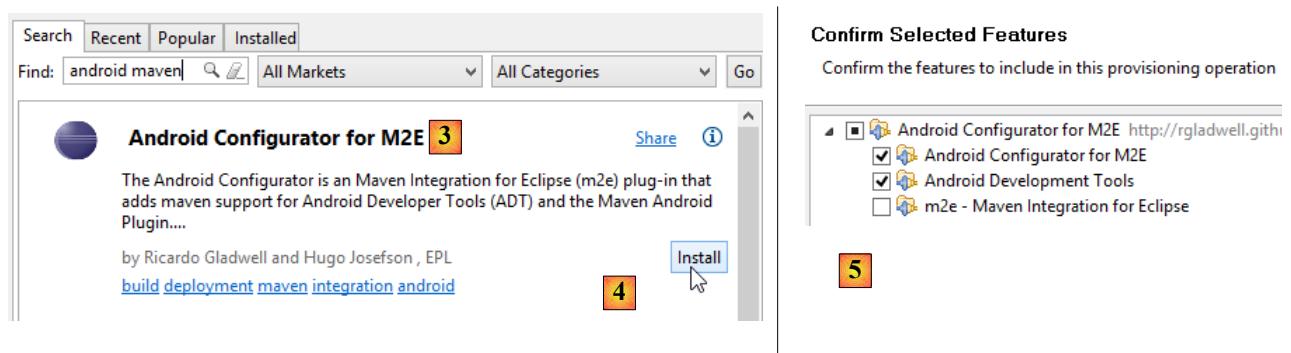
- télécharger les quatre paquetages ci-dessus ;

La suite se fait sous Eclipse.

11.5 Installation des outils Android pour Eclipse

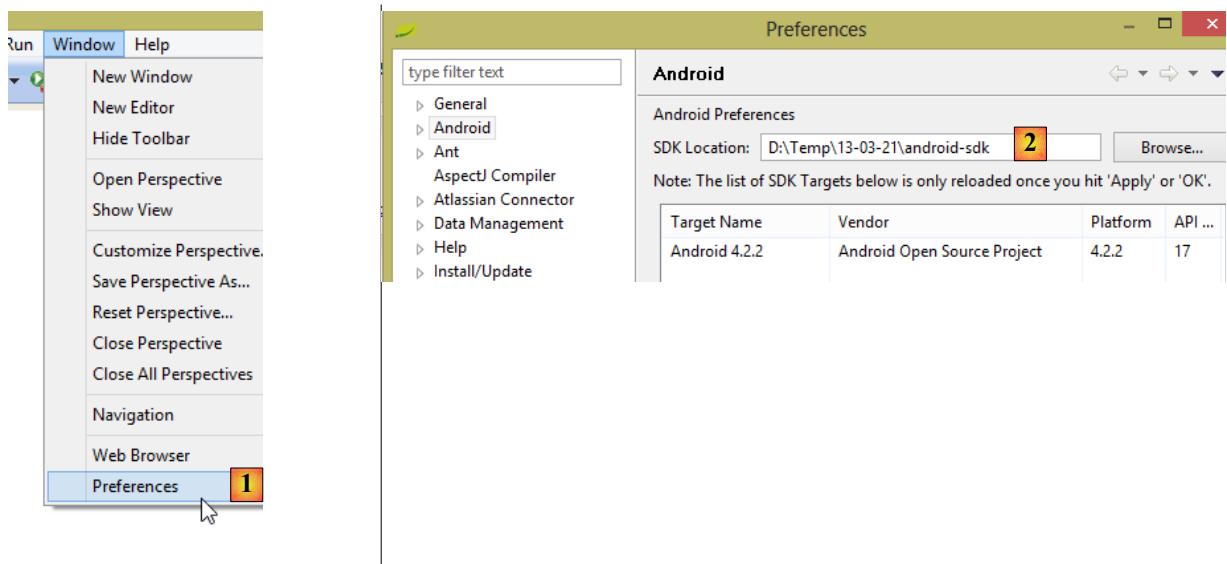


- en [1], sélectionner l'option *Help / Eclipse Marketplace* ;
- en [2], dans la zone de recherche, mettre les mots clés [android maven] pour obtenir les outils Maven pour Android ;



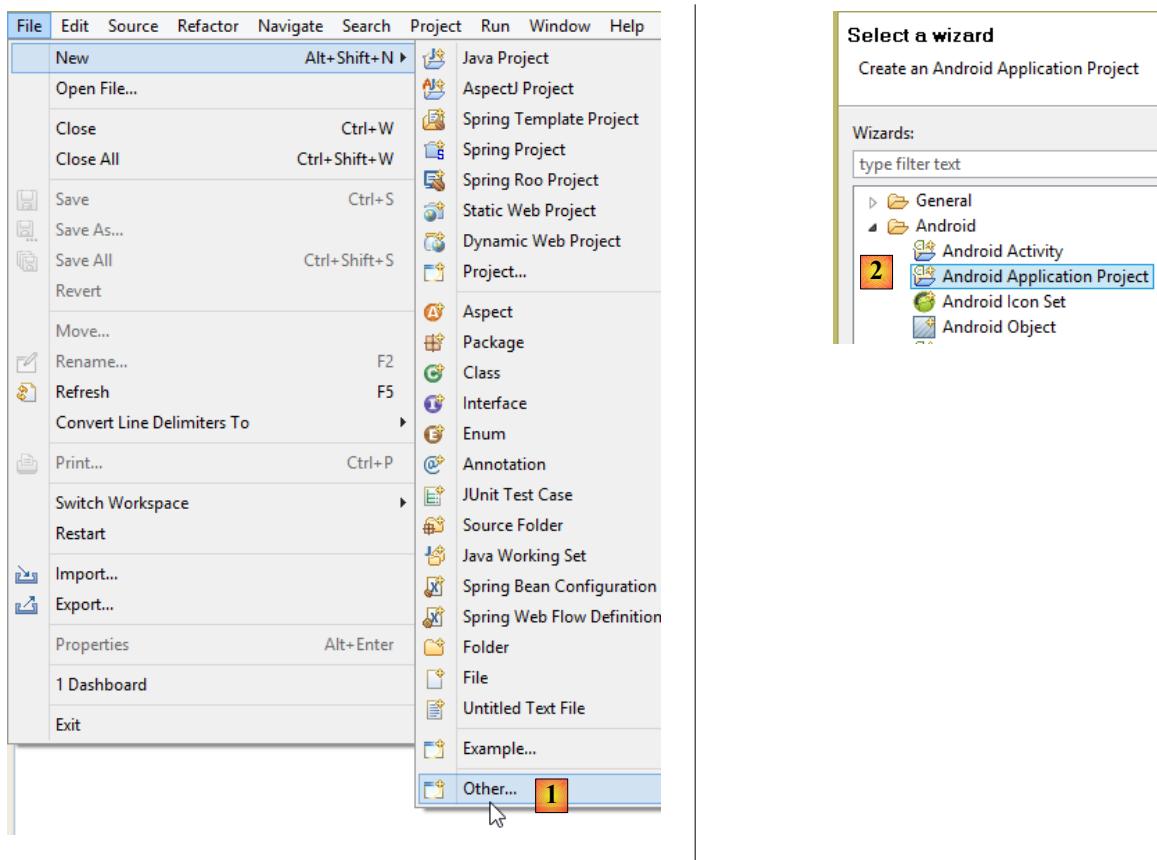
- en [3], choisir l'application [Android Configurator for M2E] et l'installer [4] ;
- en [5], désélectionner [m2e], les outils Maven pour Eclipse qui sont déjà installés puis aller jusqu'au bout de l'assistant. Redémarrer Eclipse ;

Ceci fait, on configure le plugin Android :

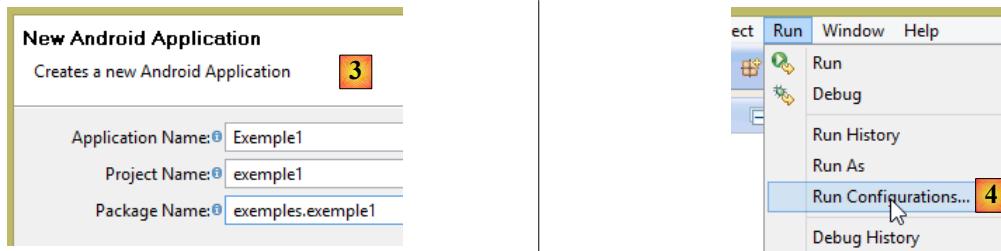


- en [1], choisir l'option *Window / Preferences* ;
- en [2], indiquez le répertoire d'installation du SDK Manager d'Android qui a été installé précédemment, puis valider ;

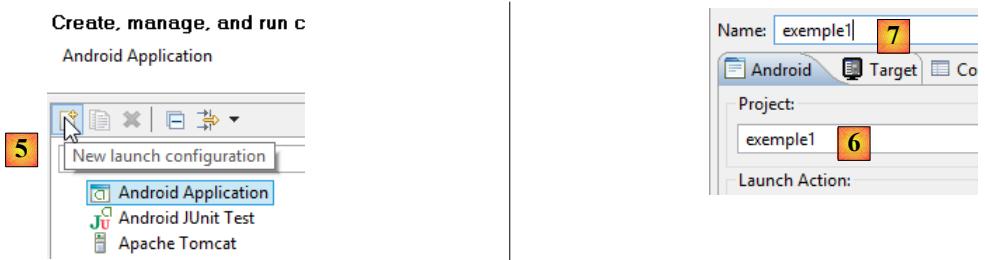
Nous allons créer un premier projet Android :



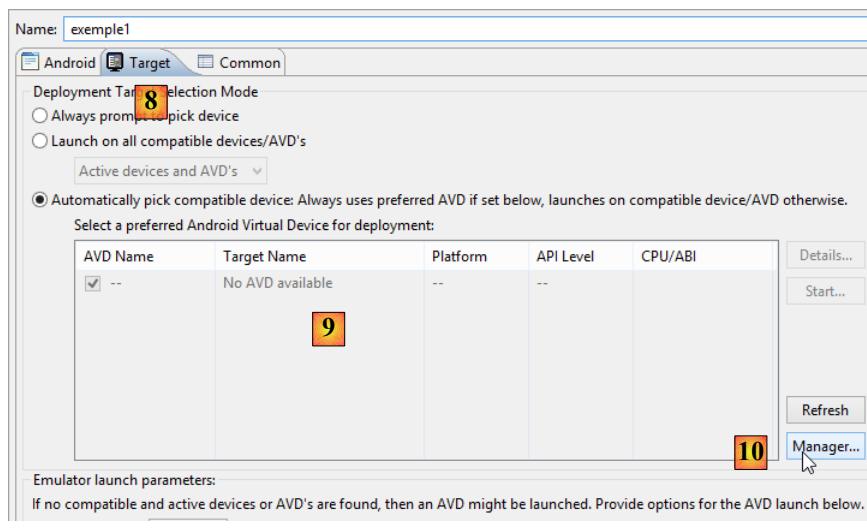
- en [1], créer un nouveau projet de type Android [2] ;



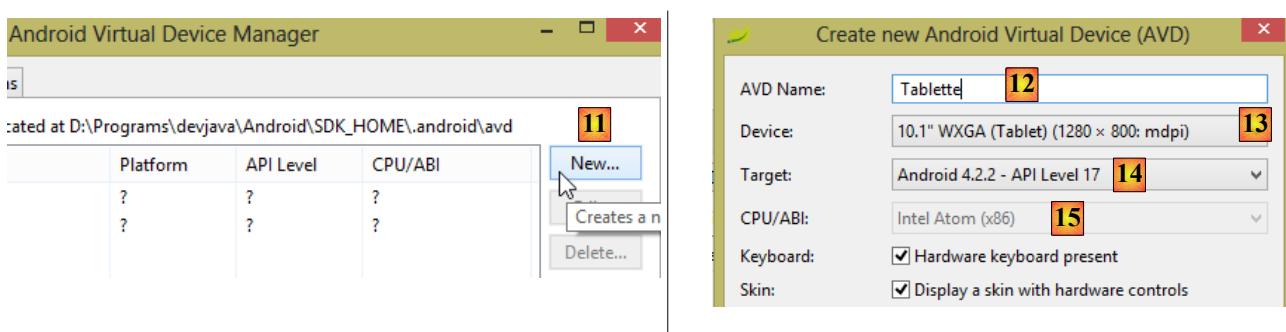
- donner les trois informations ci-dessus et aller jusqu'au bout de l'assistant en acceptant les valeurs proposées par défaut ;
- en [4], utiliser l'option *Run / Run Configurations* pour définir une configuration d'exécution pour le projet Android ;



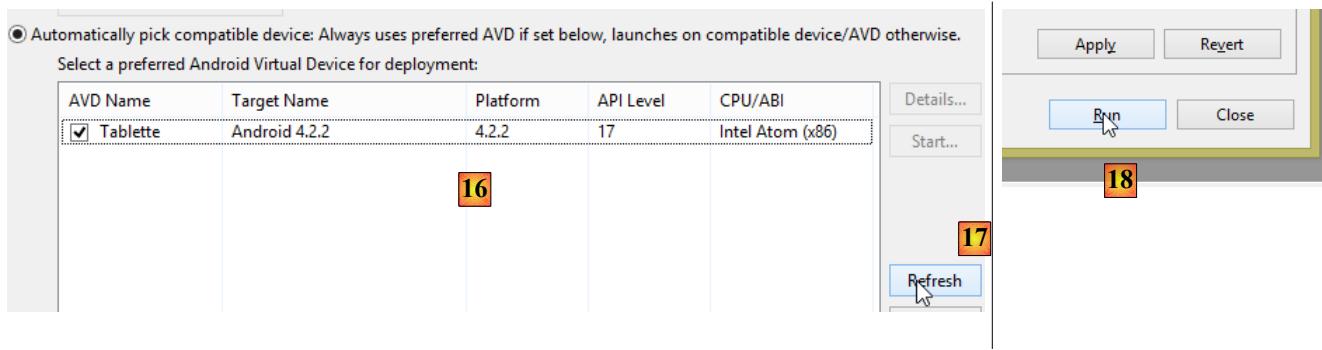
- en [5], sélectionner le type [Android Application] et cliquer l'icône + ;
- en [6], mettre le nom du projet à exécuter ;
- en [7], donner un nom quelconque à cette configuration ;



- en [8], sélectionner l'onglet [Target] ;
- en [9], il nous faut définir un émulateur de tablette Android ;
- pour cela, on utilise le bouton [10] ;

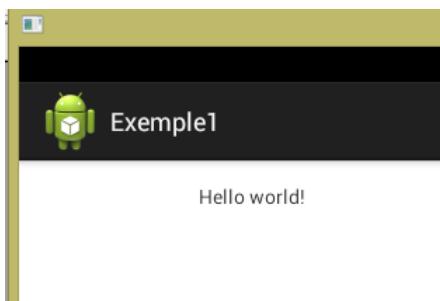


- en [11], on crée un nouvel émulateur ;
- en [12], donner un nom à l'émulateur de la tablette ;
- en [13], choisir l'objet à émuler, une tablette ;
- en [14], choisir l'API d'Android à utiliser. Précédemment, lors de l'installation du SDK Manager, une seule API a été téléchargée. C'est celle qui est proposée ;
- en [15], la saisie n'est pas possible. [Intel Atom] est l'un des quatre paquetages que nous avons téléchargés. Valider le tout ;



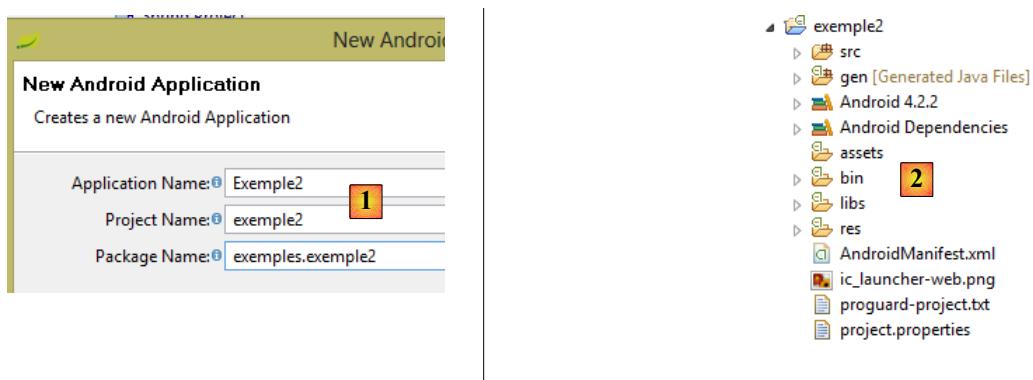
- de retour dans le gestionnaire de configuration, le nouvel émulateur doit apparaître en [16]. Utiliser [17] si ce n'est pas le cas ;
- en [18], on exécute le projet Android [exemple1] avec cette configuration ;

L'émulateur est alors lancé et l'interface visuelle du projet affichée :

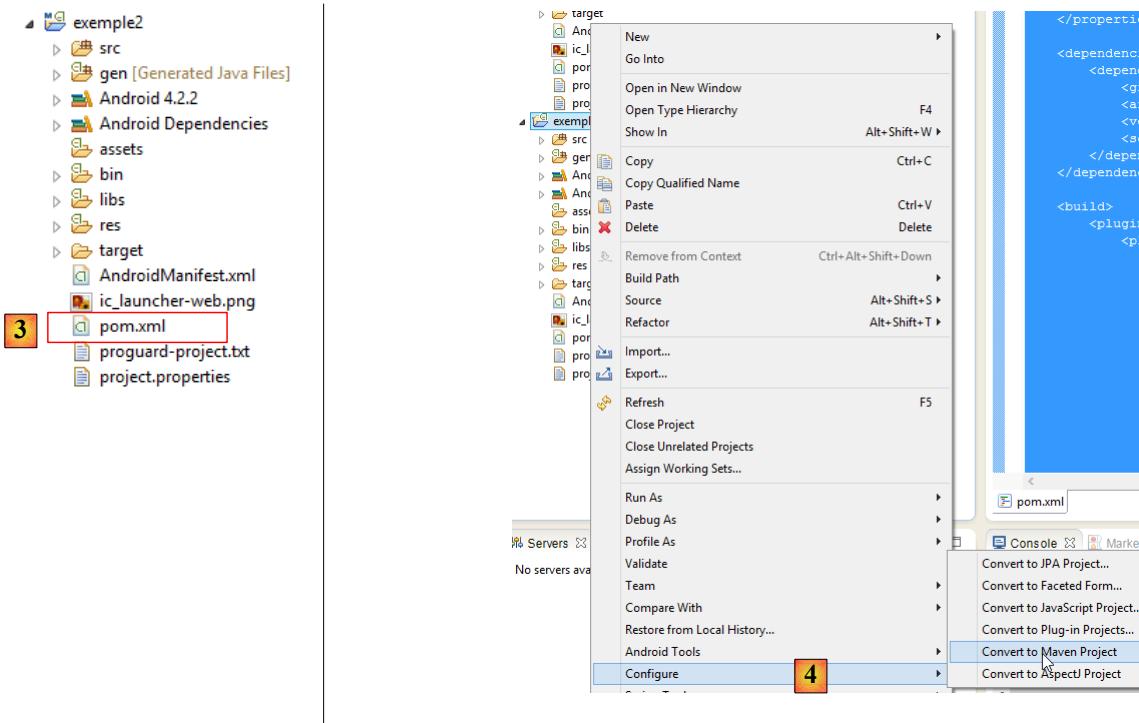


11.6 Crédation d'un projet Maven Android

Pour créer un projet Maven Android, on pourra suivre la démarche suivante :



- en [1] et [2], créer un nouveau projet Android comme il a été fait précédemment pour le projet [exemple1] ;



- en [3], ajouter à la racine du projet le fichier [pom.xml] suivant (faire un copier / coller à partir de ce document) :

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>exemples</groupId>
  <artifactId>exemple2</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>apk</packaging>

  <properties>
    <platform.android.version>4.1.1.4</platform.android.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.google.android</groupId>
      <artifactId>android</artifactId>
      <version>${platform.android.version}</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>com.jayway.maven.plugins.android.generation2</groupId>
        <artifactId>android-maven-plugin</artifactId>
        <version>3.1.1</version>
        <configuration>
          <androidManifestFile>${project.basedir}/AndroidManifest.xml</androidManifestFile>
          <assetsDirectory>${project.basedir}/assets</assetsDirectory>
          <resourceDirectory>${project.basedir}/res</resourceDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

```

<nativeLibrariesDirectory>$
{project.basedir}/src/main/native</nativeLibrariesDirectory>
    <sdk>
        <platform>16</platform>
    </sdk>
    <undeployBeforeDeploy>true</undeployBeforeDeploy>
    <proguard>
        <skip>true</skip>
    </proguard>
</configuration>
<extensions>true</extensions>
</plugin>

<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.3.2</version>
    <configuration>
        <source>1.6</source>
        <target>1.6</target>
    </configuration>
</plugin>
</plugins>
</build>
</project>

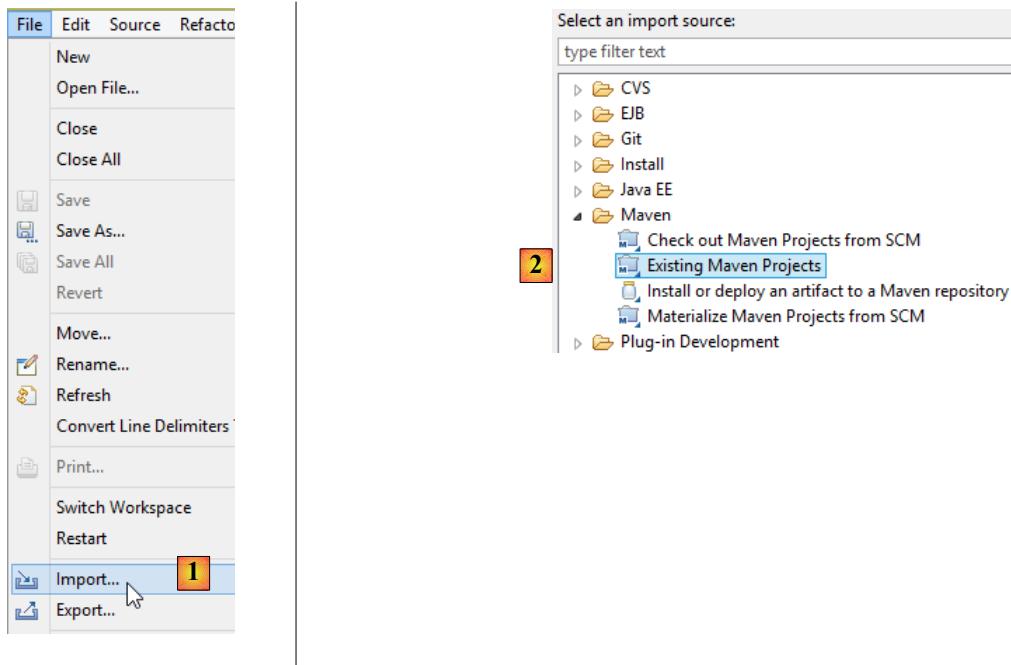
```

- puis en [4], convertir le projet en projet [Maven] puis l'exécuter comme il a été fait pour le projet [exemple1] ;

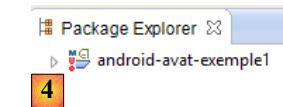
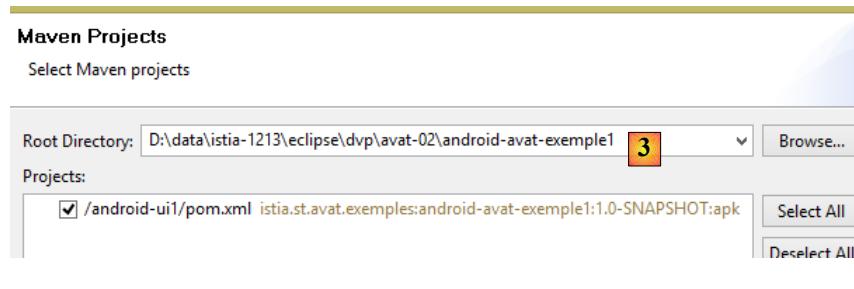


11.7 Exécution des exemples du document avec un émulateur

Importer l'exemple 1 du document dans Eclipse :

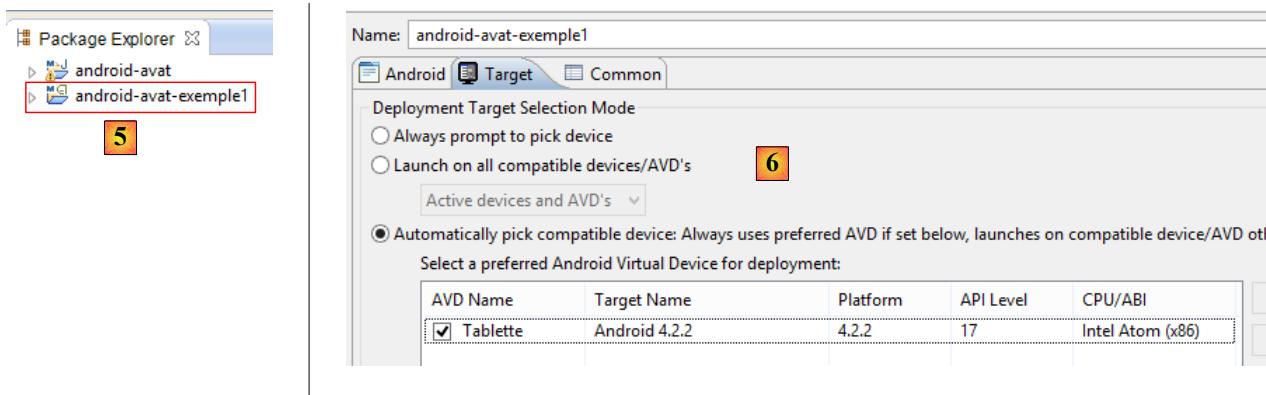


- en [1] importer un projet de type [Maven] [2] ;

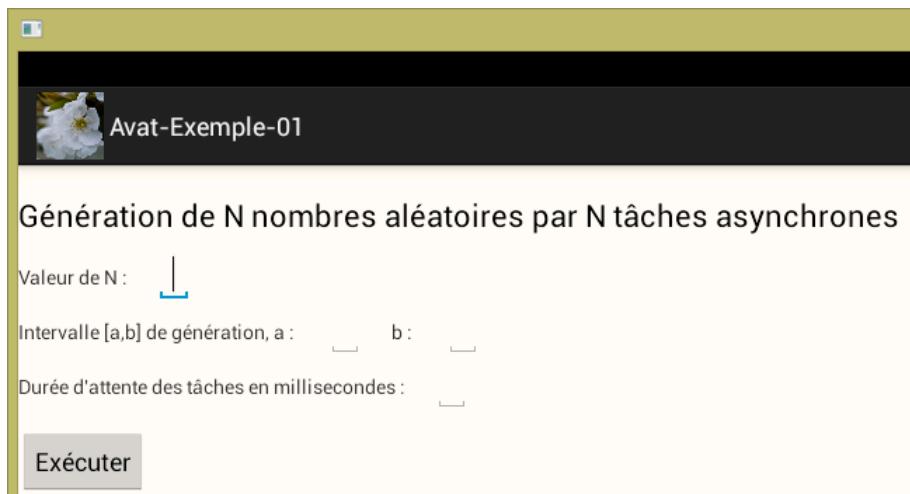


- en [3], désigner le dossier du projet et l'importer ;
- en [4], le projet importé présente une erreur. Il lui manque la dépendance Maven [android-avat] ;

En suivant la même démarche, importer le projet manquant :



- en [5], le projet [android-avat-exemple1] ne présente plus d'erreurs ;
- en suivant la démarche vue précédemment, créer une configuration d'exécution [6] pour ce projet puis l'exécuter :

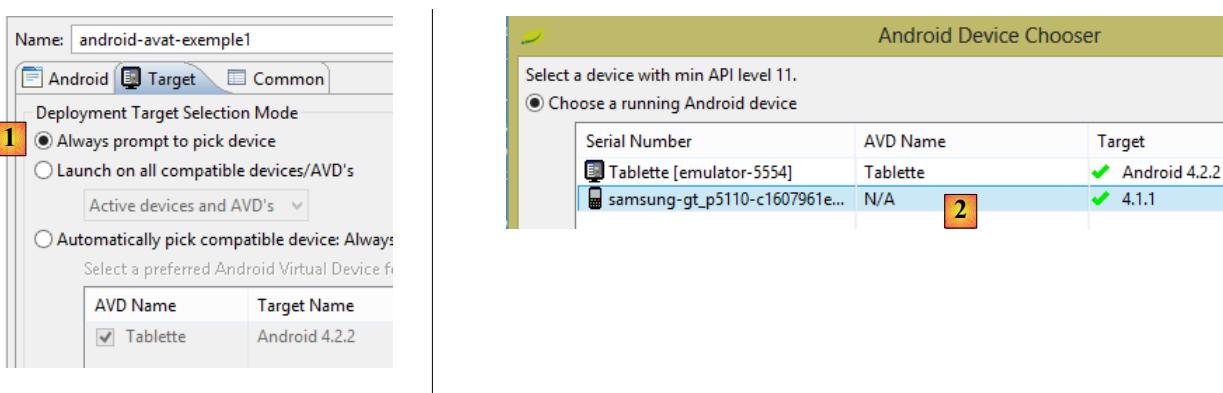


11.8 Exécution des exemples du document sur une tablette Android

La configuration qui suit a été faite avec une tablette Samsung Galaxy Tab 2 10.1.

- on doit configurer la tablette en mode développement [Paramètres / Système / Options de développement / Débogage / Débogage USB **coché**]. Cela permet d'installer sur la tablette une application Android à partir du port USB de l'ordinateur de développement ;
- brancher la tablette sur le port USB de l'ordinateur de développement ;

Charger comme il a été décrit l'application exemple [android-avat-exemple1] et modifier sa configuration d'exécution de la façon suivante :



- en [1], on demande qu'à l'exécution on nous laisse choisir où elle doit s'exécuter ;
- on lance l'exécution et en [2], Eclipse nous propose deux cibles :
 - l'émulateur déjà utilisé (ligne 1),
 - la tablette qu'on a connectée au port USB et qu'il a reconnue. On la sélectionne. L'application est alors transférée sur la tablette et exécutée.

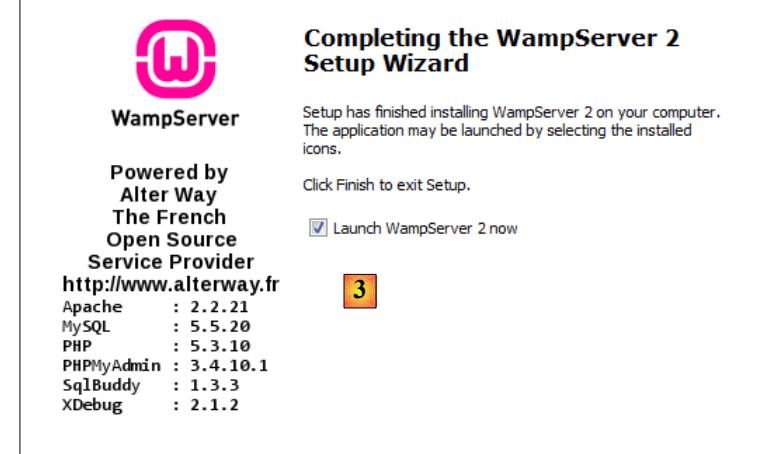
Dans le cas des applications client / serveur des exemples 2 et 7 :

- le serveur est sur le PC ;
- le client est sur la tablette ;

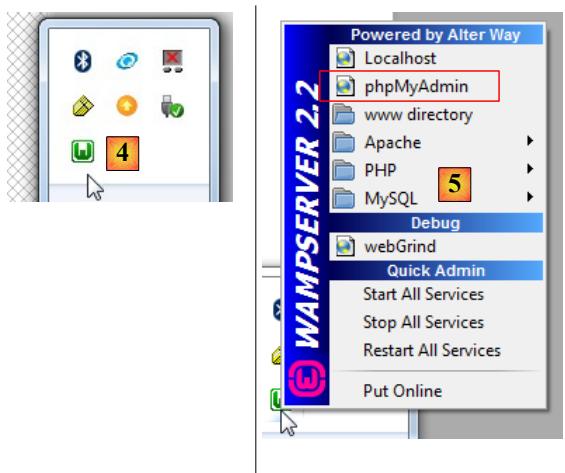
Pour qu'ils se voient, le plus simple est de les mettre sur le même réseau Wifi. Sur le PC, il faut désactiver le pare-feu voire parfois l'antivirus.

11.9 Installation de [WampServer]

[WampServer] est un ensemble de logiciels pour développer en PHP / MySQL / Apache sur une machine Windows. Nous l'utiliserons uniquement pour le SGBD MySQL.



- sur le site de [WampServer] [1], choisir la version qui convient [2],
- l'exécutable téléchargé est un installateur. Diverses informations sont demandées au cours de l'installation. Elles ne concernent pas MySQL. On peut donc les ignorer. La fenêtre [3] s'affiche à la fin de l'installation. On lance [WampServer],



- en [4], l'icône de [WampServer] s'installe dans la barre des tâches en bas et à droite de l'écran [4],
- lorsqu'on clique dessus, le menu [5] s'affiche. Il permet de gérer le serveur Apache et le SGBD MySQL. Pour gérer celui-ci, on utilise l'option [PhpPmyAdmin],
- on obtient alors la fenêtre ci-dessous,

Nous donnerons peu de détails sur l'utilisation de [PhpMyAdmin]. Nous montrerons dans le document comment l'utiliser pour créer la base de données de l'application exemple 7.

Table des matières

1 INTRODUCTION.....	5
 1.1 CONTENU DU DOCUMENT.....	5
 1.2 LES OUTILS UTILISÉS.....	8
 1.3 LES CODES SOURCE.....	9
 1.4 PRÉ-REQUIS.....	9
2 LE MODÈLE AVAT.....	10
 2.1 LE PROJET ECLIPSE.....	10
 2.2 LE COEUR DU MODÈLE.....	11
 2.3 L'IMPLÉMENTATION ANDROID DU MODÈLE AVAT.....	17
 2.4 ÉLÉMENTS D'IMPLÉMENTATION.....	26
 2.4.1 L'ACTIVITÉ ANDROID.....	26
 2.4.2 LA VUE.....	27
 2.4.3 L'ACTION.....	29
 2.4.4 LA TÂCHE ASYNCHRONE.....	31
3 AVAT- EXEMPLE 1.....	33
 3.1 LE PROJET.....	33
 3.2 L'ARCHITECTURE.....	34
 3.3 LE PROJET ECLIPSE.....	35
 3.4 LE MANIFESTE DE L'APPLICATION ANDROID.....	35
 3.5 LES DÉPENDANCES MAVEN.....	36
 3.6 L'ACTIVITÉ ANDROID.....	37
 3.7 LA FABRIQUE D'OBJETS.....	39
 3.8 LA CLASSE D'EXCEPTION.....	40
 3.9 LA VUE [VUE_01].....	41
 3.9.1 MÉTHODE [DOEXÉCUTER].....	45
 3.9.2 MÉTHODE D'ANNULATION DES TÂCHES.....	47
 3.10 L'ACTION [ACTION_01].....	47
 3.11 LA TÂCHE ASYNCHRONE [TASK_01].....	50
 3.12 LES TESTS.....	52
4 AVAT- EXEMPLE 2.....	54
 4.1 LE PROJET.....	54
 4.2 LE MANIFESTE DE L'APPLICATION ANDROID.....	54
 4.3 LE SERVEUR.....	55
 4.3.1 LA COUCHE [MÉTIER].....	55
 4.3.1.1 Le projet Eclipse.....	55
 4.3.1.2 Les dépendances Maven.....	56
 4.3.1.3 L'interface [IMetier].....	56
 4.3.1.4 L'implémentation [Metier].....	56
 4.3.1.5 La classe d'exception.....	57
 4.3.2 LE SERVICE REST.....	58
 4.3.2.1 Le projet Eclipse.....	58
 4.3.2.2 Les dépendances Maven.....	58
 4.3.2.3 Configuration du service REST.....	60
 4.3.2.4 SpringMVC.....	61
 4.3.2.5 Le serveur REST.....	64
 4.3.2.6 Déploiement et test du service REST.....	65
 4.4 LE CLIENT ANDROID.....	66
 4.4.1 LA COUCHE [METIER].....	67
 4.4.1.1 Le projet Eclipse.....	67
 4.4.1.2 Les dépendances Maven.....	67
 4.4.1.3 Implémentation de la couche [métier].....	68
 4.4.2 LA COUCHE [DAO].....	69
 4.4.2.1 Le projet Eclipse.....	70
 4.4.2.2 Les dépendances Maven.....	70
 4.4.2.3 Implémentation de la couche [DAO].....	71
 4.4.3 LA COUCHE [AVAT].....	74
 4.4.3.1 Le projet Eclipse.....	74
 4.4.3.2 Les dépendances Maven.....	75
 4.4.3.3 La tâche [Task_01].....	75
 4.4.3.4 La vue [Vue_01].....	77
 4.4.3.5 La configuration de l'application.....	78

4.4.3.6 La fabrique d'objets.....	79
5 AVAT- EXEMPLE 3.....	81
 5.1 LE PROJET.....	81
 5.2 LE PROJET ECLIPSE.....	82
 5.3 L'ACTION [ACTION_01].....	82
 5.4 LA VUE [VUE_01].....	83
 6 AVAT- EXEMPLE 4.....	85
 6.1 LE PROJET.....	85
 6.2 LE PROJET ECLIPSE.....	86
 6.3 LA SESSION.....	86
 6.4 L'ACTIVITÉ ANDROID.....	87
 6.5 L'ACTION [ACTION_01].....	88
 6.6 LA VUE [VUE_01].....	88
 6.7 LA VUE [VUE_02].....	89
 6.8 LA FABRIQUE.....	90
 7 AVAT- EXEMPLE 5.....	93
 7.1 LE PROJET.....	93
 7.2 LE PROJET ECLIPSE.....	93
 7.3 LA VUE [VUE_01].....	93
 7.4 LA VUE [VUE_02].....	94
 8 AVAT- EXEMPLE 6.....	96
 8.1 LE PROJET.....	96
 9 AVAT- EXEMPLE 7.....	99
 9.1 LE PROJET.....	99
 9.2 L'ARCHITECTURE DU PROJET.....	104
 9.3 LA BASE DE DONNÉES.....	105
 9.3.1 LA TABLE [MEDECINS].....	105
 9.3.2 LA TABLE [CLIENTS].....	105
 9.3.3 LA TABLE [CRENEAUX].....	106
 9.3.4 LA TABLE [RV].....	106
 9.3.5 GÉNÉRATION DE LA BASE.....	107
 9.4 LES PROJETS ECLIPSE DE L'APPLICATION.....	109
 9.5 LA COUCHE [MÉTIER] DU SERVEUR.....	109
 9.6 LE SERVEUR REST.....	114
 9.6.1 LE PROJET ECLIPSE.....	114
 9.6.2 LES DÉPENDANCES MAVEN.....	115
 9.6.3 CONFIGURATION DE LA COUCHE [REST].....	116
 9.6.3.1 Le fichier [web.xml].....	116
 9.6.3.2 Le fichier [rest-services-config.xml].....	117
 9.6.4 IMPLÉMENTATION DE LA COUCHE [REST].....	119
 9.6.4.1 Liste des clients.....	121
 9.6.4.2 Liste des médecins.....	122
 9.6.4.3 Liste des créneaux d'un médecin.....	122
 9.6.4.4 Liste des rendez-vous d'un médecin.....	123
 9.6.4.5 Obtenir un client identifié par son n°.....	124
 9.6.4.6 Obtenir un médecin identifié par son n°.....	124
 9.6.4.7 Obtenir un rendez-vous identifié par son n°.....	125
 9.6.4.8 Obtenir un créneau horaire identifié par son n°.....	125
 9.6.4.9 Ajouter un rendez-vous.....	126
 9.6.4.10 Supprimer un rendez-vous.....	126
 9.6.4.11 Obtenir l'agenda d'un médecin.....	127
 9.7 LA COUCHE [DAO] DU CLIENT ANDROID.....	128
 9.7.1 LE PROJET ECLIPSE.....	128
 9.8 LA COUCHE [MÉTIER] DU CLIENT ANDROID.....	129
 9.8.1 LE PROJET ECLIPSE.....	129
 9.8.2 IMPLÉMENTATION.....	129
 9.9 LA COUCHE [AVT].....	132
 9.9.1 LE PROJET ECLIPSE.....	133
 9.9.2 L'ACTIVITÉ ANDROID.....	133
 9.9.3 LA CLASSE DE CONFIGURATION.....	134
 9.9.4 LE PATRON DES VUES.....	135
 9.9.5 LA VUE [CONFIG].....	137
 9.9.6 LA VUE [LOCALVUE].....	141

<u>9.9.7</u> LA FABRIQUE.....	146
<u>9.9.8</u> LA TÂCHE [GETALLCLIENTSTASK].....	148
<u>9.9.9</u> LA TÂCHE [GETALLMEDECINSTASK].....	149
<u>9.9.10</u> LA VUE [ACCUEILVUE].....	149
<u>9.9.11</u> LA TÂCHE [GETAGENDAMEDECINTASK].....	152
<u>9.9.12</u> LA VUE [AGENDAVUE] - 1.....	152
<u>9.9.13</u> L'ADAPTATEUR [LISTCRENEAUXADAPTER].....	154
<u>9.9.14</u> LA VUE [AGENDAVUE] - 2.....	158
<u>9.9.15</u> LA TÂCHE [SUPPRIMERRVTASK].....	160
<u>9.9.16</u> LA VUE [AJOUTRVTUE].....	161
<u>9.9.17</u> LA TÂCHE [AJOUTERRVTASK].....	164
<u>9.10</u> CONCLUSION DE L'EXEMPLE 7.....	164
<u>10</u> CONCLUSION GÉNÉRALE.....	167
<u>11</u> ANNEXES.....	168
<u>11.1</u> LA BIBLIOTHÈQUE JACKSON.....	168
<u>11.2</u> LA BIBLIOTHÈQUE GSON.....	169
<u>11.3</u> INSTALLATION DE L'IDE ECLIPSE.....	170
<u>11.4</u> INSTALLATION DU SDK MANAGER D'ANDROID.....	177
<u>11.5</u> INSTALLATION DES Outils ANDROID POUR ECLIPSE.....	177
<u>11.6</u> CRÉATION D'UN PROJET MAVEN ANDROID.....	181
<u>11.7</u> EXÉCUTION DES EXEMPLES DU DOCUMENT AVEC UN ÉMULATEUR.....	183
<u>11.8</u> EXÉCUTION DES EXEMPLES DU DOCUMENT SUR UNE TABLETTE ANDROID.....	185
<u>11.9</u> INSTALLATION DE [WAMP SERVER].....	186