



**LOG8430 - Architecture logicielle et conception avancée**

**Hiver 2023**

**Travail pratique #1: Principes et patrons de conception**

**Équipe Delta**

**1954607 - Victor Kim**

**1895231 - Anis Youcef Dilmi**

**2225733 - Yasser Benmansour**

**2224721 - Mohammed Ridha Ghoul**

**Le dimanche 12 février 2023**

## Question 1 : Analyse d'architecture logicielle (20 points)

### 1.1 Identifier chacun des packages qui composent Omni-Notes (3 points)

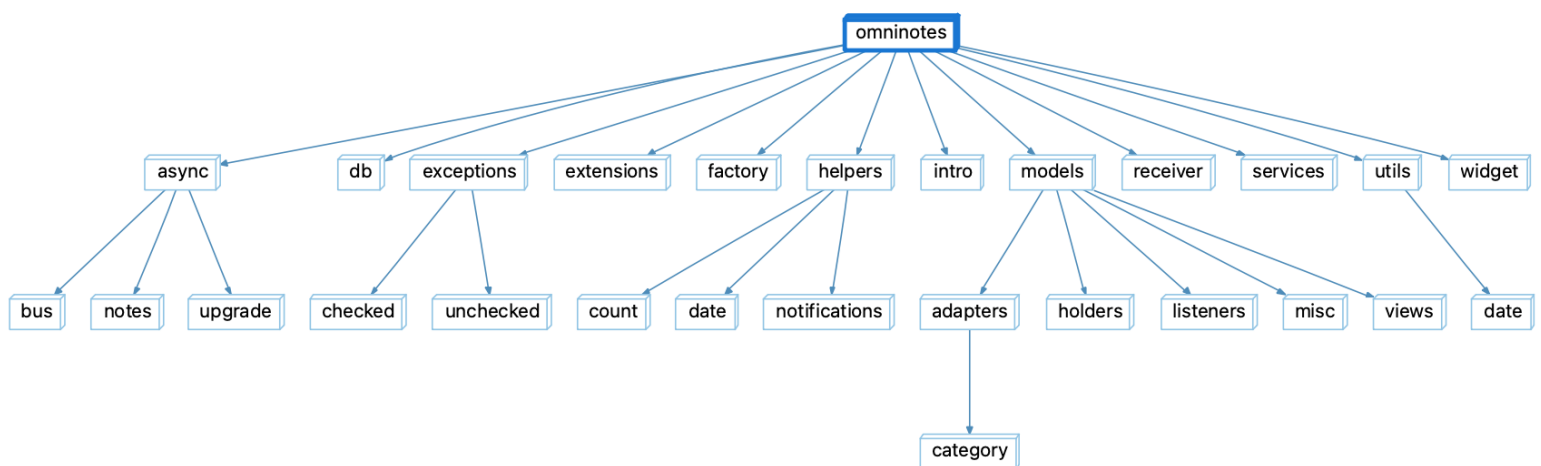
Les principaux packages qui composent Omni-Notes se trouvent dans le package suivant:

`/omniNotes/src/main/java/it/feio/android/omninotes/`

Ils sont représentés par les dossiers suivant:

- `intro/`
- `async/`
- `db/`
- `exceptions/`
- `extensions/`
- `factory/`
- `helpers/`
- `widgets/`
- `models/`
- `receivers/`
- `services/`
- `utils/`

Aussi, un autre package important se trouve au chemin `/omniNotes/src/main/res/layout/`, il contient les fichiers XML qui sont les vues présentés à l'utilisateur. Un autre package important est une dépendance du projet: `greenrobot.event.EventBus`.

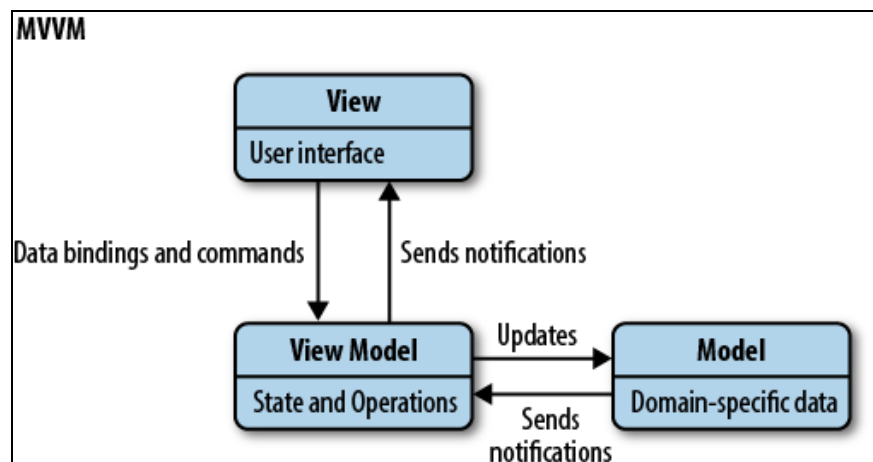
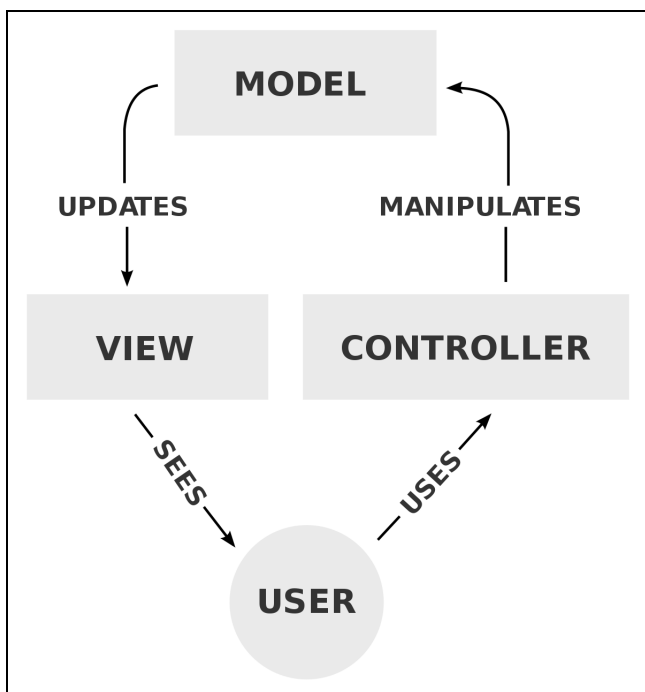


**Figure 1:** Hiérarchie des packages du projet Omni-Notes

1.2 En se référant aux styles d'architectures présentés au cours, identifier le(s) style(s) d'architectures de Omni Notes. (5 points)

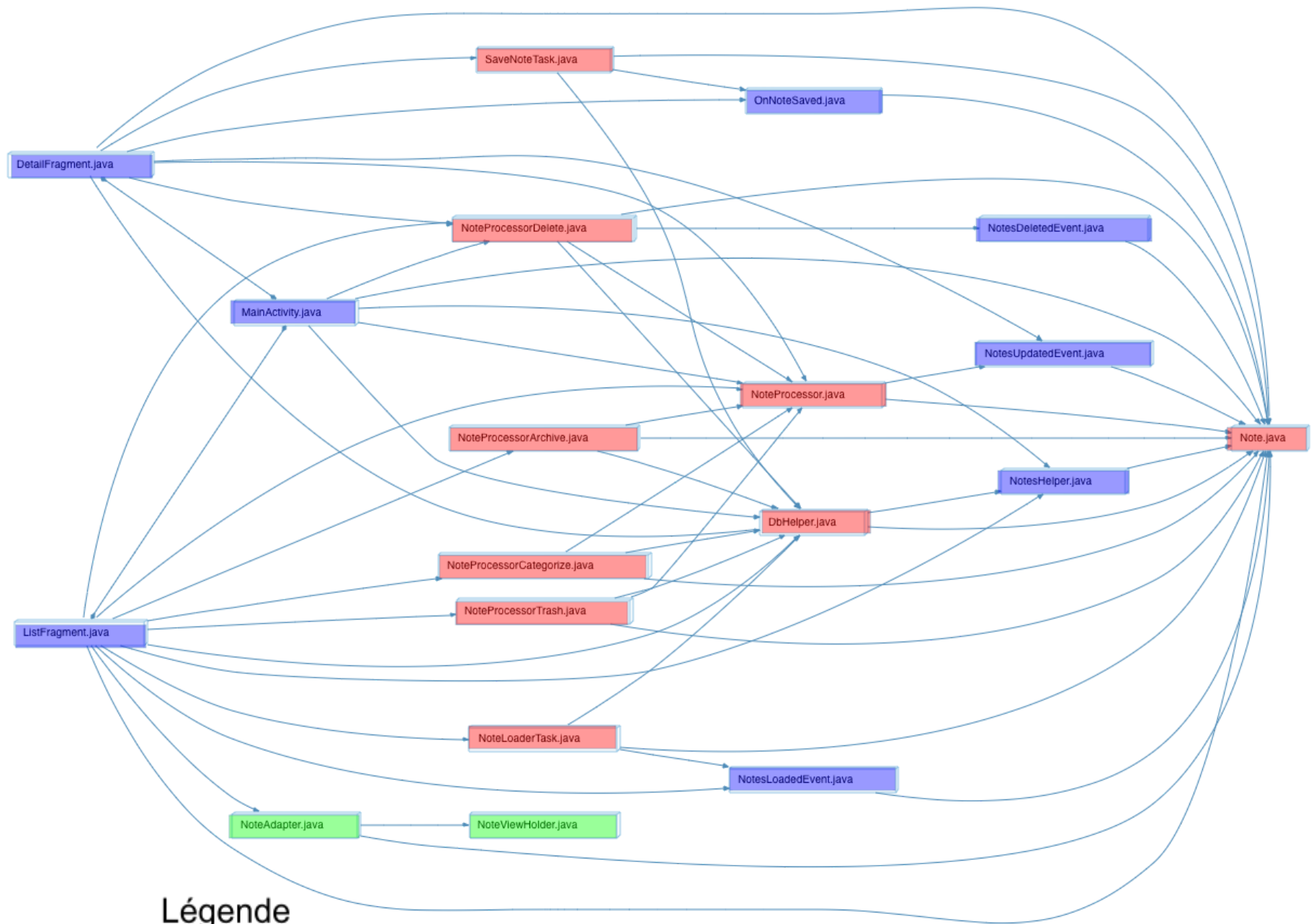
L'application Omni-Notes est construite avec une version spécifique du modèle MVC, c'est l'architecture Model-View-ViewModel (MVVM). Ce modèle architectural sépare le développement des interfaces utilisateur de celui de la logique d'affaire. Souvent, ce modèle utilise du "data-binding" pour permettre de séparer le développement des View de celui des ViewModel et Model.

Dans le cas de l'application Omni-Notes, ce sont des fichiers XML qui représentent les éléments d'interface utilisateur. Ces éléments sont liés aux classes Activity et Fragment par du data-binding, ces classes peuvent alors avoir une communication bidirectionnelle avec les éléments de l'interface utilisateur.



**Figure 3:** Architecture Model-View-ViewModel (MVVM) [2]

**Figure 2:** Architecture Model-View-Controller (MVC) [1]



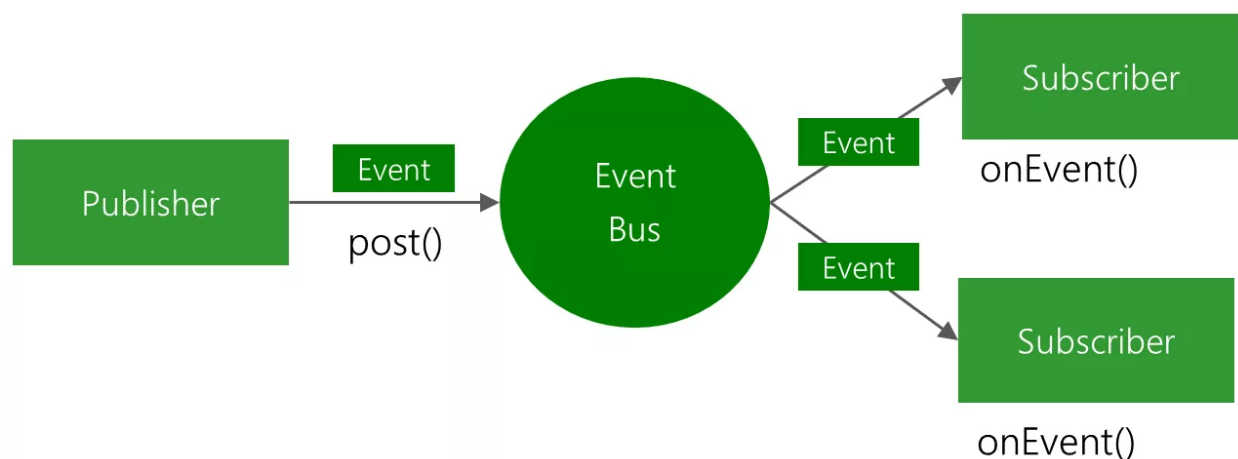
## Légende



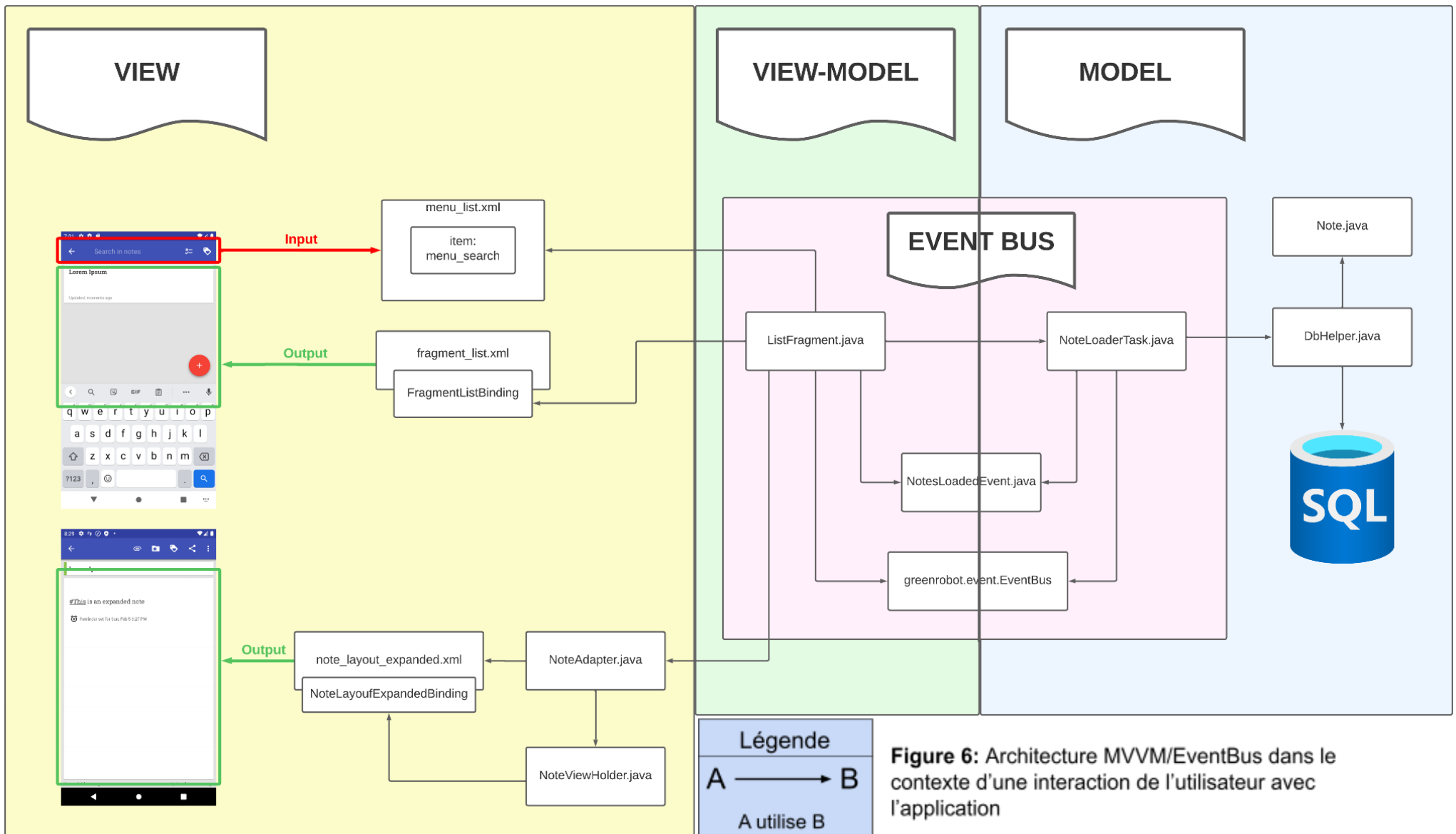
**Figure 4:** Dépendance entre fichiers de l'architecture MVVM (généré avec Understand)

La figure 4 ci-dessus a été obtenue avec le logiciel Understand avec la commande “Graph” et l’option “Dependencies”. On y voit certains fichiers qui participent au modèle architectural MVVM, en réalité il y a beaucoup plus de fichiers qui participent à MVVM mais pour des raisons de compréhension, nous avons choisi ces fichiers spécifiquement. Les **Activity**, les **Fragment** et les **Event** sont considérés comme le ViewModel, ils exécutent la logique d’affaire et font le pont entre le Model et la View. Dans ce schéma, l’**Adapter** et le **Holder** représentent la View, ils permettent d’organiser les données avant qu’elles soient affichées à l’écran. Les **Processor**, le **Task** et la classe **Note** (la principale donnée de l’application) représentent le Model, ils permettent des opérations CRUD sur la BD.

Dans la figure 6, plus bas, on peut voir le fonctionnement de l’application en allant d’une interaction spécifique de l’utilisateur avec l’écran, jusqu’à la communication avec la base de données et puis la réponse de l’application à travers l’interface utilisateur. Ce schéma nous permet de voir spécifiquement quelques fichiers qui participent à l’architecture MVVM. Ceci nous donne une idée sur l’architecture générale de l’application. On peut aussi voir sur la figure 5 ci-dessus un exemple d’architecture EventBus qui est utilisé dans l’application. Le bus d’événements sert d’intermédiaire entre les producteurs d’événements et les consommateurs d’événements. Un producteur émet les événements vers un bus et les classes qui sont inscrites aux événements (consommateurs) seront notifiées [3]. Dans le cas particulier de notre application, le **NoteLoaderTask** (Model) envoie une notification au **ListFragment** (ViewModel) par la voie du bus d’événement, cette notification l’avertit que les données ont bien été récupérées de la base de données pour qu’il puisse faire une action.

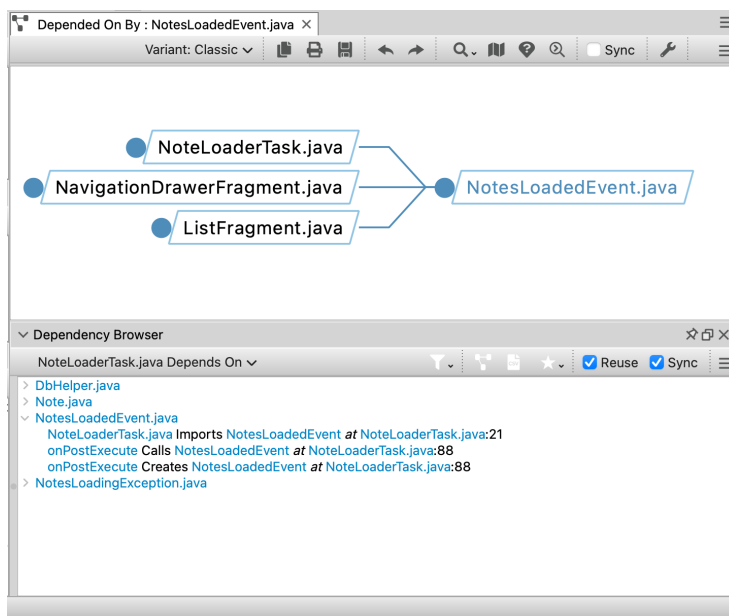


**Figure 5:** Architecture EventBus utilisé par la librairie greenrobot utilisée dans Omni-Notes [4]



1.3. En se référant à l'organisation typique recommandée par ces styles d'architecture, décrire le rôle de chaque package dans l'architecture. (5 points)

- 1) **Package omninotes:** Contient les classes qui représentent les vues du système, chacune de ces classes a comme suffixe `Activity` ou `Fragment` pour indiquer qu'elle hérite de `AppCompatActivity` ou `Fragment` respectivement. Une `Activity` est reliée à une vue plein écran tandis qu'un `Fragment` ne représente qu'une portion de l'écran. Ces deux types de classes sont couplées avec un fichier XML qui définit la structure de la page comme le ferait une structure en HTML. Ces classes font partie du ViewModel (logique d'affaire).
- 2) **Package async:** Contient le code asynchrone, c'est-à-dire qui ne bloque l'exécution de l'application ou qui s'exécute en arrière-plan. Il contient plusieurs sous packages:
  - a) **bus:** Ce package participe à l'architecture EventBus, il représente les notifications qui sont échangées. Contient les classes qui représentent les types d'événements qui peuvent circuler sur le bus. Ce package participe donc à l'architecture Event Bus. Dans l'image suivante on peut voir que 3 fichiers dépendent du fichier `NotesLoadedEvent.java`. Dans `NoteLoaderTask.java`, la fonction `postExecute()` envoie l'événement `NotesLoadedEvent` sur le bus d'événements et dans `ListFragment.java`, la fonction `onEvent()` reçoit cet événement.



**Figure 7:** Utilisations de `NotesLoadedEvent.java`

- b) **notes:** Ce package participe à l'architecture MVVM, il fait partie du Model. Contient les classes pour faire des opérations CRUD dans la base de données, ce code concerne uniquement les notes. Dans l'image suivante, on voit une classe pour modifier la catégorie d'une note, elle fait appel à la base de données avec [DbHelper](#).

```
public class NoteProcessorCategorize extends NoteProcessor {  
    Category category;  
  
    public NoteProcessorCategorize(List<Note> notes, Category category) {  
        super(notes);  
        this.category = category;  
    }  
  
    @Override  
    protected void processNote(Note note) {  
        note.setCategory(category);  
        DbHelper.getInstance().updateNote(note, false);  
    }  
}
```

**Figure 8:** Modification de la catégorie d'une note.

- c) **upgrade:** Contient le code à exécuter en arrière-plan lorsque la base de données se met à jour.
- d) **async:** Contient les classes qui font des tâches asynchrone comme la sauvegarde en arrière plan des données de l'application et la gestion des pièces jointes attachées aux notes. Toutes les classes avec suffixe "Task" sont dérivées de la classe [AsyncTask](#), celle-ci permet d'effectuer des tâches en arrière-plan et de publier les résultats sur le thread de l'UI sans avoir à manipuler les threads et/ou les handlers.
- 3) **Package db:** Ce package participe à l'architecture MVVM, il fait partie du Model. Contient le code pour analyser les fichiers de requête SQL et aussi le code pour créer et envoyer des requête à la base de données. La classe [DbHelper](#) est une exemple du patron Singleton, c'est-à-dire qu'une seule instance de cette classe est utilisée par toute l'application.



- 4) **Package exceptions:** Dans le langage java, il existe des classes qui représentent des exceptions. Une exception est un événement, qui se produit au cours de l'exécution d'un programme, qui perturbe le déroulement normal du programme. Pour pouvoir décrire chaque type d'exception et les différencier les unes des autres, nous associons une classe à chaque type d'exception.
- 5) **Package extensions:** Ce package contient deux classes: [ONDashClockExtension](#) et [PushBulletExtension](#). Ces deux classes servent d'extensions pour les fonctionnalités d'horloge pour écran verrouillé (widget) et pour le système de notification Push respectivement. Ces classes doivent être déclarées comme service dans le [AndroidManifest](#).
- 6) **Package factory:** Ce package contient une pour créer le bon type d'Uri en fonction d'un paramètre. Un URI MediaStore est un identifiant pour retrouver un élément de média stocké dans le MediaStore Android [5].
- 7) **Package helpers:** On peut traduire le mot helper par assistant. Les classes dans ce package permettent de manipuler des données et d'autres classes. Voici quelques exemples:
- a) DateHelper: sert à manipuler des objets Date.
  - b) RecurrenceHelper: sert à manipuler des récurrences dans le contexte d'un rappel récurrent sur un téléphone intelligent.
  - c) Répertoire notifications: ces classes servent à manipuler les notifications du téléphone.
  - d) AttachmentHelper: fait partie du ViewModel (logique d'affaire). Sert à manipuler les attachements aux notes comme des images ou fichiers audio.
  - e) BackupHelper: Aide à faire des opérations liées au à la sauvegarde (backup) du téléphone.
  - f) LanguageHelper: Aide à faire des opérations liées au langage du téléphone.
  - g) NotesHelper: fait partie du ViewModel (logique d'affaire). Sert à faire des manipulations sur les notes.
  - h) PermissionsHelper: fait des opérations reliés aux permissions (lecture, écriture, paramètres, etc).
- 8) **Package intro:** Ce package fait partie de la View. Il contient les classes responsable de l'intro de l'application. Lorsque l'utilisateur ouvre l'application, il voit l'intro.

## 9) Package models:

- a) models: Fait partie du Model. Contient les classes qui représentent les données de l'application, par exemple, une [Note](#), une [Category](#) ou un [Attachment](#).
- b) adapters: Fait partie de la View. Contient des classes qui permettent d'organiser les données pour qu'elles soient présentées de la bonne façon dans la View.
- c) holder: Fait partie de la View. Contient les classes qui permettent de contenir un élément de vue, comme un élément de liste par exemple.
- d) listeners: Contient des interfaces qui déclarent des méthodes callback qui réagissent à des événements.
- e) misc: Peut être traduit par divers. Il contient par exemple, une classe [PlayStoreMetadataFetcherResult](#), elle permet d'aller chercher des métadonnées sur une application du PlayStore.
- f) views: Fait partie de la View. Ces classes sont utilisées par les fichiers de vue XML pour appeler des callbacks lorsque l'utilisateur interagit avec la vue.

**10) Package receivers:** Les classes de ce package sont dérivées de [BroadcastReceiver](#), elles sont conçues pour recevoir des messages venant du OS. Le système d'exploitation peut envoyer des messages broadcast aux applications. Par exemple, il envoie des notifications lorsque divers événements système se produisent, comme le démarrage du système ou la sonnerie d'une alarme. [6]

**11) Package services:** Ce package contient une seule classe, [NotificationListener](#), qui dérive de [NotificationListenerService](#). Cette classe permet à une application de communiquer avec d'autres applications.

**12) Package utils:** Fait partie du ViewModel (logique d'affaire). Ce package contient des classes utilitaires qui font des tâches connexes qui permettent à l'application de faire sa tâche principale, soit de gérer des notes. Par exemple, [TagsHelper](#) s'occupe tags de note (étiquettes), [KeyboardUtils](#) s'occupe du clavier et [ConnectionManager](#) s'occupe de la connexion internet. Ce package ressemble au package helpers.

**13) Package widgets:** Ce package fait partie de la View, les classes qu'il contient gèrent la fonctionnalité des widgets. Un widget Android est un petit élément de fenêtre interactif qui peut être affiché sur l'écran d'accueil d'un appareil Android. Les widgets permettent d'accéder rapidement à des fonctions ou des informations fréquemment utilisées,

comme la météo actuelle, une liste de tâches ou les derniers titres de l'actualité, sans avoir à lancer une application. [7]

**14) Package res/layout :** Ce package contient les fichiers XML qui représentent la structure de chaque vue d'écran, comme des fichiers HTML en développement web.

**15) Package greenrobot.event.EventBus:** Ce package fait partie de l'architecture [EventBus](#), c'est un package externe qui est une dépendance de l'application. Ce package implémente le bus d'événements, il est utilisé pour communiquer entre les différentes parties de l'application.

1.4 Créer des diagrammes pertinents pour présenter l'architecture du système et les dépendances entre ses packages et ses modules. (Dans Understand vous pouvez trouver les diagrammes : UML Class diagrams, Butterfly-Dependency diagrams, Architecture diagram). (5 point)

Voir les figures 2, 3, 4, 5 et 6 pour des diagrammes pertinents pour présenter l'architecture du système.

## 2. Patrons de conception

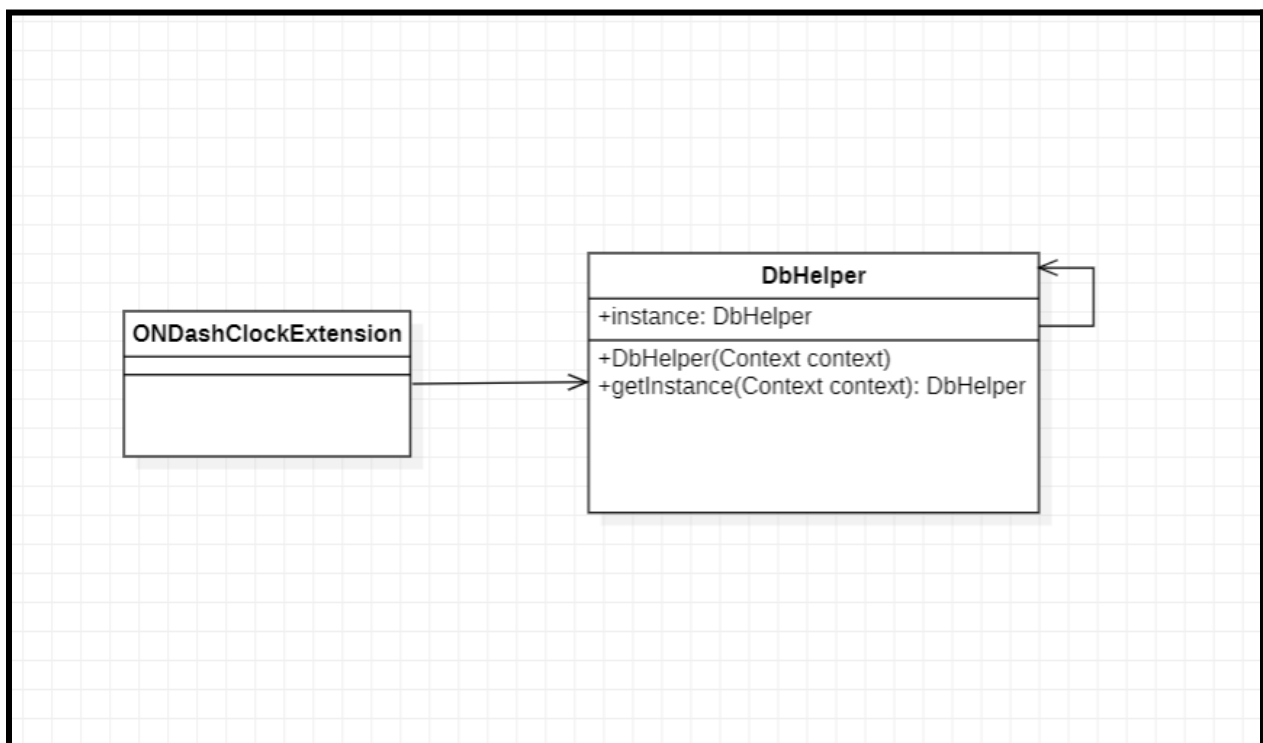
**Note:** Pour chaque patron, nous répondons aux questions 2.1, 2.2 et 2.3.

L'observation du code source de Omni Notes a permis d'identifier trois modèles de patrons .Ces modèles sont la méthode Template, Singleton et la méthode Factory .

Dans la section ci-dessous, chaque patron de conception est présenté par un diagramme UML, suivi de sa fonctionnalité et d'une description du rôle de chaque classe dans le modèle et le diagramme correspondants.

### Singleton

Le premier patron de conception identifié est le patron de conception Singleton qui a été appliqué à la classe DbHelper.java trouvée dans java/it/feio/android/omninotes/db. Pour commencer, un diagramme de classe UML est présenté pour illustrer le patron.



**Figure 9:** Diagramme illustrant le patron singleton

```

public static synchronized DBHelper getInstance(boolean forcedNewInstance) {
    if (instance == null || forcedNewInstance) {
        Context context = (instance == null || instance.mContext == null) ? OmniNotes.getAppContext()
            : instance.mContext;
        instance = new DBHelper(context);
    }
    return instance;
}

private DBHelper(Context mContext) {
    super(mContext, DATABASE_NAME, null, DATABASE_VERSION);
    this.mContext = mContext;
}

```

**Figure 10:** Code de la classe DBHelper

Le modèle de conception singleton est l'un des modèles de création les plus couramment utilisés dans ce projet. Le concept de ce patron qu'une classe ne doit posséder qu'une seule responsabilité. Donc, si une classe à un seul objectif, elle ne devrait avoir qu'une seule raison de changer. Avec ce principe on réduit la dépendance au sein d'une classe, ce qui permet de maintenir un couplage faible.[11]

De la figure ci-dessus, le patron est garanti la création d'une et une seule instance de la classe sur l'ensemble du projet. Ainsi, puisqu'une classe n'a qu'une seule et unique instance, cette seule instance devient un point d'accès global et la même référence est réutilisée partout dans le code source.

De plus, pour vérifier que le patron de conception est bien respecté, nous pouvons voir que la classe "ONDashClockExtension" dans omninotes/extensions/ONDashClockExtension utilise la classe DBHelper. Nous observons que la classe "ONDashClockExtension" appelle la méthode getInstance pour récupérer l'instance globale de la classe DBHelper.

## Factory Method

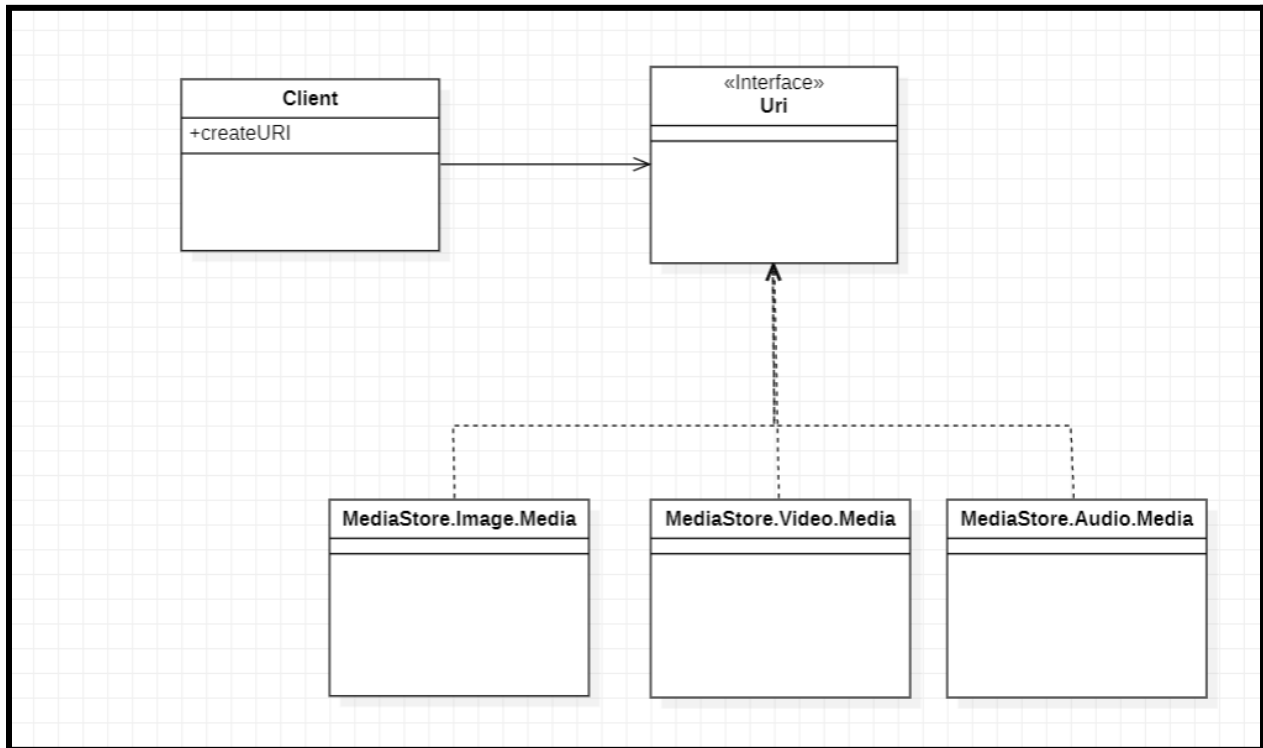


Figure 11 : Diagramme illustrant le patron de conception Factory Method

```
public class MediaStoreFactory {
    public Uri createURI(String type) {
        switch (type) {
            case "image":
                return MediaStore.Images.Media.EXTERNAL_CONTENT_URI;
            case "video":
                return MediaStore.Video.Media.EXTERNAL_CONTENT_URI;
            case "audio":
                return MediaStore.Audio.Media.EXTERNAL_CONTENT_URI;
            default:
                return null;
        }
    }
}
```

Figure 12 : Code de la classe MediaStoreFactory

Le deuxième patron de conception identifié est Factory Method, il est utilisé pour encapsuler la logique de création d'objets dans une classe distincte, appelée "factory", afin de rendre le code plus facile à maintenir et plus extensible.

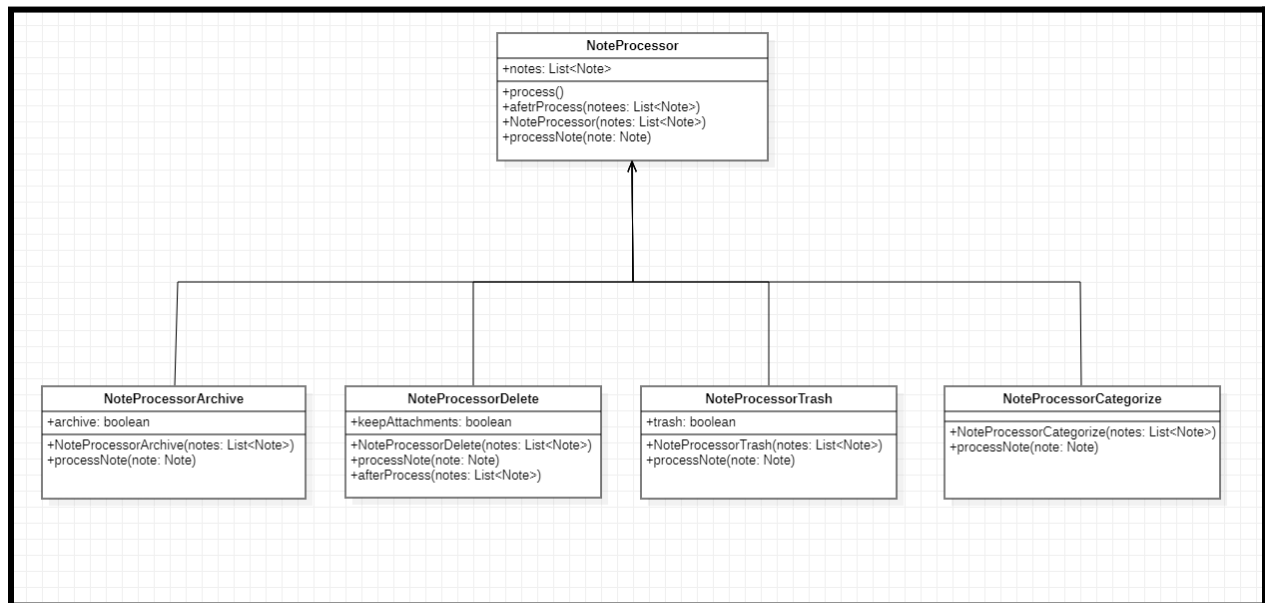
Dans ce cas précis, la classe MediaStoreFactory est utilisée pour créer des URI pour différents types de médias (images, vidéos, audio) à l'aide des classes MediaStore.Images.Media, MediaStore.Video.Media, MediaStore.Audio.Media respectivement. La méthode createURI() prend en compte un type de média (sous forme de chaîne) et renvoie l'URI approprié à l'aide d'une instruction switch/case.

## Template method

Le dernier patron de conception qu'on trouvé est le patron Template-method.

Le patron de conception Template method est un patron comportemental qui permet de définir de manière générale un algorithme de traitement dans une classe abstraite avec des sections de l'algorithme à être définies (implémentées) dans les implémentations de la classe abstraite [10].

Le diagramme UML qui illustre cette relation est disponible ci-dessous.



**Figure 13 :** Diagramme illustrant le patron de conception Template method

La classe abstraite `NoteProcessor` définit les étapes du traitement de la liste d'objets `Note`, notamment:

- La création de la méthode `AsyncTask` in `process()`.
- La mise en œuvre de la méthode `afterProcess()` qui sera appelée après le traitement.

Les sous-classes font l'implémentation concrète de l'étape de traitement en utilisant la méthode abstraite `processNote` (`Note note`). Le chemin pour trouver ce patron est :

`omniNotes\src\main\java\it\feio\android\omninotes\async\notes\.`

```
public abstract class NoteProcessor {

    List<Note> notes;

    protected NoteProcessor(List<Note> notes) {
        this.notes = new ArrayList<>(notes);
    }

    public void process() {
        NotesProcessorTask task = new NotesProcessorTask();
        task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, notes);
    }

    protected abstract void processNote(Note note);

    class NotesProcessorTask extends AsyncTask<List<Note>, Void, List<Note>> {

        @Override
        protected List<Note> doInBackground(List<Note>... params) {
            List<Note> processableNote = params[0];
            for (Note note : processableNote) {
                processNote(note);
            }
            return processableNote;
        }

        @Override
        protected void onPostExecute(List<Note> notes) {
            afterProcess(notes);
        }
    }

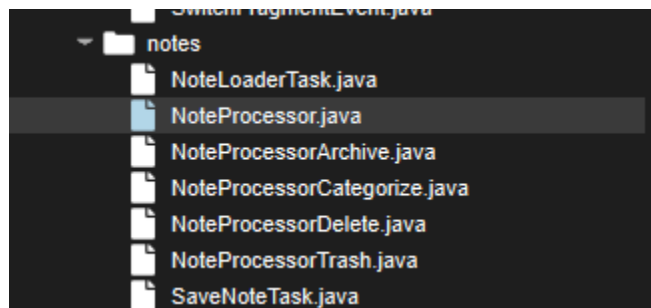
    protected void afterProcess(List<Note> notes) {
        EventBus.getDefault().post(new NotesUpdatedEvent(notes));
    }
}
```

**Figure 14 :** Code de la classe `NoteProcessor`



Les quatres sous-classes qui réagissent avec la superclasse NoteProcessor sont :

- NoteProcessorArchive
- NoteProcessorCategorize
- NoteProcessorDelete
- NoteProcessorTrash



**Figure 15:** Package notes

### NoteProcessorArchive

On commence par la sous-classe “NoteProcessorArchive” qui étend la superclasse “NoteProcessor” et implémente la méthode “processNote(Note note)”, qui est une méthode abstraite définie dans la superclasse.

Son constructeur appelle le constructeur de la superclasse pour définir la liste des “notes” et un paramètre supplémentaire booléen “archive”. La méthode “processNote(Note note)” est surchargée dans cette classe pour archiver ou désarchiver les objets “Note” en appelant la méthode “DbHelper.getInstance().archiveNote(note, archive)”, où “archive” est la valeur booléenne passée dans le constructeur.

```
public class NoteProcessorArchive extends NoteProcessor {  
    boolean archive;  
  
    public NoteProcessorArchive(List<Note> notes, boolean archive) {  
        super(notes);  
        this.archive = archive;  
    }  
  
    @Override  
    protected void processNote(Note note) {  
        DbHelper.getInstance().archiveNote(note, archive);  
    }  
}
```

**Figure 16:** Code de la classe NoteProcessorArchive

## NoteProcessorCategorize

Cette classe définit une autre implémentation concrète de la classe abstraite "NoteProcessor", Elle étend la classe "NoteProcessor" et implémente la méthode "processNote(Note note)" qui est une méthode abstraite définie dans la superclasse aussi.

Son constructeur appelle le constructeur de la superclasse pour définir la liste "notes" et un paramètre "category". La méthode "processNote(Note note)" est surchargée dans cette classe. En appelant la méthode "note.setCategory(category)", La méthode "processNote(Note note)" va être utilisée pour catégoriser les objets "Note", sachant que "category" est la valeur passée dans le constructeur.

En utilisant la méthode "DbHelper.getInstance().updateNote(note, false)", La note va être mise à jour dans la base de données.

```
public class NoteProcessorCategorize extends NoteProcessor {  
    Category category;  
  
    public NoteProcessorCategorize(List<Note> notes, Category category) {  
        super(notes);  
        this.category = category;  
    }  
  
    @Override  
    protected void processNote(Note note) {  
        note.setCategory(category);  
        DbHelper.getInstance().updateNote(note, false);  
    }  
}
```

**Figure 17:** Code de la classe NoteProcessorCategorize

## NoteProcessorDelete

Cette classe définit une autre implémentation concrète de la classe abstraite "NoteProcessor", et étend la classe abstraite "NoteProcessor" aussi. Elle implémente la méthode "processNote(Note)" et la méthode "afterProcess(List<Note> notes)" de la superclasse. La tâche principale de cette classe est de supprimer les notes.

La classe a deux constructeurs. Le premier prend une liste de notes et met "keepAttachments" à false. Le second constructeur prend deux arguments, une liste de notes et un booléen qui indique s'il faut conserver des pièces jointes "keepAttachments" ou non.

Dans la méthode "processNote(Note note)", la note est d'abord supprimée de la base de données en utilisant la méthode "deleteNote" de la classe DbHelper. Si "keepAttachments" est faux, alors toutes les pièces jointes associées à la note sont également supprimées du stockage externe.

Dans la méthode "afterProcess(List<Note> notes)", "NotesDeletedEvent" est affiché sur le Event-bus pour notifier les autres composants des notes supprimées.

```

public class NoteProcessorDelete extends NoteProcessor {

    private final boolean keepAttachments;

    public NoteProcessorDelete(List<Note> notes) {
        this(notes, false);
    }

    public NoteProcessorDelete(List<Note> notes, boolean keepAttachments) {
        super(notes);
        this.keepAttachments = keepAttachments;
    }

    @Override
    protected void processNote(Note note) {
        DbHelper db = DbHelper.getInstance();
        if (db.deleteNote(note) && !keepAttachments) {
            for (Attachment mAttachment : note.getAttachmentsList()) {
                StorageHelper
                    .deleteExternalStoragePrivateFile(OmniNotes.getAppContext(), mAttachment.getUri()
                        .getLastPathSegment());
            }
        }
    }

    @Override
    protected void afterProcess(List<Note> notes) {
        EventBus.getDefault().post(new NotesDeletedEvent(notes));
    }
}

```

**Figure 18:** Code de la classe NoteProcessorDelete

## NoteProcessorTrash

Cette classe est utilisée pour déplacer/restaurer des notes vers/à partir de la corbeille.

Le constructeur “NoteProcessorTrash” prend une liste de “notes” et un flag booléen “trash”.

La méthode “processNote” remplace la méthode de la classe parent et met à jour le statut trash d’un objet “Note” en appelant “DbHelper.getInstance().trashNote(note, trash)”.

Si le drapeau (flag) de “trash” est true, alors les méthodes “ShortcutHelper.removeShortcut” et “ReminderHelper.removeReminder” sont appelées pour supprimer le raccourci et le rappel liés à l’objet “Note”. Si l’indicateur “trash” est false, la méthode “ReminderHelper.addReminder” est appelée pour ajouter le rappel à l’objet “Note”.

Ces classes démontrent l'utilisation de l'héritage pour fournir une autre implémentation concrète des étapes de traitement définies dans la superclasse. Les implémentations concrètes ("NoteProcessorArchive", "NoteProcessorCategorize", "NoteProcessorDelete" et "NoteProcessorTrash") étendent la classe "NoteProcessor" et peuvent être utilisées indépendamment pour effectuer différentes tâches de traitement sur les objets "Note" comme les archiver, categorizer, supprimer et les déplacer.

```
public class NoteProcessorTrash extends NoteProcessor {  
    boolean trash;  
  
    public NoteProcessorTrash(List<Note> notes, boolean trash) {  
        super(notes);  
        this.trash = trash;  
    }  
  
    @Override  
    protected void processNote(Note note) {  
        if (trash) {  
            ShortcutHelper.removeShortcut(OmniNotes.getAppContext(), note);  
            ReminderHelper.removeReminder(OmniNotes.getAppContext(), note);  
        } else {  
            ReminderHelper.addReminder(OmniNotes.getAppContext(), note);  
        }  
        DbHelper.getInstance().trashNote(note, trash);  
    }  
}
```

**Figure 19:** Code de la classe NoteProcessorTrash

### 3. Principes SOLID

**Note:** pour chaque principe SOLID nous répondons aux questions 3.1, 3.2 et 3.3.

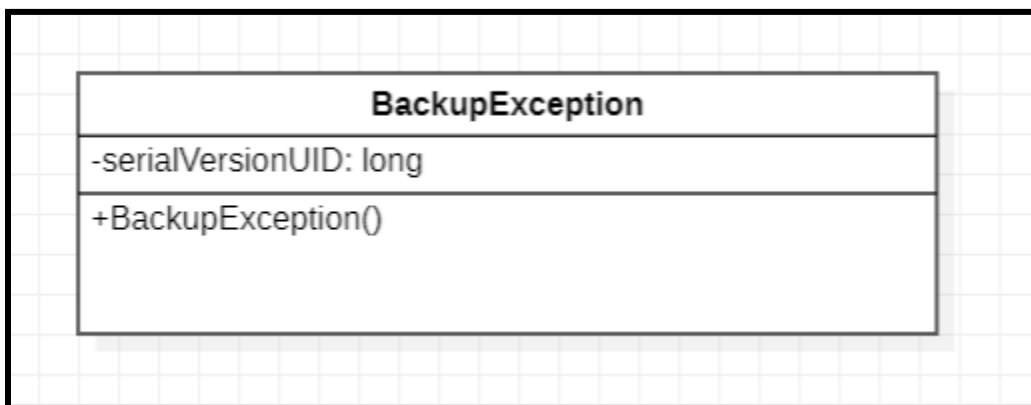
SOLID est un acronyme pour les 5 principes de conception de logiciels orientés objet :

- Principe de Responsabilité Unique (SRP)
- Principe Ouvert/Fermé (OCP)
- Principe de Substitution de Liskov (LSP)
- Principe de Ségrégation d'interface (ISP)
- Principe d'inversion de dépendance (DIP)

Dans cette section, nous étudions trois différents types de principes de conception SOLID, qui sont:

- Le principe de responsabilité unique,
- Le principe de substitution de Liskov
- Le principe de ouvert et fermé.

#### Single Responsibility principle



**Figure 20:** Diagramme de la classe BackupException

```

public class BackupException extends RuntimeException {

    private static final long serialVersionUID = 7892197590810157669L;

    public BackupException(String message, Exception e) {
        super(message, e);
    }

}

```

**Figure 21:** Code de la classe BackupException

La classe BackupException qui se trouve à “omniNotes\src\main\java\it\feio\android\omninotes\exceptions\BackupException.java” respecte le principe Single Responsibility (SRP) par ce que elle a seulement une seule raison pour se changer, qui consiste à fournir une exception personnalisée pour les erreurs liées à la sauvegarde (BackupException ).

## Liskov Substitution Principle

Nous avons observé le principe de Substitution de Liskov (LSP), qui dit, on peut remplacer les objets d’une superclasse par des objets de ses sous-classes sans impacter le reste du système. Autrement, les objets de nos sous-classes doivent se comportent de la même manière que les objets de notre superclasse. [8]

Pour ce principe nous allons donner comme exemple de la superclasse “WidgetProvider” et de ses sous-classes “ListWidgetProvider” et “SimpleWidgetProvider”. qui se trouve sur le path omniNotes\src\main\java\it\feio\android\omninotes\widget\WidgetProvider.java

“WidgetProvider” déclare la méthode “getRemoteViews” qui retourne un objet “RemoteViews” qui sert de conteneur pour la mise en page du widget. Pour mettre à jour l’interface utilisateur du widget, RemoteViews est envoyé en réponse à une demande du framework Android

“WidgetProvider” est une classe qui fournit les informations nécessaires pour créer et afficher un widget sur l’écran d’accueil d’un appareil Android.

La superclasse “WidgetProvider” est étendue par “SimpleWidgetProvider”. Pour retourner un objet RemoteViews la méthode getRemoteViews est surchargée. L’objet remoteViews représente la disposition du widget. qui dépend du paramètre isSmall. Si isSmall est vrai, la disposition R.layout.widget\_layout\_small sera utilisée, sinon c’est R.layout.widget\_layout qui sera utilisé.

La superclasse “WidgetProvider” est étendue par La classe “ListWidgetProvider”. Elle est responsable de la création et de la configuration des “RemoteViews” du widget. En fonction des paramètres transmis (widget size, layout style), la classe initialise une mise en page “RemoteViews” spécifique et définit des intents en attente pour des vues spécifiques dans la mise en page du widget.

Finalement, les classes “ListWidgetProvider” et “SimpleWidgetProvider” devraient pouvoir remplacer la superclasse “WidgetProvider” sans provoquer de comportements ou d'erreurs involontaires. Le fait que “ListWidgetProvider” et “SimpleWidgetProvider” étendent la classe “WidgetProvider” et surchargent “override” la méthode “getRemoteViews” d'une manière qui honore les contrats établis par la superclasse signifie que ces classes respectent le LSP.

```
public class SimpleWidgetProvider extends WidgetProvider {

    @Override
    protected RemoteViews getRemoteViews(Context mContext, int widgetId, boolean isSmall,
        boolean isSingleline,
        SparseArray<PendingIntent> pendingIntentsMap) {
        RemoteViews views;
        if (isSmall) {
            views = new RemoteViews(mContext.getPackageName(), R.layout.widget_layout_small);
            views.setOnClickPendingIntent(R.id.list, pendingIntentsMap.get(R.id.list));
        } else {
            views = new RemoteViews(mContext.getPackageName(), R.layout.widget_layout);
            views.setOnClickPendingIntent(R.id.add, pendingIntentsMap.get(R.id.add));
            views.setOnClickPendingIntent(R.id.list, pendingIntentsMap.get(R.id.list));
            views.setOnClickPendingIntent(R.id.camera, pendingIntentsMap.get(R.id.camera));
        }
        return views;
    }
}
```

**Figure 22:** Code de la sous-classe SimpleWidegetProvider



```

public class ListWidgetProvider extends WidgetProvider {

    @Override
    protected RemoteViews getRemoteViews(Context mContext, int widgetId,
        boolean isSmall, boolean isSingleLine,
        SparseArray<PendingIntent> pendingIntentsMap) {
        RemoteViews views;
        if (isSmall) {
            views = new RemoteViews(mContext.getPackageName(),
                R.layout.widget_layout_small);
            views.setOnClickPendingIntent(R.id.list,
                pendingIntentsMap.get(R.id.list));
        } else if (isSingleLine) {
            views = new RemoteViews(mContext.getPackageName(),
                R.layout.widget_layout);
            views.setOnClickPendingIntent(R.id.add,
                pendingIntentsMap.get(R.id.add));
            views.setOnClickPendingIntent(R.id.list,
                pendingIntentsMap.get(R.id.list));
            views.setOnClickPendingIntent(R.id.camera,
                pendingIntentsMap.get(R.id.camera));
        } else {
            views = new RemoteViews(mContext.getPackageName(),
                R.layout.widget_layout_list);
            views.setOnClickPendingIntent(R.id.add,
                pendingIntentsMap.get(R.id.add));
            views.setOnClickPendingIntent(R.id.list,
                pendingIntentsMap.get(R.id.list));
            views.setOnClickPendingIntent(R.id.camera,
                pendingIntentsMap.get(R.id.camera));

            // Set up the intent that starts the ListViewService, which will
            // provide the views for this collection.
            Intent intent = new Intent(mContext, ListWidgetService.class);
            // Add the app widget ID to the intent extras.
            intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, widgetId);
            intent.setData(Uri.parse(intent.toUri(Intent.URI_INTENT_SCHEME)));

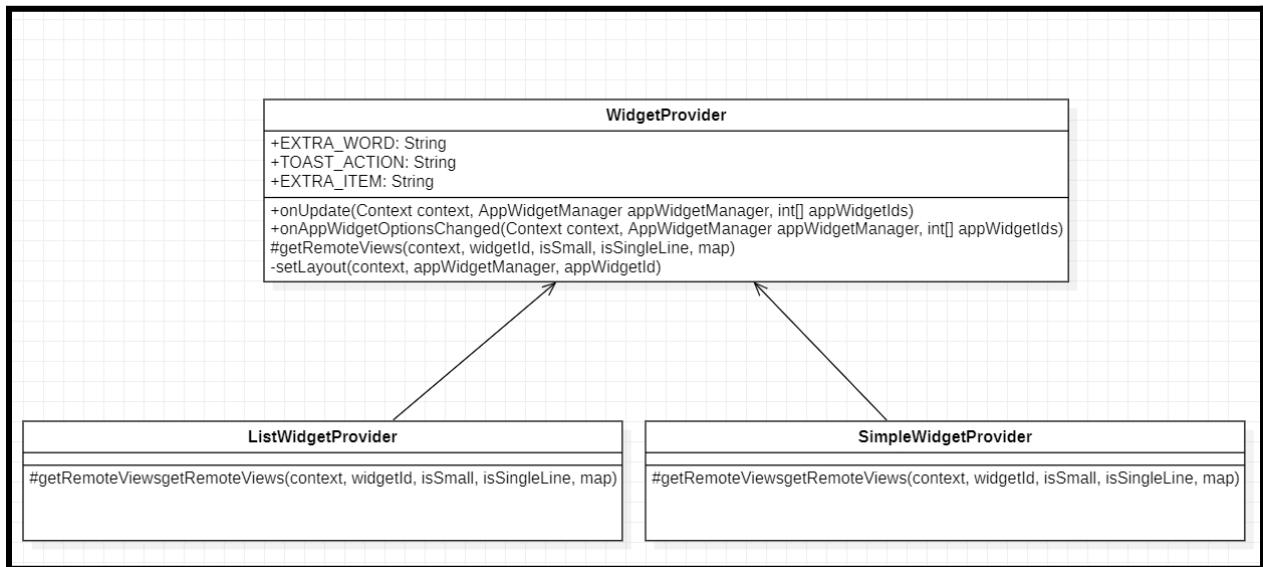
            views.setRemoteAdapter(R.id.widget_list, intent);

            Intent clickIntent = new Intent(mContext, MainActivity.class);
            clickIntent.setAction(ACTION_WIDGET);
            PendingIntent clickPI = PendingIntent.getActivity(mContext, 0,
                clickIntent, mutablePendingIntentFlag(FLAG_UPDATE_CURRENT));

            views.setPendingIntentTemplate(R.id.widget_list, clickPI);
        }
        return views;
    }
}

```

Figure 23: Code de la sous-classe ListWidgetProvider

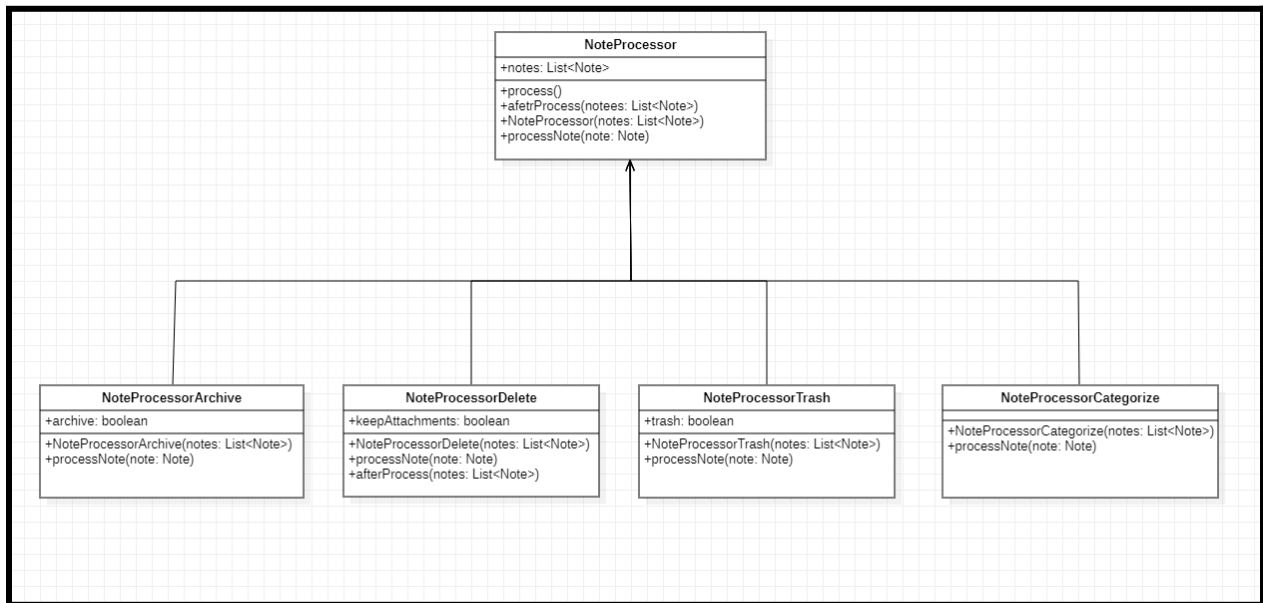


**Figure 24:** Diagramme de la classe WidgetProvide et de ses dérivées

## Principe Ouvert/Fermé (OCP)

Le troisième principe que nous avons identifié est le principe ouvert-fermé (OCP). S'il fallait le résumer en une phrase, ce serait que les entités logicielles (telles que les classes, les fonctions ou les modules) doivent être ouvertes à l'extension mais fermées à la modification.

En d'autres termes, le code doit être écrit de manière que nous puissions ajouter de nouvelles fonctionnalités au lieu de modifier le code existant. Comme cela crée une certaine séparation entre le code existant et le code modifié, nous évitons de déclencher ou de créer des erreurs potentielles et casser le programme. Nous allons examiner le groupe de classes `NoteProcessor`. (voir **Figure 25**) pour démontrer l'utilisation de OCP.



**Figure 25:** Diagramme de la classe NoteProcessor et de ses dérivées.

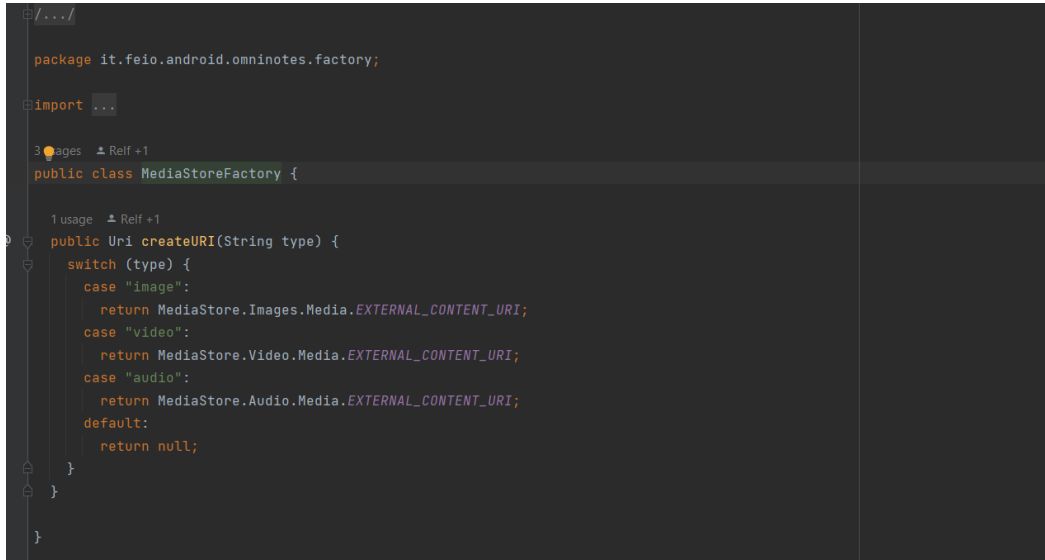
Le principe est représenté par les 4 classes du bas qui implémentent NoteProcessor. On remarque que chaque classe enfant implémente sa propre version de la fonction processNote. Nous voyons également un type attribut similaire entre les enfants « type boolean », qui change le nom en fonction de la classe dont elle provient. Ces classes de listes de lecture sont similaires entre elles mais sont utilisées dans l'application pour des raisons différentes. Les enfants de NoteProcessor sont tous des listes du traitement d'une liste d'objets Note, mais ils sont créés automatiquement par le système en fonction de certaines actions de l'utilisateur. Par exemple, la liste NoteProcessorDelete est construite sur ce que les utilisateurs peuvent supprimer les notes en fonction de l'interaction de l'utilisateur avec l'application.

## 4. Violation des principes SOLID

Les principes SOLID posent plusieurs restrictions pour les applications, mais la majorité des applications ne suivent pas ces principes à la lettre en pratique, car ces principes ne sont pas universels et peuvent complexifier l'implémentation pour rien. En analysant l'application Omni Notes, on peut confirmer qu'il y a en effet la présence de violations des principes SOLID. Un des principes de SOLID qui n'est pas respecté par l'application est le "Open-Closed principe".

Ce principe stipule que les entités logicielles doivent être ouvertes aux extensions et fermées aux modifications. Dans la **Figure 26**, on peut voir que dans la classe "MediaStoreFactory", la valeur de retour de la méthode "createURI" dépend du paramètre type donné en paramètre.

Dans la **Figure 26**, on peut voir que la classe `MediaStoreFactory` n'a qu'une seule méthode. On peut voir qu'à chaque fois qu'un nouveau type est ajouté, la méthode est modifiée à chaque fois ce qui ne respecte pas le "Open-Closed principe", car on veut avoir le moins de modifications possibles dans le code existant et seulement ajouter du code.



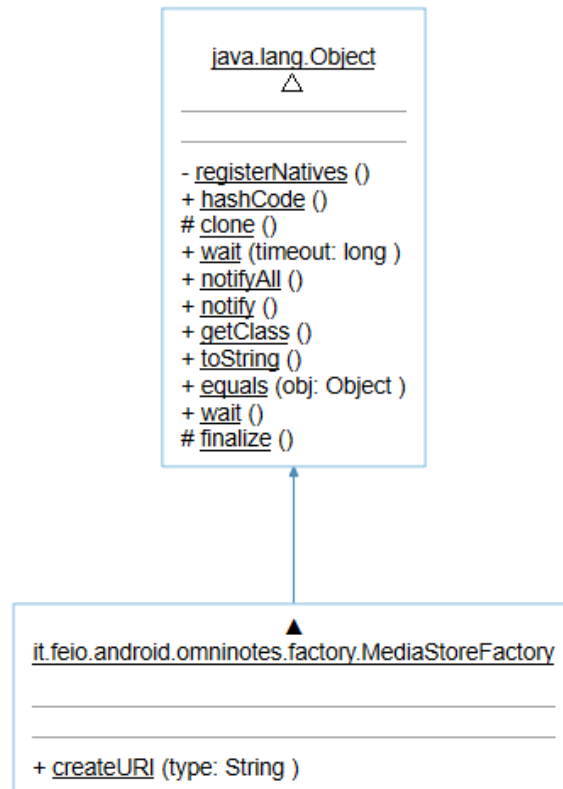
```
package it.feio.android.omninetes.factory;

import ...

public class MediaStoreFactory {

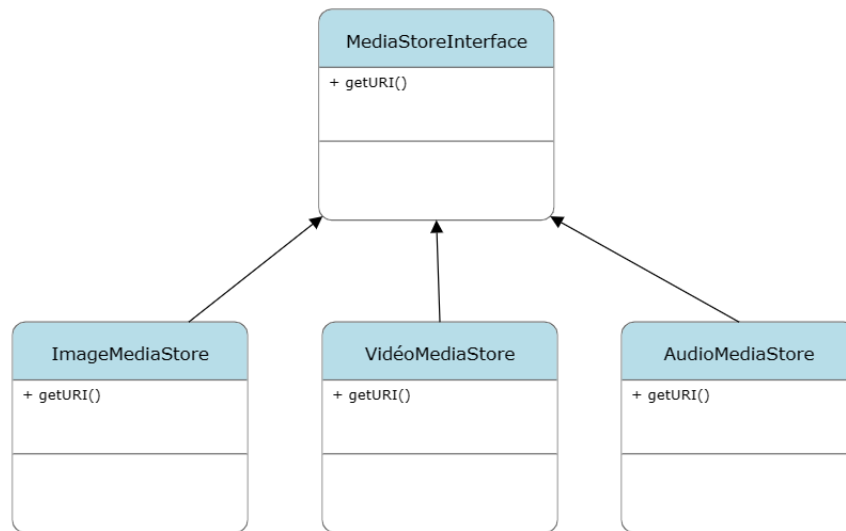
    public Uri createURI(String type) {
        switch (type) {
            case "image":
                return MediaStore.Images.Media.EXTERNAL_CONTENT_URI;
            case "video":
                return MediaStore.Video.Media.EXTERNAL_CONTENT_URI;
            case "audio":
                return MediaStore.Audio.Media.EXTERNAL_CONTENT_URI;
            default:
                return null;
        }
    }
}
```

**Figure 26:** La méthode `createURI` qui viole le Open-Closed principe



**Figure 27:** La classe `MediaStoreFactory` qui viole le Open-Closed principe

Pour respecter ce principe, on peut apporter des modifications à l'implémentation. On peut commencer par créer une interface de base "`MediaStoreInterface`" où tous les types vont hériter de cette interface. Chaque type va implémenter la méthode "`getURI`" de cette interface. Enfin, dans la méthode "`createURI`" de la classe "`MediaStoreFactory`" on a simplement à appeler la méthode "`getURI`" de l'objet de type "`MediaStoreInterface`" passé en paramètres. Avec les modifications dans la **Figure 27**, l'ajout d'un nouveau type n'affecte pas la méthode "`createURI`", on va simplement créer une nouvelle classe qui implémente l'interface de base avec sa propre méthode "`getURI`".



**Figure 28:** Les classes de type de média qui implémente l'interface de base

```
public interface MediaStoreTypeInterface {
    public Uri getURI() {
        return default;
    }
}
```

```
public class ImageMediaStore implements MediaStoreTypeInterface {
    @Override
    public Uri getURI() {
        return MediaStore.Images.Media.EXTERNAL_CONTENT_URI;
    }
}
```

```
public class VideoMediaStore implements MediaStoreTypeInterface {
    @Override
    public Uri getURI() {
        return MediaStore.Video.Media.EXTERNAL_CONTENT_URI;
    }
}
```

```

public class AudioMediaStore implements MediaStoreTypeInterface {
    @Override
    public Uri getURI() {
        return MediaStore.Audio.Media.EXTERNAL_CONTENT_URI;
    }
}

```

```

public class MediaStoreFactory {
    new *
    public Uri createURI(MediaStoreTypeInterface type) {
        return type.getURI();
    }
}

```

**Figure 29:** Les modifications à apporter au code

## Références

- [1] <https://www.oreilly.com/library/view/hands-on-software-architecture/9781788622592/2310cf5c-3faa-409a-9c52-7e4ccf1e382a.xhtml>
- [2] <https://www.oreilly.com/library/view/learning-javascript-design/9781449334840/ch10s06.html>
- [3] <https://www.oreilly.com/library/view/c-reactive-programming/9781788629775/4d5f576b-cf55-4106-ab4f-bde3a623b2a1.xhtml>
- [4] <https://greenrobot.org/eventbus/>
- [5] <https://developer.android.com/training/data-storage/shared/media>
- [6] <https://developer.android.com/guide/components/broadcasts>
- [7] <https://developer.android.com/develop/ui/views/appwidgets/overview>

[8] <https://blog.knoldus.com/what-is-liskov-substitution-principle-lsp-with-real-world-examples/#:~:text=Simply%20put%2C%20the%20Liskov%20Substitution,the%20objects%20of%20our%20superclass.>

[9] <https://firebase.google.com/docs/reference/android/io/fabric/sdk/android/fabric/services/concurrency/AsyncTask#:~:text=This%20class%20allows%20to%20perform,constitute%20a%20generic%20threading%20framework.>

[10] <https://inf1420.teluq.ca/semaine-12/activite-2-le-patron-de-conception-template/>

[11] <https://www.geeksforgeeks.org/singleton-design-pattern/>