POLYTECHNIQUE MONTRÉAL

Assignment #1
LOG8430e - Software Architecture and Advanced Design

Trimester: Fall 2021

Yankees

Alice Gong 1961605

Yuhan Li 1947497

Yanis Toubal 1960266

Groupe: 01

Présenté à :
Mohammadreza Rasolroveicy

École Polytechnique Montréal
À remettre le 4 octobre 2021

Question 1: Software Architecture Analysis

Only the *app* module has been considered for this part. Also, we would like to mention that the following diagrams were made with the UML online tool *diagrams.net* and the computer software *Understand*.

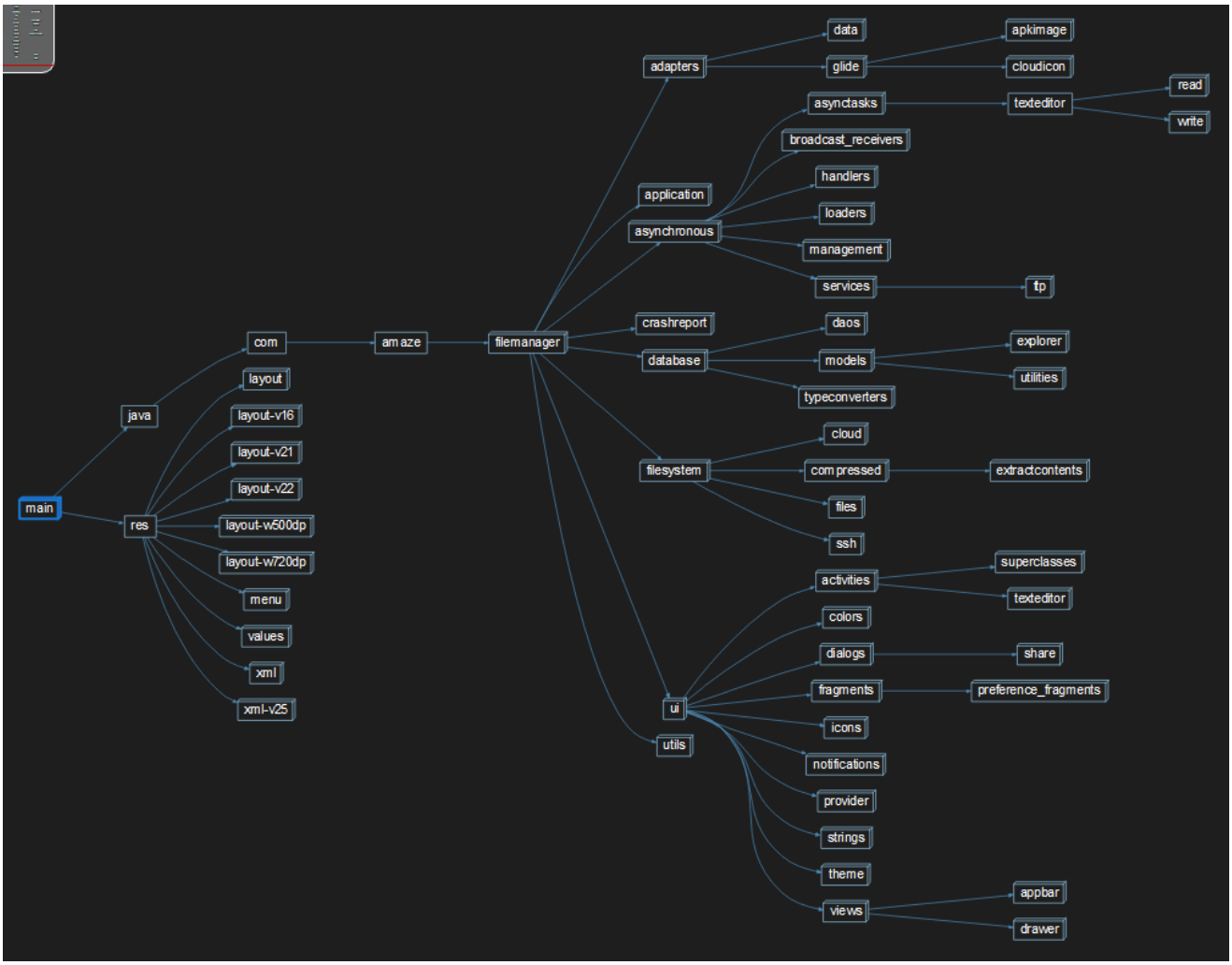

Figure 1. Overview of the AmazeFile App Architecture

Modules (packages):

- res
- ui/activites
- ui/fragments
- Reste ui/
- filesystem
- database
- asynchronous
- adapter

The overview above comes from the modules located in the app module, and more specifically the path: app > src > main > java. From this diagram, we identified one modular architectural style for the implementation of the system. We based our analysis on the standard organization pattern, which led us to identify an MVC architecture.

MVC

MVC stands for Model-View-Controller, which describes the three logical components that characterize this architecture. It is a common architecture used particularly when developing user interfaces. Each component is separated in a logical way, and plays an important role when it comes to developing specific aspects of an application.

Model

Firstly, the *Model* component is mainly dedicated to data handling. It is responsible for the communication with the database, for the storage of the application data and for managing the logic of the application. In our case, we identified the *database* module, the *filesystem* module, the *utils* module, the *application module* and the *asynchronous* module (modules in orange on Figure 2) as the ones representing the *Model* component.

We can identify the *database* module to be part of the *Model* since it gives a link to the database used, in addition to giving the logic and functionality to access it. Inside this module, we can note three other modules that serve a different purpose related to the logic of the database. For example, the *dao* module contains all the interfaces necessary to perform operations on the given databases and the *models* module contains all the classes needed to represent the database data logic.

The *filesystem* module, on the other hand, contains all the logic to deal with the files and directories on the user's device which includes operations such as adding, deleting, compressing, extracting and more.

Regarding the *utils* module, it is composed of many utility classes that contain utility methods which are used in many contexts around the whole application.

As for the *application* module, it mainly stores objects and logic in the global application state which means that the same instances are available during the whole application. Some examples are the database instances, the logic to run a *toast* and the logic to run code in the background.

Lastly, the *asynchronous* module contains all the classes that provide asynchronous operations for example searching files, moving files, compressing files, and more.

View

Secondly, we have the *View* component that is responsible for all the visual elements represented on the screen. In other words, this component offers a visualization of the data received or provided by the *Model* component.

For the AmazeFileManager application, we can say that the *res* module, and the *ui* module (except for *ui/activities* and *ui/fragments*) perfectly depict the role of the *View* (they are outlined in red on Figure 2). The *res* module contains many folders that define the general layout of the app's interface as well as other features contributing to it. Its folders mainly contain XML that define animations (*res/anim*), user interface layout (all the layout folders), application menus (*res/menu*) and simple values, such as color or style values (*res/values*). The folders *res/drawable* and *res/mipmap* also contain drawables or in other words, "graphic that can be drawn to the screen" (Google, n.d.).

As for the *ui* module, it also contains code relevant to the application's interface. It is mostly composed of custom widgets that are used in the *res* module. For example, the directory *ui/dialog* is composed of classes that create different types of dialog, such as a color picking dialog, or a dialog that appears when renaming a bookmark.

Controller

Finally, the *Controller* component serves as the logic part of the architecture. By reacting to events in the *View* and by updating data in the *Model*, the *Controller* is the one that establishes a relationship between the *View* and the *Model*. For our application, we concluded that the *activities* and *fragments* modules in ui, the *crashreport* and the *adapters* modules (in green on Figure 2) represent the *Controller*.

The activities and fragments in the *ui* module are considered to be the controller since they listen to events from the *View*, update the *View* as well as the *Model*. The activities serve as an entry point when opening a new app and navigating in the application. In the *AmazeFile* app, there are four activities, with the main one being *MainActivity*. We can note that *AboutActivity* is responsible for the "About" page, *DatabaseViewerActivity* is responsible for viewing the files and directories from the database and *PreferencesActivity* is responsible for the "Settings" page. Fragments can also be considered controllers since they can be viewed as pieces of an activity. In fact, fragments can only exist within an activity where they act as a controller on their own view.

As for the *crashreport* module, the activity is responsible for displaying an error page for the user while also reporting the issue to the developers of the app when an unexpected error occurs within the app.

Finally, an adapter in Android is an object that acts as a bridge between a view, or an ui component, and its data source (Google, n.d.). In some way, they direct the flow of data in an

appropriate format to the right view and thus, serves as a coordinator between the *View* and the *Model*. The *adapters* module contains different adapters, a folder of data to be displayed in the UI and a folder of *ViewHolders* (*adapter/holders*). A *ViewHolder* is an implementation that stores a view for a larger view (Google, n.d.). In many cases, this larger area designates a *RecyclerView*, which is an Android UI component that consists of a scrolling list where the ViewHolder consists of a list element. In brief, the *adapters* module creates the *ViewHolders* with the data that it contains which are in return displayed in a larger view.

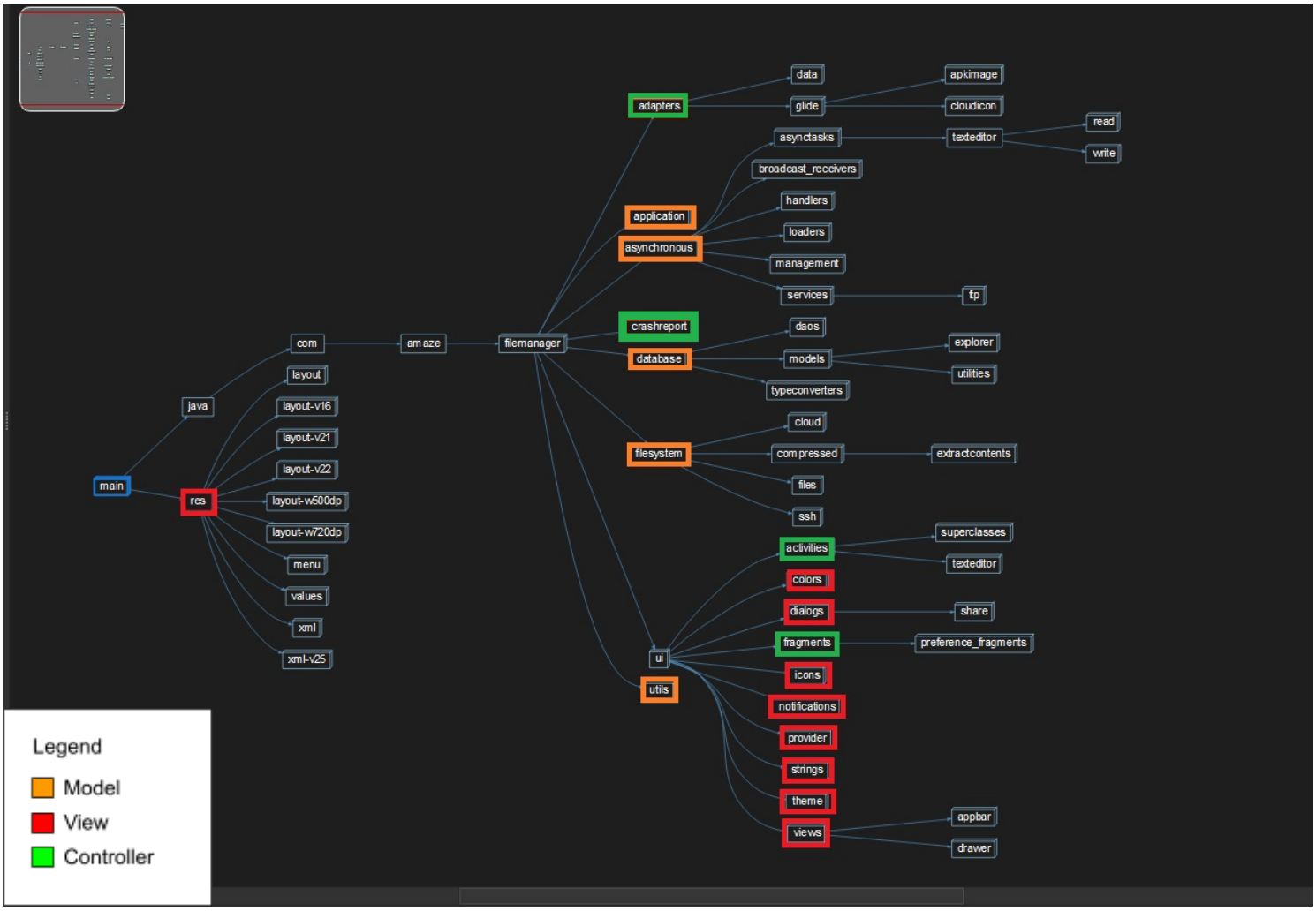

Figure 2. MVC representation in the architecture with a legend

Question 2: Design Patterns

Factory Method

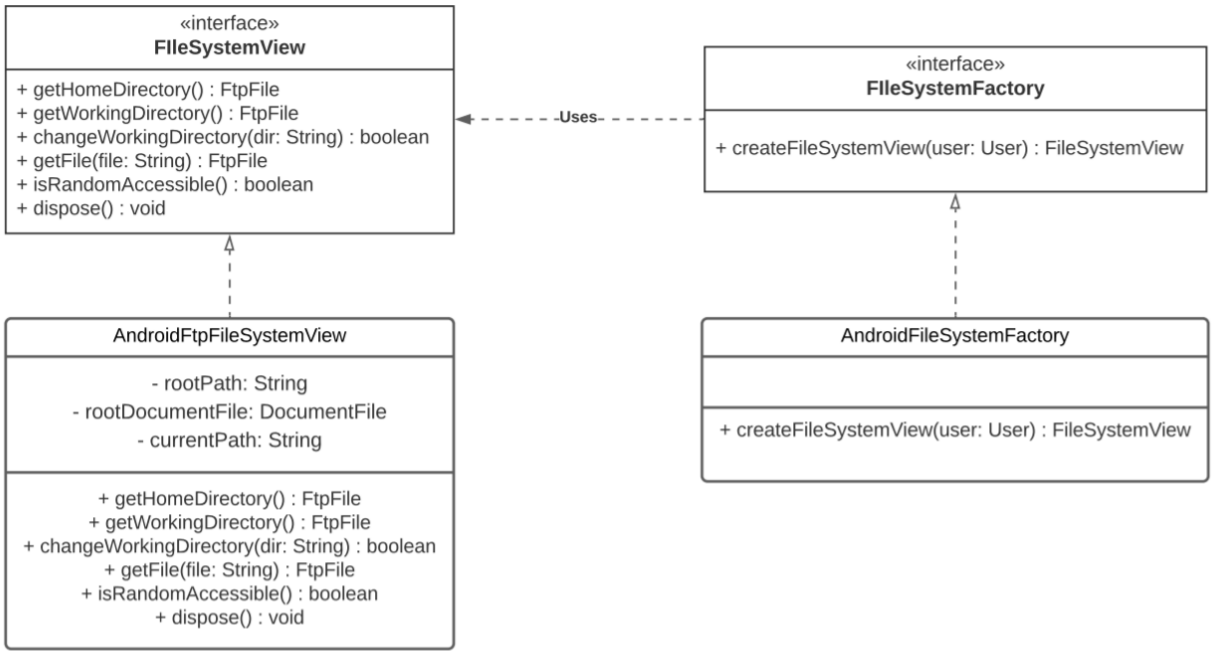

Figure 3. Diagram depicting the Factory Method's Architecture

Functionality

The Factory Method is a creational pattern, meaning that its main functionality is to make the instantiation of an object more straightforward and easily reusable. This pattern does that by defining an interface that allows the creation of multiple objects, but lets other subclasses decide which specific class to instantiate. Instead of specifying the concrete class each time we want to create an object (*new* operator), we can simply go through a factory method. Consequently, we can say that the Factory pattern is useful when we are handling many related objects.

Role in AmazeFile

We identified this pattern from the files located in the folder: app > src > main > java > com > amaze > filemanager > filesystem > ftpserver. In this case, the Factory Method pattern helps with the instantiation of the user's file system view, or in other word the object *AndroidFileSystemView*. This class can be seen as an API that allows the user to interact with files by reading, writing or managing them. More specifically, the whole architect has the interface named *FileSystemView.java*, that serves as the **Product** for the objects (Concrete

Products) produced by the factory method from the Creator. In the Factory Method pattern, a product is known as the interface responsible for templating all the concrete objects (known as Concrete Product) produced by the factory method. In our case, *AndroidFtpFileSystemView.kt* can be considered as a **Concrete Product** that implements *FileSystemView*, as we can observe in the relationship represented on the Figure 3 below. As we mentioned, the factory method is the one responsible for replacing direct object construction. This method is declared in a class known as the **Creator**, and it is an abstract method at first. This way, subclasses from the **Creator** can override the base factory method so that we can return the type of product we wish. Moreover, since the Creator is also an interface, it has **Concrete Creators** that can be represented by the file *AndroidFileSystemFactory*. In our situation, the **Creator** can be seen as the file *FileSystemFactory.java* since it has a function `FileSystemView createFileSystemView(User user)`. This function acts like a factory method that creates concrete products of the interface *FileSystemView*, hence the relationship between the two classes represented in Figure 3. Therefore, because of the different types of concrete objects and the easy way of creating products, the coupling between classes is significantly reduced and the code is a lot more flexible.

Singleton

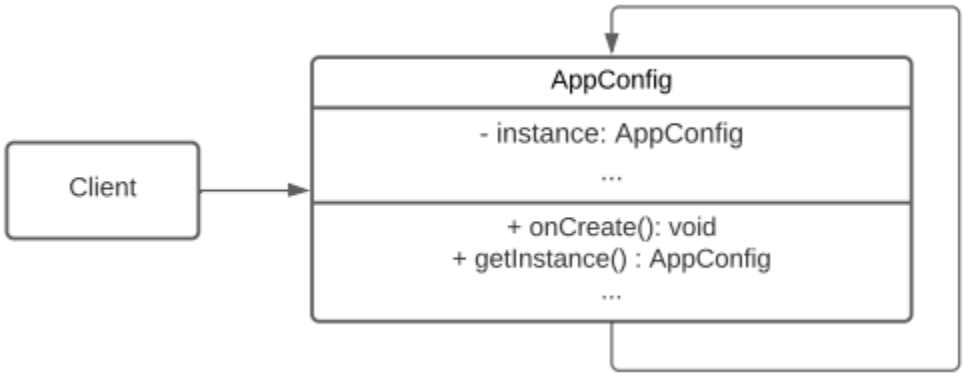

Figure 4. Diagram Depicting the Singleton Pattern's Architecture

Functionality

Just like the Factory pattern, the Singleton pattern is also a creational pattern. This means that its functionality is to simplify the project using techniques that revolve around the instantiation of an object. In the case of the Singleton pattern, the main goal is to ensure that a class can only have one instance while having a global access point. This can be done by making sure the default constructor of the Singleton class is private (prohibits an instantiation with *new* operator) and that the class has a static method as the constructor. Therefore, the Singleton should be considered if we need to have a stricter control over global variables, or if a class should have only one instance available to the clients.

Role in AmazeFile

The Singleton we identified is the class AppConfig, located in the path app > src > main > java > com > amaze > filemanager > application > AppConfig.Java. As its name depicts, this class acts as the base configuration for the application, which means that there can only be one instance at a time, hence the reason why we set the class as a Singleton. In our case, AppConfig's constructor is set as a private static object, which makes it hidden to the classes outside. To get the Singleton object, there should be only one way and it's with a static method. This is exactly the case with AppConfig, as it declares its static method with the function `public static synchronized AppConfig getInstance(){return instance;}` which returns the same instance of its own object as observed in Figure 4. This way, we let the clients access the same AppConfig instance throughout the entire program. Finally, it is important to note that this Singleton pattern does not follow its conventional structure, as there is not a default private constructor for the object. Instead of always going through the instantiation of the singleton class at the beginning, there is a public method called onCreate(), represented on the figure below, that initializes the instance (line 93: `instance = this;`).

Builder

```
178     builder
179         .customView(dialogView,  wrapInScrollView: false)
180         .widgetColor(accentColor)
181         .theme(m.getAppTheme().getMaterialDialogTheme(m.getApplicationContext()))
182         .title(title)
183         .positiveText(positiveButtonText)
184         .onPositive(positiveButtonAction);
```

Figure 5. Code snippet of *GeneralCreationDialog* class

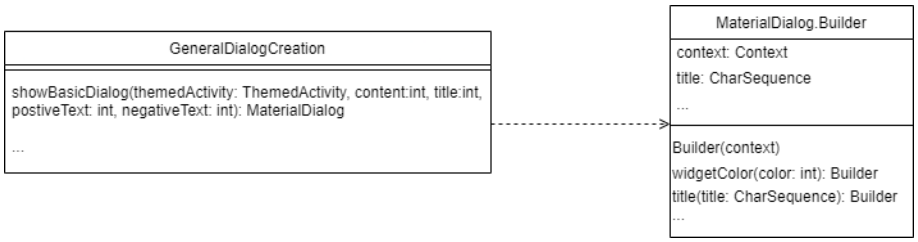

Figure 6. Class diagram illustrating the Builder pattern

Functionality

Lastly, just like the aforementioned patterns, the Builder design pattern is also creational. As its name implies, this concept helps build complex objects. Imagine we were to create an object which requires instantiating many fields; we would have to use a lengthy constructor every time we create this object. Some of the parameters might not even be necessary in some cases.

This is problematic, as the code quickly becomes hard to read, inconsistent and thus, more easily error-prone. The Builder pattern becomes handy in this case since it separates the construction of an object from its representation. It provides an easy way to build the object in a step-by-step approach while providing the required parameters. This would allow building different representations of the same object.

Role in AmazeFile

For this design pattern, we will talk about *GeneralDialogCreation* and *MaterialDialog.Builder* classes. Below (Figure 5) is a code snippet of an example of the Builder pattern in *GeneralDialogCreation*. We can clearly see the step by step creation of a dialog, while only precising the necessary parts. Theoretically, the Builder pattern should consist of a *Director*, *Builder* and *ConcreteBuilder* objects. The *Builder* is an abstract class or an Interface that defines the necessary steps in order to create the object in question. As for *ConcreteBuilder*, it inherits from *Builder* and implements the methods initially defined in the base class on top of defining its own methods. However, in our case, the *Builder* abstract interface does not exist, as shown in Figure 6. Instead, all the building methods are defined in the *ConcreteBuilder* which is directly used by the *Director*. As for the *Director*, its main role is to define the order in which we instantiate the object built by the Builder. We identified the GeneralDialogCreation as the **Director** class, since it calls the Builder and defines the order for the construction steps.

Question 3: SOLID Design Principles

This section will be about the SOLID design principles found within AmazeFile.

Single Responsibility Principle

The first principle that we observed is the single responsibility principle (SRP). In summary, this concept dictates that a class should only possess one responsibility. Therefore, if a class has one purpose, it should only have one reason to change. This is beneficial, because it reduces dependencies within a class, and thus helps maintain low coupling.

In this example, we will be talking about *EncryptedStringTypeConverter.java*, a class that encrypts and decrypts strings (we can presume that the strings in question are passwords from the name of the methods).

```java
public class EncryptedStringTypeConverter {

  private static final String TAG = EncryptedStringTypeConverter.class.getSimpleName();

  @TypeConverter
  public static StringWrapper toPassword(String encryptedStringEntryInDb) {
    try {
      return new StringWrapper(
          CryptUtil.decryptPassword(AppConfig.getInstance(), encryptedStringEntryInDb));
    } catch (Exception e) {
      android.util.Log.e(TAG, msg: "Error decrypting password", e);
      return new StringWrapper(encryptedStringEntryInDb);
    }
  }

  @TypeConverter
  public static String fromPassword(StringWrapper unencryptedPasswordString) {
    try {
      return CryptUtil.encryptPassword(AppConfig.getInstance(), unencryptedPasswordString.value);
    } catch (Exception e) {
      android.util.Log.e(TAG, msg: "Error encrypting password", e);
      return unencryptedPasswordString.value;
    }
  }
}
```

Figure 7. Code implementation of the class *EncryptedStringTypeConverter*

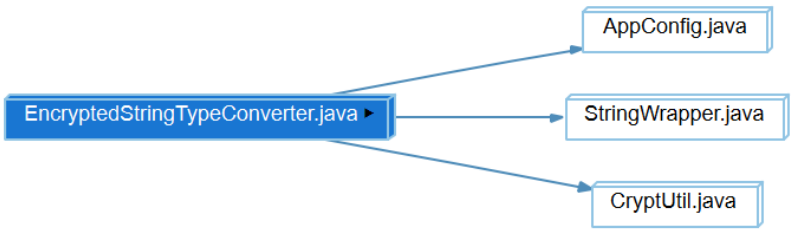

Figure 8. *EncryptedStringTypeConverter*'s class dependencies

From the figure above, we see that *EncryptedStringTypeConverter* depends on three classes, such as *AppConfig*, *StringWrapper* and *CryptUtil*. It also has a constant TAG that is used by its methods upon error. More precisely, when an exception is thrown by *toPassword* and *fromPassword*, the error is logged with TAG identifying the class where it happened. It is fair to say that this class possesses a fairly high cohesion as well as low coupling. When it comes to its methods, *EncryptedStringTypeConverter* implements two types of operations: *toPassword* and *fromPassword*. The first method, *toPassword*, decrypts a string that is saved in the database (the parameter *encryptedStringEntryInDb*). As for *fromPassword*, this method does the opposite. And so, the class' sole purpose in the application is to encrypt the users' password so that it can be stored inside the database, and to decrypt encrypted passwords inside the database in order to retrieve their initial form.

Open-Closed Principle

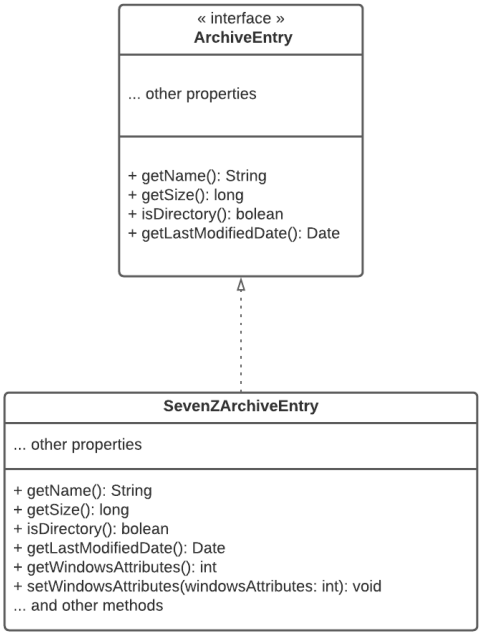

Figure 9. Class diagram illustrating *ArchiveEntry* and *SevenZArchiveEntry*

The second principle we have identified is the open-closed principle (OCP). If it were to be summed up in one sentence, it would be that software entities (such as classes, functions or modules) should be open for extension but closed for modification. In other words, code should be written in a way in which we can add new functionalities instead of modifying the existing code. Since this creates a certain separation between the existing and modified code, we avoid triggering or creating potential bugs and thus, breaking the application. We will be discussing the interface *ArchiveEntry* and the class *SevenZArchiveEntry* (see Figure 9) to demonstrate OCP.

First of all, *ArchiveEntry* is defined by the Apache Commons Compress library. As described in the code, this interface represents an entry of an archive. As for *SevenZArchiveEntry*, this class designates an entry in an 7z archive. These classes are especially useful when it comes to compressing files. When a single or multiple files are compressed, they form an “archive entry” and their sizes reduce significantly, which frees up space.

As we can see in Figure 9, *SevenZArchiveEntry* contains its parent’s methods, such as *getName*, *getSize*, *isDirectory* and *getLastModifiedDate*. These are important methods for working archive entries. Furthermore, it also contains methods that are not found in *ArchiveEntry*, such as *getHasCrc*, *setHasCrc*, *getCompressedCrc* and others. These additional methods are necessary to manipulate 7z archive entries precisely but are not needed nor supported by all types of archive entries. For example, if we take another look into its code, *SevenZArchiveEntry* contains methods useful to retrieve information about an entry’s Windows

attributes, such as *getWindowsAttributes*. A rare case of an archive entry not being supported by the Windows operating system is 7zX; 7zX archive entries can only be used on macOS, and thus, they would not need to possess Windows attributes.

In this case, these classes comply with LSP because if we needed to add more methods to handle 7z archive entries, we could simply add them directly into *SevenZArchiveEntry* instead of modifying *ArchiveEntry*. Moreover, if ever we wanted to create a new class to designate a different format of archive entry, such as JAR or RAR, we would not have to modify the *ArchiveEntry* interface. Instead, we could just extend upon it.

Liskov Substitution

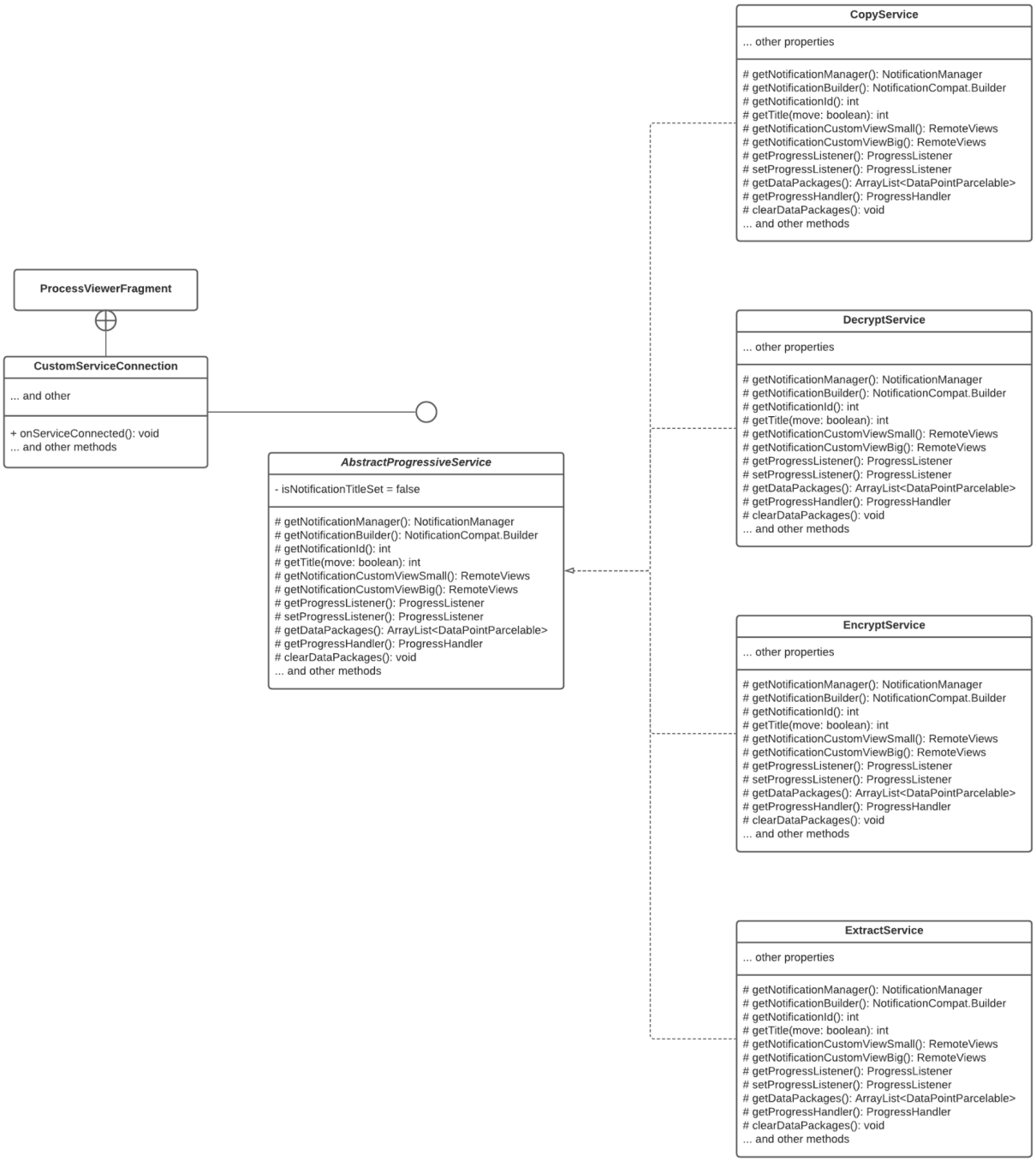

Figure 10. Class diagram illustrating *AbstractProgressiveService*

Finally, we have also observed the Liskov Substitution Principle (LSP). Simply put, we should be able to replace parent classes with their children without breaking the application. In other words, the subclasses should display similar behavior to their parents to avoid undesired side effects. To comply with this, the method of a child class needs to accept the same parameters

as its parent, and its methods should also be able to support its children's method's return value. In other words, preconditions cannot be enforced in subclasses, and postconditions cannot be more relaxed. For this principle, we will talk about *AbstractProgressiveService* and its children classes. *AbstractProgressiveService* is an abstract class and designates an Android service, which is a component that can perform long-running operations in the background (Google, n.d.). This superclass declares methods which allow us to obtain information about the service's progress, and illustrate it with a progress bar or a notification. As illustrated in Figure 10, the subclasses are *CopyService*, *DecryptService*, *EncryptService* and *ExtractService*. Since AmazeFileManager is an application for file management, *CopyService* is a service that allows copying files and moving them in different directories. *EncryptService* is useful for encrypting a file, which helps protect its content. *DecryptService*, on the other hand, decrypts a file, or in simpler terms, converts the encrypted file back into its original form. Finally, as its name states, *ExtractService* extracts files from a compressed archive.

The methods shown in Figure 10 are the abstract methods in *AbstractProgressiveService* and are overridden in the subclasses *CopyService*, *DecryptService*, *EncryptService* and *ExtractService*. Upon closer examination of their code, we can see that these methods accept identical input parameters as the method of their parent, which is also the case for their return values. Also, since all of *AbstractProgressiveService*'s subclasses inherit its members and methods, therefore, we can say that these subclasses comply with their parent's behavior. Let's demonstrate that with an example.

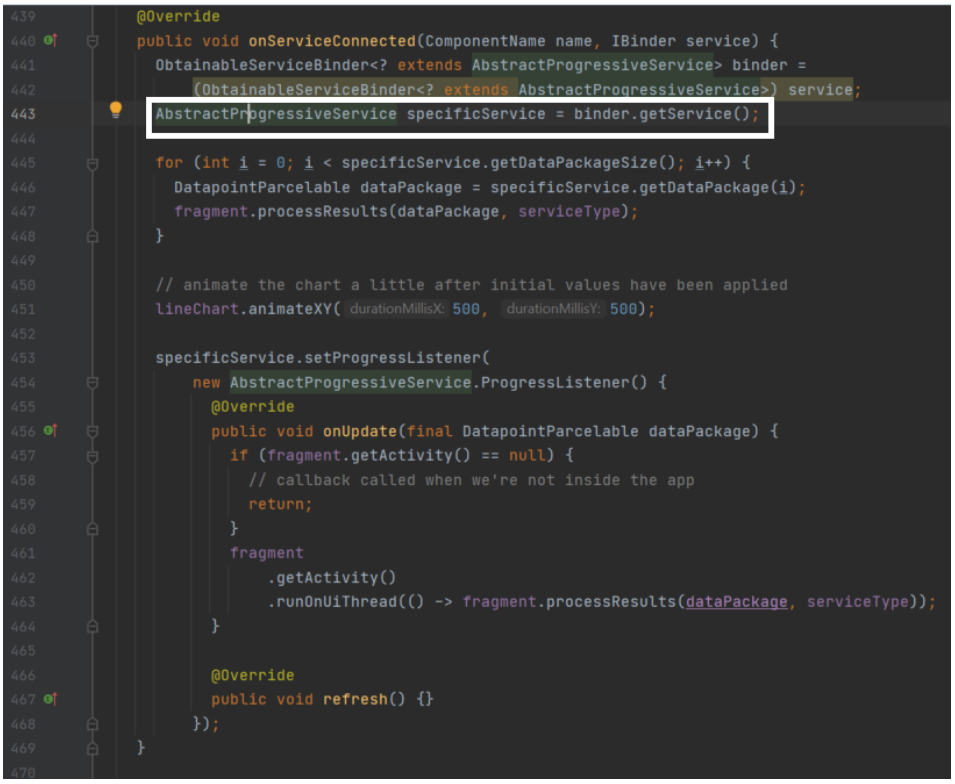

Figure 11. Code snippet of the method *onServiceConnected* of the class *CustomServiceConnection*

As illustrated in Figure 11, the class *CustomServiceConnection* utilizes *AbstractProgressiveService*, more precisely, the method *onServiceConnected*. Below is a code snippet of its implementation. We can see that on line 443, we create a variable of type *AbstractProgressiveService*, and iterate upon it on line 445. Given the fact that *CopyService*, *DecryptService*, *EncryptService* and *ExtractService* inherited the method *getDataPackageSize* from its parent, we can replace it with any of its children. Furthermore, from a subclass to another, the behaviors are actually quite similar. For example, whether it be *EncodeService*, *ExtractService* or *DecryptService*, *getDataPackages* returns a list of data packages, whose goal is to provide the necessary information to be used in the Fragment displaying the services' progress.

Question 4: Violations of SOLID Design Principles

Violation of the Interface Segregation Principle

```
abstract class CoderBase {
    private final Class<?>[] acceptableOptions;
    private static final byte[] NONE = new byte[0];

    /**
     * @param acceptableOptions types that can be used as options for this codec.
     */
    protected CoderBase(final Class<?>... acceptableOptions) {
        this.acceptableOptions = acceptableOptions;
    }
```

Figure 12: Code snippet of *CoderBase* class

```
    /**
     * @return a stream that reads from in using the configured coder and password.
     */
    abstract InputStream decode(final String archiveName,
        final InputStream in, long uncompressedLength,
        final Coder coder, byte[] password) throws IOException;

    /**
     * @return a stream that writes to out using the given configuration.
     */
    OutputStream encode(final OutputStream out, final Object options) throws IOException {
        throw new UnsupportedOperationException("method doesn't support writing");
    }
```

Figure 13: Code snippet of *Deflate64Decoder* class showing methods *encode* and *decode*

It should be noted that only the relevant parts of the *CoderBase* class have been shown in the screenshots. From the figure 13, we can see that the abstract class *CoderBase* implements the method *encode*, which throws an *UnsupportedOperationException* as the default behavior. This most likely means that not all the classes extending *CoderBase* will need an *encode* method. We can also see that the *decode* method is abstract, which means that the subclasses are obligated to provide their own implementation.

```
static class Deflate64Decoder extends CoderBase {
    Deflate64Decoder() { super(Number.class); }

    @SuppressWarnings("resource") // caller must close the InputStream
    @Override
    InputStream decode(final String archiveName, final InputStream in, final long uncompressedLength,
            final Coder coder, final byte[] password)
        throws IOException {
        return new Deflate64CompressorInputStream(in);
    }
}
```

Figure 14: Code snippet of *Deflate64Decoder* class

```
static class BCJDecoder extends CoderBase {
    private final FilterOptions opts;
    BCJDecoder(final FilterOptions opts) { this.opts = opts; }

    @Override
    InputStream decode(final String archiveName, final InputStream in, final long uncompressedLength,
            final Coder coder, final byte[] password) throws IOException {
        try {
            return opts.getInputStream(in);
        } catch (final AssertionError e) {
            throw new IOException("BCJ filter used in " + archiveName
                                  + " needs XZ for Java > 1.4 - see "
                                  + "https://commons.apache.org/proper/commons-compress/limitations.html#7Z",
                                  e);
        }
    }

    @SuppressWarnings("resource")
    @Override
    OutputStream encode(final OutputStream out, final Object options) {
        return new FlushShieldFilterOutputStream(opts.getOutputStream(new FinishableWrapperOutputStream(out)));
    }
}
```

Figure 15: Code snippet of *BCJDecoder* class

By looking at the class *Deflate64Decoder,* in figure 14, which extends the *CoderBase* abstract class and doesn't override the encode method, we found an Interface Segregation principle violation. Since the class doesn't need this method but is given by the abstract class (which by default throws an error), there is clearly a violation of ISP. As opposed to that, the *BCJDecoder* does override the *encode* method with it's own implementation, so we can see that this class needs this method.

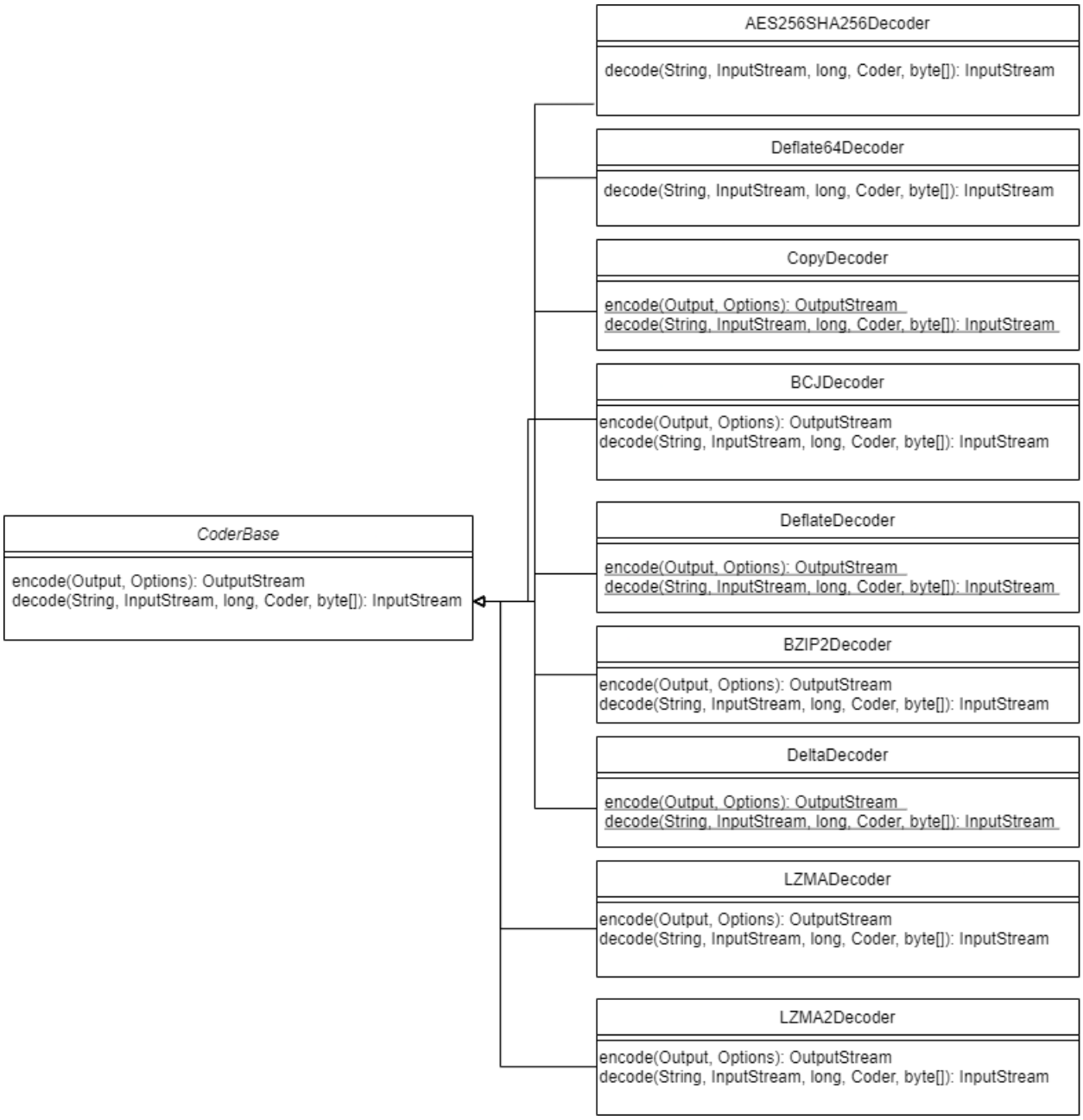

Figure 16: Class diagram of the Interface Segregation principle violation

From the figure 16, where only the relevant methods have been represented, we can see that all the Decoder classes extend from the *CoderBase* and, since *decode* is abstract, all of the subclasses have their own implementation. As for the *encode* method, only the *AES256SHA256Decoder* and the *Deflate64Decoder* don't implement it, which means that the default behavior for those classes is to throw an error.

Some problems could arise from that, for example, it would constitute a Liskov substitution principle violation since the behavior would be different from a subclass to another: some classes would have their own implementation, while others would throw an error. That could cause many unexpected behaviors when using the application. Another problem that could occur in the future is the creation of a new class that would perform an *encode* but wouldn't need a *decode*. With the current implementation, we would be forced to implement a method even if we didn't need it, which would create unnecessary code.

```
interface EncoderBase {

    /**
     * @return a stream that reads from in using the configured coder and password.
     */
    OutputStream encode(final OutputStream out, final Object options) throws IOException;
}
```

Figure 17: Code snippet of the new EncoderBase interface

```
interface DecoderBase {

    /**
     * @return a stream that reads from in using the configured coder and password.
     */
    InputStream decode(final String archiveName,
                       final InputStream in, long uncomressedLength,
                       final Coder coder, byte[] password) throws IOException;
}
```

Figure 18: Code snippet of the new DecoderBase interface

The steps used to refactor the code were to first create the two new interfaces which would respectively contain the *encode* method and the *decode* method. The next step was to move each method from the *CoderBase* class to their respective interfaces. Even though separating the *decode* method wasn't a necessity since It wasn't directly causing an ISP violation, it made the code more readable, less coupled and could prevent such violation in the future. It also felt logically more accurate to separate the two actions.

By separating the *encode* and *decode* methods in two different interfaces, we lower the coupling and It makes it a lot easier to understand the code by separating the logic of each action. By doing it this way, each subclass only has to implement the interface's methods that they need thus they aren't forced to implement an unnecessary class.

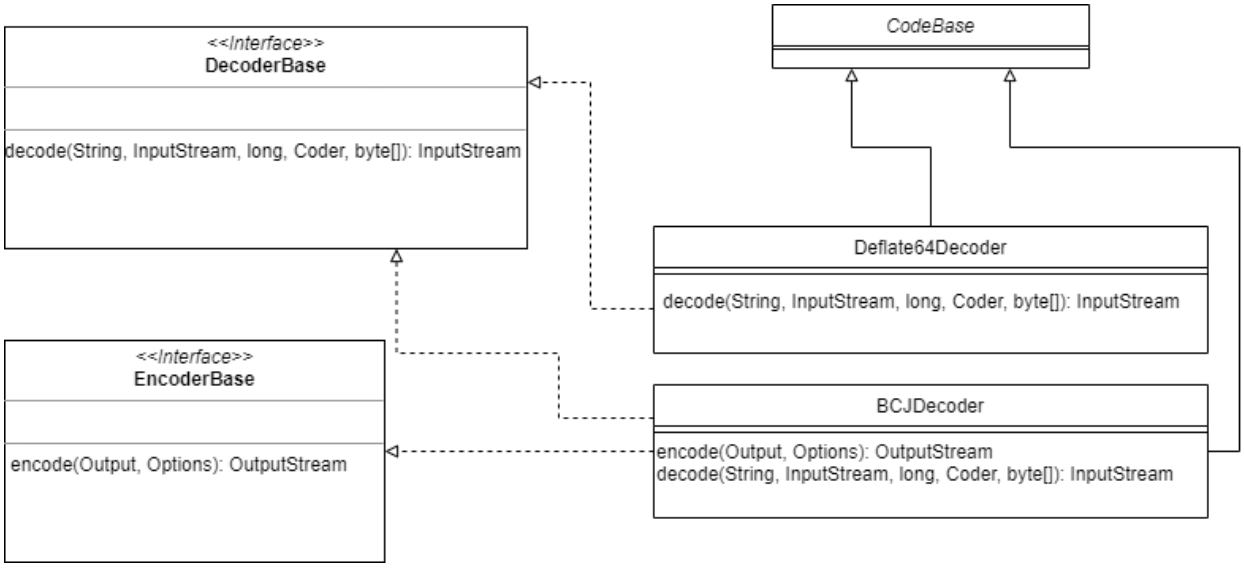

Figure 19: Class diagram of the new solution

The class diagram in the figure 19 represents the solution in action with only two subclasses (for simplicity sake), where *Deflate64Decoder* represents a subclass which doesn't implements encode and *BCJDecoer* represents a subclass which implements both *encode* and *decode*. It should be noted that the other subclasses shown previously would fall into one of the two categories.

The implemented solution does solve the problems mentioned previously. Firstly, the Liskov substitution principle isn't violated anymore since there are no errors thrown by not implemented methods. Secondly, implementing a new class with only the *decode* method wouldn't need the unnecessary implementation of the *encode* method since the two methods are now completely separated. The opposite also holds true if a new class would only need the *encode* method without a *decode* method.

References

Google. (n.d.). *Adapters*. Android for Developers. https://developer.android.com/reference/android/widget/Adapter

Google. (n.d.). *Drawable Resources*. Android for Developers. https://developer.android.com/guide/topics/resources/drawable-resource

Google. (n.d.). *RecyclerView.ViewHolder*. Android for Developers. https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.ViewHolder

Google. (n.d.). *Services*. Android for Developers. https://developer.android.com/guide/components/services

JGraph LTD. (n.d.). *diagrams.net*. Flowchart Maker and Online Diagram Software. https://app.diagrams.net/

Scientific ToolWorks, Inc. (2019). *Understand* (Version 5.1) [Computer Software]. https://www.scitools.com/