Wydział Informatyki, Elektroniki i Telekomunikacji
Katedra Informatyki

Pastebin w wersji zorientowanej na środowisko mobilne z klientem pozwalającym na oba kierunki przeklejania.

Dokumentacja techniczna

Kierunek, rok studiów:
Informatyka, rok 3
Przedmiot:
Inżynieria Oprogramowania
Prowadzący zajęcia:
mgr inż. Witold Rakoczy

Grupa (projekt):
3

Rok akad: **2012/2013**
Semestr: **letni**

Zespół autorski:
Michał Kowalski
Mateusz Sękara
Maciej Bassara
Sławomir Kulig

Spis Treści

Dokumentacja techniczna

1. Technologia wykonania

Narzędzia wykorzystywane to głównie API dla systemu Android oraz biblioteki pozwalające na zarządzanie chmurą i automatyczną sycnhronizację danych udostępnione przez Microsoft Azure

Mobilny klient jest aplikacją dedykowaną na systemy Android (wersja 2.2 - 4.2).
Aplikacja została stworzona w oparciu o następujące narzędzia:

- ADT (v 21.1.0) - środowisko w którym została stworzona aplikacja
- Android SDK (Android 2.2 API 8) – obsługa systemu Android 2.2 - 4.2, zainstalowanie aplikacji na emulatorze dołączonym do SDK
- Biblioteka google-gson (v 2.2.2) - obsługa mapowania notatake z postaci obiektowej na postać tabeli i odwrotnie.
- Biblioteka Windows Azure Mobile Services (v 0.2.0) - obsługa systemu chmurowego azure, dostęp do bazy danych, autentykacja za pomocą konta Google, pobieranie, wysyłanie i uaktualnianie rekordów w bazie.
- Google Apis - usługa autoryzują użytkowników na podstawie której przyznawany jest dostęp do bazy danych Azure

2. Spis utworzonych modułów i pakietów

2.1. GUI

- Obsługa głównych widoków czyli przeglądania notatek i ustawień: **pl.io.pastebin.app.activities**
- Wygląd i obsługa głównego widoku (czyli przeglądania notatek i edycji): **pl.io.pastebin.app.tabs.***

2.2. Zarządzanie bazą danych

- Cały moduł do zarządzania bazą danych (interfejs oraz zaimplementowana obsługa systemu MS Azure): **pl.io.pastebin.databases**

2.3. Pastebin API

- API do obsługi systemu Pastebin stworzone na potrzebny aplikacji w Javie: **pl.io.pastebin.pastebin_com**

2.4. Serializacja i zapis do pliku

- Obsługa serializacji i deserializacji na karcie pamięci: **pl.io.pastebin.app.serializer**
- Moduł wspierający serializację, zawiera zaimplantowany algorytm szyfrowania RC4 oraz specjalnie przygotowane strumienie: **pl.io.pastebin.app.encryption**
- Eksportowanie notatek do pliku tekstowego: **pl.io.pastebin.app.explorer**

2.5. Operacje na schowku i demony

- Cały moduł to zarządzania demonami i operacjami na schowku w zależności od systemu Android: **pl.io.pastebin.app.deamon**

2.6. Shared Preferences

- Wszystkie wymagane klasy odpowiadające za nazwę albo klucz do korzystania z Shared Preferences: **pl.io.pastebin.app.preferences**

2.7. Obsługa notatek

- Obsługa notatek, oraz współpraca widoku z modelem do zarządzania notatkami: **pl.io.pastebin.app.model**

3. Opis poszczególnych pakietów i klas

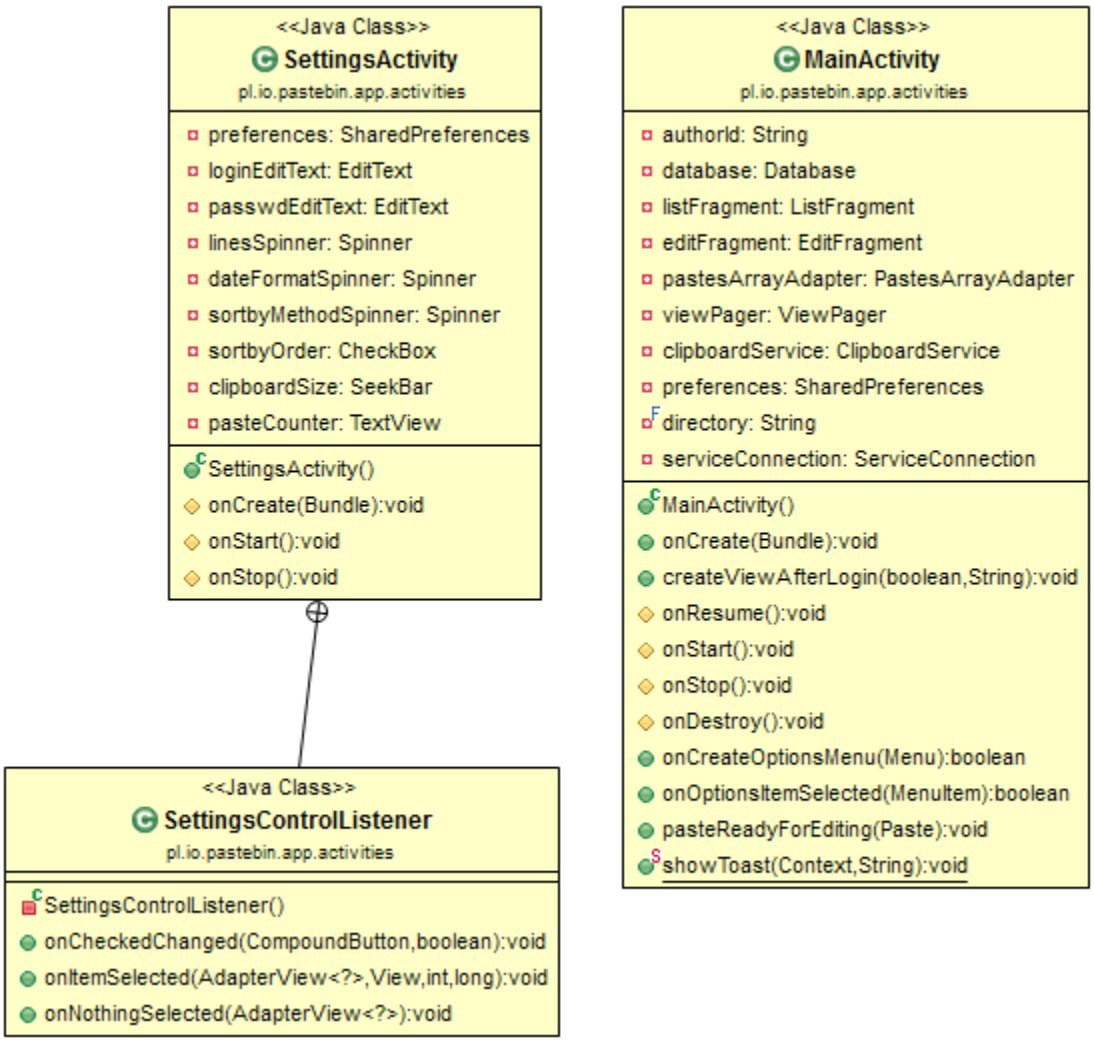
3.1. pl.io.pastebin.app.activities

3.1.1 MainActivity

Główna klasa zawierająca listę notatek lub edytor (w zależności od działania użytkownika), umożliwiająca logowanie się do serwisu Azure, ustawiająca tryb działania aplikacji

3.1.2. SettingsActivity

Klasa zawierająca wszystkie dodatkowe ustawienia dla programu: logowanie do serwisu Pastebin.com, liczbę wyświetlanych linii notatek, sposób sortowania notatek, format wyświetlanej daty oraz rozmiar rozszerzonego schowka.



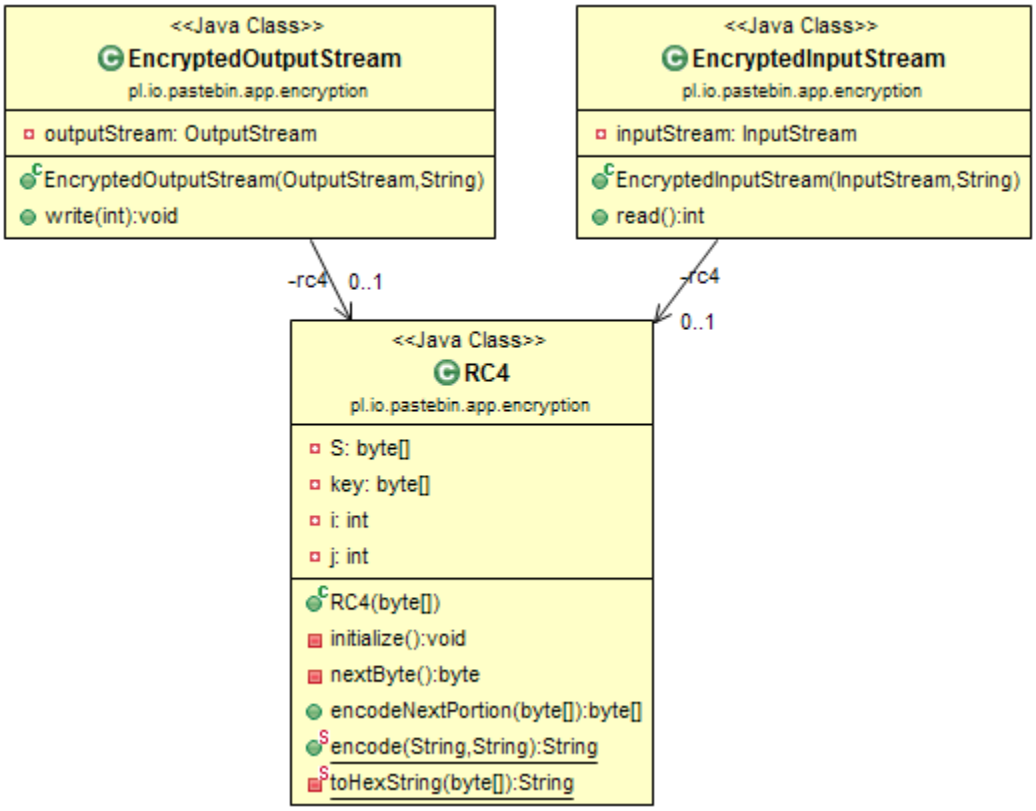
3.2. pl.io.pastebin.encryption

3.2.1. RC4

Klasa implementująca algorytm RC4 do szyfrowania zapisywanych treści

3.2.2. EncryptedOutputStream, EncryptedInputStream

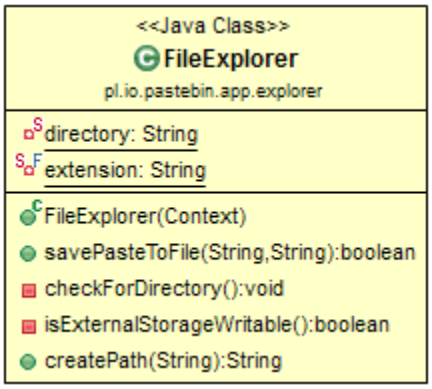
Odpowiednio strumienie do zapisywania i odczytywania zaszyfrowanych danych



3.3. pl.io.pastebin.explorer

3.3.1. FileExplorer

Klasa do zapisywania plików na karcie SD w formacie txt



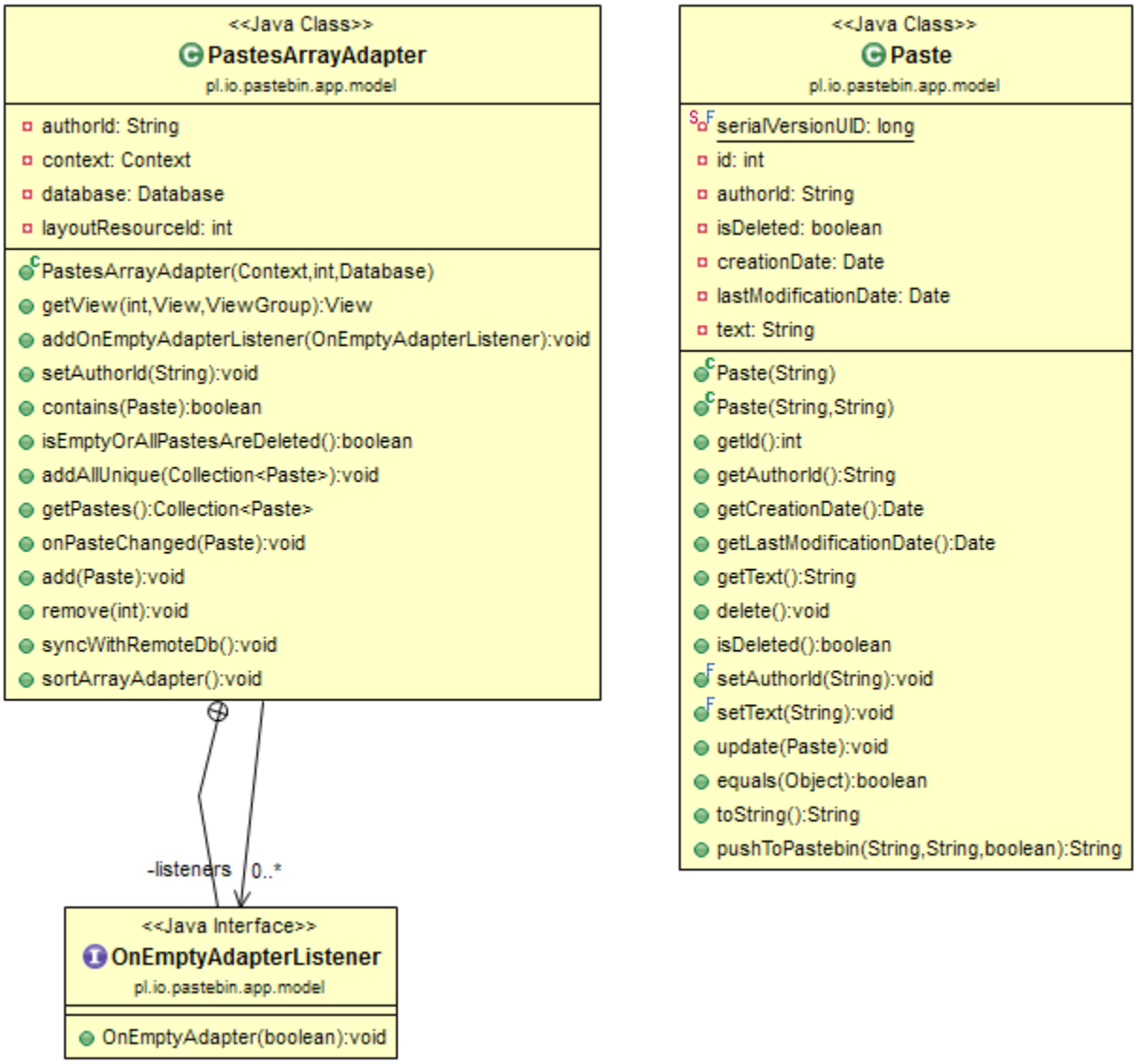
3.4. pl.io.pastebin.app.model

3.4.1. Paste

Klasa wykorzystywana do przechowywania notatek w bazie danych Azure oraz dająca możliwość przesłania notatki do serisu pastebin.com

3.4.2. PastesArrayAdapter

Klasa stanowiąca centralny punkt naszej aplikacji. Przechowuje kolekcję obiektów typu pl.io.pastebin.app.model.Paste. Dynamicznie, gdy zachodzi potrzeba odświeżenia listy notatek w ListFragment, tworzy i zwraca obiekt View dla każdej przechowywanej notatki. W zależności od preferencji użytkownika, zmienia format daty, ilość wyświetlanych linii czy sposób sortowania elementów na liście. Wszystkie operacje wykonywane na tej klasie (dodawanie, usuwanie notatek) są automatycznie wykonywane na zdalnej bazie danych.



3.4.3. PasteComparator

Klasa implementująca Comparator, wykorzystywana przy sortowaniu; daje możliwość wyboru kilku rodzajów sortowania (na podstawie treści notatki, daty stworzenia notatki, daty ostatniej modyfikacji notatki)

3.4.4. DateFormatter

Klasa pomocnicza służąca do formatowania daty. Zawiera zestaw predefiniowanych formatów, z których jeden jest wybrany przez użytkownika. Aplikacja używa metody *format(Date date)* z tej klasy, za każdym razem gdy zachodzi potrzeba wyświetlenia daty w postaci tekstowej. Dzięki temu wszystkie daty są wyświetlane przez aplikację w formacie wybranym przez użytkownika.

3.5. pl.io.pastebin.app.preferences

3.5.1. SharedPreferencesNames

Podstawowa klasa, zawiera nazwę dla obiektu SharedPreferences oraz nazwy dla wartości przechowywanych

3.5.2. ApplicationMode

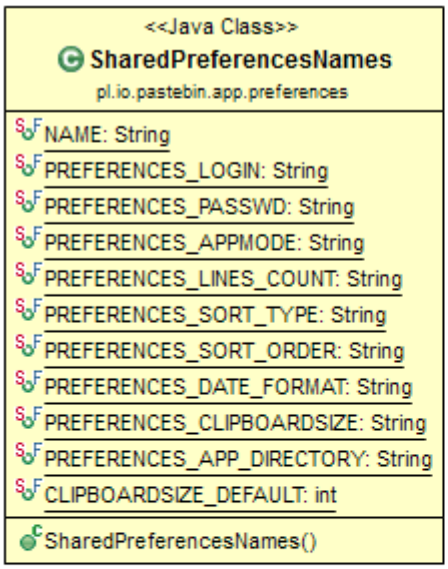
Mapuje tryb działania na aplikacji na podstawie jego nazwy na int

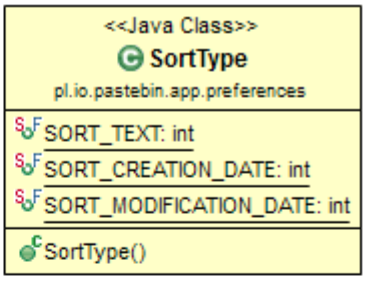
3.5.3. SortOrder

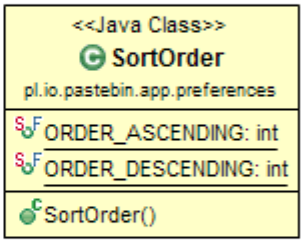
Mapuje kolejność sortowania notatek na podstawie jego nazwy na int

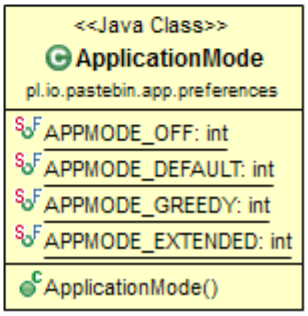
3.5.4. SortType

Mapuje rodzaj sortowania notatek na podstawie jego nazwy na int





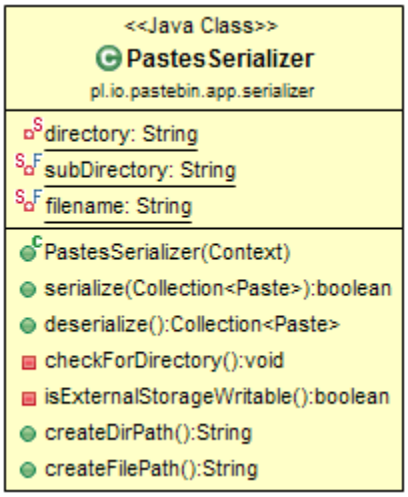




3.6. pl.io.pastebin.app.serializer

3.6.1. PastesSerializer

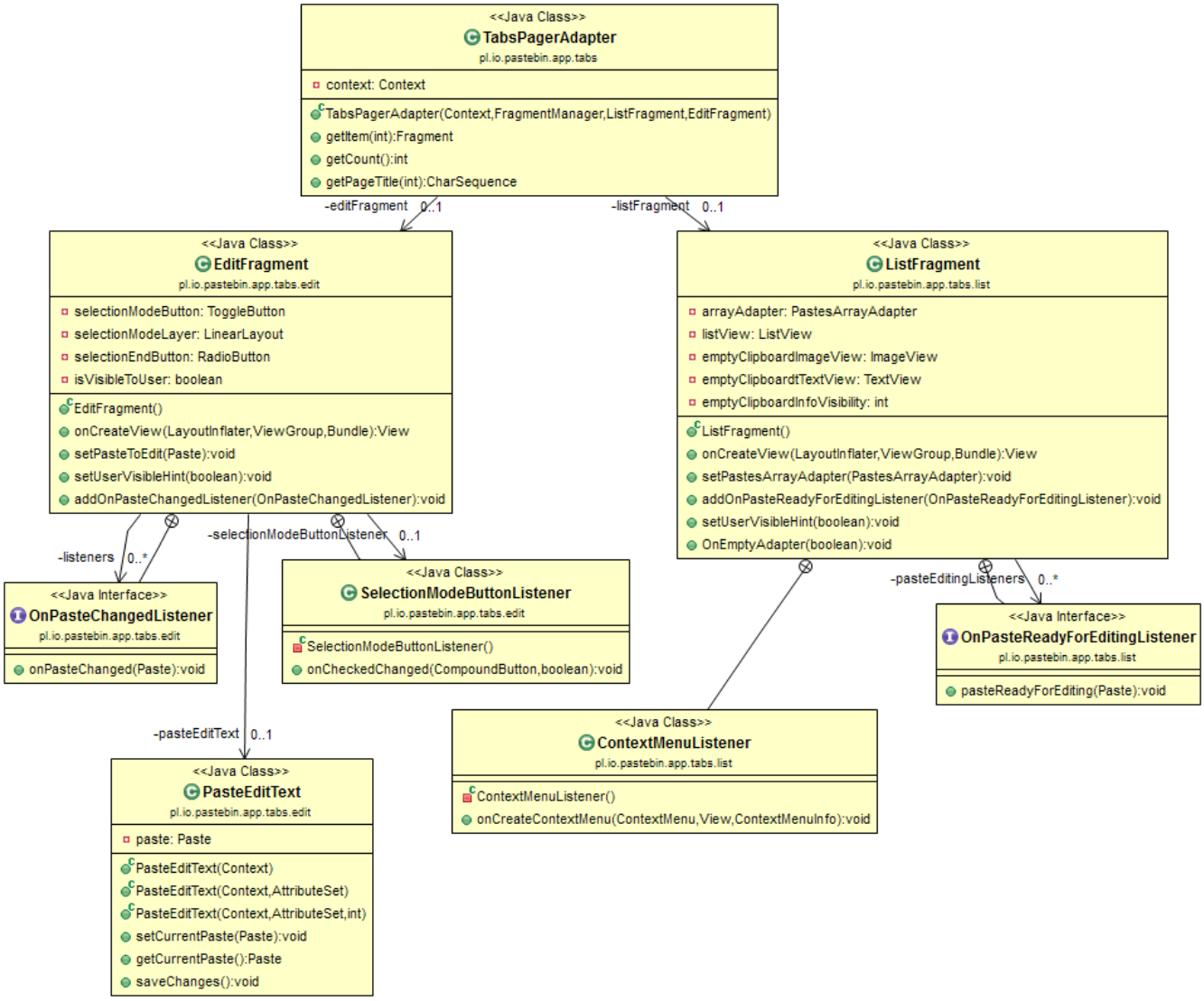
Klasa zajmująca się serializacją i deserializacją notatek na karcie SD



3.6. pl.io.pastebin.app.tabs

3.6.1. TabsPagerAdapter

Interfejs graficzny naszej aplikacji jest podzielony na dwa ekrany (Lista notatek i Edycja). Ta klasa zwraca ekran, króry w danym momencie powinien być wyświetlony oraz jego tytuł.



3.7. pl.io.pastebin.app.tabs.edit

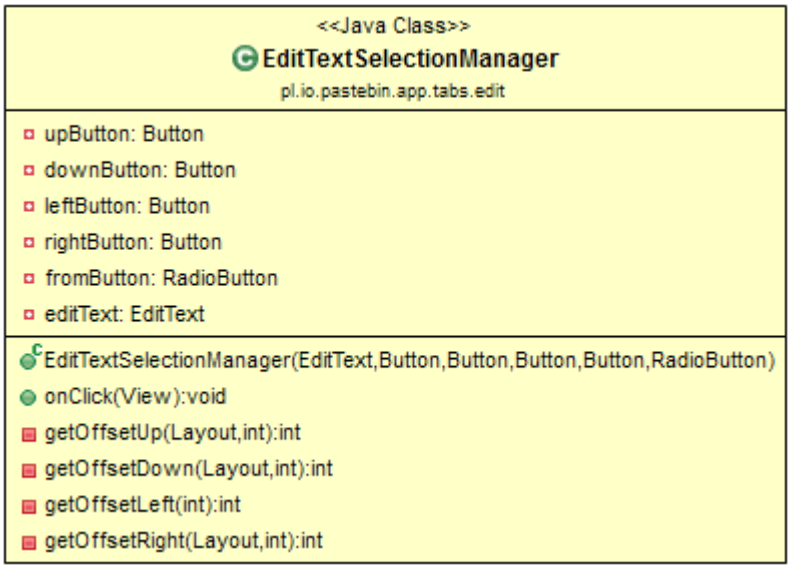
3.7.1. EditFragment

Klasa zarządzająca widokiem edycji, zawiera pole tekstowe. Po skończonej edycji, notyfikuje klasę ListFragment o zmianie stanu edytowanej notatki.

3.7.2. PasteEditText

Klasa dziedzicząca po klasie EditText. Jest to pole tekstowe w którym użytkownik edytuje notatkę. W późniejszej wersji aplikacji umożliwi dodatkowe metody zaznaczania tekstu.

3.7.3. EditSelectionManager



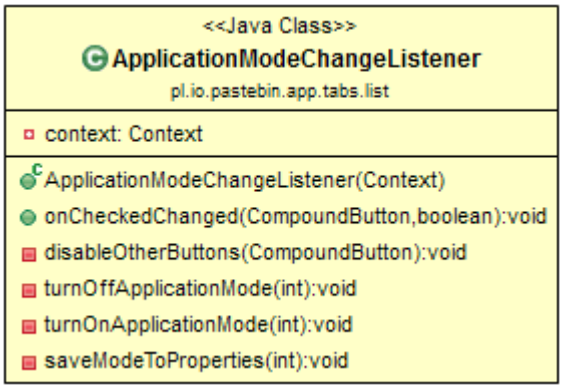
3.8. pl.io.pastebin.app.tabs.list

3.8.1. ListFragment

Klasa zarządzająca widokiem listy notatek. Dodaje menu kontekstowe umożliwiające edycje, usunięcie lub przesłanie zaznaczonej notatki do serisu pastebin.com. Dodatkowo zawiera przyciski wyboru sposobu działania aplikacji.

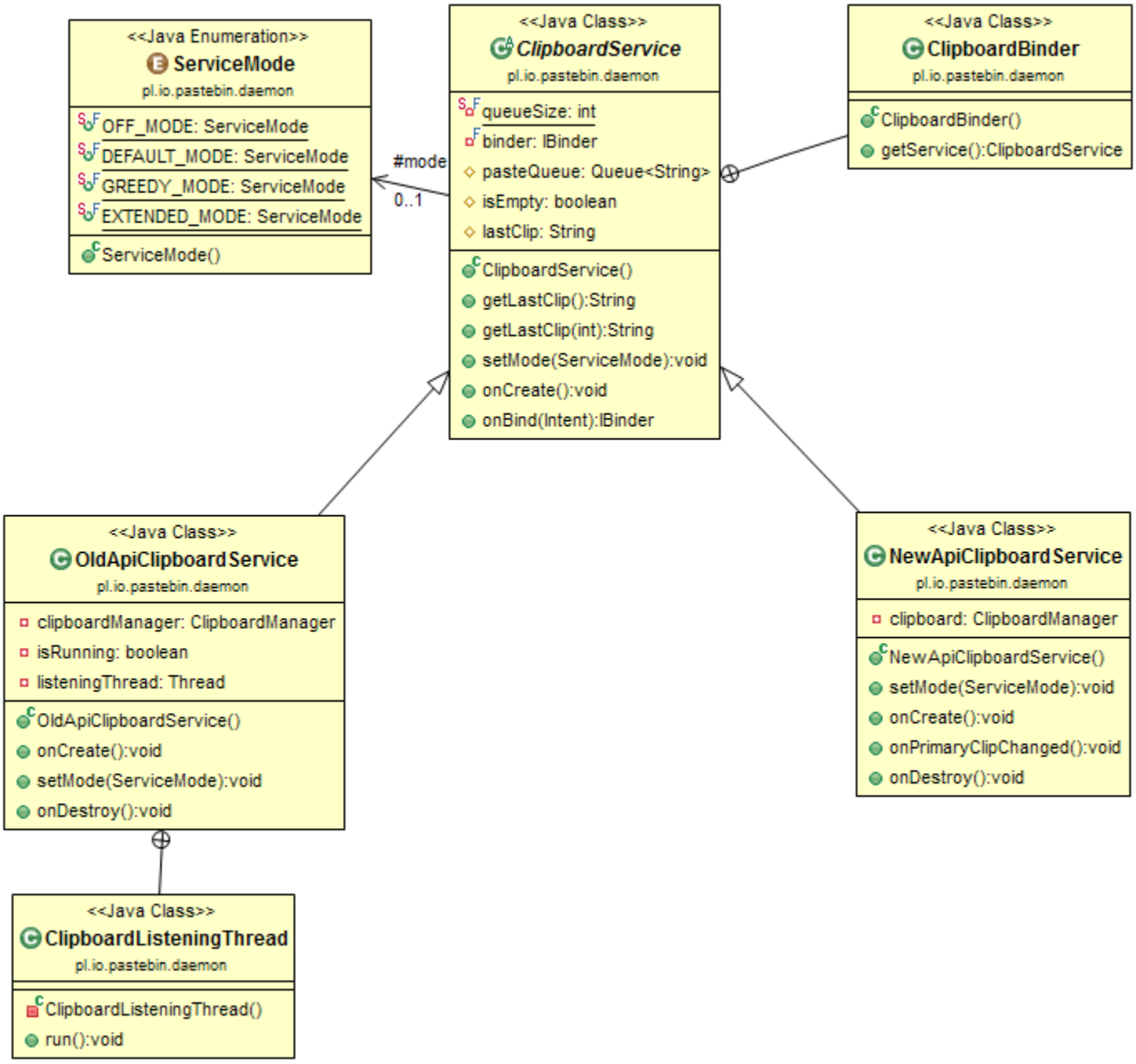
3.8.2. ApplicationModeListener

Listener do obsługi przycisków odpowiedzialnych za zmianę trybów działania aplikacji



3.9. pl.io.pastebin.daemon

Pakiet zawierający klasy odpowiadające za nasłuchiwanie schowka systemowego. Klasy te dziedziczą po klasie Service, dzięki czemu mogą działać w tle po zamknięciu aplikacji.



3.9.1. ClipboardService

Klasa bazowa dla wszystkich demonów, posiada implementację podstawowych funkcji, rozszerza klasę Service

3.9.2. NewApiClipboardService

implementacja demona nasłuchującego schowka, korzystająca z mechanizmów dostępnych w nowszych wersjach API systemu Android (wersje >= API 11).

3.9.3. OldApiClipboardService

klasa używana w przypadku gdy na urządzeniu zainstalowany jest system w wersji API mniejszej niż API 11. Z uwagi na fakt, iż w starszych wersjach systemu nie było możliwości dodania Listenera do schowka systemowego, demon uruchamia wątek **ClipboardListeningThread**,

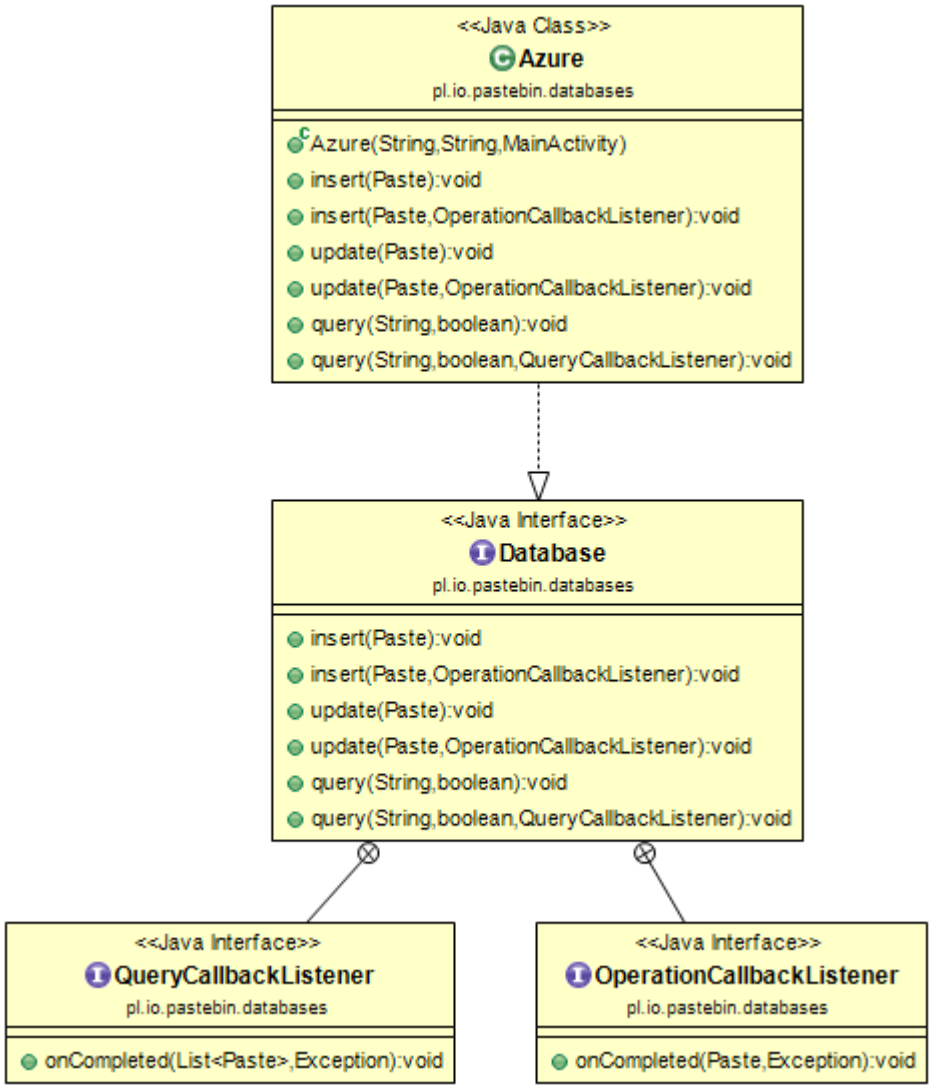
3.9.4. ClipboardListeningThread

Wątek który periodycznie sprawdza zawartość schowka systemowego i reaguje jeśli zauważy jakąś zmianę (dla API < 11)

Obie klasy, po stwierdzeniu, że w schowku znajduje się nowy element, uruchamiają aplikację poprzez wyświetlenie MainActivity, i przekazują do niej nowo zarejestrowany element.

3.10. pl.io.pastebin.databases

Pakiet zawierający klasy odpowiedzialne za persystencję notatek



3.10.1. Database

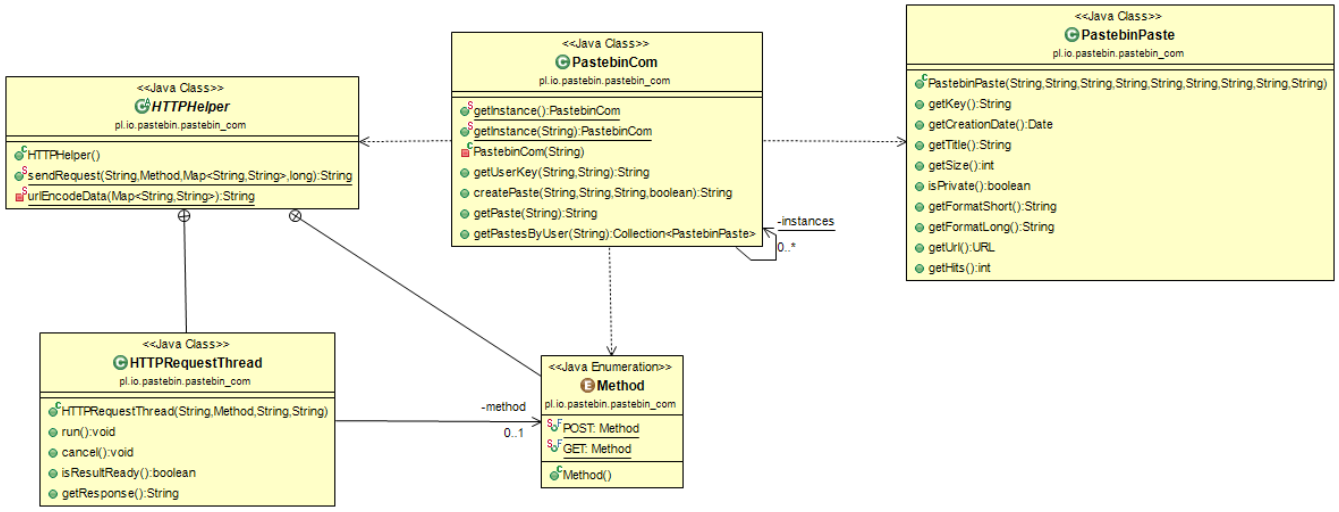
interfejs który jest wykorzystywany przez naszą aplikację do zapisu notatek w bazie danych. Możliwe jest zapewnienie wielu implementacji takiego interfejsu dzięki czemu nasza aplikacja nie jest zależna od konkretnej bazy danych

3.01.2. Azure

klasa implementująca interfejs *Database*, jest odpowiedzialna za połączenie z bazą danych platformy Windows Azure

3.11. pl.io.pastebin.pastebin_com

Pakiet zawierający klasy odpowiadające za komunikacje z serwerem serwisu Pastebin.com. Serwis ten udostępnia RESTful API dzięki któremu możemy wysyłać notatki na serwer i pobierać je.



3.11.1. HTTPHelper

Klasa pomocnicza umożliwiająca wysyłanie zapytań do serwera HTTP (metody GET i POST). Po wysłaniu rządania odpalany jest wątek HTTPRequestThread, który oczekuje na odpowiedź serwera

3.11.2. PastebinPaste

Klasa representująca pojedynczą notatkę z serwisu Pastebin.com. Zawiera wszystkie informacje które są dostępne po pobraniu notatki z serwisu.

3.11.3. PastebinCom

Klasa udostępniająca javowy interfejs do serwisu Pastebin.com. Do łączenia się z serwisem używa klasy HTTPHelper. Umożliwia:

- logowanie się do serwisu
- tworzenie nowej notatki (anonimowej i dla konkretnego użytkownika)
- listowanie notatek użytkownika
- pobieranie pojedynczych notatek

Informacje zwracane przez serwer Pastebin.com są zapisane w formacie XML. Do parsowania tych danych jest używany SAXParser.

4. Zarządzanie notatkami

(wszystkie klasy do zarządzania notatkami znajdują się w pakiecie: **pl.io.pastebin.app.model**)

Notatki są reprezentowane w postaci obiektu klasy **`Paste`** i zawierają 6 podstawowych pół zapisywanych w bazie danych:

- **`int id`** - klucz główny w bazie danych
- **`String auhtorId`** - reprezentuje identyfikator użytkownika, który stworzył notatkę
- **`boolean isDeleted`** - służy do markowania notatek, które zostały usunięte przez użytkownika
- **`Date creationDate`** - data stworzenia notatki
- **`Date lastModificationDate`** - data ostatniej modyfikacji
- **`String text`** - zawartość notatki

Bez problemu można dodawać dodatkowe pola do notatki pod warunkiem, że baza danych zostanie przygotowana do ich persystencji.

Do przechowywania notatek w trakcie pracy aplikacji służy **`PastesArrayAdapter`** rozszerzający klasę **`ArrayAdapter<Paste>,`** odpowiada również za wywoływanie synchroznizacji notatek z chmurą, sortowaniem ich we wskazanym przez użytkownika porządku oraz integrację z GUI. Notatki są usuwane i dodawane do widoku za pomocą przeciążonych metod z superklasy add i remove.

Sorotwanie notatek jest możliwe dzięki stworzeniu komparatora dla klasy Paste - **`PasteComparator.`** Uwzględnia on trzy tryby sortowania notatek:

- według zawartości
- według daty dodania notatki
- według daty ostatniej modyfikacji

Oraz umożliwia sortowanie w kolejności rosnącej oraz malejącej
Wczytywanie trybu oraz kolejności sortowania odbywa się za pomocą obiektu SharedPreferences opisanego dokładnie niżej.

Ustawianie formatu wyświetlanej daty jest realizowane za pomocą klasy **`DateFormatter`**.
Formaty są sortowań również są przekazywane za pomocą obiektu SharedPreferences.
Dodanie nowego formatu jest bezproblemowe, oprócz zdefiniowania jego obsługi w klasie **`DateFormatter`** należy również dodać go do menu w **`SettingsActivity`**

5. Współdzielone informacje - SharedPreferences

Ze względu na to, że wartości chcemy przechowywać niezmiennie między uruchomieniami aplikacji bądź przekazywać między różnymi elementami aplikacji zdecydowaliśmy się do tego celu użyć Androidowego mechanizmu SharedPreferences.
SharedPreferences to nic innego jak zbiór par (nazwa, wartość), który może być dzielony między róznymi fragmentami aplikacji, a nawet między różnymi aplikacjami.

Wszystkie nazwy które mogą zostać uzyte są zapisywane w klasie **`SharedPreferencesNames`**, wszystkie opcje które muszą zostać zmapowane na int posiadają oddzielne klasy odpowiadające za mapowanie na przykład (ze względu na niemożliwość zastosowania enumów w SharedPreferences):

- **`ApplicationMode`** `(APPMODE_OFF = 0, APPMODE_DEFAULT = 1, APPMODE_GREEDY = 2, APPMODE_EXTENDED = 3;=)`
- **`SortOrder`**`(ORDER_ASCENDING = 1, ORDER_DESCENDING = 2)`
- **`SortType (`**`SORT_TEXT = 1, SORT_CREATION_DATE = 2, SORT_MODIFICATION_DATE = 3)`

Atrybut **`public static final String NAME`** definiuje nazwę dla współdzielonych preferencji. Przykładowe otwieranie obiektu SharedPreferences:

```
context.getSharedPreferences(SharedPreferencesNames.NAME,
Activity.MODE_PRIVATE)
```

Przykładowy odczyt wartości na podstawie jej nazwy:

```
preferences.getInt(SharedPreferencesNames.PREFERENCES_APPMODE,
                           ApplicationMode.APPMODE_OFF)
```

Oraz przykładowy zapis wartości:

```
Editor prefsEditor = context.getSharedPreferences(
SharedPreferencesNames.NAME, Activity.MODE_PRIVATE).edit();

     prefsEditor.putInt(SharedPreferencesNames.PREFERENCES_APPMODE,
                        applicationMode);
prefsEditor.commit();
```

Dodanie każdej kolejnej pary (nazwa, wartość) wiążę się ze stworzeniem jaj nazwy w **`SharedPreferencesNames`** (a gdy potrzebne jest mapowanie albo jakiś inny typ podstawowy, dodanie klasy ze zdefiniowanymi wartościami do pakietu: **`pl.io.pastebin.app.preferences`**

6. Tryby aplikacji - demony

6.1. Tryby aplikacji

Aplikacja posiada 4 wbudowane tryby użytkownika wpływające na zachowanie się aplikacji oraz operacji na schowku. Przełączanie trybów jest realizowane za pomocą trzech ToggleButton'ów pojawiających się w głównym ekranie aplikacji:

- **Tryb Off** - wszystkie przyciski są wyłączone, aplikacja znajduje się w trybie pasywnym, nie reaguje w żaden sposób na zmiany schowka
- **Tryb Default** - podstawowy tryb działania aplikacji, wciśnięty przycisk "Default", gdy chowamy aplikację za pomocą przycisku Home uruchamiany jest demon nasłuchujący zmian w schowku systemowym. W momencie gdy wracamy do aplikacji automatycznie tworzona jest nowa notatka z zawartością schowka.
- **Tryb Greedy** - wciśnięty przycisk "Greedy", gdy chowamy aplikację za pomocą przycisku Home uruchamiany jest demon nasłuchujący zmian w schowku systemowym. Każda zmiana w schowku (czyli dodanie jakiegoś nowego elementu do niego) powoduje natychmiastowe wywołanie aplikacji na wierzch oraz automatyczne utworzenie notatki ze skopiowanym tekstem do schowka.
- **Tryb Extended** - wciśnięty przycisk "Extended", gdy chowamy aplikację za pomocą przycisku Home uruchamiany jest demon nasłuchujący zmian w schowku systemowym. Każda zmiana na schowku powoduje dodanie jego zawartości do rozszerzonego schowka, jego rozmiar jest określany w Settings

`ToggleButton`'y odpowiedzialne za zmianę trybów zostały objęte w **`RadioGroup`** tak aby naciśnięcie jednego przycisku powodowało wyłączenie innych co za tym idzie, aby zmiana trybów pyła płynna. Layout opisujący pozycję przycisków znajduje się w **res/layouts/fragment_list.xml.**

Cała obsługa buttonów do zmiany trybu znajduje się w pakiecie: **pl.io.pastebin.app.tabs.list** Inicjalizacja przycisków oraz listenerów znajduję się w klasie **`ListFragment`**, natomiast logika dotyczą zmian trybów jest zaimplementowana w klasie **`ApplicationModeChangeListener`**. **`ApplicationModeChangeListener`** implementuje **`CompoundButton.OnCheckedChangeListener`** i jest dodawana jako listener do każdego **`ToggleButton`**'a reprezentującego tryb oraz do **`RadioGroup`** i odpowiada również za wyłączania pozostałych przycisków.

Przesyłanie stanu pomiędzy **`ApplicationModeChangeListener`**, a **`MainActivity`**, które zajmuje się obsługą demonów oraz zachowaniem aplikacji w zależności od trybu jest realizowane przy pomocy obiektu **`SharedPreferences`** (opisane dokładnie wyżej). Do obsługi trybów wykorzystywana jest para:
(**`SharedPreferencesNames.PREFERENCES_APPMODE. ApplicationMode`**)

Do każdego trybu jest przyporządkowany integer, który jest zapisywany w preferencjach - **ApplicationMode**).

W **MainActivity** w metodzie **onResume** odczytywany jest aktualny tryb aplikacji ustawiony przez przyciski i na podstawie jego wartości jest podejmowana odpowiednia akcja, w zależności od trybu.

Rozszerzenie aplikacji o dodatkowe tryby jest bardzo proste, wystarczy dodać kolejnego **`ToggleButton`**'a do **`RadioGroup`** w **`ListFrament`**, dodać do **`AppliactionMode`** identyfikator do kolejnego trybu oraz obsługę trybu w **`onResume`** w **`MainActivity`**.

6.2. Obsługa demonów

(wszystkie klasy zajmujące się obsługą demonów znajdują się w pakiecie **pl.io.pastebin.daemon**)

Nasłuchiwanie zmian na schowku w momencie gdy aplikacja jest zminimalizowana i pracuje w którymś z trzech trybów (Default, Greedy lub Extended) wymagało stworzenia demonów, które będą pracować w tle i w zależności od wybranego trybu oferować określoną funkcjonalność.

Demon jest uruchamiany na samym początku aplikacji w metodzie **`onCreate (MainActivity),`** następnie w **`onResume`** jest włączany tryb Off, czyli demon przestaje nasłuchiwać na jakiekolwiek zmiany w schowku w trakcie gdy aplikacji jest na wierzchu, ostatecznie w momencie minimalizacji aplikacji czyli w metodzie **`onStop`** ustawiane jest jego zachowanie w zależności od wybranego trybu aplikacji.

Każdy demon rozszerza klasę abstrakcyjną **`PastebinService`**, która niesie ze sobą podstawową funkcjonalność oraz zajmuje się obsługą serwisu i bindowaniem tak aby możliwe było pobieranie danych z serwisu, dodatkowo ułatwia dynamiczną wymianę demonów dzięki zastosowaniu wzorca Strategy.

Ponieważ aplikacja wspiera wersję androida od 2.2 (API 8), dlatego zostały przygotowane 2 dedykowane demony do osbługi schowka:

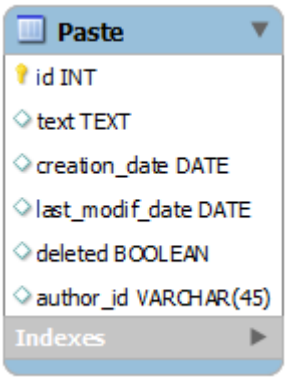
- **`OldApiClipboardService`** (dla API < 11) - zrealizowane w bardzo prosty sposób, odpalany jest dodatkowy wątek **`ClipboardListeningThread`** (wewnętrzna klasa w OldApiClipboardService), który co sekundę dokonuje sprawdzenia zawartości schowka, jeśli zawartość się zmieni to podejmowana jest odpowienia akcja w zależności od ustawionego trybu
- **`NewApiClipboardService`** (dla API >= 11) - zrealizowane za pomocą implementacji interfejsu **`OnPrimaryClipChangedListener`**, dzięki czemu jeśli serwis ustawi na sobie listenera przy każdej zmianie schowka jest automatycznie o tym informowany za pomocą metody **`public void onPrimaryClipChanged(),`** która na podstawie trybu podejmuje określoną akcję

Stworzenie dodatkowego demona jest bardzo proste i wiążę się tylko i wyłącznie z rozszerzeniem klasy **`PastebinService`** oraz w dodaniem dodatkowego warunku na wybór demona w metodzie **`onCreate`** w **`MainActivity`**.

7. Baza danych i serializacja

Jako bazy danych używamy cloudowego serwisu Microsoft Azure do którego podpięta jest prosta baza danych składająca się z jednej tabeli do przechowywania notatek użytkowników. Dodatkowo dostęp do poszczególnych kolumn jest realizowany przy pomocy autentykacji przez Google, co za tym idzie, użytkownik, który się poprawnie nie zaloguje nie jest w stanie otrzymać dostępu do tabeli z danymi. Google API automatycznie generuje dla każdego uzytkownika trwałe i niepowtarzalne identyfikatory (w postaci Stringa) dzięki czemu jesteśmy w stanie rozróżniać właścicieli poszczególnych notatek. Dane pobierane z tabeli są mapowanę na obiekt klasy **pl.io.pastebin.app.model.Paste** przy pomocy mechanizmu gson.

7.1. Schemat bazy danych



- id - klucz główny, identyfikator notatek
- text - zawiera treść notatki
- creation_date - data wraz z godziną stworzenia notatki
- last_modif_date - data wraz z godziną ostatniej modyfikacji notatki
- author_id - indywidualny identtyfikator użytkownika generowany na podstawie usługi autoryzującej

7.2. Interfejs Database

Baza danych jest wymienialna, w każdym momencie można zrezygnować z dotychczasowego rozwiązania chmurowego oferowanego przez MS Azure.

W celu podstawienia innej bazy danych wystarczy zaimplementować interfejs Database (**pl.io.pastebin.databases**) oraz dwa zagnieżdżone w nim interfejsy służące do powiadomień zwrotnych:

- **`public interface OperationCallbackListener`** - zawiera metodę onCompleted(Paste paste, Exception exception), pozwala zweryfikować czy operacja dodawania bądź uaktualniania rekordy w bazie danych przebiegła pomyślnie i nie wystąpił żaden błąd (na podstawie obiektu Exception) oraz otrzymać wstawiony obiekt (obiekt Paste)
- **`public interface QueryCallbackListener`** - zawiera metodę onCompleted(List<Paste> pastes, Exception exception), pozwala zweryfikować czy pobieranie danych z bazy danych przebiegło pomyślnie i nie wystąpił żaden błąd (na podstawie obiektu Exception) oraz otrzymać listę pobranych obiektów z bazy (za pomocą obiektu List<Paste>)

- Wstawianie notatki do bazy realizowane za pomocą dwóch metod, druga zawiera obiekt OperationCallbackListener, który odpowiada za zwrotną informację od bazy danych o poprawności wykonania operacji
  - **`public void insert(Paste paste)`**
  - **`public void insert(Paste paste, OperationCallbackListener listener)`**

- Uaktualnianie istniejącej już notatki jest realizowane w analogiczny sposób jak wstawianie, dwie funkcje, przyjmują obiekt Paste, który ma być uaktulaniony oraz druga metoda jest rozszerzona o obiekt OperationCallbackListener
  - **`public void update(Paste paste)`**
  - **`public void update(Paste paste, OperationCallbackListener listener)`**

- Pobieranie notatek z bazy danych realizowane za pomocą dwóch metod, przyjmujących identyfikator użytkownika, który jest otrzymywany po autentykacji (String authorID), informację o tym czy zapytanie ma zwrócić również notatki usunięte przez użytkownika (boolean includeDeleted) oraz analogicznie druga metoda posiada obiekt QueryCallbackListener do weryfikacja poprawności wykonania zapytania oraz zwrócenia listy notatek z bazy danych
  - **`public void query(String authorID, boolean includeDeleted)`**
  - **`public void query(String authorID, boolean includeDeleted, QueryCallbackListener listener)`**

7.3. Serwis chmurowy Azure

W tym momencie jako baza danych wykorzystywany jest serwis chmurowy Azure oferowany przez Microsoft, cała funkcjonalność obdługi chmury jest zrealizowane w klasie **pl.io.pastebin.databases.Azure.**

Połączenie z bazą danych Azure odbywa się za pomocą obiektu **`MobileServiceClient`**:

```
mServiceClient = new MobileServiceClient(https://io-pastebin.azure-mobile.net/",
                        "dIdzUPEndVeHwScernaQRnInFLDCUI16", context);
```

Kolejne elementy konstruktora MobileService do: adres URL aplikacji, klucz aplikacji oraz kontekst aplikacji (w naszym przypadku jest to MainActivity).

Logowanie i autentykacja jest również zrealizowana z poziomu serwisu Azure za pomocą obiektu **`MobileServiceClient`**:

```
mServiceClient.login(MobileServiceAuthenticationProvider.Google,
                            new UserAuthenticationCallback() { … });
```

Azure udostępnia nam kilka możliwych sposobów autentykacji dzięki obiektowi **`MobileServiceAuthenticationProvider`**, można zrezygnować z Google na rzecz Facebook'a, Twittera lub Microsoft Store. Dodatkowo listener **`UserAuthenticationCallback`** `UserAuthenticationCallback` pozwala zweryfikować poprawność logowania oraz otrzymać wygenerowany przez usługę logują unikalnego identyfikatora użytkownika, który jest wykorzystywany w bazie danych do rozróżniania notatek różnych użytkowników.

Następnie po stworzeniu klienta oraz zalogowaniu się do Azure pobierany jest obiekt do zarządzania rekordami w tabeli Paste **`(MobileServiceTable<Paste>)`**:

```
mServiceTable = mServiceClient.getTable(Paste.class);
```

Od tej pory wszystkie operacja na tabeli Paste (czyli insert, query oraz update) są wykonywane za pomocą obiektu mServiceTable.

ORM jest automatycznie obłsługiwany przez Azure, nie wymaga to żadnej większej ingerencji ze strony programisty poza ustawieniem adnotacji `(`**`@com.google.gson.annotations.SerializedName("columnName")`**`)` przy atrybutach klasy Paste, które mają być zapisywane do bazy danych, gdzie columnName to nazwa kolumny w bazie danych.

7.4. Lokalna serializacja notatek

Aby zapewnić persystencję danych w przypadku utraty połączenia z siecią, nasza aplikacja wykorzystuje serializację danych w pliku. Za serializację odpowiada klasa `PastesSerializer` która zawiera dwie publiczne metody:

```
public boolean serialize(Collection<Paste> pastes)
public Collection<Paste> deserialize()
```

Metoda `serialize()` przyjmuje jako argument kolekcję notatek i korzystając z faktu że klasa Paste implementuje interfejs `Serializable`, zapisuje ją do pliku przez `ObjectOutputStream`. Domyślnie jest to plik: `<sdcard>/IOPastes/.appdata/pastes.dat`

Deserializacja przebiega analogicznie. Wykorzystywany jest `ObjectInputStream` i jeśli odczytany obiekt może być zrzutowany na kolekcję to taka odczytana kolekcja jest zwracana. Jeśli wystąpi jakikolwiek błąd, zwracana jest pusta lista.

Cały proces zapisu i odczytu notatek jest wykonywany w `MainActivity`, w metodach `onStart()` i `onStop()`.

W `onStop()` wyciągane są wszystkie notatki z `PastesArrayAdaptera` i następnie są serializowane `PastesSerializerem`:

```
PastesSerializer serializer = new PastesSerializer(this);
if (pastesArrayAdapter != null)
      serializer.serialize(pastesArrayAdapter.getPastes());
```

W `onStart()` zdeserializowane notatki są umieszczane w `PastesArrayAdapterze` poprzez metodę `addAllUnique()` która dodaje notatki których jeszcze nie ma w adapterze.

```
if (pastesArrayAdapter != null)
      pastesArrayAdapter.addAllUnique(
            new PastesSerializer(this).deserialize()
      );
```

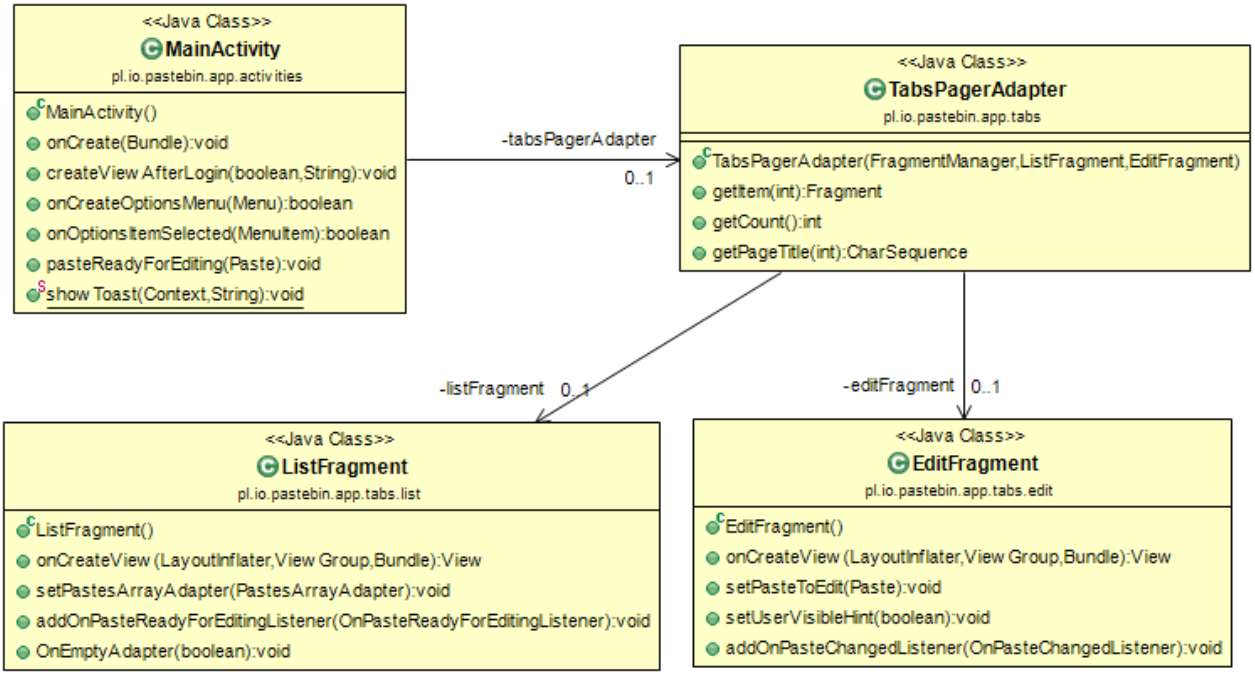
Dzięki takiemu podejściu za każdym razem gdy nasza aplikacja znika z ekranu, cała zawartość listy notatek zostaje zapisana w pliku i jest wczytywana gdy aplikacja ponownie pojawia się na ekranie. Jest więc możliwość korzystania z aplikacji gdy urządzenie nie ma połączenia z siecią. Następnie po poprawnym zalogowaniu, notatki są synchronizowane z zdalną bazą danych.

8. Interfejs użytkownika

Interfejs graficzny w naszej aplikacji składa się z dwóch ekranów:

- **ekranu listy notatek** - tutaj użytkownik może przeglądać wszystkie swoje notatki, może sortować listę według własnego uznania, może wykonywać różne czynności dotyczące notatek: usunięcie, otwarcie do edycji, zapisanie do pliku, wysłanie na Pastebin.com
- **ekranu edycji** - po otwarciu notatki do edycji użytkownik zostaje przeniesiony na ten ekran w którym treść notatki umieszczona jest w polu tekstowym. Po wyedytowaniu notatki wystarczy wyonać gest powrotu do poprzedniego ekranu (przeciągnąc palcem poziomo od lewej strony do prawej) aby zapisać zmiany w notatce i powrócić do ekranu listy.

Diagram klas:



W systemie Android graficzny interfejs składa się z Activities, czyli pełnoekranowych widoków zawierających kontrolki udostępnione użytkownikowi.

W naszej aplikacji całość interfejsu użytkownika zawarliśmy w MainActivity, którego jedynym elementem jest `ViewPager`. W `MainActivity.onCreate()` `ViewPager` ma ustawiany `FragmentPagerAdapter` który zarządza wyświetlaniem poszczególnych Fragmentów, które reprezentują poszczególne strony, w naszym przypadku `ListFragment` oraz `EditFragment`.

8.1. ListFragment

Zawiera `ListView` w którym przechowywane są widoki dla poszczegolnych notatek oraz trzy przyciski `TooggleButton` które służą do przełączania trybu pracy demona.

W `ListView` widoki dla poszczególnych elementów tworzone są dynamiczine przy odświeżaniu layoutu. Za ich tworzenie jest odpowiedzialny `ArrayAdapter` który przechowuje liste obiektów będących modelami dla elementów listy oraz zawiera metodę `getView()` która odpowiada za tworzenie pojedynczych elementów. Nasza implementacja `ArrayAdaptera` czyli `PastesArrayAdapter` została przedstawiona w innym miejscu niniejszej dokumentacji.

W celu stworzenia menu kontekstowego dla listy notatek `ListView` ma ustawiany `ContextMenuListener`, który jest naszą implementacją interfejsu `OnCreateContextMenuListener`.

```
listView.setOnCreateContextMenuListener(new ContextMenuListener());
```

`ContextMenuListener` jest notyfikowany kiedy użytkownik chce wyświetlić menu kontekstowe. Na podstawie wybranej opcji oraz elementu listy który został naciśnięty zostaje wykonana odpowiednia akcja.

`ListFragment` zawiera także listę obiektów implementujących `OnPasteReadyForEditingListener`, które są notyfikowane kiedy użytkownik wybierze z menu kontekstowego opcję “edit”. Jednym z listenerów jest `MainActivity`, którego implementacja metody `pasteReadyForEditing()` wygląda następująco:

```
public void pasteReadyForEditing(Paste paste) {
      editFragment.setPasteToEdit(paste);
      viewPager.setCurrentItem(1);
}
```

Kiedy użytkownik wybierze opcję “edit”, `MainActivity` przełącza aktualnie wyświetlany ekran na `EditFragment` oraz przekazuje do odbiektu `editFragment` obiekt `paste` który został wybrany do edycji.

8.2. EditFragment

Zawiera pole tekstowe `PasteEditText` w którym użytkownik może edytować notatkę oraz listę `OnPasteChangedListenerów EditFragment` przeciąża metodę `setUserVisibleHint(isVisible)` która jest wywoływana gdy Fragment pojawia się bądź znika z ekranu. Jeśli `EditFragment` znika z ekranu oraz w polu tekstowym znajduje się jakaś notatka to zmiany dokonane w zawartości notatki są utrwalane oraz `OnPasteChangedListenerzy` są notyfikowani.

PasteEditText jest klasą dziedziczącą po standardowej `EditText`, która dodatkowo przechowuje edytowaną notatkę. Między innymi zawiera metody:

```
public void setCurrentPaste(Paste paste) {
      this.paste = paste;
      setText(paste.getText());
}

public void saveChanges() {
      if (paste != null)
            paste.setText(getText().toString());
}
```

`setCurrentPaste()` ustawia aktualnie edytowaną notatkę i wypełnia pole tekstowe zawartością tej notatki

`saveChanges()` podmienia zawartość notatki na tekst który znajduje się w danym momencie w polu tekstowym

8.3. SettingsActivity

Zawiera ekran do edycji ustawień aplikacji. Umożliwia zmianę takich parametrów jak:

- dane logowania do Pastebin.com
- kolejność sortowania notatek na liście
- maksymalną ilość linijek wyświetlanych dla pojedynczej notatki
- format daty
- rozmiar “rozszerzonego schowka”

`SettingsActivity` korzysta z mechanizmu `SharedPreferences`, który został już opisany w ninejszej dokumentacji. Zmiana poszczególnych ustawień powoduje przypisanie wartości do `SharedPreference` o konkretnej nazwie, przykładowo ustawianie formatu daty:

```
public void onItemSelected(AdapterView<?> parent, View view,
            int position, long viewID) {
      Editor editor = preferences.edit();
      switch (parent.getId()) {
            (...)
            case R.id.settings_date_format_spinner:
                  editor.putInt(
                        SharedPreferencesNames.PREFERENCES_DATE_FORMAT,
                        position
                  );
                  break;
            (...)
      }
      editor.commit();
}
```