LOG8430E - Software Architecture and Advanced Design

Fall 2022

TP1: Design Principles and Patterns - Object-oriented Architectures

By

Team Alpha

1949477 - Ming Xiao Yuan

1956576 - Laura Beaudoin

1957959 - Reetesh Dooleea

2168201 - Mohsen Rahmati

Submitted to: **Zoreh Sharafi**

October 4th 2022

Table of contents

Introduction

This assignment focuses mainly on the analysis of object-oriented architectures and design principles. The project analyzed is Phonograph [1], a designated local music player for Android. In the first section, a thorough software architecture analysis of the main java code will be done, followed by an analysis of different design patterns found in the project. In the second section, an analysis of the different SOLID Design Principles used on Phonograph will be presented. Lastly, a violation of the SOLID Principles found in the project will be analyzed.

1. Software architecture analysis

1.1. Software architecture

First of, in order to discuss Phonograph's architecture, its structure is made into a few package diagrams represented by Figure 1.1 and 1.2.

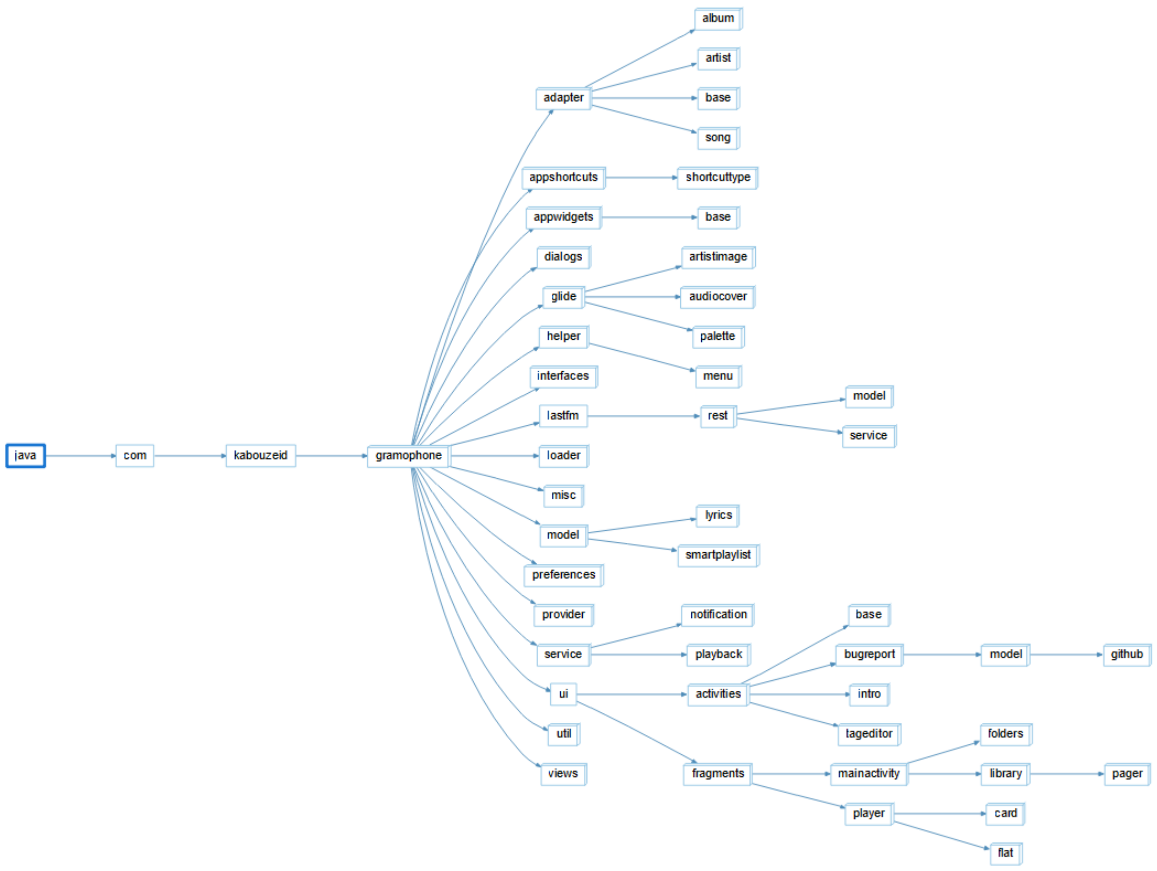


Figure 1.1 Packet diagram of java package with Understand

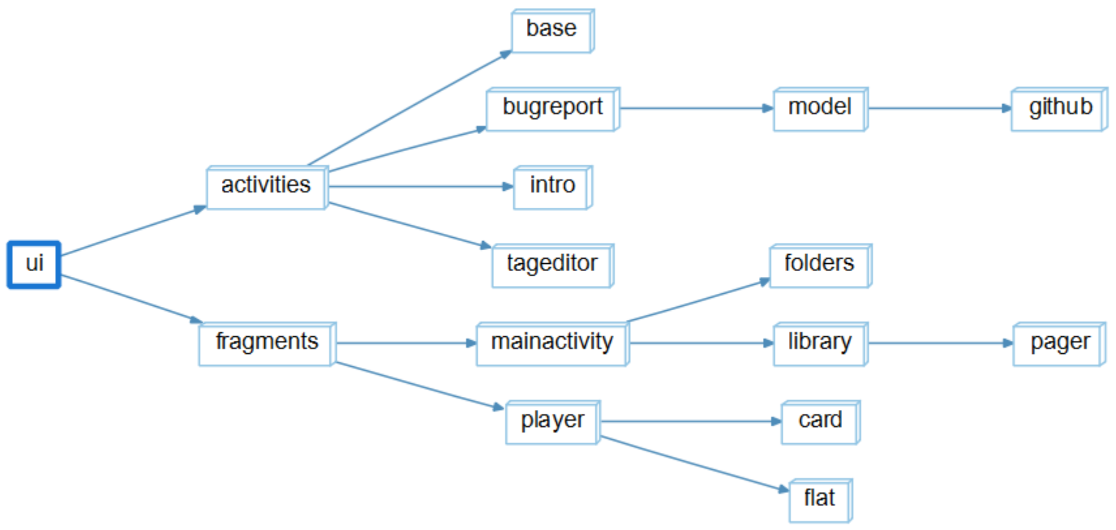


Figure 1.2 Packet diagram of ui package with Understand

Before identifying the specific architecture style of Phonograph, it is important to look at what are the main three architecture categories.

First of all, the first category is the **Monolithic Architecture** and it's a single program with a single point of change. There is no distinction between the data, the logic, the UI and the program is mainly developed and maintained by a sole team. Its advantage is that it's very secure and the program is often very efficient because there are no network communications.

Second of all, the second category is **Distributed Architectures**. It is a variation of Modular architecture but the modules are distributed in different places. The development is shared among different teams and the software is more flexible and dynamic than the other categories. It increases flexibility to use different providers of hardware and software so it's easier to add new resources.

Last but not least, the final category is **Modular Architectures** which separates the system into many modules. There is a more distinguished separation of responsibilities and the modules are developed separately. The modules are smaller and easier to manage which makes the unit testing easier.

1.2. MVC Model

With a little analysis with the two package diagrams, it is easy to witness that the Phonograph app is a MVC Model or Model-View-Controller from the Modular Architecture. There are 3 layers in the MVC Model, the Model layer, the View layer and the Controller layer.

The **Model layer** is responsible for the data treatment, which is directly linked to the database. In our case, it contains the database, and the different data structures and maps the data into usable values in the application.

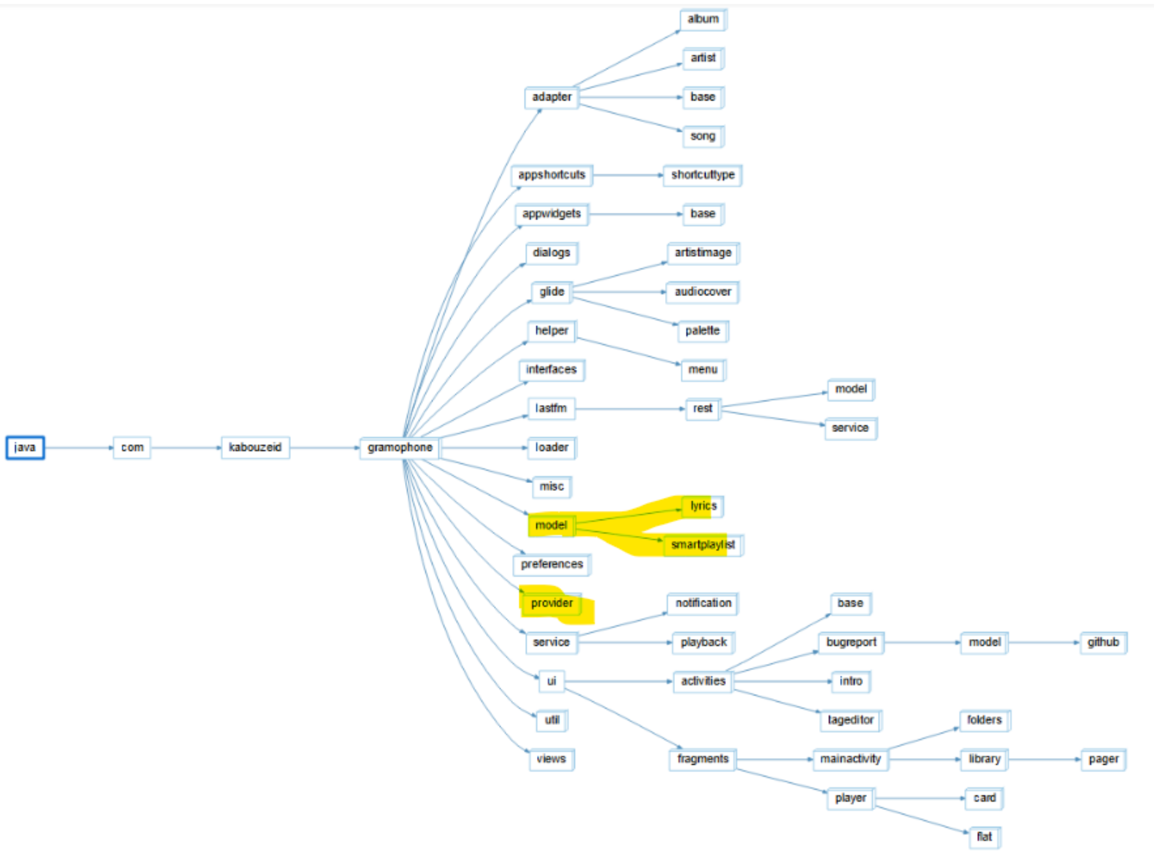


Figure 1.3 Package diagrams with Model layer highlighted with Understand

The specific packages that are involved in the Model layer are highlighted. It contains the *gramophone/model*, *lyrics* and *smartplaylist* and *provider* as the database. They also communicate with the Controller layers.

The **View layer** is responsible for the representation of the different data perceived by the user. Different views are possible to witness the same data and it can be represented by charts or graphs. In modern terms, it could be the GUI (Graphical User Interface).

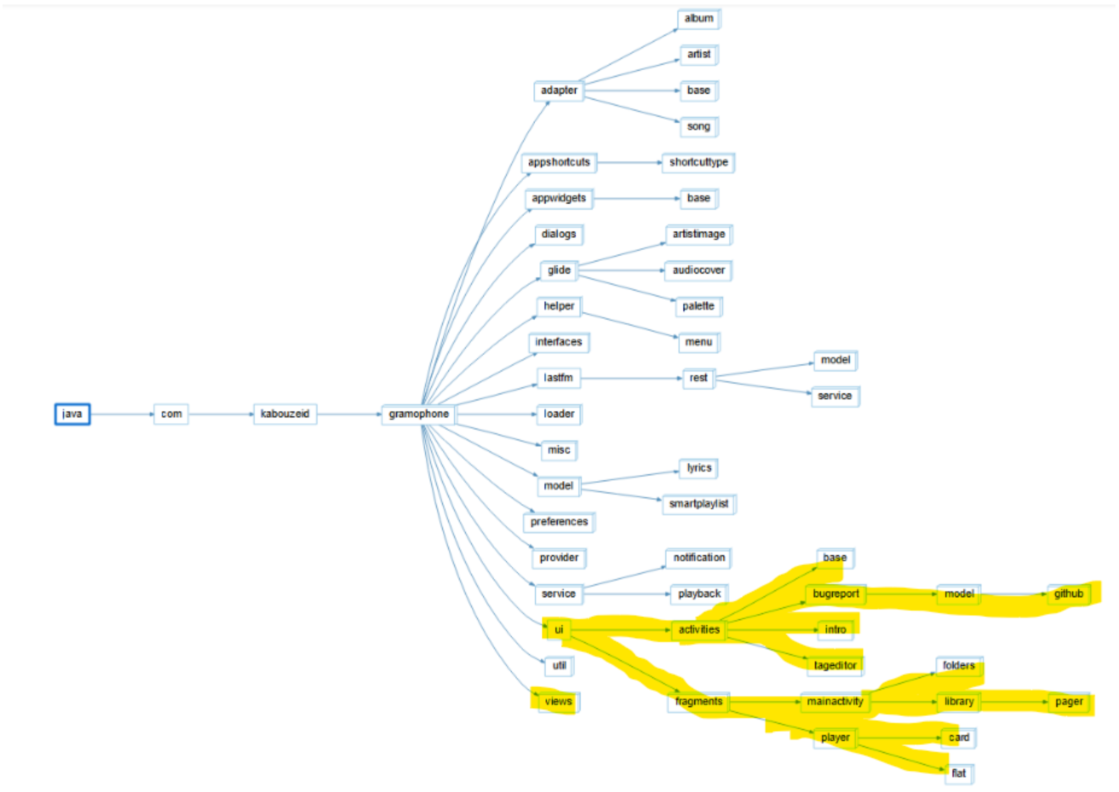


Figure 1.4 Package diagrams with View layer highlighted with Understand

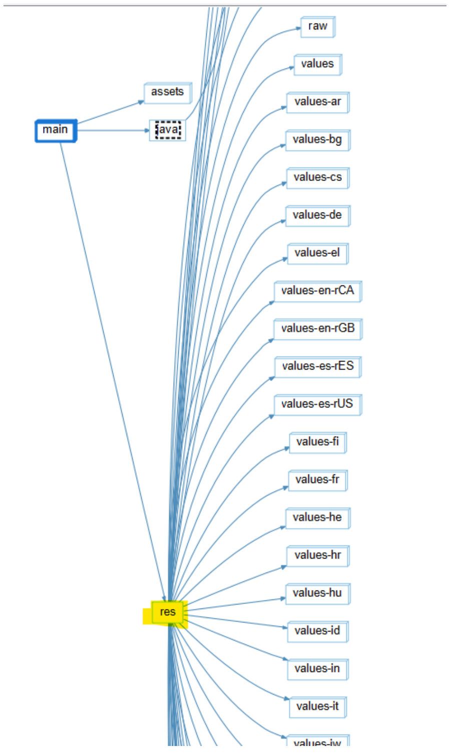


Figure 1.5 Package diagrams with View layer highlighted with Understand

In Phonograph, the View layer by the *gramophone/ui* root package and the *views* package can be clearly distinguished. The *view* package contains different functions concerning the treatment of different actions that the user and the layout. The *ui* package contains files that determine show or hide logic. Lastly, the *res* package contains all the different XML files that are needed to render the view. The View layer also communicates with the Controller layer.

Finally, the **Controller layer** is responsible for all the logic and for modifying the data presented to the user. It is essentially the bridge between the View and Model layers. It reacts to the events in the view and knows how to change and manipulate the model.

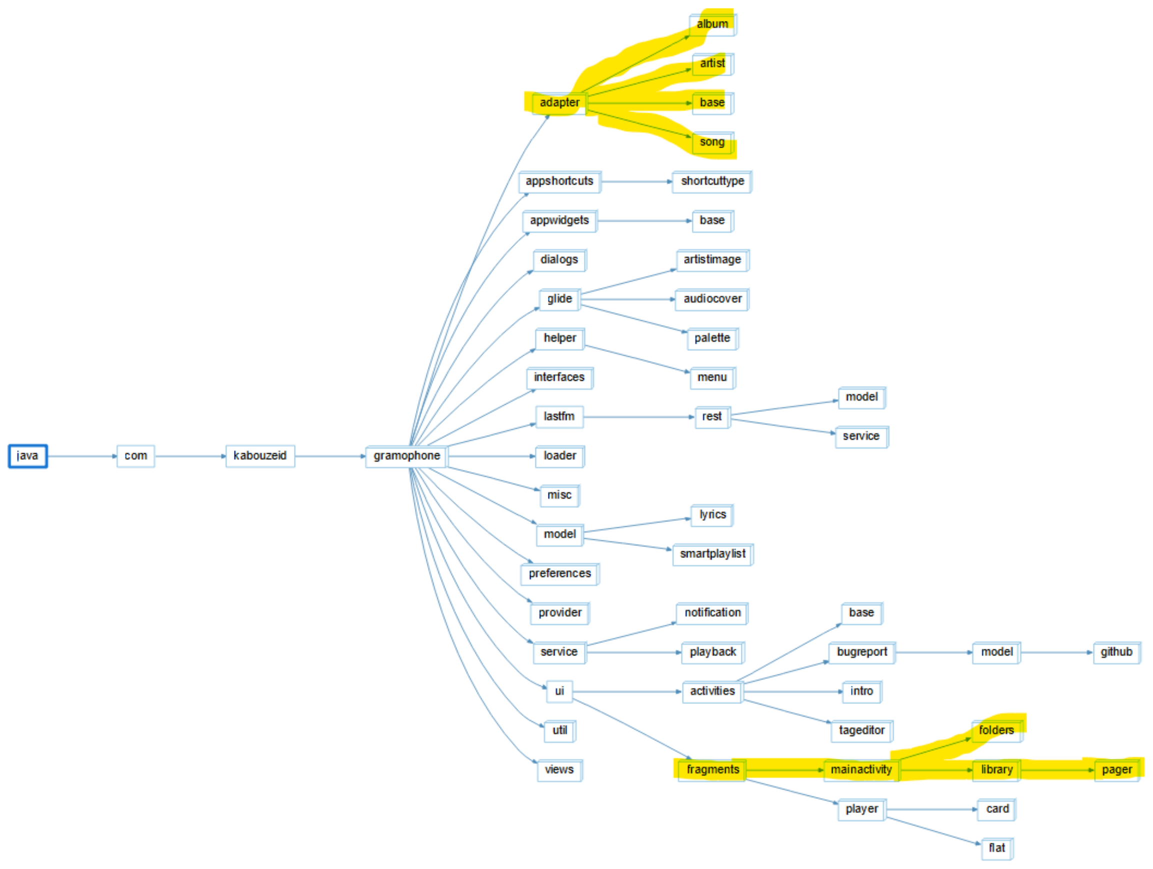


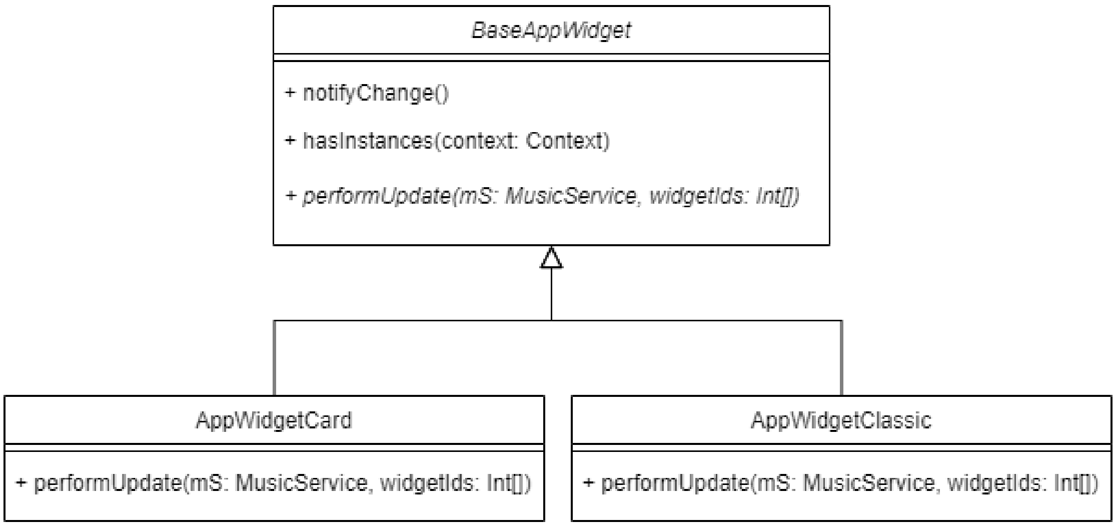
Figure 1.6 Package diagrams with Controller layer highlighted with Understand

In Phonograph, the Controller layer is represented in the *gramophone/adapter* and *gramophone/ui/fragments* packages. The *adapter* package is used specifically to link the view with new data. The use of *notifyDataSetChanged()* function is able to update the said views when the model changes. In the *fragments* package and more specifically in the *mainactivity* package, a lot of the application logic is stored there.

2. Design patterns

After exploring the code source of Phonograph, three design patterns that are worth mentioning were identified. These design patterns are the Template method, Singleton and the Facade design patterns. In the section below, each design pattern will be presented with a UML diagram, followed by its functionality as well as a description of the role of each class in the corresponding pattern and diagram.

2.1. Template method



BaseAppWidget
+ notifyChange()
+ hasInstances(context: Context)
+ performUpdate(mS: MusicService, widgetIds: Int[])
AppWidgetCard
+ performUpdate(mS: MusicService, widgetIds: Int[])
AppWidgetClassic
+ performUpdate(mS: MusicService, widgetIds: Int[])

Figure 2.1 Template method based on BaseAppWidget class

The first design pattern identified is the Template method design pattern. This is a behavioral design pattern which implements the logic or algorithm a method called *template method* found in the parent class but lets the child class define or override certain specific steps of the algorithm. This means that some steps of the algorithm may be common for all algorithms. Hence, the parent can already implement some steps of the algorithm but the logic that is distinct and specific is defined by a child class.

For this example, the template method *notifyChange* of the superclass *BaseAppWidget,* found in *gramophone/appwidgets/base,* is triggered whenever there is a new notification. To apply this design pattern, this method calls the *abstract* method named *performUpdate* which is only defined by the subclasses. In this case, the template method corresponds to the *notifyChange* method since it consists of the algorithm or logic implemented when there is a notification. For proof of concept, one of the concrete child classes of the *BaseAppWidget* class is the *AppWidgetCard* class, located at *gramophone/appwidgets.* Hence, the latter (*AppWidgetCard*) has to define its custom logic for the *performUpdate* method in order to continue the steps in the super class which is the *BaseAppWidget* class. It is to be noted that the template method is defined in the parent class. It is the method which contains all the inner steps of the algorithm. Hence, the child class can only define custom steps without modifying the structure of the template method itself.

2.2. Singleton

The second design pattern identified is the Singleton design pattern which has been applied to the class of *PreferenceUtil.java* found in *gramophone/util*. To begin with, a UML class diagram is presented to illustrate the pattern.

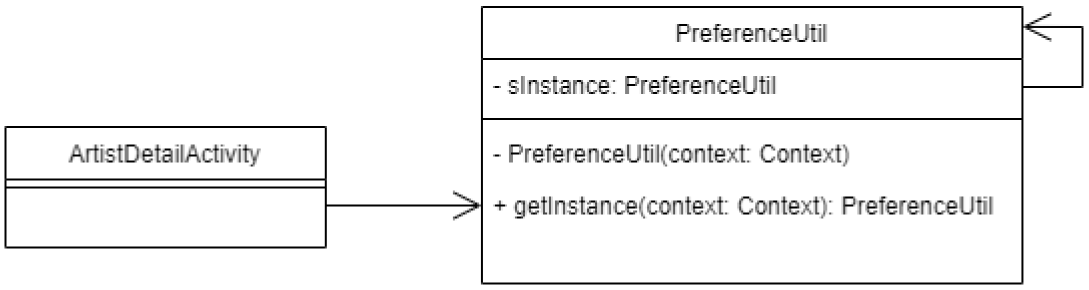

PreferenceUtil
- sInstance: PreferenceUtil
- PreferenceUtil(context: Context)
+ getInstance(context: Context): PreferenceUtil
ArtistDetailActivity

Figure 2.2 Singleton UML design pattern applied to PreferenceUtil Java class

The singleton design pattern is one of the most common creational patterns used in a project. The goal of the pattern is to guarantee the creation of one and only one instance of the class across the whole project. Hence, since a class has a single unique instance, this sole instance becomes a global access point and the same reference is reused everywhere in the code source rather than having to create multiple instances with the *new* keyword.

Since we only create a unique instance of the class, the *getInstance* method always ensures to return the same instance of its own class. In that way, if the class is not instantiated, it creates an instance of the private variable of the own class (*sInstance*) with the *new* keyword, saves the *sInstance* reference and returns it. From now on, the same unique instance will be used. We observe that it is well the case for the *PreferenceUtil* class.

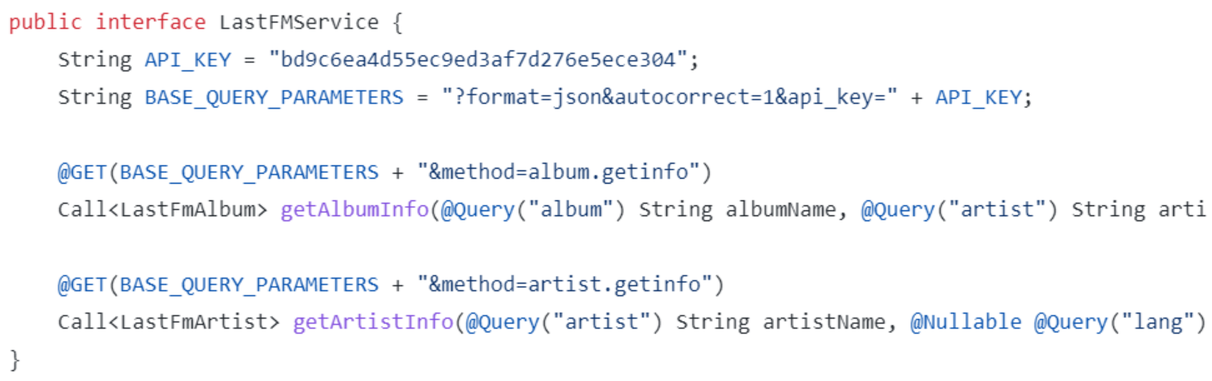
Furthermore, to verify that the design pattern is well respected, we can see that the class *ArtistDetailActivity* in *gramophone/ui/activities* uses the *PreferenceUtil* class. We observe that *ArtistDetailActivity* does not create an instance of the *PreferenceUtil* class with the *new* keyboard. In any case, it can’t because the constructor of the *PreferenceUtil* class is *private*. Only the *getInstance* method of *PreferenceUtil* can create an instance of that class. Instead, the class *ArtistDetailActivity* calls the *getInstance* method to retrieve the global instance of the *PreferenceUtil* class. However, for this pattern to work, it is essential for the *getInstance* method to be *static* so it is not possible for another class to create an instance of the singleton class.

All of these above mentioned criteria indicate that the Singleton design pattern is well applied in this situation.

2.3. Facade

The third design pattern to analyze is the Facade design pattern. This one consists of a structural design pattern that provides a simpler access or interface to a component, library or other group of interfaces. For this demonstration, the Retrofit library will be analyzed through code snippets to demonstrate how this library simplifies the logic behind the scenes for the client and only provides a high level interface in the Phonograph application.

For this example, if our client needs to fetch some information about an album from an external API, it needs to make HTTP requests to that API. Ideally, for this feature, the client should be able to retrieve a specific object without having to understand the inner implementation or details of the Retrofit or API client. First of all, the client needs to provide an interface to represent the API data structure and methods. In this example, the class *LastFMService*, located at *gramophone/lastfm/rest/service*, specifies the calls or HTTP methods that it wants to make as well as the object the API should return. In that way, the object returned from the API will be casted to the appropriate specified class model. Hence, the facade design pattern hides complex logic behind the scenes and provides a cleaner access to a dense subsystem to the client.



```
public interface LastFMService {
    String API_KEY = "bd9c6ea4d55ec9ed3af7d276e5ece304";
    String BASE_QUERY_PARAMETERS = "?format=json&autocorrect=1&api_key=" + API_KEY;

    @GET(BASE_QUERY_PARAMETERS + "&method=album.getinfo")
    Call<LastFmAlbum> getAlbumInfo(@Query("album") String albumName, @Query("artist") String arti

    @GET(BASE_QUERY_PARAMETERS + "&method=artist.getinfo")
    Call<LastFmArtist> getArtistInfo(@Query("artist") String artistName, @Nullable @Query("lang")
}
```

Figure 2.3.1 API interface for Retrofit for a HTTP requests

To use this interface, the client only needs to call the *getAlbumInfo* or *getArtistInfo* methods to get the desired data as an object without having to manipulate how the HTTP requests are being implemented. For instance, simply invoking the *getAlbumInfo* method will return a *LastFMAlbum* object. Similarly, calling the *getArtistInfo* method will make a HTTP GET request and return a *LastFmArtist* object.

That being said, the *LastFMRestClient* class in the Phonograph application, located at *gramophone/lastfm/rest*, needs to configure a Retrofit instance as the *LastFMService* class in order to be able to invoke the *getAlbumInfo* or *getArtistInfo* methods. This is done by creating a new Retrofit builder and calling the *create* method at line 39 in the figure below with the API interface to be able to access the two methods defined in that class which will allow us to make external HTTP requests. The rest is encapsulated from the client and is inside the Retrofit framework since the client does not need to know any more information.

```java
public class LastFMRestClient {
    public static final String BASE_URL = "https://ws.audioscrobbler.com/2.0/";

    private LastFMService apiService;

    public LastFMRestClient(@NonNull Context context) {
        this(createDefaultOkHttpClientBuilder(context).build());
    }

    public LastFMRestClient(@NonNull Call.Factory client) {
        Retrofit restAdapter = new Retrofit.Builder()
                .baseUrl(BASE_URL)
                .callFactory(client)
                .addConverterFactory(GsonConverterFactory.create())
                .build();

        apiService = restAdapter.create(LastFMService.class);
    }
```

Figure 2.3.2 Configuration of Retrofit to use the HTTP methods of LastFMService in LastFMRestClient class

3. SOLID Design Principles

We discuss three different kinds of SOLID design principles including the Single Responsibility principle, the Liskov Substitution principle, and the Open-Close Surgery principle. First, we have explained their UML diagrams, functionalities, and roles.

3.1. Single Responsibility Principle

In Phonograh's code, firstly the Single Responsibility Principle (SRP) can be found. In this principle, each class needs to have one and only one responsibility. It means that the class has one and only one reason to change. So, if the programmer wants to change the class, the change is predictable. Single responsibility for each class makes writing, debugging, and testing our program easier. Also, if the programmer will change the code, the side effects of these changes will be minimized.

As you can see from the following UML diagram, there is one responsibility for the *gramophone/service/MusicService.MusicBinder* which is the *getService* method.

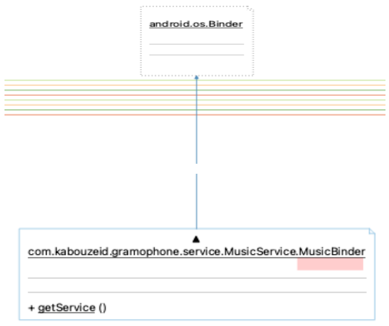


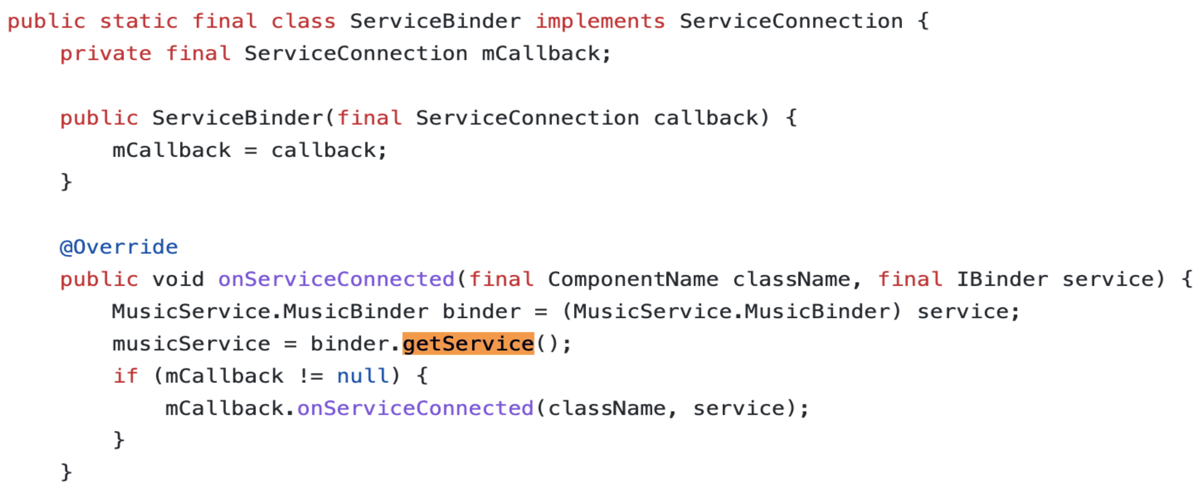
Figure 3.1.1 MusicBinder Class UML diagram proves the Single Responsibility Principle

The class of MusicBinder has one and only one function and it is returning the *MusicService*. It makes it clear that this class only has one responsibility, as the SOLID principle wants.

```
public class MusicBinder extends Binder {
    @NonNull
    public MusicService getService() {
        return MusicService.this;
    }
}
```

Figure 3.1.2 MusicBinder Class code proves the Single Responsibility Principle

As you see in the next code, the class of *MusicBinder* has been used in the class *ServiceBinder*, located at *gramophone/helper/MusicPlayerRemote.MusicBinder*, and the role of this class is to return the MusicService.

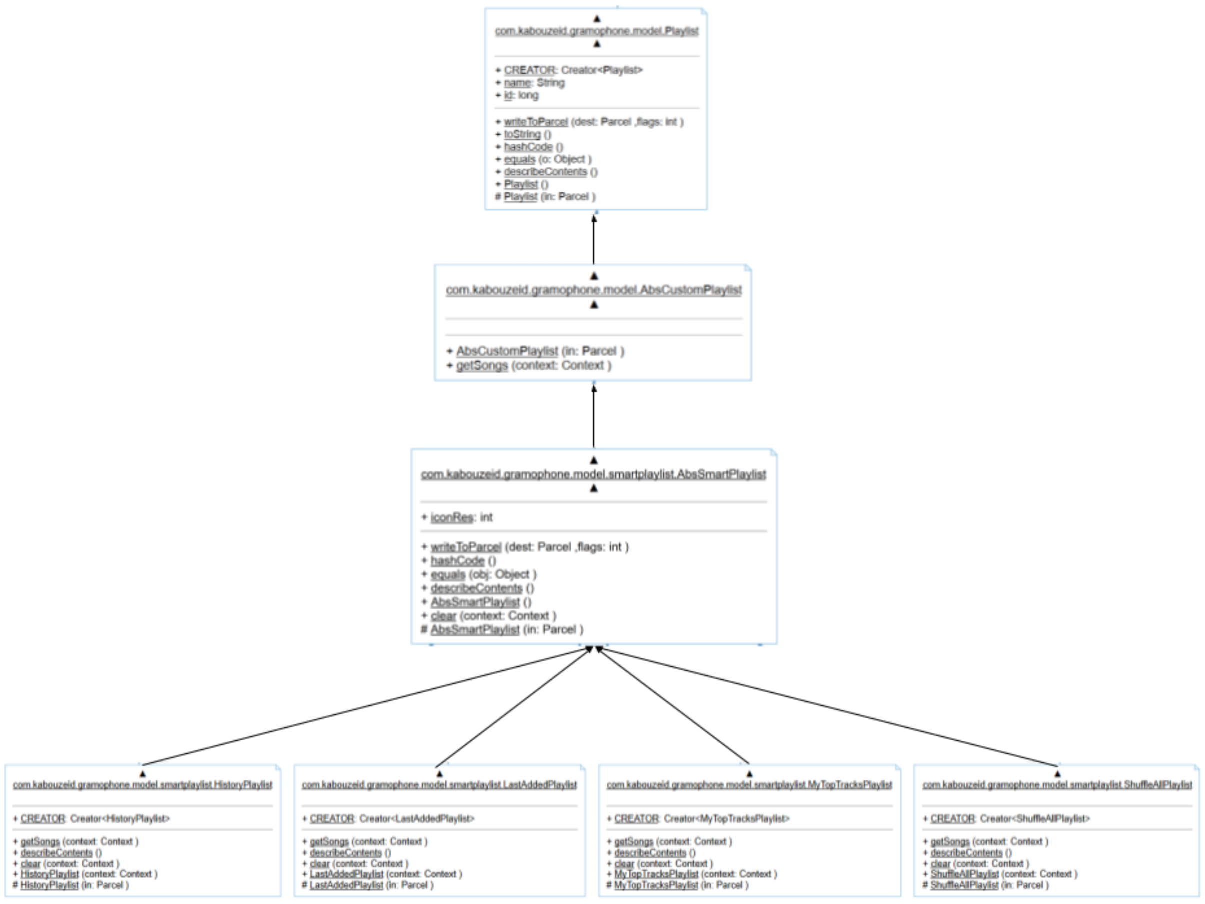


```java
public static final class ServiceBinder implements ServiceConnection {
    private final ServiceConnection mCallback;

    public ServiceBinder(final ServiceConnection callback) {
        mCallback = callback;
    }

    @Override
    public void onServiceConnected(final ComponentName className, final IBinder service) {
        MusicService.MusicBinder binder = (MusicService.MusicBinder) service;
        musicService = binder.getService();
        if (mCallback != null) {
            mCallback.onServiceConnected(className, service);
        }
    }
```

Figure 3.1.3 MusicBinder used in the class ServiceBinder

3.2. Liskov Substitution Principle

The second SOLID principle identified in Phonograh's base code is the Liskov Substitution Principle (LSP). To summarize, this principle says that a child that extends a parent must be able to be used in any context the parent is used. First, for that to be possible, the methods overridden in the children have to accept the same parameters as the parent wants. There can't be more and they can't be more strict. Secondly, the return values also have to be the same for the inherited methods as for the original one in the parent. The UML diagram below presents one instance of LSP in Phonograph.

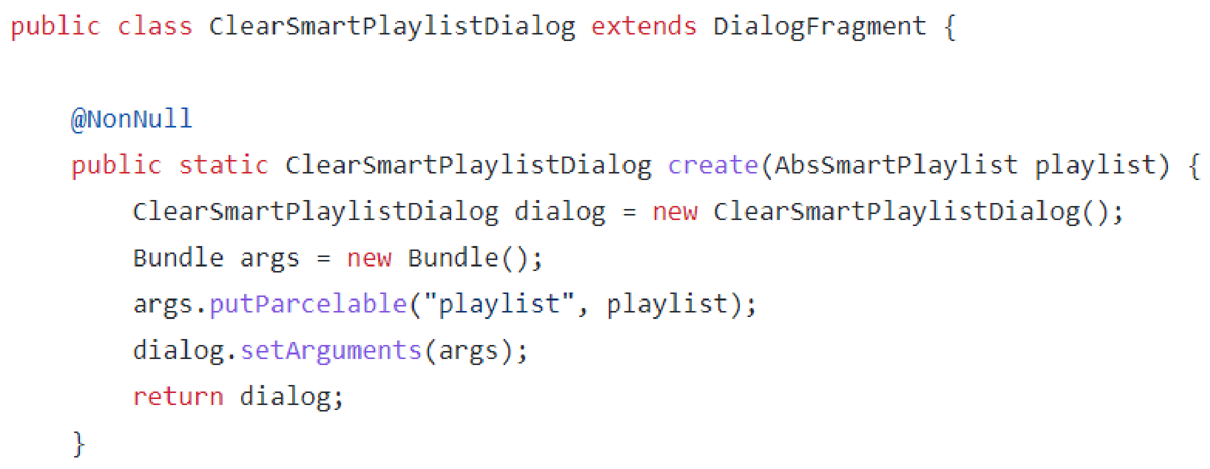


com.kabouzeid.gramophone.model.Playlist
+ CREATOR: Creator<Playlist>
+ name: String
+ id: long
+ writeToParcel (dest: Parcel ,flags: int)
+ toString ()
+ hashCode ()
+ equals (o: Object)
+ describeContents ()
+ Playlist ()
# Playlist (in: Parcel)
com.kabouzeid.gramophone.model.AbsCustomPlaylist
+ AbsCustomPlaylist (in: Parcel)
+ getSongs (context: Context)
com.kabouzeid.gramophone.model.smartplaylist.AbsSmartPlaylist
+ iconRes: int
+ writeToParcel (dest: Parcel ,flags: int)
+ hashCode ()
+ equals (obj: Object)
+ describeContents ()
+ AbsSmartPlaylist ()
+ clear (context: Context)
# AbsSmartPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.HistoryPlaylist
+ CREATOR: Creator<HistoryPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ HistoryPlaylist (context: Context)
# HistoryPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.LastAddedPlaylist
+ CREATOR: Creator<LastAddedPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ LastAddedPlaylist (context: Context)
# LastAddedPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.MyTopTracksPlaylist
+ CREATOR: Creator<MyTopTracksPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ MyTopTracksPlaylist (context: Context)
# MyTopTracksPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.ShuffleAllPlaylist
+ CREATOR: Creator<ShuffleAllPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ ShuffleAllPlaylist (context: Context)
# ShuffleAllPlaylist (in: Parcel)

Figure 3.2.1 Playlist hierarchy illustrating the LSP principle

Here, there is a concrete class, *gramophone/model/Playlist*, that has *AbsCustomPlaylist* as a child, who then has *gramophone/model/smartplaylist/AbsSmartPlaylist* as a child. Finally, that last class has 4 children. In this hierarchy, all the classes below override the functions according to the method signatures of the parent who declared it (same parameter types and same return type). This way, if the developer wants to pass an *AbsSmartPlaylist* object as a parameter, he doesn't have to worry if it is in fact an instance of a child, because if the override is done correctly, they can be treated the same.

This principle is really useful for the playlist functionality. The user can have different playlist types that have specific functions related to them. For example, here we can see a *gramophone/model/smartplaylist/HistoryPlaylist*, a *LastAddedPlaylist*, a *MyTopTracksPlaylist* and a *ShuffleAllPlaylist*. Of course, they can be treated differently from one another, but if for a part of the code we only want to manage a playlist, independently from its type, we could handle a *Playlist*, *AbsCustomPlaylist* or *AbsSmartPlaylist*, depending on our needs by taking advantage of the polymorphism.



```java
public class ClearSmartPlaylistDialog extends DialogFragment {

    @NonNull
    public static ClearSmartPlaylistDialog create(AbsSmartPlaylist playlist) {
        ClearSmartPlaylistDialog dialog = new ClearSmartPlaylistDialog();
        Bundle args = new Bundle();
        args.putParcelable("playlist", playlist);
        dialog.setArguments(args);
        return dialog;
    }
```

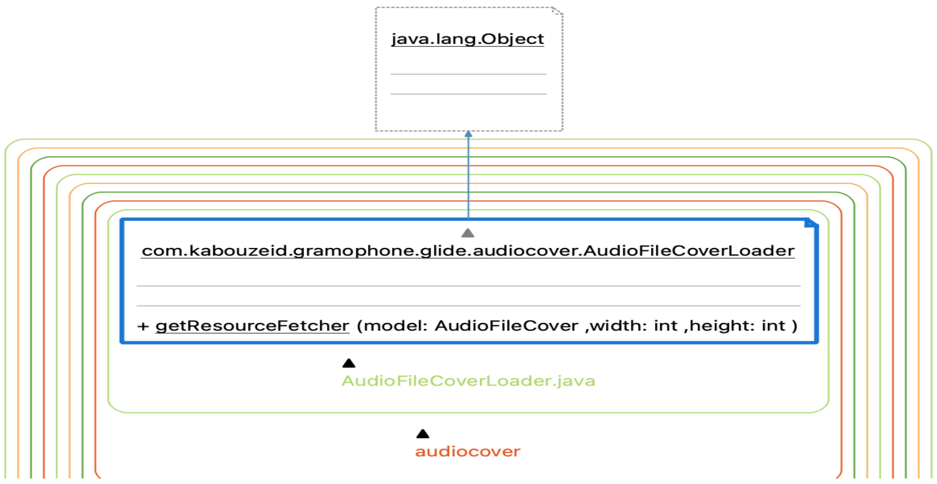
Figure 3.2.2 Code snippet to prove the polymorphism is used

As pictured above, in the *gramophone/dialogs/ClearSmartPlaylistDialog*, the create() function takes a *AbsSmartPlaylist* as a parameter. Since the latter is an abstract class, it actually absolutely has to accept a children class as a parameter, but it won't affect the behavior, since the LSP principle is respected and polymorphism can be used.

3.3. Open-Close Surgery Principle

In this principle, the programmer needs to create codes that are open for extension and close for modification. it means that the programmer can extend the behavior of this code without changing the source code. For example, there could be a payment class that works for different classes including credit cards, debit cards, and cash. Then if the programmer adds the PayPal class, the PayPal class can use the payment class. So, the payment class is open for extension but is closed for modification.

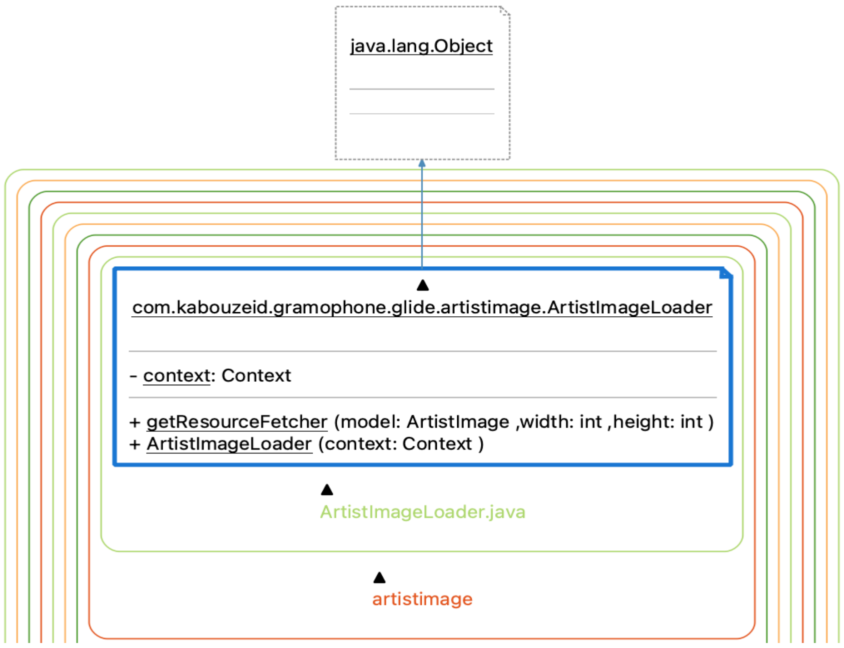
The interface of *StreamModelLoader [1]* is used in two classes, *gramophone/glide/audiocover/AudioFileCoverLoader* and *gramophone/glide/artistimage/ArtistImageLoader*. So, if there is an X class that wants to load the streaming model, it could be done by implementing *StreamModelLoader*. So, it is open to extension but close to modification. You can see the UML diagram of *AudioFileCoverLoader* which uses *StreamModelLoader* (*AudioFileCover*).



java.lang.Object
com.kabouzeid.gramophone.glide.audiocover.AudioFileCoverLoader
+ getResourceFetcher (model: AudioFileCover ,width: int ,height: int)
AudioFileCoverLoader.java
audiocover

Figure 3.3.1 AudioFileCoverLoader UML diagram proves the Open-Close Surgery Principle

The UML diagram shows how *ArtistImageLoader* uses *StreamModelLoader* (*ArtistImage*).



java.lang.Object
com.kabouzeid.gramophone.glide.artistimage.ArtistImageLoader
- context: Context
+ getResourceFetcher (model: ArtistImage ,width: int ,height: int)
+ ArtistImageLoader (context: Context)
ArtistImageLoader.java
artistimage

Figure 3.3.2 ArtistImageLoader UML diagram proves the Open-Close Surgery Principle

The functionality of *StreamModelLoader* is to help *AudioFileCoverLoader* and *ArtistImageLoader* to load their models.

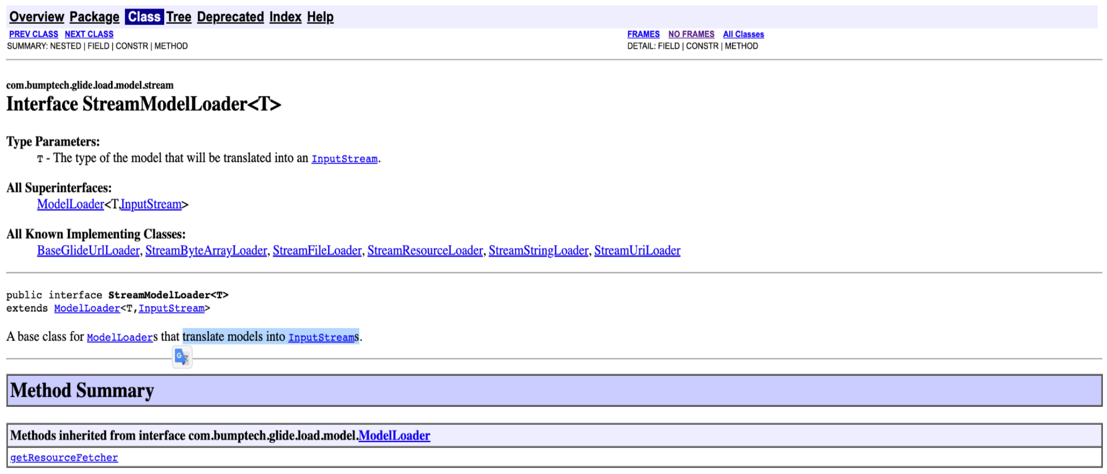
```java
public class AudioFileCoverLoader implements StreamModelLoader<AudioFileCover> {

    @Override
    public DataFetcher<InputStream> getResourceFetcher(AudioFileCover model, int width, int height) {
        return new AudioFileCoverFetcher(model);
    }

    public static class Factory implements ModelLoaderFactory<AudioFileCover, InputStream> {
        @Override
        public ModelLoader<AudioFileCover, InputStream> build(Context context, GenericLoaderFactory factories) {
            return new AudioFileCoverLoader();
        }
```

Figure 3.3.3 AudioFileCoverLoader class code proves the Open-Close Surgery Principle

```java
public class ArtistImageLoader implements StreamModelLoader<ArtistImage> {
    private Context context;

    public ArtistImageLoader(Context context) {
        this.context = context;
    }

    @Override
    public DataFetcher<InputStream> getResourceFetcher(final ArtistImage model, int width, int height) {

        return new ArtistImageFetcher(model, PreferenceUtil.getInstance(context).ignoreMediaStoreArtwork());
    }
```

Figure 3.3.4 ArtistImageLoader class code proves the Open-Close Surgery Principle

The role of StreamModelLoader is to translate models into InputStreams.


Overview Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD
FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD

com.bumptech.glide.load.model.stream
Interface StreamModelLoader<T>

Type Parameters:
T - The type of the model that will be translated into an InputStream.

All Superinterfaces:
ModelLoader<T,InputStream>

All Known Implementing Classes:
BaseGlideUrlLoader, StreamByteArrayLoader, StreamFileLoader, StreamResourceLoader, StreamStringLoader, StreamUriLoader

```
public interface StreamModelLoader<T>
extends ModelLoader<T,InputStream>
```

A base class for ModelLoaders that translate models into InputStreams.

Method Summary

Methods inherited from interface com.bumptech.glide.load.model.ModelLoader
getResourceFetcher

Figure 3.3.5 The role of StreamModelLoader [3]

4. Violation of SOLID Design principles

Although some SOLID principles were respected in the application Phonograph, we have identified an instance of a violated one. There were a lot of them, but we are going to look into the Interface Segregation Principle (ISP). This principle says that it is better to have multiple interfaces than having a single interface having multiple functions. This also applies to abstract classes. So, instead of having one class that all other classes inherit from, there should be more classes to inherit from, so the children only inherit from what they need.

As discussed in the previous section, we have a *gramophone/model/Playlist* hierarchy that looks like this:

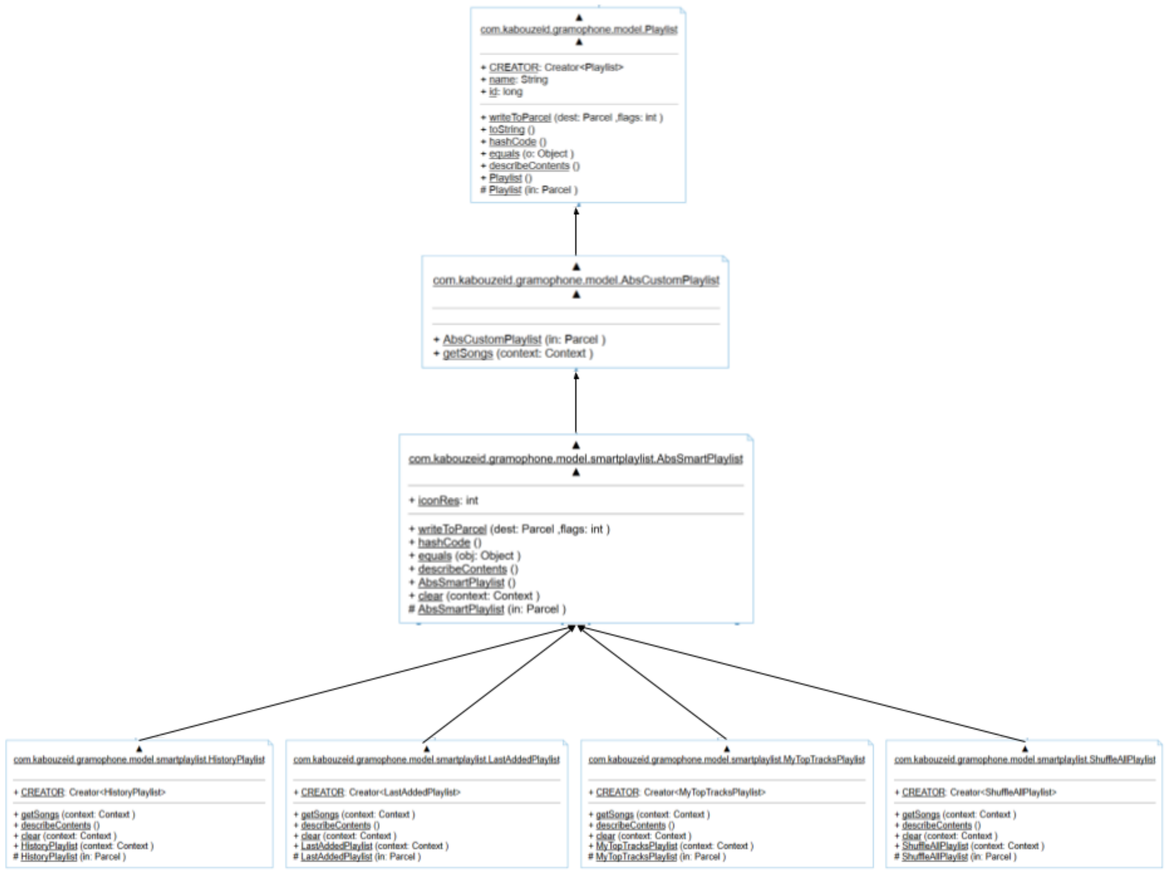
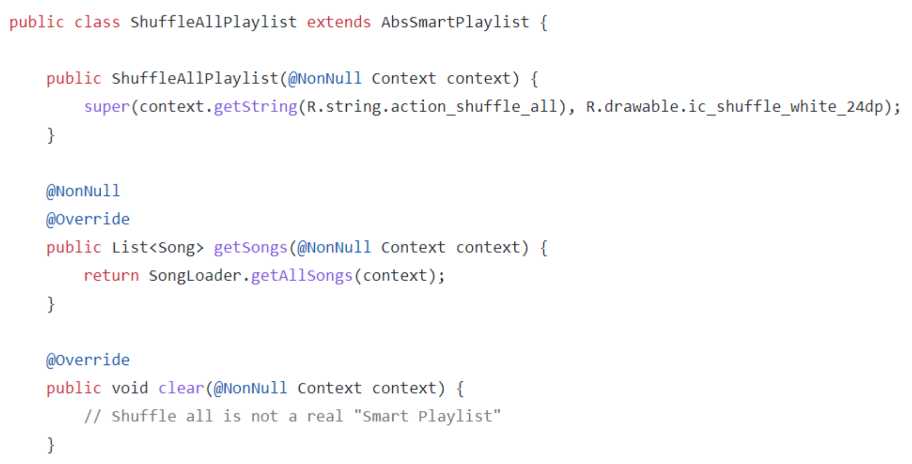

com.kabouzeid.gramophone.model.Playlist
+ CREATOR: Creator<Playlist>
+ name: String
+ id: long
+ writeToParcel (dest: Parcel ,flags: int)
+ toString ()
+ hashCode ()
+ equals (o: Object)
+ describeContents ()
+ Playlist ()
# Playlist (in: Parcel)
com.kabouzeid.gramophone.model.AbsCustomPlaylist
+ AbsCustomPlaylist (in: Parcel)
+ getSongs (context: Context)
com.kabouzeid.gramophone.model.smartplaylist.AbsSmartPlaylist
+ iconRes: int
+ writeToParcel (dest: Parcel ,flags: int)
+ hashCode ()
+ equals (obj: Object)
+ describeContents ()
+ AbsSmartPlaylist ()
+ clear (context: Context)
# AbsSmartPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.HistoryPlaylist
+ CREATOR: Creator<HistoryPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ HistoryPlaylist (context: Context)
# HistoryPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.LastAddedPlaylist
+ CREATOR: Creator<LastAddedPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ LastAddedPlaylist (context: Context)
# LastAddedPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.MyTopTracksPlaylist
+ CREATOR: Creator<MyTopTracksPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ MyTopTracksPlaylist (context: Context)
# MyTopTracksPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.ShuffleAllPlaylist
+ CREATOR: Creator<ShuffleAllPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ ShuffleAllPlaylist (context: Context)
# ShuffleAllPlaylist (in: Parcel)

Figure 4.1 Playlist hierarchy related to the ISP principle violation

At first glance, everything looks fine. We expect all the children of *gramophone/model/smartplaylist/AbsSmartPlaylist* to actually be a smart playlist and to behave the same. The problem is, when you dig deeper in the code, you can see that *gramophone/model/smartplaylist/ShuffleAllPlaylist* is not in fact a *AbsSmartPlaylist*, as you can see in the code snippet below:



```java
public class ShuffleAllPlaylist extends AbsSmartPlaylist {

    public ShuffleAllPlaylist(@NonNull Context context) {
        super(context.getString(R.string.action_shuffle_all), R.drawable.ic_shuffle_white_24dp);
    }

    @NonNull
    @Override
    public List<Song> getSongs(@NonNull Context context) {
        return SongLoader.getAllSongs(context);
    }

    @Override
    public void clear(@NonNull Context context) {
        // Shuffle all is not a real "Smart Playlist"
    }
```

Figure 4.2 Code snipped to illustrate the violation of ISP principle with ShuffleAllPlaylist

Without any doubt, this is a violation of the ISP principle, because the *clear()* method that *ShuffleAllPlaylist* inherits from *AbsSmartPlaylist* is not useful for that type of playlist. Right now, the issue is not major, because it simply doesn't do anything, and the return type is void, so it doesn't cause an error when it is called.

However, it is not good for understandability, because the UML diagram makes us believe that *ShuffleAllPlaylist* is a *AbsSmartPlaylist*, when it is in fact not. It should not be treated the same.

On another hand, it also affects the maintainability of the code if we want to add functionality in the future. Imagine we add another function applicable to all *AbsSmartPlaylist* and this time the function is supposed to return something. There is once again going to be an issue with *SuffleAllPlaylist*, because it will not be able to return anything relevant. Actually, this could lead to an LSP violation, because derived classes need to be able to be substituted for the classes that they inherit from.

To fix this issue, we must change the way the hierarchy is done for the *Playlist* and all his descendants to stop pretending *ShuffleAllPlaylist* is a *AbsSmartPlaylist*. The reason *ShuffleAllPlaylist* has been grouped with the other 3 playlist types is probably because some of *AbsSmartPlaylist*'s methods are useful. Also, this way, we can treat all playlist classes the same in the code.

So, we want to make sure that *ShuffleAllPlaylist* inherits from the methods implemented by *AbsSmartPlaylist* that are useful to this type of playlist, but not more than that.

To make it work, we would need a new class that would combine smart and non-smart playlist (*AbsPlaylist*), with all common functionalities, and then create a child class containing all smart playlist only functionalities (*AbsSmartPlaylist*). The *ShuffleAllPlaylist* can extend *AbsPlaylist* and can benefit from all the functions he used to, without the smart playlist related functions like *clear()*. The following figure represents the desired architecture.

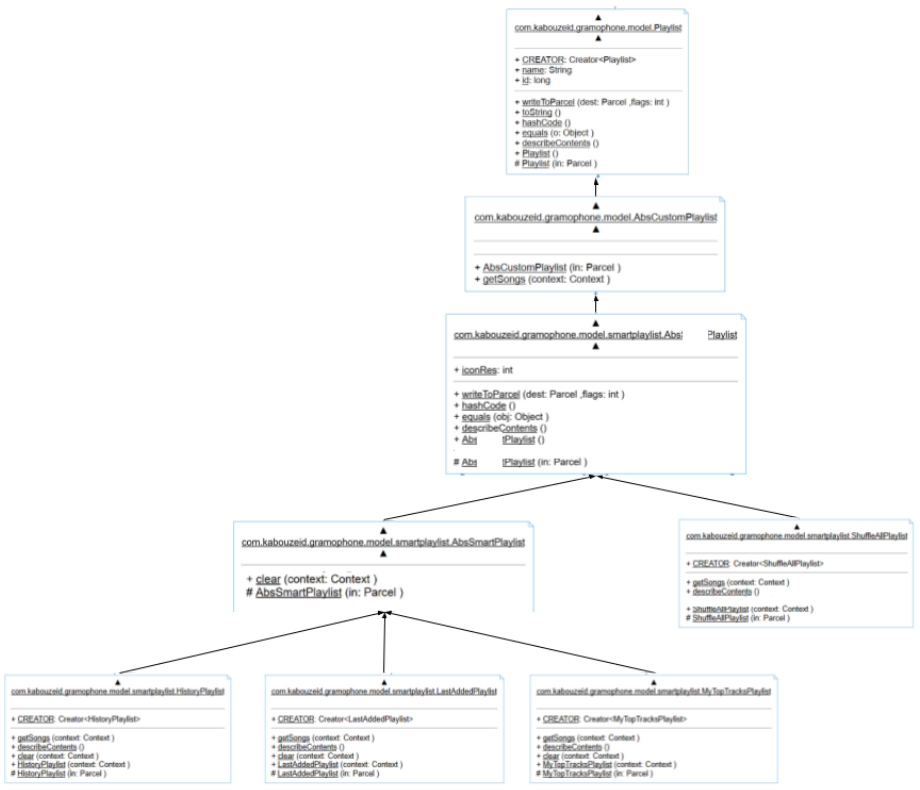
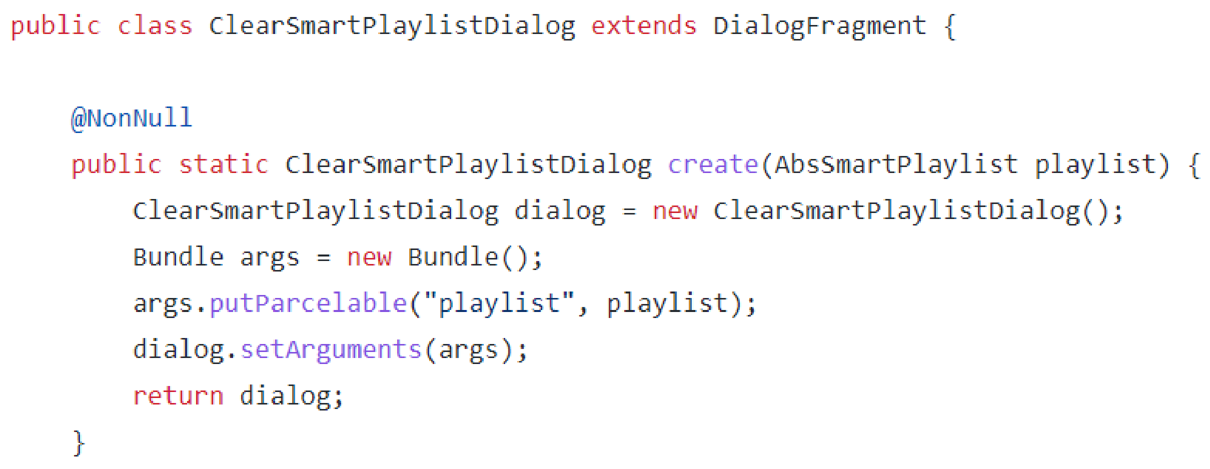

com.kabouzeid.gramophone.model.Playlist
+ CREATOR: Creator<Playlist>
+ name: String
+ id: long
+ writeToParcel (dest: Parcel ,flags: int)
+ toString ()
+ hashCode ()
+ equals (o: Object)
+ describeContents ()
+ Playlist ()
# Playlist (in: Parcel)
com.kabouzeid.gramophone.model.AbsCustomPlaylist
+ AbsCustomPlaylist (in: Parcel)
+ getSongs (context: Context)
com.kabouzeid.gramophone.model.smartplaylist.Abs Playlist
+ iconRes: int
+ writeToParcel (dest: Parcel ,flags: int)
+ hashCode ()
+ equals (obj: Object)
+ describeContents ()
+ Abs tPlaylist ()
# Abs tPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.AbsSmartPlaylist
+ clear (context: Context)
# AbsSmartPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.ShuffleAllPlaylist
+ CREATOR: Creator<ShuffleAllPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ ShuffleAllPlaylist (context: Context)
# ShuffleAllPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.HistoryPlaylist
+ CREATOR: Creator<HistoryPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ HistoryPlaylist (context: Context)
# HistoryPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.LastAddedPlaylist
+ CREATOR: Creator<LastAddedPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ LastAddedPlaylist (context: Context)
# LastAddedPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.MyTopTracksPlaylist
+ CREATOR: Creator<MyTopTracksPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ MyTopTracksPlaylist (context: Context)
# MyTopTracksPlaylist (in: Parcel)

Figure 4.3 New architecture respecting ISP with AbsPlaylist class

As explained, *AbsSmartPlaylist* and *ShuffleAllPlaylist* now both inherit from *AbsPlaylist*, which contains what *AbsSmartPlaylist* used to have, minus the smart related functions. So, the *clear()* function is now specific to *AbsSmartPlaylist*, and it has been removed from the *ShuffleAllPlaylist* class.

In the code, when we used to refer to *AbsSmartPlaylist*, we have to either change it to *AbsPlaylist*, when the execution is applicable to any playlist type, or leave it *AbsSmartPlaylist* when it only concerns smart playlist functionalities. This way, we will never call, for example, *clear()* on a non-smart playlist.

For example, the following function used to receive an *AbsSmartPlaylist* object, located in *gramophone/dialogs/ClearSmartPlaylistDialog* :

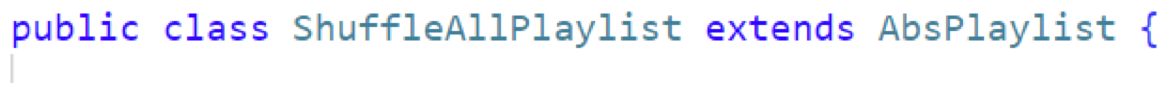


```java
public class ClearSmartPlaylistDialog extends DialogFragment {

    @NonNull
    public static ClearSmartPlaylistDialog create(AbsSmartPlaylist playlist) {
        ClearSmartPlaylistDialog dialog = new ClearSmartPlaylistDialog();
        Bundle args = new Bundle();
        args.putParcelable("playlist", playlist);
        dialog.setArguments(args);
        return dialog;
    }
```

Figure 4.4 Example where AbsSmartPlaylist would still be used

Without thinking, we could replace it with the new class, *AbsPlaylist*, but in this case, since the function is only working for smart playlist, we would leave it with the *AbsSmartPlaylist* parameter, which now only works for strictly smart playlists.

On the other hand, for the *ShuffleAllPlaylist* class found at *gramophone/model*, since it's not a smart playlist, it would now extend from *AbsPlaylist* instead of AbsSmartPlaylist:



```java
public class ShuffleAllPlaylist extends AbsPlaylist {
```

Figure 4.5 Example where AbsPlaylist would be used now instead of AbsSmartPlaylist

With this new architecture, we can easily add new methods only applicable to smart playlists.

In the future, if the developers want to add new playlists that are non-smart, they could add an abstract class (*AbsNotSmartPlaylist*) containing all the non-smart methods. The new hierarchy could look like this:

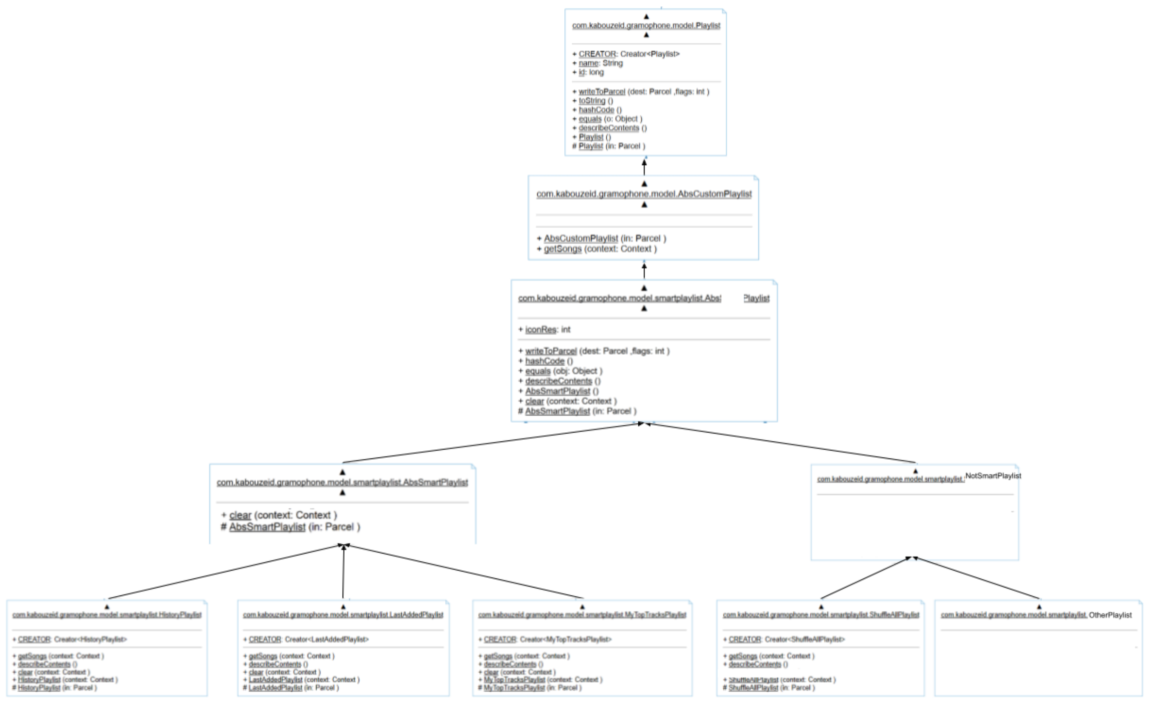
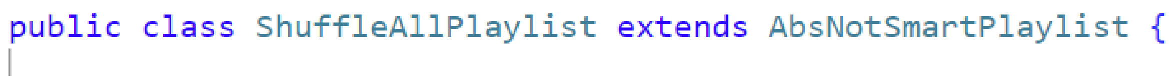

com.kabouzeid.gramophone.model.Playlist
+ CREATOR: Creator<Playlist>
+ name: String
+ id: long
+ writeToParcel (dest: Parcel ,flags: int)
+ toString ()
+ hashCode ()
+ equals (o: Object)
+ describeContents ()
+ Playlist ()
# Playlist (in: Parcel)
com.kabouzeid.gramophone.model.AbsCustomPlaylist
+ AbsCustomPlaylist (in: Parcel)
+ getSongs (context: Context)
com.kabouzeid.gramophone.model.smartplaylist.Abs Playlist
+ iconRes: int
+ writeToParcel (dest: Parcel ,flags: int)
+ hashCode ()
+ equals (obj: Object)
+ describeContents ()
+ AbsSmartPlaylist ()
+ clear (context: Context)
# AbsSmartPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.AbsSmartPlaylist
+ clear (context: Context)
# AbsSmartPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.NotSmartPlaylist
com.kabouzeid.gramophone.model.smartplaylist.HistoryPlaylist
+ CREATOR: Creator<HistoryPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ HistoryPlaylist (context: Context)
# HistoryPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.LastAddedPlaylist
+ CREATOR: Creator<LastAddedPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ LastAddedPlaylist (context: Context)
# LastAddedPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.MyTopTracksPlaylist
+ CREATOR: Creator<MyTopTracksPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ clear (context: Context)
+ MyTopTracksPlaylist (context: Context)
# MyTopTracksPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.ShuffleAllPlaylist
+ CREATOR: Creator<ShuffleAllPlaylist>
+ getSongs (context: Context)
+ describeContents ()
+ ShuffleAllPlaylist (context: Context)
# ShuffleAllPlaylist (in: Parcel)
com.kabouzeid.gramophone.model.smartplaylist.OtherPlaylist

Figure 4.6 Architecture if multiple non-smart playlists were needed

This other option is not much more complicated, and makes sure that the code will be able to be maintained if new non-smart playlists become necessary.

The *ShuffleAllPlaylist* would now extend from *NotSmartPlaylist*, just like showed in the figure below:



```
public class ShuffleAllPlaylist extends AbsNotSmartPlaylist {
```

Figure 4.7 Example how AbsSmartPlaylist would be replaced with a non-smart playlists class

In conclusion, this fixes this instance of the ISP principle that was violated and even prevents an LSP problem from occurring in the future, since the inheritance logic is more accurate and children will always be able to be treated like their parents. With those changes, the code will be easier to understand and maintain, which reflects a better code quality.

References

[1] Link for Phonograph Github repository: https://github.com/kabouzeid/Phonograph

[2] “Retrofit design pattern,” *ProgrammerSought*. [Online]. Available: https://programmersought.com/article/32613915952/. [Accessed: 18-Sep-2022].

[3] “The Catalog of Design Patterns,” *Refactoring Guru*. [Online]. Available: https://refactoring.guru/design-patterns/catalog. [Accessed: 20-Sep-2022].

[4] “Understanding about SOLID - Interface Segregation Principle,” *Manh Phan*. [Online]. Available: https://ducmanhphan.github.io/2020-01-15-Understanding-about-SOLID-part-4/. [Accessed: 24-Sep-2022].

[5] “Polymorphism in Java,” *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/polymorphism-in-java/. [Accessed: 25-Sep-2022].

[6] “StreamModelLoader,” *BumpTech Github.* [Online]. Available: https://bumptech.github.io/glide/javadocs/331/com/bumptech/glide/load/model/stream/StreamModelLoader.html. [Accessed: 26-Sep-2022].