LOG8430 - Architecture logicielle et conception avancée

Travail Pratique #1 Principe et patron de conception

Équipe: Atwater

1871842-Ben Jemaa Manel

2024717- Henri Larocque

2051212 - Carrie Kam

1621855 - Ahmed Gafsi

2065803 – Vincent Boogaart

Remis à George Salib

Hiver 2024

Table des matières

Introduction

La conception de logiciels efficaces repose souvent sur des principes et des modèles bien établis. Dans cette analyse, nous explorerons les principes SOLID, un ensemble de cinq principes de conception de logiciel visant à créer des systèmes plus flexibles et évolutifs. Nous examinerons également des violations potentielles de ces principes dans le projet Timber, ainsi que l'application de patrons de conception et une analyse de son architecture logicielle. Cette approche méthodique nous permettra de mieux comprendre la structure et les défis de ce projet.

1. Principes SOLID

Les principes SOLID sont un ensemble de cinq principes de conception de logiciel qui ont été formulés pour créer des systèmes logiciels plus flexibles, compréhensibles et évolutifs. L'acronyme SOLID représente les cinq principes fondamentaux: Principe de Responsabilité Unique (SRP), Principe Ouvert/Fermé (OCP), Principe de Substitution de Liskov (LSP), Principe de Ségrégation d'interface (ISP) et Principe d'inversion de dépendance (DIP).

Dans cette section, nous étudions trois différents types de principes de conception SOLID, qui sont: Principe de Responsabilité unique (SRP), Principe Ouvert/Fermé (OCP) et Principe de Substitution de Liskov (LSP).

1.1 Principe de Responsabilité unique (SRP)

Le SRP stipule qu'une classe doit avoir une seule responsabilité. Analysons la classe *QueueLoader* afin de vérifier si elle respecte ce principe.

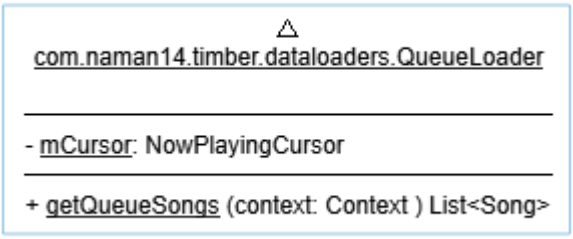

Figure 1.1: Diagramme de classe UML du QueueLoader

La classe *QueueLoader* a une seule responsabilité: charger les chansons de la file d'attente. Elle ne s'occupe pas d'autres tâches, telles que la lecture des chansons, la gestion de la file d'attente ou l'affichage des informations sur les chansons. Elle ne possède qu'une seule méthode publique, *getQueueSongs(Context)*, qui correspond à sa seule responsabilité. Ainsi, toutes les opérations effectuées dans la classe sont directement liées à la responsabilité de charger les chansons de la file d'attente. Il n'y a pas d'opérations qui relèvent d'autres responsabilités.

En fonction de cette analyse, la classe *QueueLoader* semble bien respecter le SRP. Elle a une responsabilité claire et spécifique, ce qui rend la classe facile à comprendre, à maintenir et à adapter aux changements futurs.

1.2 Principe d'ouverture/fermeture (OCP)

Le principe d'ouverture/fermeture (OCP) est un concept fondamental en développement logiciel. Il stipule que les entités logicielles (classes, fonctions, modules) doivent être ouvertes à l'extension, mais fermées à la modification. En d'autres termes, une classe devrait pouvoir être étendue pour ajouter de nouvelles fonctionnalités sans modifier le code source existant.

La classe *MusicPlaybackTrack* respecte le principe d'ouverture/fermeture (OCP). En effet, la classe permet d'étendre ses fonctionnalités sans modifier son code source existant. Cela est réalisé en fournissant plusieurs constructeurs pour initialiser l'objet avec différentes configurations. Par exemple, le constructeur *MusicPlaybackTrack(long id, long sourceId, TimberUtils.IdType type, int sourcePosition)* est utilisé pour créer une instance de *MusicPlaybackTrack* en spécifiant l'ID, la source, le type et la position. De nouveaux constructeurs peuvent être ajoutés pour prendre en charge de nouvelles configurations sans modifier les constructeurs existants. De plus, en fournissant un *Creator* statique pour la classe *Parcelable* comme le montre la figure 1.2, la classe offre la possibilité d'être étendue pour de futures fonctionnalités sans avoir à modifier la classe elle-même. De nouveaux types de *Parcelable* pourraient être ajoutés sans altérer le comportement existant de *MusicPlaybackTrack*. Enfin, le comportement de la classe peut être étendu en ajoutant de nouveaux constructeurs ou en modifiant les constructeurs existants pour prendre en charge de nouvelles fonctionnalités sans compromettre la cohérence ou la stabilité du code existant. Par exemple, si de nouvelles fonctionnalités liées à la lecture de musique sont introduites, de nouveaux paramètres pourraient être ajoutés aux constructeurs existants ou de nouveaux constructeurs pourraient être créés pour gérer ces cas spécifiques.

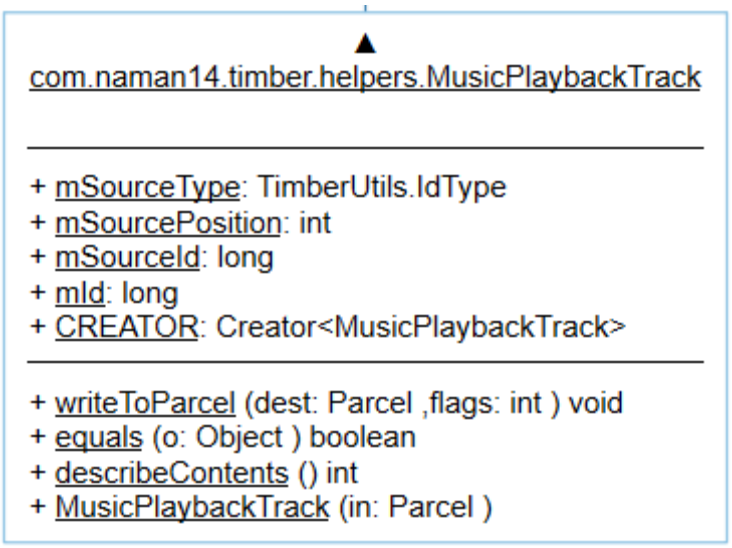

Figure 1.2: Diagramme de classe MusicPlaybackTrack

En résumé, la classe *MusicPlaybackTrack* illustre bien le principe d'ouverture/fermeture en permettant l'extension de ses fonctionnalités sans modification de son code source existant, ce qui favorise la maintenabilité et l'évolutivité du système logiciel.

1.3 Principe de substitution de Liskov (LSP)

Le principe de substitution de Liskov (LSP) stipule que nous pouvons remplacer un objet d'une superclasse par un objet de sa sous-classe sans affecter le reste du système. Autrement, les objets de notre sous-classe doivent se comporter de la même manière que les objets de notre superclasse. [3]

Dans le cadre de l'analyse de la classe *SimpleITransitionListener* et de ses sous-classes, nous pouvons observer la conformité au LSP. En effet, l'observation de la figure 1.3 des méthodes dans la classe de base *SimpleITransitionListener* et dans ses sous-classes révèle que les signatures des méthodes sont identiques. Cela signifie que les sous-classes peuvent être utilisées partout où une instance de la classe de base est attendue, car elles offrent la même interface que la classe de base. Par exemple, les méthodes *onTransitionEnd* et *onTransitionStart* ont les mêmes signatures dans la classe de base et dans ses sous-classes. De plus, ces classes ne lèvent pas d'exceptions qui ne sont pas prévues par la classe de base. Cela assure une cohérence dans le comportement des sous-classes lorsqu'elles sont utilisées dans des contextes

où la classe de base est attendue. Ainsi, le code qui utilise la classe de base peut traiter les instances des sous-classes de manière uniforme, sans risque d'exceptions inattendues.

Les méthodes *onTransitionStart* et *onTransitionEnd* ont un comportement identique dans les sous-classes lorsqu'elles sont utilisées. Cela signifie que le comportement défini dans la classe de base est respecté dans toutes les sous-classes, conformément au principe de substitution de Liskov. La cohérence du comportement garantit que l'utilisation des sous-classes ne perturbera pas le fonctionnement attendu du système.

En résumé, la classe *SimpleITransitionListener* et ses sous-classes respectent pleinement le principe de substitution de Liskov, ce qui garantit une hiérarchie de classes cohérente et prévisible. Cette conformité facilite l'extension du système, la réutilisation du code et la maintenance à long terme, en assurant que les modifications apportées aux sous-classes n'affectent pas le comportement des parties du code qui utilisent la classe de base.

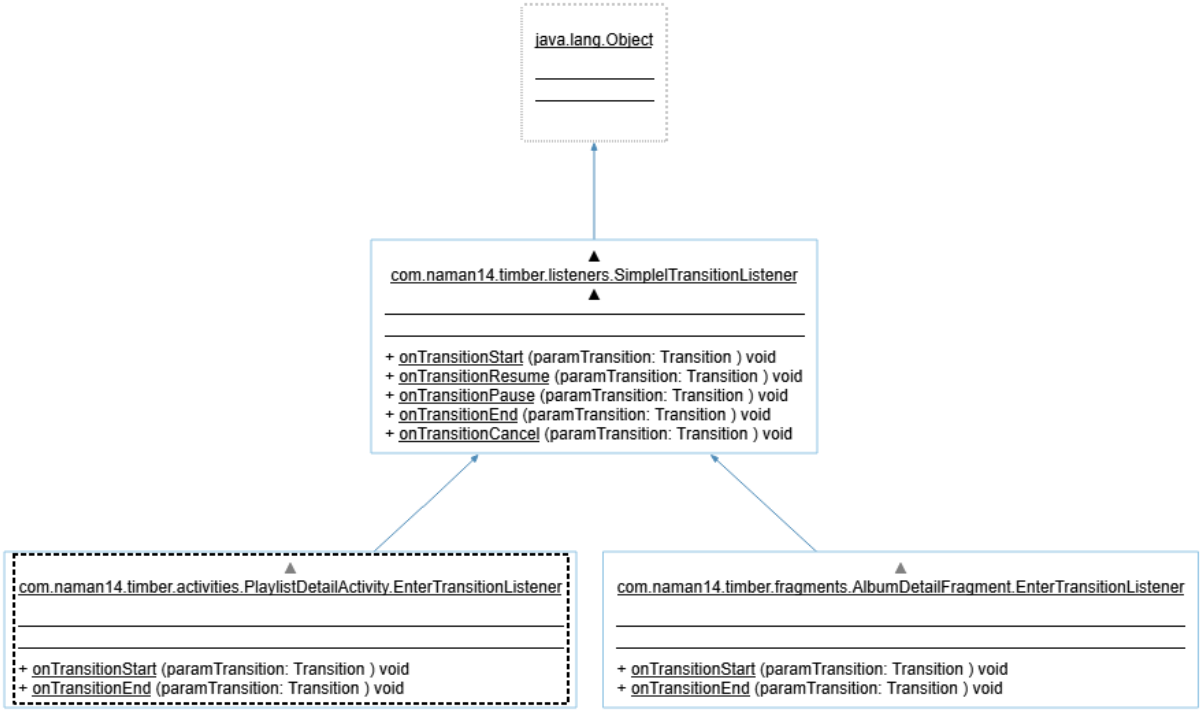

Figure 1.3: Diagramme de classe SimpleITransitionListener

2. Violation des principes SOLID

2.1 Violation d'un principe SOLID dans Timber

La majorité du temps, les principes SOLID sont bien appliqués dans le projet Timber. Par contre, ce n'est pas le cas pour tout le projet. Par exemple, dans la classe *TextDrawable*, que nous pouvons voir dans la figure ci-dessous sous forme de diagramme UML, l'*Open-Closed Principle* (OCP) n'est pas respecté:

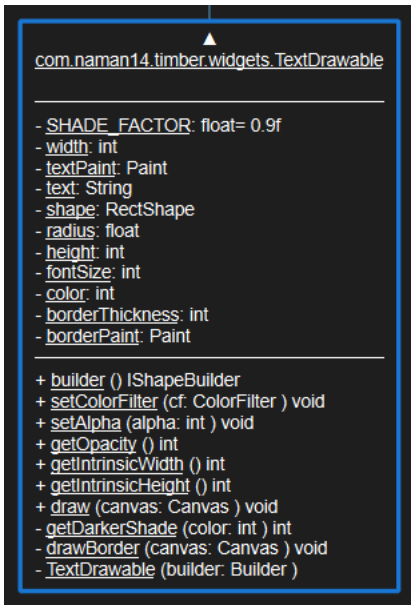

Figure 2.1: Diagramme UML de la classe TextDrawable

2.2 Justification de la violation du principe SOLID

La lettre « O » du principe SOLID représente l'*Open-Closed Principle (OCP)*. Pour suivre ce principe, il faut qu'une entité soit ouverte aux extensions, mais fermée aux modifications. Ici, la méthode *drawBorder()* de la classe *TextDrawable* ne permet pas facilement l'extension de nouvelles formes de bordure sans modifier le code existant. Actuellement, la méthode *drawBorder* utilise des vérifications de type spécifiques (*OvalShape*, *RoundRectShape*, etc.) pour déterminer le type de forme de bordure et effectuer le dessin correspondant. Si une nouvelle forme de bordure doit être ajoutée, il serait nécessaire de modifier directement cette méthode, ce qui viole le principe *Open-Closed*.

```java
private void drawBorder(Canvas canvas) {
    RectF rect = new RectF(getBounds());
    rect.inset(borderThickness / 2, borderThickness / 2);

    if (shape instanceof OvalShape) {
        canvas.drawOval(rect, borderPaint);
    } else if (shape instanceof RoundRectShape) {
        canvas.drawRoundRect(rect, radius, radius, borderPaint);
    } else {
        canvas.drawRect(rect, borderPaint);
    }
}
```

Figure 2.2: Code de la méthode drawBorder

La méthode *drawBorder* est appelée par cette méthode:

```java
@Override
public void draw(Canvas canvas) {
    super.draw(canvas);
    Rect r = getBounds();


    // draw border
    if (borderThickness > 0) {
        drawBorder(canvas);
    }

    int count = canvas.save();
    canvas.translate(r.left, r.top);

    // draw text
    int width = this.width < 0 ? r.width() : this.width;
    int height = this.height < 0 ? r.height() : this.height;
    int fontSize = this.fontSize < 0 ? (Math.min(width, height) / 2) : this.fontSize;
    textPaint.setTextSize(fontSize);
    canvas.drawText(text, width / 2, height / 2 - ((textPaint.descent() + textPaint.ascent()) / 2), textPaint);

    canvas.restoreToCount(count);

}
```

Figure 2.3: Méthode draw qui appelle la méthode drawBorder

2.3 Correction de la violation du principe SOLID

Afin de respecter le principe OCP, il est essentiel que le code soit facilement extensible sans nécessiter de modifications. Par conséquent, en subdivisant la méthode en trois parties distinctes, cela facilitera l'ajout d'extensions, telles que le dessin de bordures pour un losange, sans avoir besoin de modifier le code au sein d'une méthode. Auparavant, des modifications étaient nécessaires dans la méthode *drawBorder* pour effectuer des changements, mais désormais, cela n'est plus requis. Toutes les méthodes peuvent être appelées en une seule fois dans la méthode *draw*.

```
private void drawBorderOval(Canvas canvas) {
    RectF rect = new RectF(getBounds());
    rect.inset(borderThickness / 2, borderThickness / 2);
    if (shape instanceof OvalShape) {
        canvas.drawOval(rect, borderPaint);
    }
}
```

```
private void drawBorderRoundRect(Canvas canvas) {
    RectF rect = new RectF(getBounds());
    rect.inset(borderThickness / 2, borderThickness / 2);
    if (shape instanceof RoundRectShape) {
        canvas.drawRoundRect(rect, radius, radius, borderPaint);
    }
}
```

```
private void drawBorderRect(Canvas canvas) {
    RectF rect = new RectF(getBounds());
    rect.inset(borderThickness / 2, borderThickness / 2);
    if (!(shape instanceof RoundRectShape || shape instanceof OvalShape)) {
        canvas.drawRect(rect, borderPaint);
    }
}
```

Figure 2.4: Méthodes drawBorderOval, drawBorderRoundRect et drawBorderRect qui remplacent drawBorder

```java
@Override
public void draw(Canvas canvas) {
    super.draw(canvas);
    Rect r = getBounds();


    // draw border
    if (borderThickness > 0) {
        drawBorderOval(canvas);
        drawBorderRoundRect(canvas);
        drawBorderRect(canvas);
    }

    int count = canvas.save();
    canvas.translate(r.left, r.top);

    // draw text
    int width = this.width < 0 ? r.width() : this.width;
    int height = this.height < 0 ? r.height() : this.height;
    int fontSize = this.fontSize < 0 ? (Math.min(width, height) / 2) : this.fontSize;
    textPaint.setTextSize(fontSize);
    canvas.drawText(text, width / 2, height / 2 - ((textPaint.descent() + textPaint.ascent()) / 2), textPaint);
    canvas.restoreToCount(count);

}
```

Figure 2.5: Méthode draw modifiée qui appelle les nouvelles méthodes

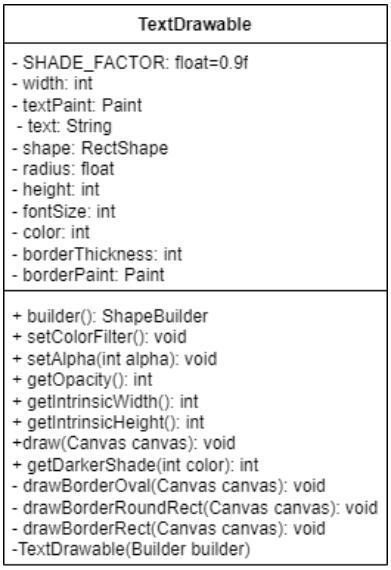

Figure 2.6: Diagramme UML de la classe TextDrawable modifiée

3. Patrons de conception

Les patrons de conceptions offrent des solutions à des problèmes typiques rencontrés lors de la conception logicielle et permettent aux développeurs de communiquer efficacement en utilisant un langage commun. Le projet Timber utilise de nombreux patrons de conceptions, nous allons en identifier 3 ici, soit les patrons Singleton, Observateur et patron de méthode.

3.1 Singleton

Le Singleton est un patron de conception qui garantit qu'il n'y ait qu'une seule instance d'une classe dans une application. Cela peut être utile pour gérer des ressources partagées, comme la configuration ou les journaux.

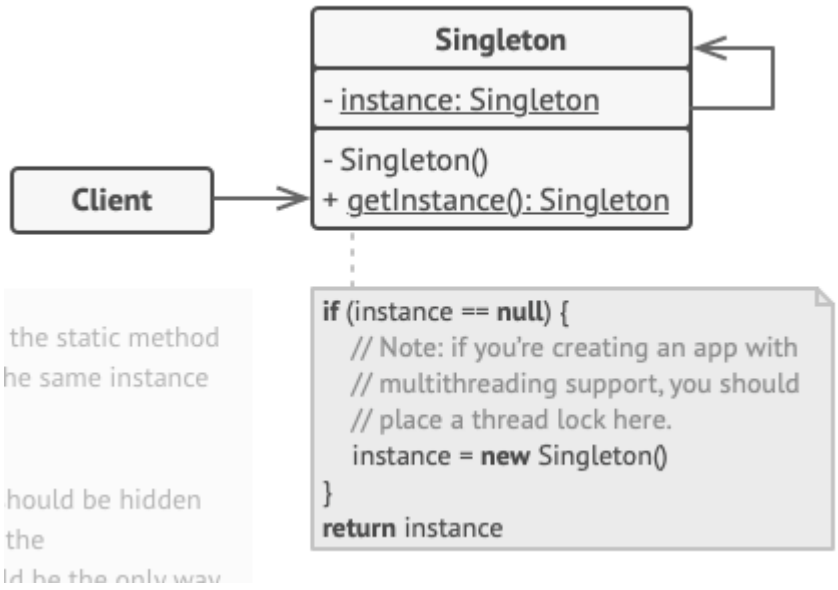

Figure 3.1: Diagramme illustrant le patron Singleton

La classe Singleton déclare la méthode statique *getInstance* qui renvoie la même instance de sa propre classe.

Le constructeur de Singleton doit être caché du code client. Appeler la méthode *getInstance* devrait être le seul moyen d'obtenir l'objet Singleton.

La fonctionnalité du patron de conception Singleton dans la classe *TimberApp* est de garantir qu'il n'y ait qu'une seule instance de la classe *TimberApp* dans une application. Cela permet de garantir que les journaux sont toujours écrits au bon endroit et avec le bon format.

La classe *TimberApp* est le seul point d'accès aux journaux. Elle fournit des méthodes pour écrire des messages dans les journaux, pour gérer la configuration des journaux et pour obtenir l'instance unique de la classe.

La classe *TimberApp* est implémentée comme un Singleton, représentant ainsi l'application elle-même et servant à gérer ses ressources telles que les préférences, les listes de lecture et les lecteurs de musique. En tant que classe d'application, elle hérite de la classe *MultiDex Application* nécessaire pour le développement Android.

L'utilisation d'un Singleton pour *TimberApp* offre plusieurs avantages :

- **Cohérence** : Une seule instance de l'application garantit que toutes les parties de celle-ci utilisent les mêmes données, assurant ainsi la cohérence de l'application.

- **Disponibilité** : L'application est toujours disponible quand l'utilisateur en a besoin, car elle est instantanément accessible à partir de n'importe quelle partie du code.

- **Simplicité** : En évitant la création manuelle d'instances de l'application dans chaque partie du code qui en a besoin, l'utilisation d'un Singleton simplifie la structure globale de l'application.

En plus de ces avantages, l'implémentation d'un singleton pour *TimberApp* contribue à la simplification du code en éliminant la nécessité de passer l'instance de l'application en paramètre à chaque méthode qui doit accéder aux ressources de l'application. Cela simplifie la gestion des ressources et améliore la lisibilité du code.

La méthode *getInstance()* assure le retour d'une seule instance de la classe *TimberApp*, garantissant ainsi son unicité au sein de l'application. La déclaration de la variable statique *mInstance* permet de stocker cette instance unique de *TimberApp*. Son constructeur étant privé,

seul la classe *TimberApp* peut y accéder, ce qui restreint son instanciation à l'intérieur de la classe et initialise l'attribut *mInstance* avec la nouvelle instance de *TimberApp*.

```
private static TimberApp mInstance;
```

Figure 3.2 : Initialisation de l'attribut mInstance

La méthode statique *getInstance()* assure la récupération de l'instance unique de *TimberApp* de manière synchronisée, garantissant qu'elle est appelée par un seul thread à la fois. Cette approche permet de renvoyer l'unique instance de *TimberApp*, assurant ainsi la cohérence et la sécurité de l'application.

```
public static synchronized TimberApp getInstance() {
    return mInstance;
}
```

Figure 3.3 : Méthode statique getInstance()

L'initialisation de l'instance de *TimberApp* se fait dans la méthode *onCreate()*. C'est à cet endroit que la seule instance de la classe est créée, assurant ainsi sa présence unique dans l'application. Cette approche garantit que l'instance est instanciée de manière appropriée au démarrage de l'application, prête à être utilisée par d'autres composants tout au long de son cycle de vie.

```
@Override
public void onCreate() {
    super.onCreate();
    mInstance = this;
    // Autres initialisations...
}
```

Figure 3.4 : Initialisation de l'instance dans la méthode onCreate()

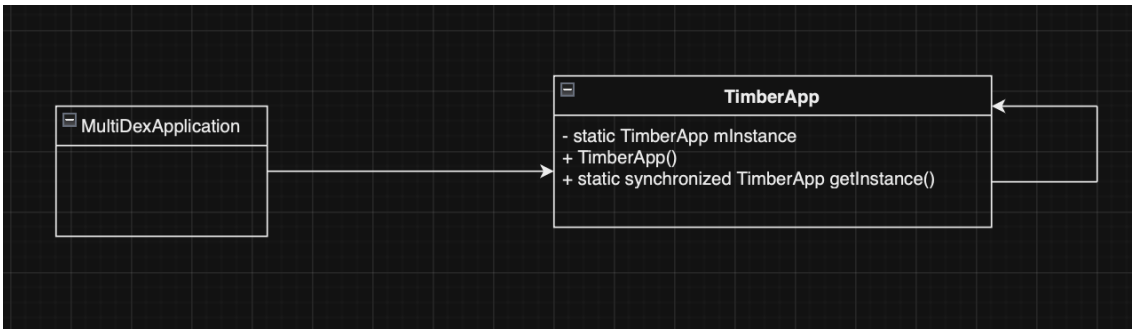

Figure 3.5: Diagramme illustrant le patron singleton

Le Singleton dans cette classe fonctionne de la manière suivante : lors du démarrage de l'application, le système Android appelle la méthode *onCreate()* de la classe *TimberApp*. Dans cette méthode, une nouvelle instance de *TimberApp* est créée et stockée dans la variable statique *mInstance*. Lorsque d'autres parties de l'application ont besoin d'accéder à cette instance, elles invoquent la méthode statique *getInstance()*. Cette méthode renvoie alors l'unique instance de *TimberApp* qui a été précédemment stockée dans *mInstance*. Ainsi, tout au long de l'exécution de l'application, il n'y a qu'une seule instance de la classe *TimberApp*.

3.2 Observateur

Le patron observateur permet à des objets de suivre les changements d'état d'un éditeur. L'éditeur émet des événements lorsqu'il change, et les abonnés réagissent en conséquence. Les abonnés peuvent recevoir des détails sur l'événement pour agir de manière appropriée. Le client crée des éditeurs et des abonnés séparément, puis enregistre les abonnés pour les mises à jour de l'éditeur.

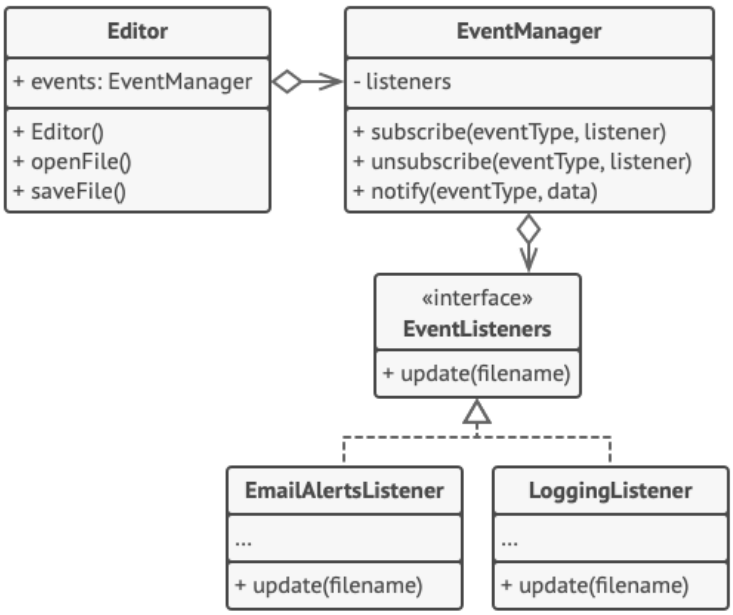

Figure 3.6: Diagramme illustrant le patron de conception observateur

Dans notre cas de projet, le patron observateur est implémenté dans la classe *BaseNowplayingFragment* à travers l'interface *MusicStateListener*. Cette interface définit trois méthodes : *restartLoader()*, *onPlaylistChanged(),* et *onMetaChanged()*. Les fragments qui étendent *BaseNowplayingFragment* peuvent ensuite implémenter cette interface pour écouter les changements d'état de la musique.

Lorsque l'état de la musique change, par exemple lorsqu'une nouvelle piste est chargée (*onMetaChanged()*), les fragments qui écoutent cet événement peuvent réagir en conséquence. Cela permet une séparation claire des préoccupations dans le code, car les fragments n'ont pas

besoin de connaître les détails de la gestion de la musique ; ils peuvent simplement écouter les événements et mettre à jour leur interface utilisateur en conséquence.

```java
/**
 * Listens for playback changes to send the the fragments bound to this activity
 */
Prempal Singh, il y a 8 ans | 2 authors (naman14 and others)
public interface MusicStateListener {

    /**
     * Called when {@link com.naman14.timber.MusicService#REFRESH} is invoked    "naman": Unknown word.
     */
    void restartLoader();

    /**
     * Called when {@link com.naman14.timber.MusicService#PLAYLIST_CHANGED} is invoked    "naman": Unknown word.
     */
    void onPlaylistChanged();

    /**
     * Called when {@link com.naman14.timber.MusicService#META_CHANGED} is invoked    "naman": Unknown word.
     */
    void onMetaChanged();

}
```

Figure 3.7: Code de l'interface MusicStateListener

Le patron observateur est implémenté dans le système de lecteur musical en permettant à plusieurs fragments (observateurs) d'écouter les changements d'état via l'interface commune *MusicStateListener*. Les classes *BaseNowplayingFragment* et ses sous-classes sont des observateurs et implémentent cette interface. Lorsque des événements tels que les changements de liste de lecture ou de métadonnées se produisent dans la classe *MusicService*, les méthodes appropriées comme *notifyPlaylistChanged()* et *notifyMetaChanged()* sont appelées, ce qui déclenche l'exécution des méthodes correspondantes *onPlaylistChanged()* et *onMetaChanged()* chez tous les observateurs enregistrés. Chaque observateur peut réagir de manière spécifique à ces événements, assurant ainsi un découplage entre le sujet et les observateurs et rendant le code plus modulaire et extensible.

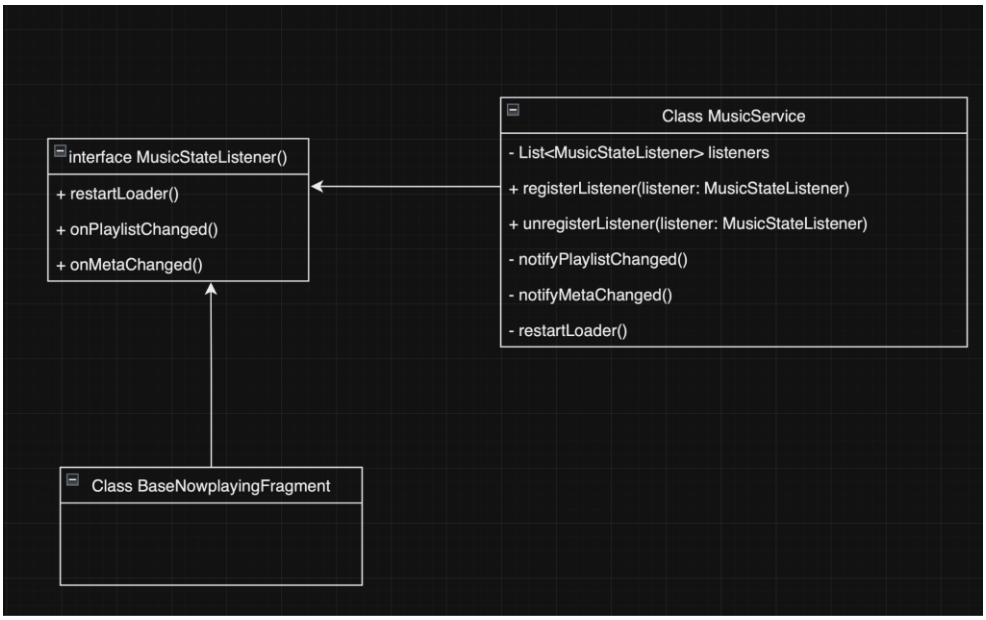

Figure 3.8: Diagramme illustrant le patron observateur dans le projet TimberApp

Le patron observateur gère la lecture de musique et notifie les changements d'état aux composants concernés. L'objet principal, *MusicService*, gère la lecture et conserve l'état à observer. Les observateurs, comme *BaseNowplayingFragment*, s'enregistrent auprès du *MusicService* via l'interface *MusicStateListener*. Les notifications sont déclenchées par des changements d'état tels que les changements de chanson ou la lecture/pause, et sont gérées par des méthodes telles que *notifyPlaylistChanged()*. Les observateurs sont notifiés via *notifyMetaChanged()*, appelant des méthodes spécifiques comme *restartLoader()* ou *onPlaylistChanged()*. Les observateurs réagissent en mettant à jour l'interface utilisateur.

3.3 Patron de méthode

Le patron de méthode est un modèle de conception comportemental qui définit le squelette d'un algorithme dans la classe de base, mais permet aux classes dérivées de remplacer des étapes spécifiques de l'algorithme sans en changer la structure.

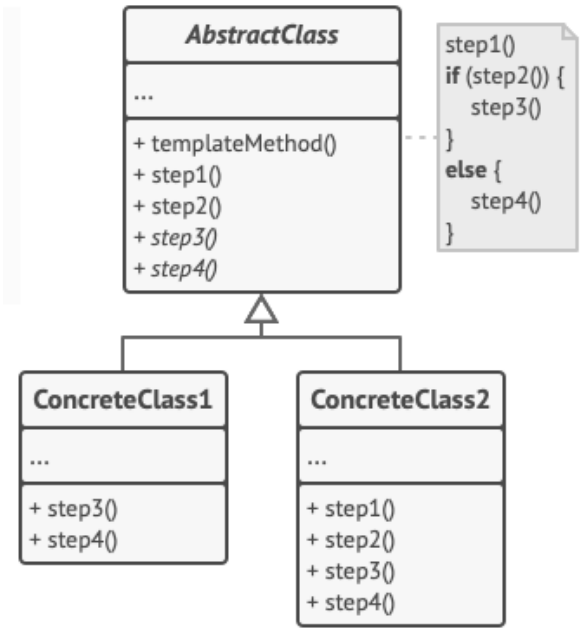

Figure 3.9: Diagramme illustrant le patron de méthode

La classe abstraite déclare des méthodes qui agissent comme des étapes d'un algorithme, ainsi que la méthode de modèle réel qui appelle ces méthodes dans un ordre spécifique. Les étapes peuvent être déclarées abstraites ou avoir une implémentation par défaut.

Les classes concrètes peuvent remplacer toutes les étapes, mais pas la méthode de modèle elle-même.

Dans les classes du projet *TimberApp* pour ce projet, *BaseSongAdapter* représente la classe modèle et ses sous-classes (*AlbumSongsAdapter*, *ArtistSongAdapter*, *FolderAdapter*, *SearchAdapter*, et *SongsListAdapter*) personnalisent ou fournissent des implémentations spécifiques pour certaines de ses méthodes.

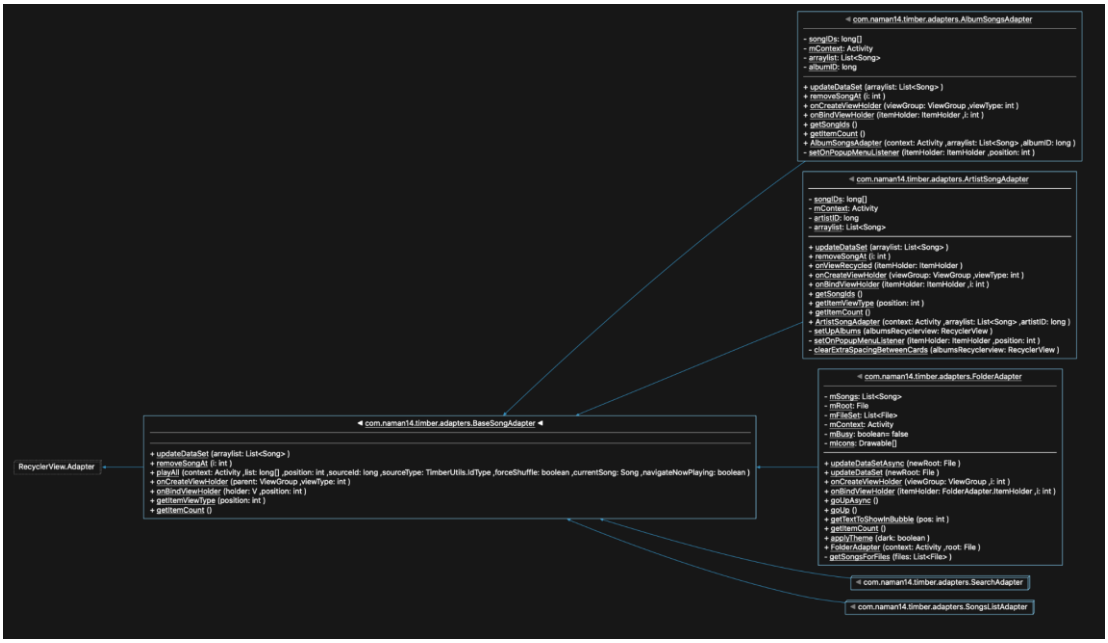

Figure 3.10: Diagramme illustrant le patron de méthode du projet TimberApp

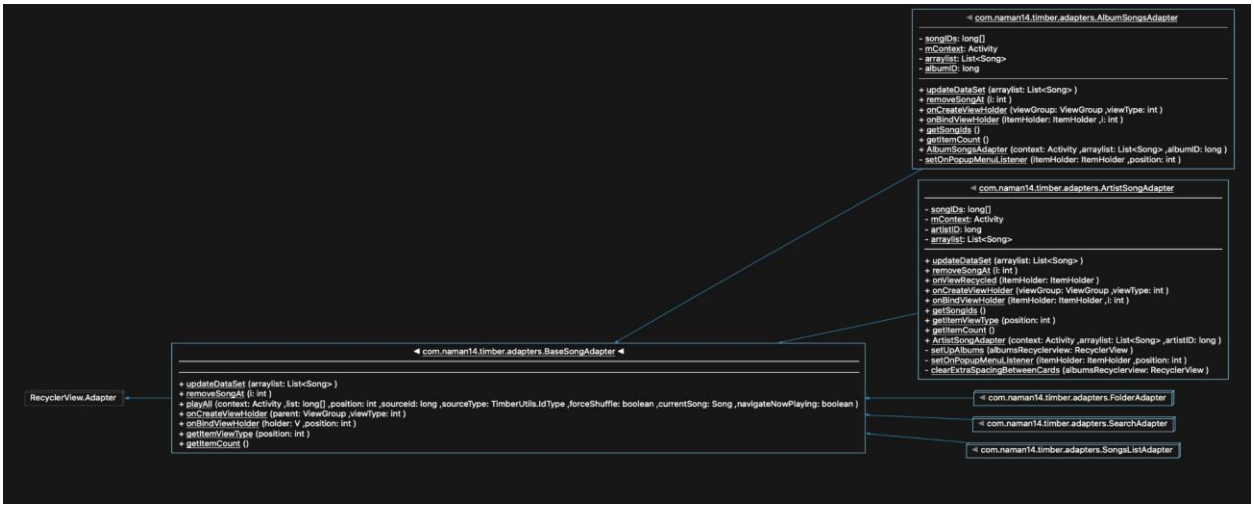

Figure 3.11: Diagramme illustrant le patron de méthode du projet TimberApp

Dans notre cas, le patron de méthode est utilisé pour standardiser la manipulation des chansons dans les *RecyclerViews*. Tout d'abord, la classe *BaseSongAdapter* fournit un schéma d'algorithmes génériques pour cette manipulation, avec des méthodes telles que *onCreateViewHolder***,** *onBindViewHolder***,** *getItemCount***,** etc. Certaines de ces méthodes sont

définies dans la classe de base, tandis que d'autres sont abstraites, laissant aux sous-classes le soin de les implémenter selon leurs besoins spécifiques.

Les différentes sous-classes telles que *AlbumSongsAdapter*, *ArtistSongAdapter*, *FolderAdapter*, *SearchAdapter* et *SongsListAdapter* étendent *BaseSongAdapter* pour fournir des implémentations spécifiques à chaque type d'adaptateur. Elles remplacent les méthodes du modèle par des implémentations concrètes adaptées à leurs exigences particulières, tout en introduisant éventuellement des méthodes et des classes supplémentaires spécifiques à chaque adaptateur.

L'invocation du patron de méthode se produit lorsque les méthodes définies dans *BaseSongAdapter* sont appelées dans le code, formant ainsi une séquence d'actions cohérente. Les sous-classes peuvent personnaliser des étapes spécifiques de ce modèle, ce qui permet une flexibilité et une extensibilité significatives sans altérer la structure globale du modèle. En utilisant une interface commune via l'utilisation de génériques, toutes les sous-classes partagent une manière uniforme d'interagir avec les adaptateurs, ce qui facilite la maintenance et la cohérence du code.

L'utilisation du patron de méthode dans notre exemple de projet garantit une structure et une interface communes pour manipuler des chansons dans les *RecyclerViews*, tout en permettant aux sous-classes de personnaliser des étapes spécifiques selon leurs besoins particuliers. Cela favorise la réutilisation du code et simplifie l'extension ou la modification du comportement de chaque adaptateur de manière indépendante.

4. Analyse d’architecture logicielle

4.1 Paquets de Timber

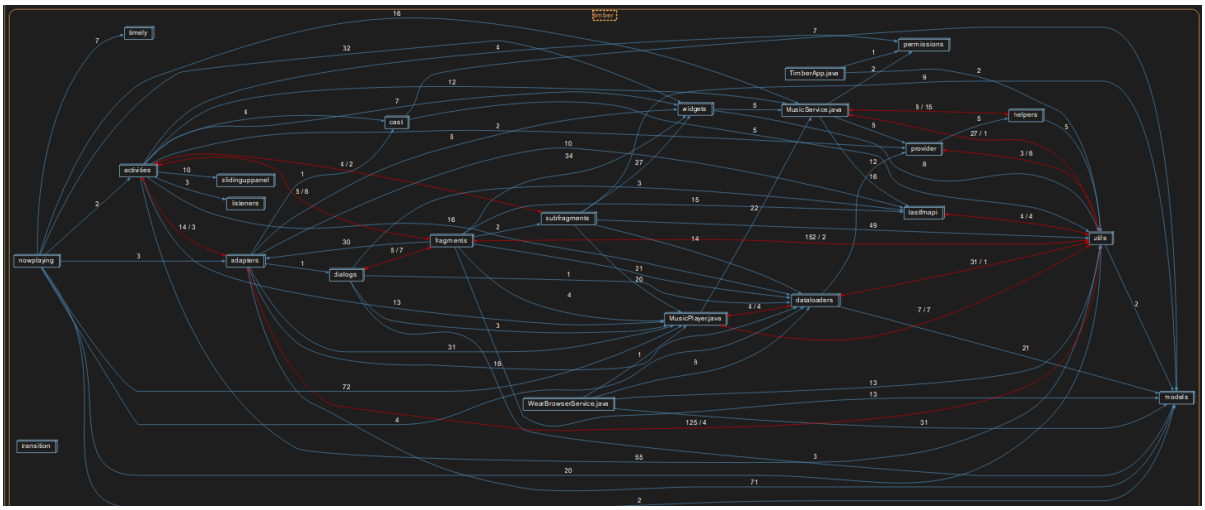

Figure 4.1: Diagramme de paquetage constituant L’application Timber

L’architecture de Timber comprend une multitude de paquetages qui se subdivisent en 4 catégories. Tout d’abord, il y a les paquetages qui s’occupent d’implémenter la logique de présentation de l’application. Dans cette catégorie, les paquetages *activities*, *widgets*, *fragments*, *subfragments* et *dialogs* implémentent différentes composantes de la vue. Les *activities* sont formées de *fragments* et les *fragments* de *subfragments*. De plus, tous ces paquetages peuvent importer des *widgets* ou des *dialogs* selon leurs besoins. Cette relation entre les paquets de l’application se révèle en explorant les liens d’utilisation entre les paquetages de la vue. La logique de présentation possède aussi le paquet *nowPlaying* qui possède plusieurs vues de l’application. Finalement, il possède des paquets d'utilitaires qui fournissent des fonctionnalités à la présentation de l’application. Ceux-ci sont les paquets *Listeners* qui font la gestion d’état, *Cast* qui envoie gère l’état de l’authentification et le paquet *timely* qui est un utilitaire.

Par la suite, l’application Timber comprend une couche de logique d’application qui expose plusieurs services et API qui sont utilisés par la logique de présentation. Parmi les services, on retrouve le *WearBrowserService* qui gère la session des utilisateurs, *TimberApp* qui gère la configuration de l’application selon un thème visuel, *Dataloaders* un service qui s’occupe du chargement des données de l’application, *MusicService* un service qui s’occupe de

l'implémentation de la gestion de la musique, *Permissions* un paquet qui fait l'authentification de l'utilisateur et *MusicPlayer* un paquet qui gère la logique pour faire jouer la musique. Cette couche possède aussi un API REST *lastfmAPI* et un paquet d'utilitaires nommé *helper*.

Puis, l'application Timber possède également une couche d'accès aux données représentée par le paquet *providers* qui fournit un point d'accès vers une base de données.

Finalement, L'application possède un paquet d'utilitaires qui sont co-dépendant avec quasiment l'entièreté des autres paquetages. C'est le paquetage *utils* qui expose des outils aux autres paquetages. L'application possède aussi un paquetage *models* qui définit et centralise les formats des objets, données et interfaces qui traversent l'application.

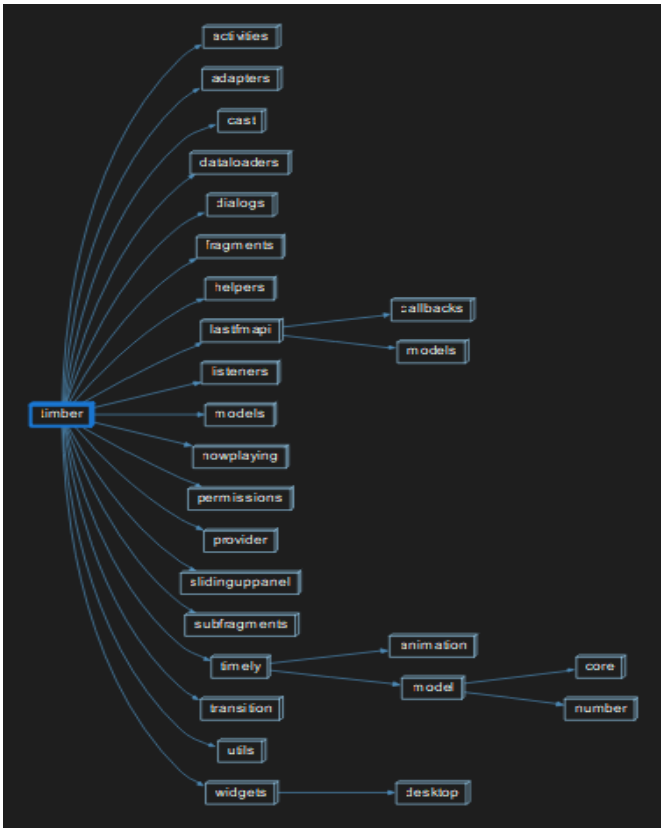

Figure 4.3: Diagramme Architecture de système Timber

4.2 Style architectural de Timber

L'architecture de l'application Timber est complexe. Sa conception ne reflète pas d'un style d'architecture en particulier, mais emprunte plutôt certains éléments de plusieurs styles.

Concrètement, l'application possède des éléments de l'architecture monolithique, de l'architecture orientée service et de l'architecture par couche. L'architecture du système s'inspire donc de plusieurs modèles afin de former un hybride.

Monolithique:

L'architecture du système s'inspire du monolithique dû au paquetage *utils*. En effet, celui-ci est utilisé par toutes les couches du système, soit la vue, la couche d'application et la couche d'accès des données. De plus, elle est non seulement utilisée, mais plutôt interdépendante. Cela signifie qu'une modification à ce paquetage peut exiger la modification de plusieurs autres paquetages. Cette caractéristique du système est une conséquence de l'utilisation d'une architecture monolithique. Cette utilisation du paquetage enlève aussi la distinction entre les couches du système, on enlève donc une certaine distinction entre ceux-ci. Cela est aussi présent dans une architecture monolithique

Orientée service:

L'architecture de Timber s'inspire également de la philosophie orientée service. Cette réalité s'explique par la présence de plusieurs paquetages relativement indépendants et atomiques. Par exemple les modules TimberApp, WearBrowserService et permissions sont tous des paquetages qui partagent quelques similarités. Entre autres, ils sont petits, ils n'ont pas d'interdépendances avec d'autres paquetages et leurs responsabilités peuvent être résumées en une phrase. Ces caractéristiques sont typiques d'une architecture orientée service.

Par couche:

Finalement, L'architecture du système emprunte aussi des éléments de l'architecture par couche. Notamment, cela s'exprime par le flux des dépendances du système. Dans le diagramme, si on ignore le paquetage *utils*, la grande majorité des dépendances (90 %) vont de la gauche vers la droite. De plus, les composantes de la vue sont placées à gauche, les composantes de la couche d'application à droite. Il n'y a donc pas de scénario ou une couche inférieure dépend d'une couche supérieure. Ce principe constitue de l'inversion de contrôle (IoC). Cette caractéristique est typique d'une architecture multicouche.

Conclusion

La conception de logiciels efficaces repose souvent sur des principes et des modèles bien établis. Dans cette analyse, nous avons exploré les principes SOLID, un ensemble de cinq principes de conception de logiciel visant à créer des systèmes plus flexibles et évolutifs. Nous avons examiné des violations potentielles de ces principes dans le projet Timber, ainsi que l'application de patrons de conception et une analyse de son architecture logicielle. Cette approche méthodique nous a permis de mieux comprendre la structure et les défis de ce projet, mettant en lumière l'importance de suivre des principes de conception solides pour garantir la qualité et la maintenabilité du logiciel.

Références

[1] Refactoring Gure (2024) Template Method. [En ligne]. Disponible: https://refactoring.guru/design-patterns/template-method

[2] S. A. (2024) Create dynamic lists with RecyclerView. [En ligne]. Disponible: https://developer.android.com/develop/ui/views/layout/recyclerview

[3] Moxit Saxena (2021) What Is Liskov Substitution Principle (LSP)? With Real World Examples! [En ligne]. Disponible: 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