Document d'architecture du projet HP Bookstore

1. Contexte du projet

1.1. Objectifs du projet

Ce projet a pour but de présenter une application de test, proposant à un utilisateur d'effectuer une commande de livres au sein d'un catalogue de livres "Henri Potier", en ajoutant ces livres à un panier. Cet utilisateur bénéficie d'une offre de réduction sur l'ensemble du panier en fonction du nombre de livres qu'il contient. Il peut alors commander l'ensemble de ces livres.

1.2. Postulats de base

L'objectif est de fournir une application avec une construction similaire à une vrai application cliente, tout en servant les besoins de la démonstration technique. Partant de ce principe, l'application doit répondre aux postulats suivants :

- L'architecture de l'application doit être prévue pour un maintien à long terme aisé. De fait, l'application doit être façile à faire évoluer (ajout de fonctionnalités ou ajout sur l'existant).

- L'application doit rester de qualité au cours de son évolution, ce qui signifie que le code doit être maintenable au sein d'une équipe de projet. De même, l'application doit être testable afin de maintenir un niveau de qualité du code.

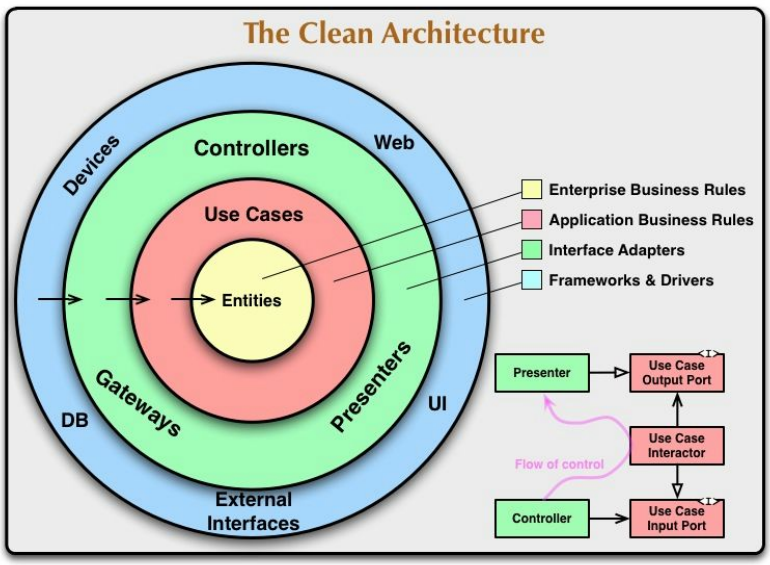
- L'application doit être installable sur plusieurs types de devices android (smartphone et tablette) ainsi que selon plusieurs configurations (portrait et paysage).

- Enfin, dans un but de démonstration, l'architecture de l'application peut utiliser des bibliothèques connus pouvant être amené à être utilisé dans un réel projet, même si la taille de cette application ne le justifie pas.

2. Choix de l'architecture

Au regard des postulats de base, le choix d'architecture doit satisfaire des besoins en évolutivité, maintenabilité et testabilité.

Une architecture qui répond bien à ces besoins, ayant été popularisé ces dernières années, est la "Uncle ben's Clean Architecture". Cette architecture base son fonctionnement sur l'inpédendance des différentes couches de l'application, les faisant communiquer en suivant la logique des "cas d'utilisation" :



- La première couche, la couche graphique, reste ici très simple. Elle se contente de transmettre les actions de l'utilisateur à la couche de présentation et d'afficher ce que cette couche lui demande d'afficher.

- La couche de présentation ensuite, se charge de transmettre ou de récupérer auprès de la couche d'interaction les données appropriées, en suivant un cas d'utilisation précis, puis les formatte pour les adapter à la couche graphique et lui demande de se mettre à jour par rapport à ces informations.

- La couche d'interaction regroupe les différents processus métiers que l'application doit pouvoir gérer (enchaîner des opérations de téléchargements, de transformation des données et de stockage par exemple). Elle fait pour ça appel à la couche entité pour réaliser ces opérations puis informe la couche de présentation une fois la donnée prête

- La couche entité enfin, fourni à la couche d'interaction les entités contenant les données, les accès aux sources de données (réseau, fichiers, base de données) et les opérations de stockage (gestion du cache par exemple)

Cet organisation du code permet une séparation très claire des responsabilités du code et une indépendance de chacun de ces niveaux.

Pour correspondre à cet organisation, le projet est divisé en deux module Android :

- Le module de données, appelé "**HpData**", qui contient la couche entité et la couche d'interaction
- Le module de présentation, appelé "**HpPresentation**", qui contient la couche de présentation et la couche graphique.

3. Module de données HpData

3.1. Structure du module

Ce module a pour objectif de fournir à l'application un accès aux sources de données et au stockage de l'application, par le biais de "cas d'utilisations" (qui seront ici appelé ***Interactor***).

Le module se divise en plusieurs packages :

- **definition** : l'ensemble des interfaces et classes abstraites permettant de définir les méthodes de récupération des données et de stockage des données.
- **exceptions** : l'ensemble des exceptions que ce module peut envoyer en plus des exceptions standards.
- **hpbooks** : Une instance concrète d'une source de données permettant la récupération des livres "Henri Potier" et des offres commerciales par leurs webservices associés, et fournissant des méthodes de stockage en cache de ces informations.
- **interactors** : L'ensemble des Interactor que l'application peut utiliser. Contient pour le moment les Interactor permettant de récupérer la liste des livres et celui pour les offres commerciales.
- **utils** : Ensemble de classes utilitaires.

L'idée derrière ce découpage est de permettre, à chaque ajout de nouvelles sources de données, de créer un nouveau package avec le nom de la source en question et d'y placer son implémentation.

3.2. Bibliothèques utilisées

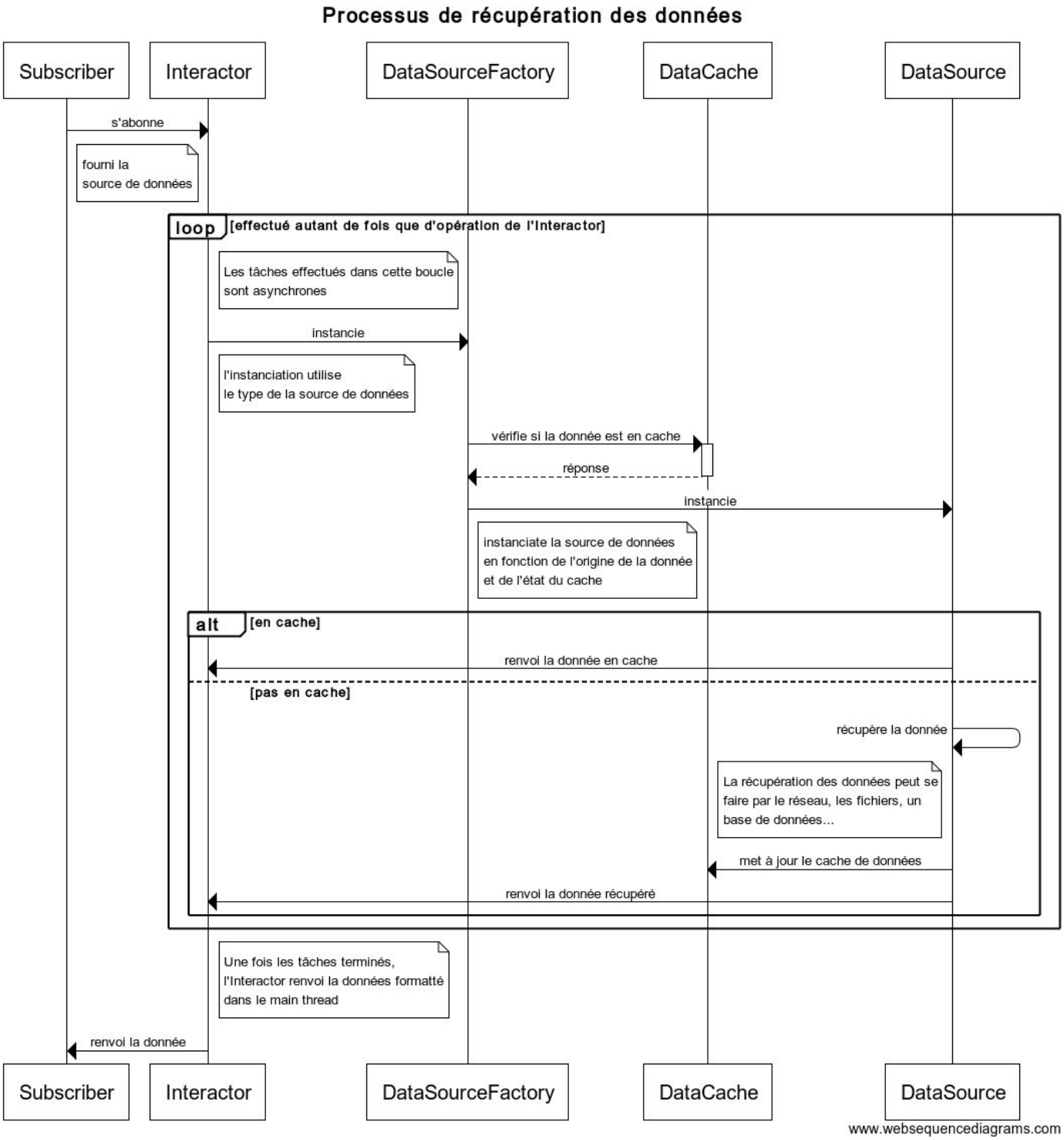
Pour permettre la récupération des données ainsi que l'enchaînement des opérations, plusieurs bibliothèques ont été incluses afin de faciliter la tâche :

RxAndroid : une déclinaison de RxJava pour Android, permettant de faire de la programmation réactive fonctionnelle. Cette bibliothèque est utilisée pour réaliser les Interactor, puisqu'elle permet de gérer un enchaînement de processus asynchrones et d'en traiter les résultats à la suite pour obtenir façilement les données voulus.

Retrofit : une bibliothèque permettant d'accéder à un catalogue de webservices REST proposé par un serveur web. Elle fournit les opérations réseaux standard, elle est très utilisée aujourd'hui et simple d'utilisation, ce qui permet une maintenabilité facilité.

Mockito : une bibliothèque permettant de "mocker" des classes (leur donner des comportements par défaut) à des fins de tests, tests réalisés avec JUnit.

3.3. Processus de récupération des données



4. Module de présentation HpPresentation

4.1. Structure du module

Ce module a pour objectif de fournir la partie visuelle de l'application, connectée aux sources de données par le biais des Interactor du module HpData (de fait, ce module inclus le module HpData).

Le module se divise en plusieurs packages :

- **dependencies** : L'ensemble des dépendences dont le module peut avoir besoin. Les classes de ce package sont des classes utilisées par la bibliothèque Dagger (voir plus loin) pour injecter des dépendances tel que les sources de données, les Interactor et les Presenter au classes les nécessitant.
- **model** : Un ensemble de classes contenant les données qui seront exploitées par la vue. Contrairement aux entités contenus dans HpData, ces classes sont spécialisés pour la vue et peuvent, par exemple, contenir des messages prêt à être affichés.
- **presenter** : L'ensemble des classes Presenter de l'application. Un Presenter est un classe jouant le rôle de la couche de présentation : il récupère les données utiles via les Interactors et le modèle de données, réagit à l'état de la vue et aux actions de l'utilisateur et met à jour la vue.
- **views** : L'ensemble des classes représentant les vues de l'application. Contient les Activity, les Fragments, les Adapter et les composants personnalisés.

Le découpage est ici différent pour bien marquer la structure MVP (Model-Presenter-View) du module. Les différentes fonctionnalités sont séparées à l'intérieur du package **views** (sous-packages *cart* et *catalog*, les classes de définition étant dans le sous-package *base*).

4.2. Bibliothèques utilisées

De même ici, pour faciliter le traitement des données et la gestion de la vue, plusieurs bibliothèques ont été intégrées :

Dagger 2 : une bibliothèque permettant l'injection de dépendences. Son utilitée réside dans la centralisation des classes gérant les données et dans l'injection de celles-ci directement dans les classes ayant besoin d'eux.

Note : Dagger est une bibliothèqe assez lourde à mettre en place, et son utilisation dans un projet de petite taille n'est pas recommandée. Toutefois elle est intégrée ici dans un but de démonstration et pour montrer le genre d'outils à intégrer dans le cadre d'une application amenée à beaucoup évoluer.

Butterknife : une autre bibliothèque d'injection de dépendances, permettant à une classe "graphique" de charger directement les vues provenant d'une ressource xml dans le code et d'intéragir avec elles.

Picasso : une bibliothèque permettant de charger des images dynamiquement depuis une source de données distante ou non (ressource en ligne ou locale), de la mettre en cache et de la redimensionner pour correspondre à la vue dans laquelle elle doit être affichée.

RecyclerView / CardView : ces éléments graphique ne sont pas des bibliothèques externes, elles font partis de la bibliothèque de support appcompat-V7 d'Android. Elles sont utilisées ici pour fournir des listes ou grilles de vues dynamiques et optimisées pour le chargement de nombreuses vues.

Note : ici comme pour Dagger, l'utilisation de ces éléments n'est pas la plus adaptée puisque la quantité de données récupérées est très limitée. Ils ont été mis pour les mêmes raisons, c'est-à-dire dans un but de démonstration et pour anticiper d'éventuels besoins futurs.

4.3. Architecture des vues

Quelques précisions ici concernant l'architecture des vues, la navigation et les éléments graphiques utilisés.

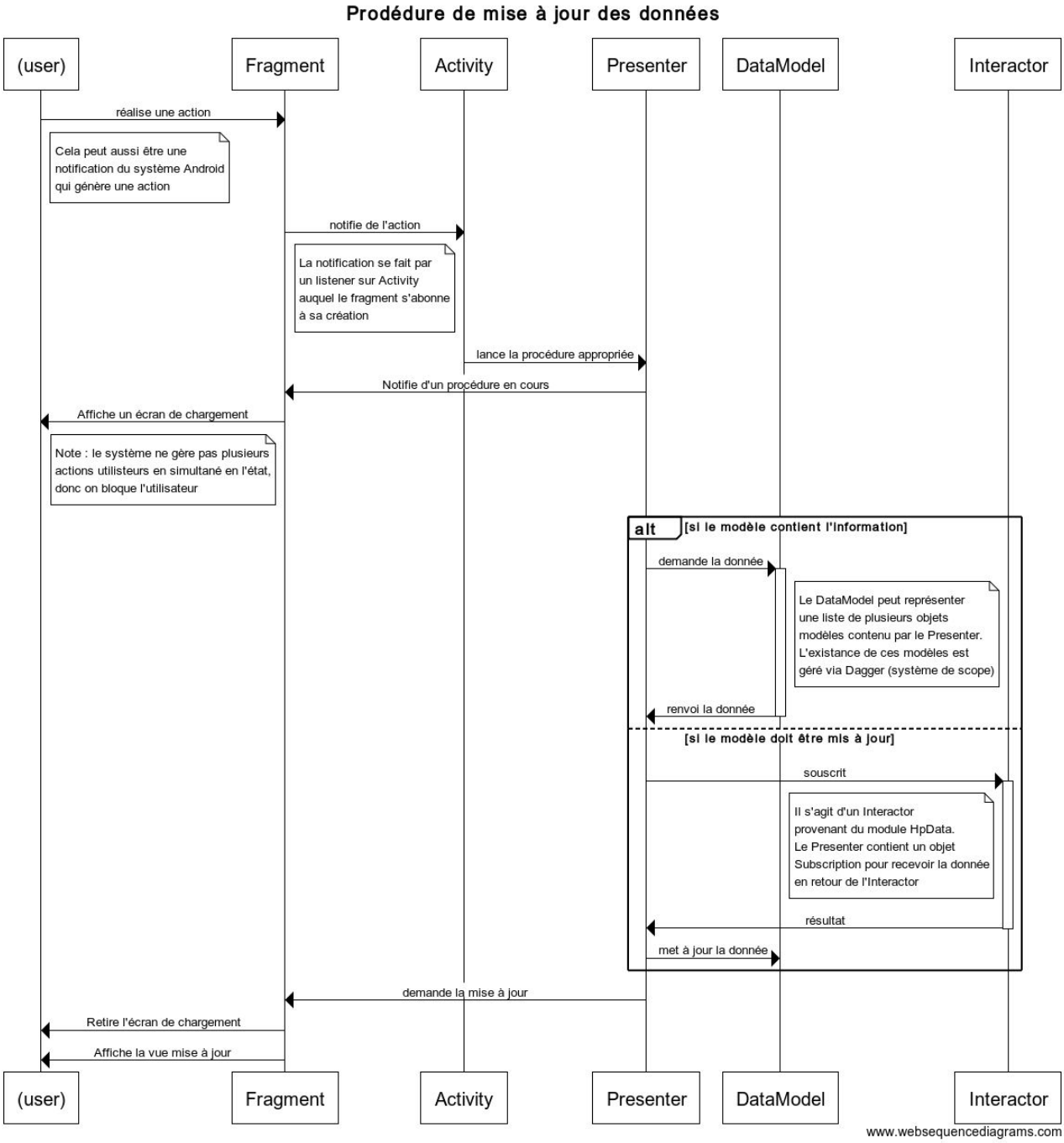
L'application adopte une structure multi-activités, c'est-à-dire que pour chaque écran / fonctionnalité de l'application il y a une activité. Chaque activité possède au moins un fragment qu'il affiche, fragment qui se contente *uniquement* de gérer la vue. Le fragment en question possède une référence sur son activité (*listener*), à qui il transmet les actions de l'utilisateur.

Chaque activité possède également un Presenter (injecté via Dagger) qui va prendre le fragment comme référence (appelée "*PresentableView*"). L'activité va réagir aux évènements du fragment en appelant son Presenter, qui va effectuer ses opérations pour récupérer la donnée puis enfin les envoyer au fragment pour qu'il se mette à jour.

Cette structure, même si elle peut paraître complexe pour pas grand chose, à le gros avantage de rendre le fragment et donc la vue concrète complètement indépendante de son activité, et donc être réutilisable à souhait. Cela est très utile pour avoir une version tablette de l'application notamment, mais également pour la gestion du mode paysage. De plus, cela maintien l'activité dans son rôle de semi-contrôleur qui ne connait pas le contenu de la vue qu'il manipule.

Enfin, chaque activité se voit injecter un NavigationManager, une classe fournissant les méthodes pour naviguer entre les activités dans l'application. Pour l'heure, la seule navigation disponible est celle menant du catalogue de vue vers le panier, et le retour du panier vers le catalogue.

4.4. Procédures d'affichage des données



5. Evolutions envisagées

5.1. Affichage des détails d'un livre

Une des évolutions envisagée pour ce projet aurait été d'ajouter une nouvelle source de données capable de fournir des détails sur les livres. Cette source de données aurait pu être un fichier JSON contenant les détails avec pour clé le numéro ISBN du livre associé.

Un nouvel écran aurait été créé afin d'afficher ces détails à l'utilisateur, au clic sur un des livres affiché dans le catalogue.

Cette fonctionnalité aurait été intéressante pour montrer la souplesse de l'architecture intégrée : il suffirait ici d'ajouter une nouvelle source de donnés au niveau du module HpData, de modifier l'Interactor pour ajouter aux livres récupérés du webservice la partie description, puis au niveau du module HpPresentation ajouter les vues et le Presenter associé.

5.2. Version tablette

Une autre amélioration possible était d'avoir une version spécifique pour tablette. Avec les deux vues (catalogue et panier) cela n'aurait pas eu d'intérêt, cependant avec l'ajout d'une vue de détail sur un livre on aurait très bien pu avoir un écran scindé en deux, avec dans la partie gauche le catalogue de livres et dans la partie droite les détails du livre sélectionné.

Cette fonctionnalité aurait mis en avant les bénéfices de la séparation activité / fragment / présenteur. En effet, avec cette structure, intégrer deux fragments au lieu d'un dans l'activité "catalogue" aurait été façile. L'activité aurait fait un très bon contrôleur pour transmettre d'un fragment à l'autre les actions de l'utilisateur, tout en continuant de transmettre les actions aux présenteurs associés (il y aurait eu autant de présenteur que de fragments dans l'activité).

5.3. Gestion de comptes

Enfin, une dernière amélioration intéressante aurait été l'intégration d'une gestion de comptes. Cette fonctionnalité aurait permit à un utilisateur de se créer un compte pour sauvegarder l'état de son panier, rendant l'application plus "crédible" (une application de vente de livres sans gestion de compte ne peut simplement pas fonctionner puisqu'il faut un adresse où livrer les livres).

Au-delà de ça, cela aurait intégré dans l'application un principe de session que l'architecture aurait très bien su gérer : le système d'injection de dépendences, d'une part, peut fournir à la vue les présenteurs adaptés à la session de l'utilisateur en cours. Mais en plus de ça, la session aurait pu être gérée directement par la couche d'interaction, qui aurait accédée à la session en cours et s'en serait servi pour récupérer les données associés à l'utilisateur courant (sans presque aucune modifications au niveau du code grâce à RxJava).