Université de Namur

[INFOM218] Evolution de systèmes logiciels

Database reverse engineering and assessment

Pretend You're Xyzzy

Janvier 2024

Contents

1 Étape 1

Lien vers le Répo GITHUB

1.1 Description et analyse du schéma physique

Lors de notre analyse avec l'outil **sqlinspect**, nous avons découvert que cette application faisait appel à deux bases de données. La première porte sur l'aspect du jeu, et la deuxième sur le système de chat disponible.

Nous avons choisi de nous concentrer sur la base de données liée au jeu plutôt que sur celle du chat. Ce choix ne s'est pas basé sur le nombre de tables, car les deux bases en possédaient autant, mais sur le fait que la base du jeu comportait un plus grand nombre de colonnes. Cela offrait un potentiel plus important pour découvrir des clés étrangères et d'autres *code smells* spécifiques au domaine de l'ingénierie des bases de données, ce qui constitue l'objectif principal de ce rapport.

Une autre raison ayant guidé notre choix tient dans le fait que l'application se vendant sur son aspect ludique, nous avons voulu nous intéresser aux mécanismes de bases de données utilisés dans ce cadre.

Le schéma physique obtenu pour cette base de données nous apprend donc l'existence de 7 tables.

D'une part, nous avons quatre tables : `Decks`, `Cards`, `CR_cast_decks` et `Starred_decks`. Toutes, à l'exception de `CR_cast_decks`, possèdent un identifiant défini comme étant un entier unique et portant le même nom *id*. Des similitudes sont observables, comme les colonnes *watermark*, *name* et *lastUsed* qui sont présentes dans les trois tables possédant `decks` dans leur nom. D'autres aspects attirent notre attention sur les dynamiques entre ces tables. Nous examinerons ces éléments en détail tout au long de ce rapport. Cela inclut la présence de la colonne *remoteid* dans trois tables, les variations dans l'utilisation des tables de **decks**, l'identification potentielle de clés étrangères comme *deck_id* dans Cards, et la signification de noms de colonnes et de tables ambigus. Nous nous renommeront ces éléments dans notre schéma logique. L'objectif est que les requêtes SQL et le fonctionnement interne de l'application permette de faire plus de sens.

D'autre part, le schéma comporte deux tables. Ce sont `Cr_cast _decks_temp` et `Starred_decks_temp` qui semblent correspondre chacune à une aux deux tables au nom similaire sans le temp présentées précédemment. La différence étant la colonne favorite qui manque à `Cr_cast _decks_temp`, mais qui est présente dans sa table correspondante.

La dernière table, que nous avons appelée `Favorite_Cards`, est associée dans notre schéma physique à une seconde base de données. Comme expliqué plus en détail dans la section 1.2.8 **Cards_V2**, nous incluons cette unique table d'une autre

base de données dans notre reconstruction. La raison est qu'elle est utilisée pour les mêmes données métier de l'application. De plus, la création d'une seconde base de données pour cela découle selon nous d'une mauvaise pratique de design. Cette table possède elle aussi un identifiant défini comme étant un entier unique sous le nom *id*. Elle possède en plus de cela les colonnes *remoteId*, *blackCard* et *whiteCard*. Ces trois colonnes sont communes à la table `Card` de notre première base de données, ce qui nous mènera à tenter de les fusionner dans notre analyse **"What IF?"**.

1.2 Analyse des résultats de requêtes

Dans cette section, nous explorerons les éléments permettant d'enrichir le schéma décrit précédemment et d'obtenir notre **schéma physique enrichi** et notre **schéma logique**. Pour ce faire, nous nous sommes appuyés sur les requêtes SQL trouvées dans le code de l'application, sur des intuitions découlant de notre schéma logique, ainsi que sur nos connaissances du domaine des bases de données et des bonnes pratiques de programmation qui y sont associées.

1.2.1 Les tables temporaires

Dans le schéma physique obtenu à l'aide de **sqlinspect**, nous avons remarqué que deux tables portent le mot `temp` dans leur nom. Le concept de tables temporaires nous intriguait, et nous avons trouvé l'endroit dans le code qui semble expliquer pourquoi elles sont utilisées. Comme le montre le code suivant, dans la méthode `onUpgrade` située dans le fichier **CustomDecksDatabase.java**, on peut voir que ces tables sont utilisées comme des variables.

```
1  @Override
2  public void onUpgrade(SQLiteDatabase db, int oldVersion, int
       newVersion) {
3      Log.d(TAG, "Upgrading from " + oldVersion + " to " +
       newVersion);
4      switch (oldVersion) {
5          case 9:
6              [...]
7          case 10:
8          case 11:
9              db.execSQL("CREATE TABLE cr_cast_decks_tmp (name TEXT NOT
       NULL, watermark TEXT NOT NULL UNIQUE, description TEXT NOT NULL
       , lang TEXT NOT NULL, private INTEGER NOT NULL, state INTEGER
       DEFAULT NULL, created INTEGER DEFAULT NULL, whites_count
       INTEGER NOT NULL, blacks_count INTEGER NOT NULL, lastUsed
       INTEGER NOT NULL)");
10             db.execSQL("INSERT INTO cr_cast_decks_tmp SELECT * FROM
    cr_cast_decks");
11             db.execSQL("DROP TABLE cr_cast_decks");
12             db.execSQL("ALTER TABLE cr_cast_decks_tmp RENAME TO
    cr_cast_decks ");
13             db.execSQL("ALTER TABLE cr_cast_decks ADD favorite INTEGER
        NOT NULL DEFAULT 0");
```

```
14          case 12:
15              db.execSQL("CREATE TABLE starred_decks_tmp (id INTEGER
    PRIMARY KEY UNIQUE, shareCode TEXT NOT NULL UNIQUE, name TEXT
    NOT NULL, watermark TEXT NOT NULL, owner TEXT NOT NULL,
    cards_count INTEGER NOT NULL, remoteId INTEGER UNIQUE, lastUsed
     INTEGER NOT NULL DEFAULT 0)");
16              db.execSQL("INSERT INTO starred_decks_tmp SELECT * FROM
starred_decks");
17              db.execSQL("DROP TABLE starred_decks");
18              db.execSQL("ALTER TABLE starred_decks_tmp RENAME TO
starred_decks");
19          case 13:
20              [...]
21          case 14:
22              [...]
23      }
24      Log.i(TAG, "Migrated database from " + oldVersion + " to " +
    newVersion);
25  }
```

Tout d'abord, on peut observer que ces deux tables sont créées uniquement dans les cas où l'on doit modifier les tables `Cr_cast_decks` et `Starred_decks`. Ensuite, la table nouvellement créée est remplie avec le contenu des colonnes de sa correspondante sans le `temp`, la correspondante se faisant supprimer après cette opération. Enfin, la table temporaire est modifiée pour reprendre le nom de celle dont elle vient de prendre la place qui elle est supprimée.

Dans le cas de la *version 11*, la nouvelle table se voit ajouter la colonne *favorite* qui est un concept certainement apparu et utilisé dans le code lors de cette mise à jour.

Il apparaît également que l'utilisation de ces tables a été mise en place au fur et à mesure de l'évolution de l'application. Ceci montre, selon nous, un manque de vision cohésive et cohérente des exigences fonctionnelles dès le début du développement de l'application et à travers les versions consécutives de celle-ci. Notre hypothèse est que de nouvelles fonctionnalités ont été ajoutées sans trop de conceptualisation. Ces fonctionnalités nécessitant la mise à jour de `Cr_cast_deck` puis de `Starred_decks`, nous pouvons ainsi les observer dans le *switch case* visible dans le code ci-dessus.

Nous avons aussi fait des recherches dans la **codebase java** et nous avons trouvé un autre indice étayant notre hypothèse. Le terme *overloaded* y apparaît de manière récurrente. Il définit la possibilité ou non de pouvoir accéder à certaines fonctionnalités.

On peut, par exemple observer, la séparation du code dans le fichier `NewUserInfo Dialog.java` qui est spécifiée par un commentaire précisant la zone de code concernée par cette notion ainsi que la présence de classes et méthodes spécifiquement dédiées à la gestion de ce concept.

```
//region Overloaded
if (Overloaded Utils.isSignedIn() && overloaded) {
    binding.userInfoDialogAddFriend.setVisibility(View.GONE);
    binding.userInfoDialog Overloaded Info.setVisibility(View.GONE);
    binding.userInfoDialog Overloaded InfoLoading.setVisibility(View.
   VISIBLE);
    binding.userInfoDialog Overloaded InfoLoading.showShimmer(true);
    [...]
    Overloaded Api.get().getProfile(username)
    [...]
```

Afin de nous assurer de la signification d'*overloaded*, nous avons entrepris d'émuler l'application et avons rapidement découvert que ***overloaded*** faisait référence à un mode **premium** payant. Ce mode permet, entre autres, d'utiliser la fonctionnalité de *chat* (d'où la présence de l'autre base de données que nous n'avons pas choisie pour ce rapport), de *créer des decks à plusieurs*, d'avoir un système d'*amis* dans le jeu et de pouvoir *mettre des decks en favoris*.

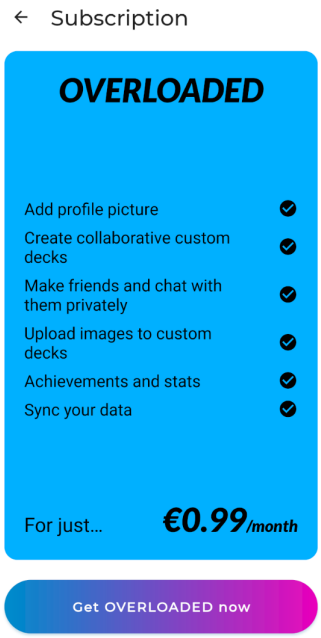


Figure 1: Overloaded

Nous avons donc décidé de supprimer ces tables de notre schéma conceptuel car elles n'apportent aucune information sémantique et ne font que la complexifier inutilement.

1.2.2 Réaffirmation de contraintes

Dans le code, nous avons également constaté que la contrainte d'unicité de la colonne *name* de la table `Decks` était vérifiée avant l'utilisation de sa valeur, bien que cette contrainte ait déjà été déclarée lors de la création de la table comme le

montre notre **schéma physique enrichi**.

Cette méthode *isNameUnique* est présente dans le fichier `CustomDecksDatabase.java` et permet de réaliser cette vérification.

```
public boolean isNameUnique (@NonNull String name) {
    SQLiteDatabase db = getReadableDatabase();
    db.beginTransaction();
    try (Cursor cursor = db.rawQuery("SELECT COUNT(*) FROM decks
    WHERE name=?", new String[]{name})) {
        if (cursor == null || !cursor.moveToNext()) return false;
        else return cursor.getInt(0) == 0;
    } finally {
        db.endTransaction();
    }
}
```

La raison de cette vérification pourrait découler du fait que la colonne *name* est effectivement définie comme *unique*, mais cela se limite à la table `Decks`. Dans les tables `Cr_cast_decks` et `Starred_decks`, cette colonne n'est pas marquée de la même manière. Dans la troisième partie de ce rapport, nous explorons les répercussions de déclarer cette colonne comme unique dans ces deux tables.

Après vérification du fichier **changelog.md** inclus dans le répertoire GitHub, nous avons finalement observé qu'une autre raison possible était une réparation rapide lors de la **version 5.0.6** à l'aide du *if* afin de permettre l'importation de plusieurs decks possédant le même nom.

```
## [5.0.6] - 11-01-2021
### Fixed
- Fixed crash at startup

### Added
- Allow importing deck with duplicate name after changing it
```

1.2.3 Les colonnes watermark

Les trois tables `decks` de notre base de donnée possèdent une colonne commune nommée *watermark*. Cette dernière a attiré notre attention car elle n'est déclarée *unique* que dans la table `Cr_cast_decks`.

Dans la méthode *contains* du fichier `BasicCustomDeck.java`, la notion de *watermark* est utilisée pour vérifier la présence d'un deck parmi les decks d'un utilisateur. La condition de vérification s'effectue ainsi : si le nom du deck trouvé correspond à celui du deck en cours d'examen et que la *watermark* du deck à trouver est soit nulle (s'il n'en possède pas), soit égale à celle du deck en cours, alors la méthode retourne vrai, indiquant que le deck recherché est présent dans la liste. Sinon, elle

retourne faux.

```
public static boolean contains (@NotNull List<BasicCustomDeck>
    decks, @NotNull Deck find) {
     for (BasicCustomDeck deck : decks)
         if (deck.name.equals(find.name) && (find.watermark == null
    || deck.watermark.equals(find.watermark)))
              return true;

     return false;
}
```

Il semble donc que le concept de *watermark* a un but d'identification. On peut aussi l'observer l'exemple suivant dans le fichier `CustomDecksDatabase.java` et ses méthodes *getCrCastDeckLastUsed* et *updateCrCastDeckLastUsed*. Il convient également de noter que le code se réfère également à ce concept en utilisant le terme *deckCode*.

```
@Nullable
public Long getCrCastDeckLastUsed (@NonNull String deckCode) {
     [...]
     try (Cursor cursor = db.rawQuery("SELECT lastUsed FROM
    cr_cast_decks WHERE watermark =?", new String[]{deckCode})) {
     [...]
}

public void upadateCrCastDeckLastUsed(@NonNull String deckCode, long
    lastUsed) {
     [...]
         ContentValues values = new ContentValues();
         values.put("lastUsed", lastUsed);
         db.update("cr_cast_decks", values, " watermark =?", new
    String[]{deckCode});
         db.setTransactionSuccessful();
     [...]
}
```

Afin de compléter l'analyse de cette colonne, nous avons également émulé l'application pour comprendre à quoi correspondait cette notion. Il apparaît ainsi que la *watermark* fait office de label permettant de classifier les cartes et les decks.

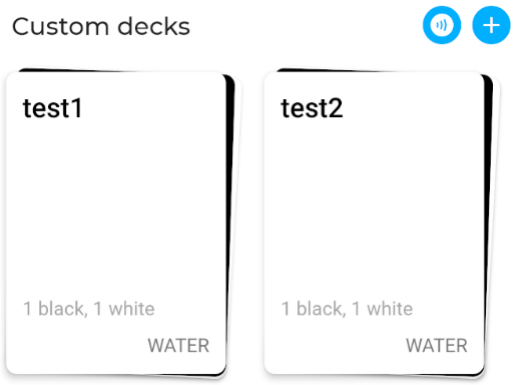


Figure 2: Watermark "Water" sur les cartes

1.2.4 Les colonnes blackCard et whiteCard

La présence de deux colonnes distinctes dans la table `Cards` a suscité notre interrogation quant à leur utilité. Nous avons découvert que la distinction entre les deux types de cartes (*blackCard* et *whiteCard*) est effectuée à l'aide d'un simple *booléen*.

La valeur du champ texte de ces colonnes est assignée en fonction du type de carte représenté par l'entrée.

```
1  public final class CrCastCard extends BaseCard {
2      [...]
3      private final boolean black;
4      [...]
5
6      CrCastCard(@NonNull String deckCode, boolean black, @NonNull
       JSONObject obj) throws JSONException {
7          [...]
8          this.text = reformatText(obj.getString("text"), black);
9          [...]
10     }
11
12     @NonNull
13     private static String reformatText(@NonNull String text,
       boolean black) {
14         if (!black) return  text;
15
16         if (!text.contains("_")) return text + "____";
17         else return text.replaceAll("_+", "____");
18     }
19
20 }
```

Cette distinction entre cartes noires et blanches sous forme de booléen est également observée dans d'autres classes telles que `ContentCard`, `GameRound`, `RoundCard` et `GameCards`, où un booléen *black* est utilisé pour identifier le type de carte. Plus

généralement, ces deux colonnes sont utilisées partout comme un booléen. Seulement, cette colonne est déclarée de type *text* dans la table `Cards`. Nous discuterons de cette différence de type dans la partie sur les scénarios **WhatIf**.

```
int type = blacks[i] ? CARD_TYPE_BLACK : CARD_TYPE_WHITE;
```

Dans la troisième partie, nous enquêterons sur les répercussions possibles de séparer la table **Cards** en deux pour différencier ces deux types de cartes et essayer ainsi d'apporter une solution plus élégante à ce problème.

1.2.5 Les colonnes remote_id

La colonne *remote_id* apparaît dans trois tables distinctes de la base de données : `Decks`, `Cards`, et `Starred_decks`. Nous avons examiné le code associé à ces tables pour comprendre si le *remote_id* agit en tant que clé étrangère référençant d'autres tables. Les investigations ont révélé plusieurs utilisations de cette colonne.

Dans un premier temps, pour retrouver le *remote_id*, l'identifiant *id* d'une entrée de la table `Cards` est utilisé. Ensuite, pour obtenir l'identifiant *id* d'une entrée de la table `Decks`, le *remote_id* est employé. Cependant, aucune définition explicite du *remote_id* n'est trouvée dans le code, et il semble être utilisé uniquement lors des opérations d'ajout et de retrait de cartes d'un deck.

```
@Nullable
private Long getCardRemoteId(int cardId) {
  SQLiteDatabase db = getReadableDatabase();
  db.beginTransaction();
  try (Cursor cursor = db.rawQuery("SELECT remoteId FROM cards WHERE
   id =?",
      new String[] { String.valueOf(cardId) })) {
    if (cursor == null || !cursor.moveToNext())
      return null;
    else
      return cursor.getLong(0);
  } finally {
    db.endTransaction();
  }
}

@Nullable
public Integer getDeckIdByRemoteId(long remoteId) {
  SQLiteDatabase db = getReadableDatabase();
  db.beginTransaction();
  try (Cursor cursor = db.rawQuery("SELECT id FROM decks WHERE
  remoteId=?",
      new String[] { String.valueOf(remoteId) })) {
    if (cursor == null || !cursor.moveToNext())
      return null;
    else
      return cursor.getInt(0);
```

```
26    } finally {
27      db.endTransaction();
28    }
29  }
```

Enfin, nos recherches nous ont menés jusqu'à l'un des rares commentaires présents dans ce code qui semblait répondre à un autre commentaire ayant été effacé.

```
1  // The remote ID can be that simple because it is used only when
       removing cards
```

Le *remote_id* n'est donc défini nulle part, et on peut observer qu'il n'apparaît que dans ces opérations d'ajout et de retrait de cartes d'un deck (tas de cartes).

Une autre hypothèse est que le *remote_id* pourrait être impliqué dans le maintien de la cohérence des données entre la base de données locale et les données du jeu via l'API, potentiellement pour des raisons de performance en évitant d'enregistrer toutes les données directement dans la base de données.

```
1  //region Sync logic
2  public static void syncStarredCards (@NonNull Context context,
       @Nullable OnCompleteCallback callback) {
3      [...]
4      StarredCardsDatabase db = StarredCardsDatabase.get(context);
5      [...]
6      @Override
7      public void onResult(@NonNull StarredCardsUpdateResponse
       result) {
8          if (result.remoteIds == null) {
9              Log.e(TAG, "Received invalid response when syncing starred
       cards (no remoteIds).");
10         } else {
11             if (result.remoteIds.length == update.localIds.length) {
12                 for (int i = 0; i < update.localIds.length; i++)
13                     db.setRemoteId(update.localIds[i], result.
       remoteIds[i]);
14                 Log.i(TAG, "Updated starred cards on server,
       count: " + result.remoteIds.length);
15
16                 if (result.leftover != null && result.leftover.
       length() > 0) {
17                     db.loadUpdate(result.leftover, false, null)
       ;
18                     Log.i(TAG, "Updated leftover starred cards,
       count: " + result.leftover.length());
19                 }
20             } else {
21                 Log.e(TAG, String.format("IDs number doesn't match,
       local: %d, remote: %d", update.localIds.length, result.remoteIds.
       length));
22             }
```

```
        }
    [...]
    }
    [...]
}
```

1.2.6 Les relations des Decks

Dans cette sous-section, nous avons tenté, à l'aide des requêtes SQL seules, étant donné la documentation presque inexistante, de comprendre les interrelations encore non explorées dans ce rapport entre les tables `Decks`, `Starred_decks` et `Cr_cast_decks`.

Nous avons identifié que la table `Decks` joue un rôle central en tant que structure de stockage, car les tables `Starred_decks` et `Cr_cast_decks` sont composées de copies de données des instances de cette table `Decks`. Nous avons pu retrouver plusieurs indices nous menant à cette conclusion. Dans le fichier `BasicCustomDeck.java`, nous trouvons la méthode *contains* qui itère sur tous les decks à l'aide d'une liste d'instances de la classe `BasicCustomDeck`. Il y a donc une notion sous-entendue de types de decks différents regroupés sous le nom de Deck.

```
public static boolean contains(@NotNull List<BasicCustomDeck> decks
    , @NotNull Deck find) {
    for (BasicCustomDeck deck : decks)
        if (deck.name.equals(find.name) && (find.watermark == null
    || deck.watermark.equals(find.watermark))
            return true;

    return false;
}
```

Or, la méthode *getAllDecks* est utilisée uniquement par l'interface utilisateur. Elle permet de récupérer tous les decks. Le regroupement de ceux-ci se réalise en récupérant, d'une part, les `decks normaux`. Ensuite, selon la configuration de l'utilisateur, la méthode inclut également ceux qualifiés de `CrCastDecks` et/ou de `StarredDecks`.

```
/**
 * Get all decks. This method MUST be used only to display UI
   because it filters decks:
 * - Overloaded starred decks won't be displayed if Overloaded isn
   't signed in.
 * - CrCast decks won't be displayed if CrCast isn't signed in.
 *
 * @return A safely modifiable list of decks
 */
@NonNull
public List<BasicCustomDeck> getAllDecks() {
    List<BasicCustomDeck> decks = new LinkedList<>(getDecks());
```

```
11      if (OverloadedUtils.isSignedIn()) decks.addAll(getStarredDecks(
        false));
12      if (CrCastApi.hasCredentials()) decks.addAll(getCachedCrCastDecks
        ());
13      return decks;
14  }
```

Un autre indice montrant cette relation est que `CustomDeck`, `CrCastDeck` et `StarredDeck` étendent tous les trois la classe `BasicCustomDeck`. `CustomDeck` est l'équivalent de la table `Decks`, comme peut le montrer la récupération de données de ce type toujours en utilisant la méthode *getDecks()*, qui retourne une liste de `CustomDeck` contenant les instances de la table `Decks`.

```
1  // Signature de la classe CustomDeck
2  public final class CustomDeck extends BasicCustomDeck{
3
4  // Signature de la classe CrCastDeck
5  public final class CrCastDeck extends BasicCustomDeck{
6
7  // Signature de la classe StarredDeck
8  public static class StarredDeck extends BasicCustomDeck{
9
10 // Recuperation de decks via la methode getDecks
11 List<CustomDeck> decks = db.getDecks();
```

Un extrait de code dans le fichier `CustomDecksDatabase.java` montre la façon dont `Starred_decks` est mis à jour. Cette mise à jour est réalisée grâce à des valeurs reprises depuis une instance de la table `Deck`. Afin de remonter la trace de cette opération, nous nous sommes penchés sur la classe `SyncUtils` du package `overloaded`. Là, la méthode *loadStarredDecksUpdate* est appelée. L'argument *"deck"* de l'instruction *obj.getJSONObject("deck")* pose la question de ce qu'il pourrait référencer. Pour le découvrir, nous avons remonté le flux d'exécution présumé en nous intéressant à la classe `OverloadedSyncApi` dans laquelle *"deck"* ne fait référence qu'à des instances de la classe `CustomDeck`.

```
1  // CustomDeckDatabase
2  public void loadStarredDecksUpdate(@NonNull JSONArray update, boolean
       delete, @Nullable Long revision) {
3      [...]
4      JSONObject obj = update.getJSONObject(i);
5      JSONObject deckObj = obj.getJSONObject("deck");
6
7      ContentValues values = new ContentValues();
8      values.put("shareCode", deckObj.getString("shareCode"));
9      values.put("name", deckObj.getString("name"));
10     values.put("watermark", deckObj.getString("watermark"));
11     values.put("owner", obj.getString("owner"));
12     values.put("cards_count", deckObj.getInt("count"));
13     values.put("remoteId", obj.getLong("remoteId"));
14     values.put("lastUsed", System.currentTimeMillis());
```

```
15      db.insert("starred_decks", null, values);
16      [...]
17  }
18
19  // SyncUtils
20  OverloadedSyncApi.get().updateStarredCustomDecks(ourRevision, update
        .update, new OverloadedSyncApi.Callback<
        StarredCustomDecksUpdateResponse>() {
21      @Override
22      public void onResult(@NonNull StarredCustomDecksUpdateResponse
        result) {
23          [...]
24          db.loadStarredDecksUpdate(result.leftover, false, null);
25          [...]
26      }
27  }
28
29  // OveloadedSyncApi
30  public void patchCustomDeck[...]
31      [...]
32      .put("deck", deck)
33      [...]
```

De nombreux autres aspects et instructions contenus dans le code nous confortent dans nos hypothèses. Par exemple, certains endroits du code appliquent sur des instances de `Starred_decks` des opérations définies dans `CustomDecks`, ailleurs on peut observer la façon dont `Cr_cast_deck` est peuplé à partir de données d'instances de la table `Decks`, ...

Nous émettons donc comme hypothèse finale que la table `Decks` agit comme conteneur commun pour les données des tables `Starred_decks` et `Cr_cast_decks`. Nous représentons cela dans notre schéma conceptuel par une relation d'héritage entre `Decks` et `Starred_decks`/`Cr_cast_decks`. Cette relation est partielle, car, comme démontré par l'existence et l'utilisation de la classe `CustomDecks`, il existe des decks repris dans les instances de la table `Decks` n'étant ni qualifiés de `StarredDecks` ni de `Cr_cast_decks`.

1.2.7 Renommage de tables

La table `cr_cast_decks` a deux interprétations possibles que nous n'avons pas su départager en raison d'un manque de documentation claire à ce propos.

Premièrement, ***card_cast_decks***, venant du nom cité à différents endroits dans le code et dont la signification nous reste floue. Voici un commentaire du code qui nous a tout d'abors fait penser à cette hypothèse.

```
1  //region Custom decks CrCast (old CardCast)
```

Deuxièmement, ***chrome_cast_decks***, car en explorant le répertoire GitHub et le

nom des releases, il est apparu que de nombreuses modifications ont été réalisées afin de permettre l'utilisation de cette application avec la technologie Chromecast de Google. D'où la supposition de l'utilisation de cette table pour mettre en place cela et d'où le nom évoquant cette technologie. Cela est corroboré par la notion d'*enabled* (d'activation) présente pour cette table à de nombreux endroits du code comme dans l'exemple qui suit.

```
public boolean crCast Enabled () {
    return getOrDefault("CR_CAST_ ENABLED ");
}
```

Nous avons renommé la table `Starred_decks` en `Favourite_Decks`. Ceci s'explique par la possibilité, en mode premium (**overloaded**), de marquer des decks comme favoris à l'aide d'une étoile, comme détaillé dans la section 1.2.5 traitant de la colonne *remote_id*. Cette modification vise à rendre le nom de la table plus clair et compréhensible.

1.2.8 Le cas Cards_v2

Comme expliqué dans la section 1.1 **Description du schéma physique**, nous avons identifié une table supplémentaire, que nous avons tout d'abord renommée `Cards_v2` dans le schéma physique enrichi. Cette table appartient à une seconde base de données. Bien qu'elle ne contienne qu'une seule table, elle est cruciale pour les données du jeu. Initialement, elle portait le nom de `Card` dans le code source, mais pour éviter toute confusion, nous l'avons renommée `FavoriteCards` dans le schéma enrichi.

Notre découverte tardive de cette base de données s'explique en partie par le fait que l'outil ***SQLInspect*** mettait en avant la commande SQL de création de la table `Card_v2`, mais ne mentionnait pas explicitement la base de données ni la table elle-même. Ceci s'explique par le fait que ***SQLInspect*** ne considérait pas la requête de création de la table comme ajoutant une table supplémentaire au schéma général de la base de données, car il existait déjà une table avec le même nom.

```
"CREATE TABLE IF NOT EXISTS cards (  id INTEGER PRIMARY KEY UNIQUE,
   deck_id INTEGER NOT NULL, type INTEGER NOT NULL, text TEXT NOT
   NULL, creator TEXT DEFAULT NULL, remoteId INTEGER UNIQUE)"

"CREATE TABLE IF NOT EXISTS cards (id INTEGER UNIQUE PRIMARY KEY NOT
    NULL, blackCard TEXT NOT NULL, whiteCards TEXT NOT NULL,
   remoteId INTEGER UNIQUE)"
```

Cependant, une analyse plus approfondie a révélé que les deux bases de données étaient distinctes, comme en témoignent les requêtes de création de table présentes dans les classes `CustomDecksDatabase` et `StarredCardsDatabase`. D'une part, car ce sont deux classes différentes pour la création et la gestion de base de données.

D'autre part, les deux tables créées possèdent des colonnes différentes. Cette différence entre les colonnes des deux tables nous a tout d'abord laissé perplexe, car cela apporte une incohérence dans les modèles de la base de données. Nous avons d'abord pensé à du code mort, mais il s'est avéré que ce n'était pas le cas. L'autre explication possible est la présence de deux bases de données distinctes. Nous pouvons nous conforter dans cette intuition en observant que les classes `CustomDecksDatabase` et `StarredCardsDatabase` implémentent toutes deux le *pattern singleton*. Dans le **schéma conceptuel**, cela se traduira à travers des cardinalités de 1 pour chacune d'entre elles.

```
public final class StarredCardsDatabase extends SQLiteOpenHelper {
    [...]
    private static StarredCardsDatabase instance = null;

    private StarredCardsDatabase(@Nullable Context context) {
        super(context, "starred_cards.db", null, 1);
    }

    @NonNull
     public static StarredCardsDatabase get(@NonNull Context context
    ) {
        if (instance == null) instance = new StarredCardsDatabase(
    context);
        return instance;
    }
    [...]
}
```

```
public final class CustomDecksDatabase extends SQLiteOpenHelper {
    [...]
    private static CustomDecksDatabase instance;

    private CustomDecksDatabase(@Nullable Context context) {
        super(context, "custom_decks.db", null, 15);
    }

    @NonNull
     public static CustomDecksDatabase get(@NonNull Context context)
    {
        if (instance== null)
            instance = new CustomDecksDatabase(context);
        return instance;
    }
    [...]
}
```

Dans le **schéma logique**, nous proposons de renommer cette base de données en `StarredCards` pour plus de clarté. Dans le **schéma conceptuel** et **logique**, nous considérons ces deux bases de données comme une seule entité, car elles sont toutes deux liées à la logique métier de l'application. Une hypothèse plausible est que l'implémentation de l'option premium, permettant l'accès à cette fonc-

tionnalité, a été ajoutée ultérieurement dans le développement de l'application. Cette hypothèse pourrait expliquer la création d'une nouvelle classe pour définir une base de données distincte, à côté de celle déjà existante. Le schéma physique peut refléter cette différenciation.

1.2.9 La clé étrangère deck_id

Dans le code source de l'application, l'utilisation de la variable *deck_id* permet d'établir une relation entre les tables `Decks` et `Cards`. Cette relation peut raffiner notre **schéma physique enrichi** et **logique** en créant un lien entre une carte spécifique et le deck auquel elle est associée. On peut remarquer dans le code que l'on récupère un `CustomDeck` via la méthode *getDeck*, en y passant en paramètre un *deckId*. De plus, lors de la sélection d'une carte dans la table `cards`, la condition *WHERE deck_id=?* est utilisée pour filtrer les cartes en fonction de leur *deckId*. Cette requête permet ainsi de récupérer toutes les cartes d'un deck spécifique. Ce genre de construction est présent à plusieurs endroits du code. Pour chaque cas, il est possible d'établir ce lien entre `Decks` et `Cards` en remontant le flux de la codebase. Ceci établit donc une relation entre les cartes et le deck via la clé étrangère *deck_id*. La clé étrangère trouvée est une clé étrangère standard dans la table `Cards` qui fait référence à l'identifiant de la table `Decks`.

```
1  @NonNull
2  public List<CustomCard> getCards(int deckId) {
3      CustomDeck deck = getDeck(deckId); // Utilisation de deckId
       pour obtenir un objet CustomDeck
4      if (deck == null) throw new IllegalStateException();
5
6      SQLiteDatabase db = getReadableDatabase();
7      db.beginTransaction();
8      try (Cursor cursor = db.rawQuery("SELECT * FROM cards WHERE
       deck_id=?", new String[]{String.valueOf(deckId)})) { //
       Utilisation de deckId pour obtenir tous les cartes ayant ce
       deckId
9          if (cursor == null) return new ArrayList<>(0);
10
11         List<CustomCard> list = new ArrayList<>(cursor.getCount())
       ;
12         while (cursor.moveToNext()) list.add(new CustomCard(deck,
       cursor));
13         return list;
14     } finally {
15         db.endTransaction();
16     }
17 }
```

1.2.10 Les colonnes revision et lastUsed

Les colonnes *revision* et *lastUsed* présentent toutes deux un problème similaire : elles sont déclarées comme étant de type *integer* dans les commandes SQL permettant de créer les tables où elles apparaissent, alors que dans le code Java, on leur

assigne des valeurs de type *long*.

En ce qui concerne leur signification, *lastUsed* représente la période de temps écoulée depuis la dernière utilisation du deck ou de la carte, tandis que *revision* représente la période de temps depuis le dernier patch sur le serveur (uniquement pour les decks). La notion de **patch** demeure cependant floue même après nos recherches.

```
// CrCastDeck
@NonNull
public static CrCastDeck fromCached(@NonNull Cursor cursor) {
    [...]

    return new CrCastDeck(cursor.getString(cursor.getColumnIndex("
    name")),
            [...]
            cursor.getLong(cursor.getColumnIndex("lastUsed")));
}

@NonNull
public static CrCastDeck parse(@NonNull JSONObject obj, @NonNull
    CustomDecksDatabase db, boolean favorite)        [...]
    lastUsed = System.currentTimeMillis(); // Retourne un type Long
    [...]
}

// StarredCardDatabase
long revision = OverloadedApi.now();

// OverloadedApi
public static long now() {
    if (TrueTime.isInitialized()) return TrueTime.now().getTime();
    else return System.currentTimeMillis();
}
```

Finalement, nous avons appris que la raison derrière cette différence de typage avec la base de données est due à la non-prise en charge du type *long* par SQLite. Les *long*, étant stockés sur 8 octets, peuvent être stockés sous forme d'entier dans la base de données. Ceci nous mènera, dans la suite de notre rapport à discuter de cette manière de faire. Ce cas étant similaire à celui des colonnes *white_cardes/black_cards*.

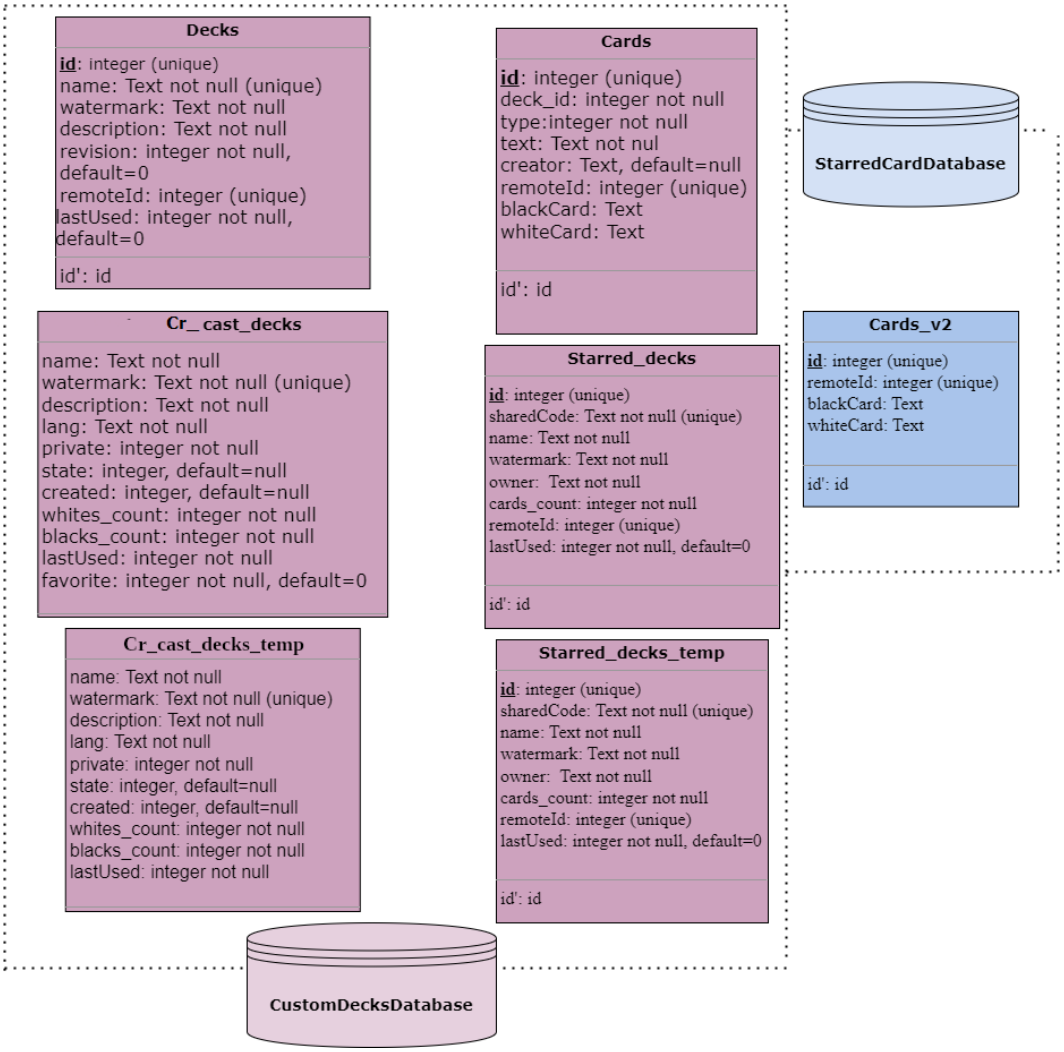
1.3 Schéma physique enrichi

En conclusion de l'analyse que nous avons mené, nous en avons construit ce **schéma physique enrichi**. Il est composé de deux bases de données distinctes, l'une composée de 6 tables : `Decks`, `Cards`, `Cr_cast_decks`, `Starred_decks`, `Cr_cast_decks_temp` et `Starred_decks_temp`; et l'autre composé d'une seule table `Favourite_cards`.

Les quatres premières tables, à l'exception de `Cr_cast_decks`, possèdent un identifiant unique définit comme un entier et nommé *id*. Des similitudes entre les colonnes *watermark*, *name* et *lastUsed* ont été observées dans les trois tables contenant `decks` dans leur nom. Des dynamiques entre ces tables, telles que l'utilisation de la colonne *remoteId* dans trois tables et les variations d'utilisation des tables de decks, ont également été examinées en détail.

Les deux tables temporaires, `Cr_cast_decks_temp` et `Starred_decks_temp`, semblent correspondre respectivement aux tables `Cr_cast_decks` et `Starred_decks`, avec certaines variations comme la colonne *favorite* manquante dans `Cr_cast_decks_temp`. La table `Favorite_cards`, associée à une autre base de données, a été incluse dans notre reconstruction en raison de son utilisation pour les mêmes données métier de l'application.[1]

[1] Ayant reçus des directives contraires, nous avons décidé de suivre les conventions présentes dans la slide 34 du Chapitre 1/ introduction du cours pour ce qu'il faut reprendre où non dans le schéma physique.



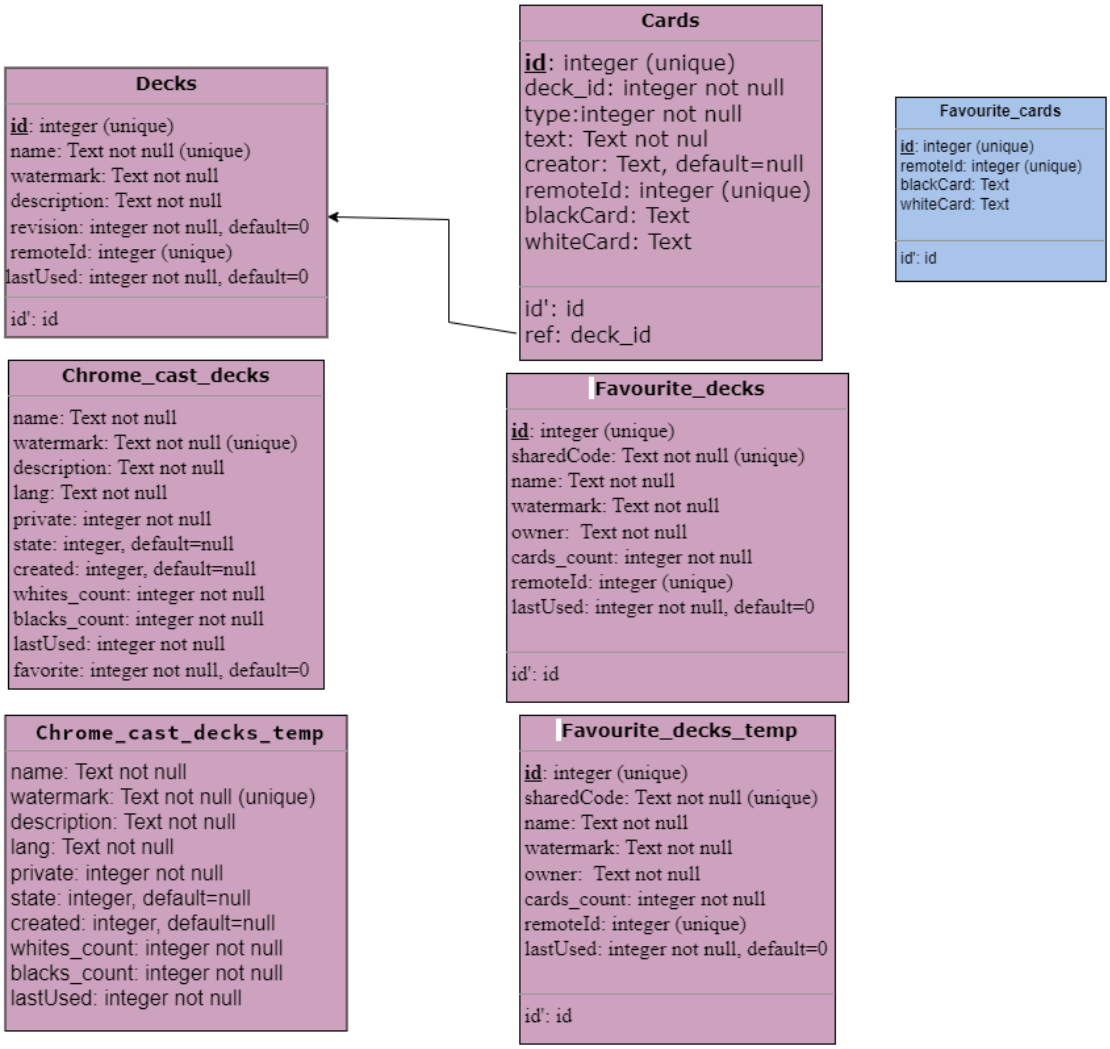
1.4 Schéma logique

Nous avons identifié l'existence d'une clé étrangère importante, celle de *deck_id*, établissant une relation entre les tables `Decks` et `Cards`. Cette clé étrangère permet de lier chaque entrée dans la table `Cards` à un deck spécifique dans la table `Decks`.

De plus, la codebase réaffirme certaines contraintes observées dans le **schéma physique**. Par exemple, la contrainte d'unicité de la colonne *name* dans la table `Decks` est vérifiée avant l'utilisation de sa valeur. D'autres éléments tels que les colonnes *watermark*, *blackCard* et *whiteCard* ont été analysés en détail pour comprendre leur utilisation dans le code. Par exemple, la colonne *watermark* est utilisée comme un label permettant de classifier les cartes et les decks, tandis que les colonnes *blackCard* et *whiteCard* dans la table `Cards` sont des booléens utilisées pour distinguer les types de cartes.

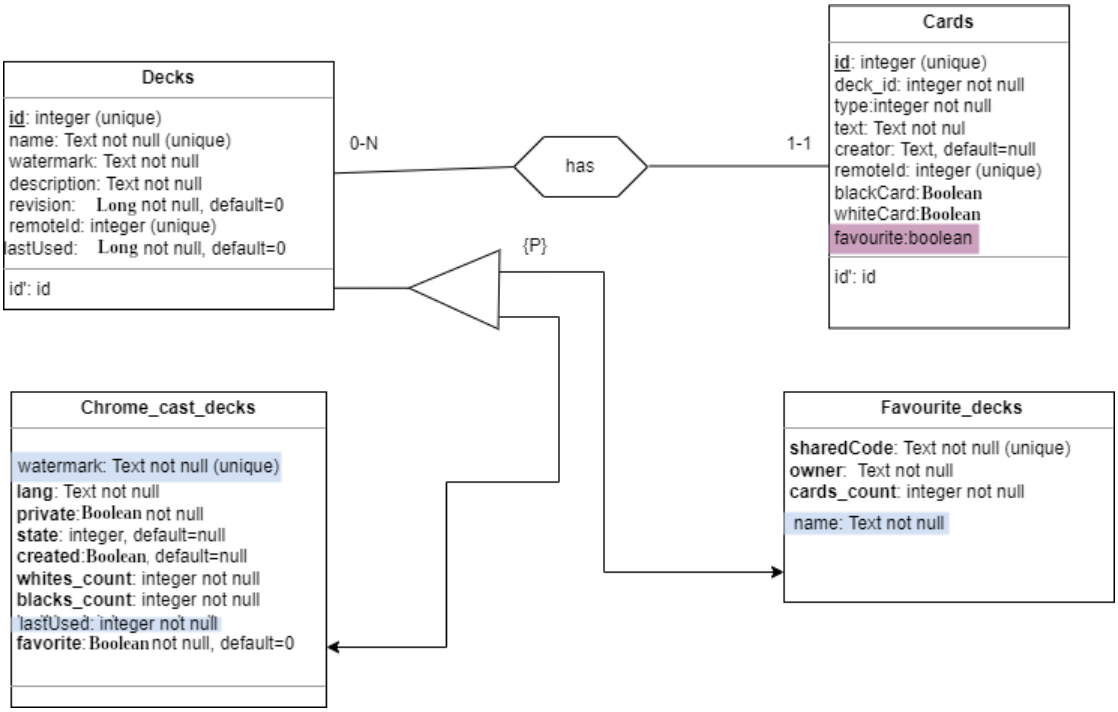
Le renommage de certaines tables, comme `Cards_V2` en `Favourite_Decks`, vise à rendre les noms plus clairs et compréhensibles. De plus, la découverte tardive

de la base de données `StarredCardsDatabase` et son intégration dans la base de donnée `CustomDecksDatabase` ont été prises en compte fusionnant les deux pour plus de clarté dans le **schéma logique**.



1.5 Schéma conceptuel

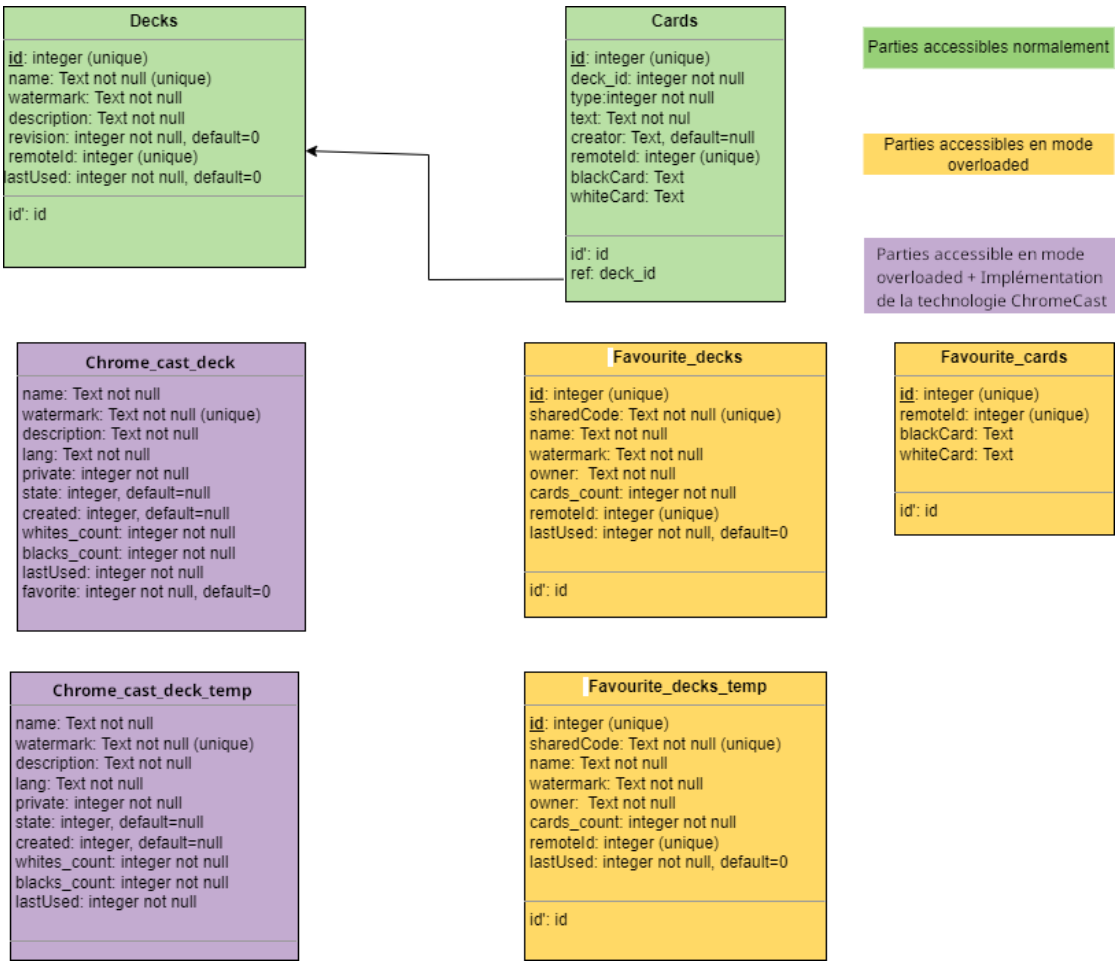
Le **schéma conceptuel** intègre les résultats de l'analyse des relations entre les tables, y compris la découverte que la table `Decks` agit comme un conteneur commun pour les données des tables `Starred_decks` et `Cr_cast_decks`. Une relation d'héritage partielle est établie entre `Decks` et `Starred_decks`/`Cr_cast_decks`, reflétant la structure de stockage des données. De plus, la clé étrangère standard ajoutée au schéma logique est transformée en relation *has* avec comme cardinalité `{0-N;1-1}`


Decks
id: integer (unique)
name: Text not null (unique)
watermark: Text not null
description: Text not null
revision: Long not null, default=0
remoteId: integer (unique)
lastUsed: Long not null, default=0
id': id
0-N
has
1-1
Cards
id: integer (unique)
deck_id: integer not null
type:integer not null
text: Text not nul
creator: Text, default=null
remoteId: integer (unique)
blackCard:Boolean
whiteCard:Boolean
favourite:boolean
id': id
{P}
Chrome_cast_decks
watermark: Text not null (unique)
lang: Text not null
private:Boolean not null
state: integer, default=null
created:Boolean, default=null
whites_count: integer not null
blacks_count: integer not null
lastUsed: integer not null
favorite:Boolean not null, default=0
Favourite_decks
sharedCode: Text not null (unique)
owner: Text not null
cards_count: integer not null
name: Text not null

2 Étape 2

2.1 Logical Sub-Schema (LSS)

Pour dériver le **sous-schéma logique** effectivement utilisé par les programmes, nous avons examiné attentivement les interactions entre les différentes parties du schéma logique initial et les dépendances des programmes. L'objectif était de distiller uniquement les éléments du schéma logique qui sont effectivement utilisés par les programmes, éliminant ainsi tout ce qui relève du *code mort*. Seulement, toutes les parties du schéma logique sont utilisés par l'application. Donc nous avons divisé le schéma logique en 3 parties. Nous avons coloré en vert les tables qui sont utilisées par l'application dans les cas basiques d'utilisation. En jaune, ce sont les tables utilisées uniquement en mode overloaded. Enfin, en mauve, ce sont les tables utilisées en mode overloaded pour la technologie Chromecast.



2.2 Analyse de la Répartition des Requêtes SQL et mesure de Complexité Cognitive

Dans cette section, nous présentons une analyse détaillée de la répartition des requêtes SQL au sein de la base de données, en distinguant les requêtes que nous avons appelé de *"premier ordre"* et de *"second ordre"*. De plus, nous évaluons une sorte de complexité cognitive associée à chaque type de requête.[2]

2.3 Répartition des Requêtes SQL de Premier Ordre

Nous commençons par examiner la répartition des requêtes SQL directes, que nous qualifions de "premier ordre". Ces requêtes sont exécutées directement à l'intérieur des classes dont le nom est associé à une base de données. Les deux classes correspondant à cette description sont `CustomDecksDatabase` et `StarredDecksDatabase`.

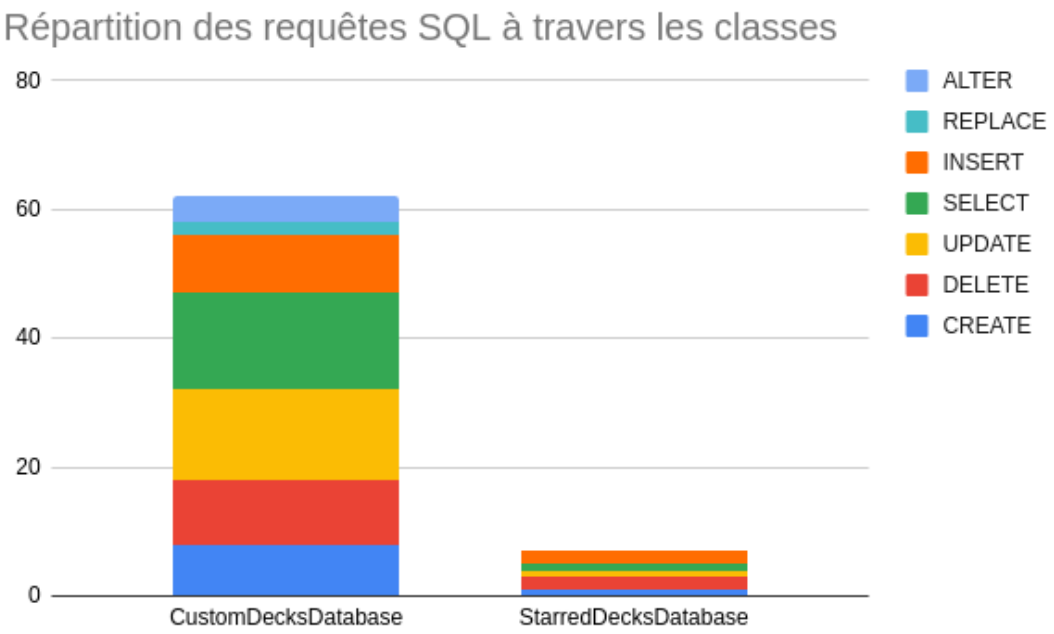


Figure 3: Répartition des Requêtes SQL de Premier Ordre dans la Codebase

La Figure 3 illustre la répartition des différents types de requêtes SQL de premier ordre. On peut remarquer que le nombre de requêtes SQL pour `CustomDecksDatabase` est bien plus important que celui pour `StarredDecksDatabase`, ce qui est cohérent avec notre hypothèse d'ajout rapide et incrémental discutée dans la section 1.2.8 abordant les deux bases de données.

[2] Toutes les données sur les requêtes du premier ordre et second ordre sont disponibles sur ce document Microsoft Excel partagé.

2.4 Répartition Globale des Requêtes SQL

En élargissant notre analyse aux requêtes que nous appelons de "second ordre", c'est-à-dire celles qui sont effectuées à travers des méthodes appelées sur la variable `db`, nous obtenons une vue plus complète de la répartition des requêtes SQL dans la codebase.

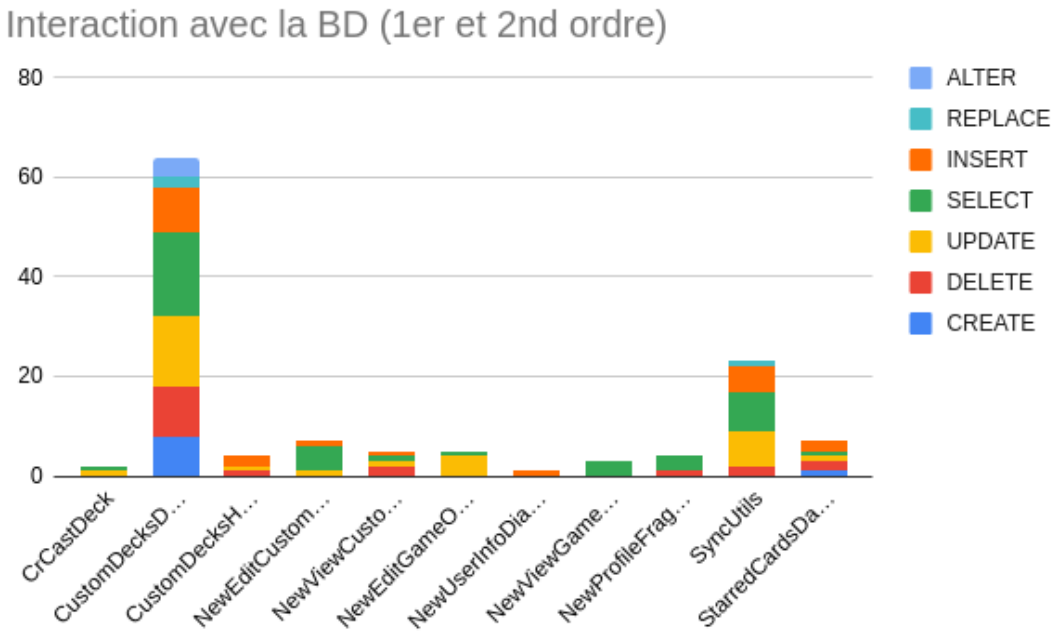


Figure 4: Répartition Globale des Requêtes SQL dans la Codebase

La Figure 4 présente la répartition globale des requêtes SQL, y compris celles de second ordre. Elle permet de visualiser l'ensemble des interactions avec la base de données. `CustomDecksDatabase` reste largement la classe dans laquelle le nombre de requêtes est le plus prédominant, mais `StarredCardsDatabase` ne prend pas la seconde place. En effet, on peut voir que `SyncUtils` effectue bien plus de requêtes SQL si l'on prend en compte les requêtes du second ordre. Ceci peut être intéressant pour une analyse d'impact. `SyncUtils` semble être une classe utilitaire utilisée pour la synchronisation des données entre l'API applicatif et la base de données.

Ensuite, nous avons examiné comment chaque type de requête, qu'il soit de premier ou de second ordre, est utilisé au sein de la codebase.

La Figure 5 détaille l'utilisation de chaque type de requête, offrant une vision spécifique des interactions avec la base de données.

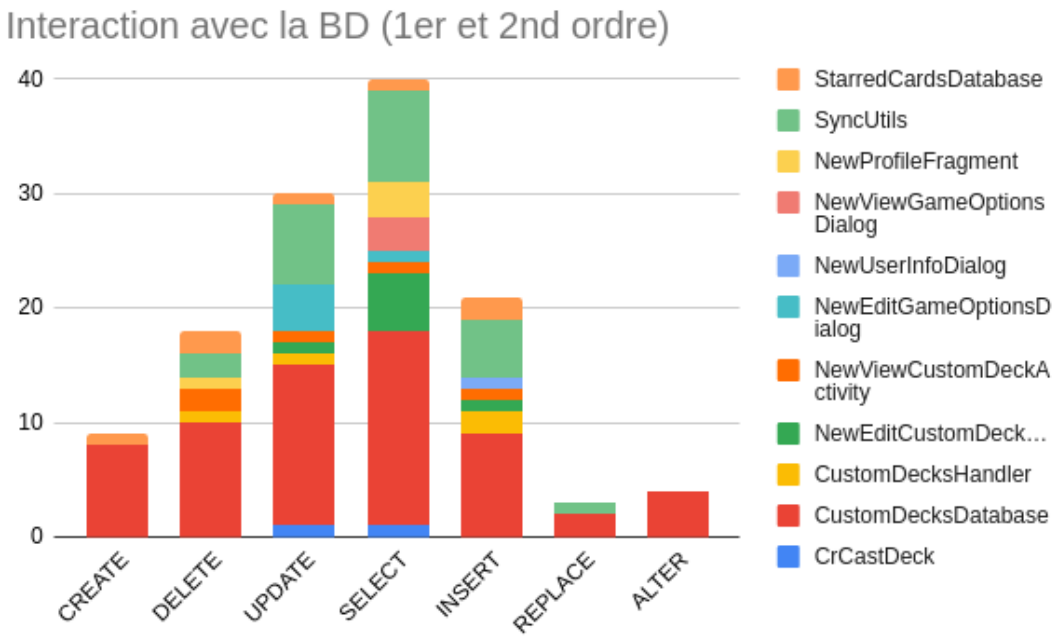


Figure 5: Utilisation de Chaque Type de Requête dans la Codebase

2.5 Complexité Cognitive des Requêtes

Enfin, nous évaluons la complexité cognitive associée à chaque type de requête, en tenant compte de critères définis. La complexité cognitive est mesurée en points, représentant la difficulté de compréhension de la requête et des variables associées.

Table 1: Facteurs Influant sur la Complexité Cognitive des Requêtes SQL

Facteur	**Description**	**Points**
Requête SQL	Présence d'une requête SQL	1
Structure de Requête	Présence d'un paramètre de substitution (?) dans la requête	+1 par ?
Variables de Programme	Utilisation de variables du programme pour compléter les ?	+1 par variable

Facteur	Description	Points
value.put Structure	Utilisation de `value.put` pour spécifier le contenu de la requête	+1
Conditions et Boucles	Exécution de la requête dépendant de structure conditionnelle ou itérative	+1
Méthodes de Second Ordre		
Arguments de Fonction	Nombre d'arguments dans les méthodes appelées	+1 par argument
Origine des Variables	Provenance des variables utilisées (hors du scope de la fonction courante)	+1 par variable
Variables de requête	Si variable provient d'un résultat de requête à la BD	+ complexité de la requête
Interaction avec l'API	Si la variable provient du résultat d'un appel à leur API	+10

La Figure 6 présente la complexité cognitive de chaque type de requête par classe qui les utilisent, ainsi que la complexité moyenne. Cette mesure offre un aperçu de la charge cognitive impliquée dans la compréhension des interactions avec la base de données.

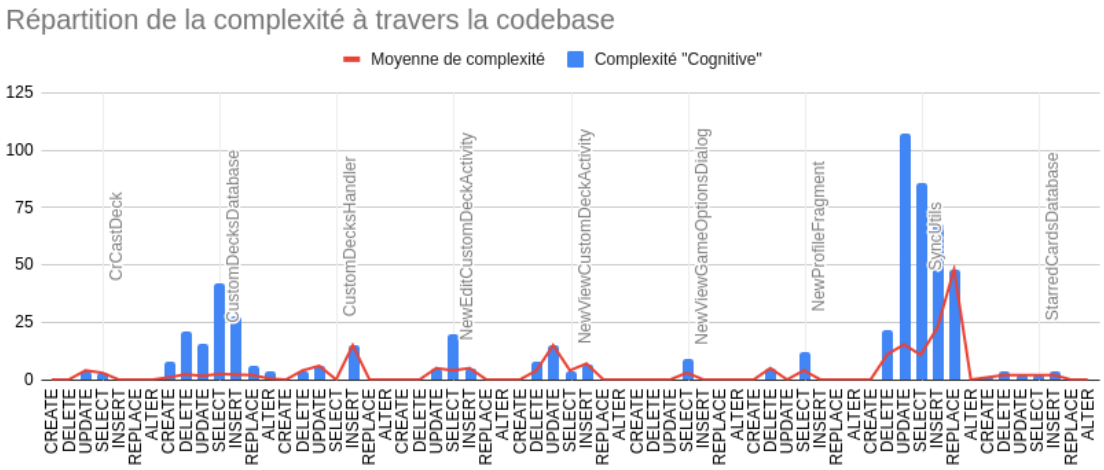


Figure 6: Complexité Cognitive et Complexité Moyenne de Chaque Type de Requête au travers de la codebase

3 Étape 3

Premièrement afin d'évaluer l'impact de nos scénarios, nous avons trouvé judicieux de définir une mesure afin de pouvoir quantifier plus aisément et l'avons divisée en trois gradation afin de facilité sa clarté.

Impact faible

Lorsque les modifications entraînées par le scénario n'implique que la création de moins de quatre requêtes SQLite (ne se suivant pas dans la même méthode, car sinon nous considérons également cet impact comme faible peu importe le nombre de requêtes ajoutées se suivant dans le code).

Impact moyen

Lorsque au moins trois classes différentes sont impactées par les modifications à effectuer ou que plus de trois requêtes SQLite sont à ajouter dans des méthodes différentes.

Impact critique

Ce cas reprend principalement les scénarios engendrant des modifications importantes à la fois dans le schéma de la base de données, dans différentes classes et dans la sémantique globale les liant.

Ensuite, avant de débuter l'exploration de nos scénarios, nous souhaitions mettre en avant les lignes directrices qui nous ont guidées dans leur sélection et leur rédaction. Celles-ci sont:

- **Maintenance et amélioration de la cohérence des données**
  dans la base comme dans leur utilisation par le code (cela avant mais surtout après la mise en place hypothétique de nos scénarios)
- **Amélioration de la qualité de la base de donnée**
  (et du code l'utilisant) en vue de la **réduction des code smells** (mauvaises pratiques de programmation, ici, dans le concept du développement de bases de données).
- **Facilité de mise en place**
  de la solution apportée par le scénario (même si nous avons tout de même sélectionner certains scénarios jugés comme ayant un impact de grade critique cela dû à leur intérêt d'un point de vue théorique et une curiosité quant-aux solutions disponibles pour ceux-ci).

3.1 Ajouter une contrainte `CHECK` sur la colonne *favorite* de la table `Chrome_cast_decks`

Justification

Nous avons tout d'abord entrepris d'apporter une modification du type de la colonne *favorite* de la table `Chrome_cast_decks` en booléen car la valeur de cette colonne est déjà utilisée de cette manière dans le code.

```
db.beginTransaction();
try {
    for (CrCastDeck deck : toUpdate) {
        [...]
        values.put("favorite", deck.favorite ? 1 : 0);
        [...]
        db.insertWithOnConflict("cr_cast_decks", null, values,
    SQLiteDatabase.CONFLICT_REPLACE);
    }
    db.setTransactionSuccessful();
} finally {
    db.endTransaction();
}
```

Seulement, nous avons appris par la suite que **SQLite** ne prend pas en charge le type booléen. Nous avons donc pensé à garder le type `INTEGER` et à ajouter une contrainte pour s'assurer que les seules valeurs autorisées sont *0* et *1*, représentant respectivement *false* et *true*.

Analyse d'impact : Faible

La modification consistant à ajouter une contrainte `CHECK` sur la colonne *favorite* lors de la création de la table a un impact global faible sur le système. Le reste du code n'a pas besoin de subir de modifications, bien qu'une documentation concernant le casting de la valeur récupérée de la base de données dans le code serait la bienvenue.

```
db.execSQL("ALTER TABLE cr_cast_decks ADD favorite INTEGER NOT
    NULL DEFAULT 0 CHECK (favorite IN (0, 1))");
```

3.2 Suppression des tables `temp`

Justification

Les tables temporaires `temp` sont utilisées à un seul endroit du code pour effectuer une mise à jour de la base de données lors des cas 11 et 12 du switch case de la méthode onUpgrade de l'application. Dans ces situations, les tables temporaires sont créées ; les données des tables originales y sont recopiées ; la table originale est supprimée et enfin, la table temporaire est renommée pour prendre le nom de la table originale. Cette suite d'opérations semble ne pas avoir d'influence directe sur le résultat final mais présente tout de même un risque d'introduction d'incohérence dans la base de données **CustomDecksDatabase** en cas d'erreurs lors du déroulement de cette suite.

```
1  @Override
2  public void onUpgrade(SQLiteDatabase db, int oldVersion, int
       newVersion) {
3      Log.d(TAG, "Upgrading from " + oldVersion + " to " +
       newVersion);
4      switch (oldVersion) {
5          case 9:
6              [...]
7          case 10:
8          case 11:
9              db.execSQL("CREATE TABLE cr_cast_decks_tmp (name TEXT
       NOT NULL, watermark TEXT NOT NULL UNIQUE, description TEXT NOT
       NULL, lang TEXT NOT NULL, private INTEGER NOT NULL, state
       INTEGER DEFAULT NULL, created INTEGER DEFAULT NULL,
       whites_count INTEGER NOT NULL, blacks_count INTEGER NOT NULL,
       lastUsed INTEGER NOT NULL)");
10             db.execSQL("INSERT INTO cr_cast_decks_tmp SELECT *
       FROM cr_cast_decks");
11             db.execSQL("DROP TABLE cr_cast_decks");
12             db.execSQL("ALTER TABLE cr_cast_decks_tmp RENAME TO
       cr_cast_decks");
13             db.execSQL("ALTER TABLE cr_cast_decks ADD favorite
       INTEGER NOT NULL DEFAULT 0");
14         case 12:
15             db.execSQL("CREATE TABLE starred_decks_tmp (id INTEGER
        PRIMARY KEY UNIQUE, shareCode TEXT NOT NULL UNIQUE, name TEXT
       NOT NULL, watermark TEXT NOT NULL, owner TEXT NOT NULL,
       cards_count INTEGER NOT NULL, remoteId INTEGER UNIQUE, lastUsed
        INTEGER NOT NULL DEFAULT 0)");
16             db.execSQL("INSERT INTO starred_decks_tmp SELECT *
       FROM starred_decks");
17             db.execSQL("DROP TABLE starred_decks");
18             db.execSQL("ALTER TABLE starred_decks_tmp RENAME TO
       starred_decks");
19         case 13:
20             [...]
21         case 14:
22             [...]
23     }
```

```
24      Log.i(TAG, "Migrated database from " + oldVersion + " to " +
    newVersion);
25  }
```

Analyse d'impact : Faible

Nous proposons de supprimer ces requêtes inutiles qui introduisent ces deux tables temporaires, engendrant un plus grand risque et étant idempotent lorsque tout se passe bien. L'impact est assez faible car la modification ne touche qu'une partie très localisée du code car les modifications ne touchent que deux cas d'un switch case de cette même méthode onUpgrade.

```
1  @Override
2  public void onUpgrade(SQLiteDatabase db, int oldVersion, int
       newVersion) {
3       Log.d(TAG, "Upgrading from " + oldVersion + " to " +
       newVersion);
4       switch (oldVersion) {
5           case 9:
6               [...]
7           case 10:
8           case 11:
9               db.execSQL("ALTER TABLE cr_cast_decks ADD favorite
       INTEGER NOT NULL DEFAULT 0");
10          case 12:
11          case 13:
12              [...]
13          case 14:
14              [...]
15      }
16      Log.i(TAG, "Migrated database from " + oldVersion + " to " +
       newVersion);
17 }
```

3.3 Déclaration de la colonne *name* en tant qu'Unique dans les tables `Favourite_decks` et `Chrome_cast_decks`

Justification

La colonne *name* est unique dans la table `Decks`. Selon l'hypothèse formulée dans la partie 1.2.6 intitulée **Les relations des Decks**, avec la relation d'héritage entre les decks, il est souhaitable de rendre la colonne *name* unique dans les tables `Favourite_decks` et `Chrome_Cast_Decks` afin d'assurer la cohérence des données. Nous n'avons pas trouvé de contre-indication à ce choix. Il existe même une méthode *isNameUnique* dans le code, utilisée pour vérifier l'unicité du nom et employée dans la partie UI pour empêcher l'utilisateur de créer un nom déjà existant en lui affichant un message d'erreur.

Analyse d'impact : Faible

Les modifications à effectuer se situent uniquement dans les déclarations de la base de données. Il y aura des modifications aux requêtes de création dans `CustomDecksDatabase` aux lignes {79 ; 80 ; 88 ; 91 ; 97 ; 104}. Ces requêtes se succédant dans la même méthode onCreate, cela justifie le niveau d'impact évalué à faible.

```
db.execSQL("CREATE TABLE IF NOT EXISTS starred_decks (id INTEGER
    PRIMARY KEY UNIQUE, shareCode TEXT NOT NULL UNIQUE, name TEXT
    NOT NULL UNIQUE, watermark TEXT NOT NULL, owner TEXT NOT NULL,
    cards_count INTEGER NOT NULL, remoteId INTEGER UNIQUE, lastUsed
     INTEGER NOT NULL DEFAULT 0)");
```

3.4 Ajout de la colonne *owner* et *name* de `Favourite_decks` en identifiant secondaire

Justification

Les colonnes *owner* et *name* de la table `Favourite_decks` sont utilisées en combinaison comme identifiant pour récupérer les instances de la classe `CustomDecks` ayant le nom et le propriétaire spécifiés dans la partie *main* de l'application. Cette utilisation nous pousse à rendre cet identifiant secondaire explicite afin d'améliorer la lisibilité du code et de clarifier sa sémantique. Pour garantir l'unicité de la composition de ces deux colonnes en tant qu'identifiant secondaire, il est nécessaire d'explorer le cas WhatIf de la section 3.3, déclarant la colonne *name* comme unique dans les tables `Favourite_decks` et `Chrome_cast_decks`. Cependant, **SQLite** ne permet pas de définir des identifiants secondaires. En effet aucun mécanisme n'est défini dans la documentation de SQLite afin de prendre en charge cet aspect de la conception de base de données. Nous avons donc envisagé de remplacer l'identifiant primaire *id* par l'identifiant primaire composite *(name, owner)*. Cependant, l'utilisation d'un identifiant numérique présente plusieurs avantages, notamment une utilisation plus simple, une facilitation de la gestion des jointures et de meilleures performances pour les bases de données optimisées pour les opérations sur des clés primaires numériques. Nous avons donc choisi de conserver l'identifiant *id* et d'ajouter un *index* sur la combinaison des colonnes *(name, owner)* pour rendre cette relation explicite et améliorer l'efficacité des requêtes utilisant ces colonnes ensemble.

Analyse d'impact : Faible

L'impact est faible car il suffit pour cela d'ajouter la requête *CREATE INDEX* après la création des tables en question.

```
db.execSQL("CREATE INDEX idx_name_owner ON favourite_decks (name,
    owner)");
```

3.5 Remplacer la base de données Starred_cards par une colonne dans la table Cards

Justification

Afin d'éviter la redondance des tables, nous pouvons supprimer la table `favourite_cards` de notre schéma de base de données, avec l'intégration d'une nouvelle colonne nommée `favourite` dans la tabel **Cards**. La colonne sera définie comme un type entier (integer), contenant exclusivement les valeurs 1 ou 0. Cette configuration vise à simuler un type booléen, comme expliqué dans la section 3.1.

Cette modification a pour but de simplifier la logique de l'application en éliminant les redondances et en réduisant le nombre de jointures nécessaires pour récupérer les préférences des utilisateurs. En incorporant l'indicateur de favori directement dans la table `cards`, nous éliminons le besoin d'une table séparée pour gérer ces informations. Cela aura pour avantage de diminuer la complexité des requêtes et d'améliorer les performances de la base de données, tout en conservant une interface simple pour les opérations de marquage des cartes en tant que favorites.

L'ajout de la colonne `favourite` permettra aux requêtes qui sélectionnent les cartes favorites d'être plus directes et plus rapides, car elles pourront se passer d'une jointure avec une autre table. Cette évolution reflète notre engagement à fournir une architecture de données à la fois performante et intuitive, qui soutient efficacement les fonctionnalités de l'application et l'expérience utilisateur.

Analyse d'impact : Moyen

Pour mener ces changements à bien, il faut une restructuration partielle du programme. Au lieu de faire appel à la table `Favourite_cards`, il faudrait désormais vérifier si la carte est considéré comme étant favorite dans la table `Cards`, ce changement est donc à opérer dans différentes méthodes de différentes classes de notre programme.

Ces changements doivent notamment être effectués uniquement dans les classes suivantes :

- AnotherGameManager
- NewProfileFragment
- SyncUtils

3.6 Fusionner les bases de données CustomDecksDatabase et StarredCardsDatabase

Justification

L'objectif principal de cette fusion est d'intégrer la table cards_v2 provenant de la base de données `StarredCardDatabase` dans la base de données `CustomDecksDatabase`. Cette intégration vise à centraliser les informations relatives aux cartes et aux decks dans une unique base de données, permettant ainsi une gestion plus cohérente et simplifiée des données.

La table `Cards_v2`(1.2.8) contient des informations essentielles sur les cartes qui sont utilisées conjointement avec les decks définis dans `CustomDecksDatabase`. L'intégration de ces données dans une seule base de données facilitera les requêtes inter-tables et renforcera l'intégrité référentielle des données. Cela signifie que toutes les opérations de création, de mise à jour et de suppression seront désormais gérées de manière uniforme et centralisée.

Analyse d'impact : Critique

L'impact de cette fusion est considérable et nécessite une attention particulière. Voici la manière dont nous procéderions pour appliquer ce cas *WhatIf* :

- **Relocalisation des éléments dans le code source :** Déplacer la création des tables et l'attribution de leurs fonctions au sein de la classe `CustomDecksDatabase` au lieu de la classe `StarredCardsDatabase`. Cela implique des modifications substantielles dans le code existant, notamment dans la manière dont les fonctions sont appelées et dans les interactions avec la base de données.
- **Wrapper pour la base de données :** Création d'un wrapper autour de la base de données qui contient la logique nécessaire pour gérer la table `Favourite_cards`.
- **Utilisation du design pattern Adapter :** Utiliser le design pattern Adapter implémentant l'interface attendue par les méthodes déjà existantes interagissant avec l'ancienne table. Cet adaptateur utilisera le wrapper pour effectuer les opérations sur la nouvelle base de données. De cette manière, l'ancien code peut continuer à utiliser les méthodes existantes durant la transition, et les méthodes introduites par la suite pourront implémenter la nouvelle logique.
- **Transition progressive avec le wrapper :** Remplacement graduel des anciennes méthodes pour les faire utiliser directement le wrapper.

Bien sûr, le wrapper n'a pas à se limiter uniquement à la table qui a migré. Une implémentation d'un wrapper utilisé comme une API, montrant au code une vue conceptuelle de la base de données, permettrait une meilleure gestion et évolution

du code. Le code métier ne doit être modifié que lors de changements conceptuels du schéma de la base de données. Autrement, les modifications seront effectuées dans le wrapper. Cependant, cette approche nécessite du temps pour la mise en place du wrapper et surtout une vision conceptuelle du schéma de la base de données, ce qui ne semble pas être fait de base dans ce projet.

Ces changements doivent notamment être effectués dans les classes suivantes :

- AnotherGameManager
- NewProfileFragment
- SyncUtils

3.7 Remplacer la colonne cards_count Favourite_decks par une méthode

Description

Dans le schéma conceptuel (1.5), on peut observer dans la table `Favourite_decks` que le calcul du nombre de cartes est l'objet de sa propre colonne, or, cela n'étant pas nécessaire, nous avons décidé de les remplacer par l'ajout de requêtes sql permettant de compter le nombre d'instances de la table `Favourite_cards` relié à chacune des instances `Favourite_decks`.

Justification

Le fait de réaliser des opérations sur des valeurs pouvant être extraites d'autres tables et de les entreposer dans une colonne représente un code smell dans le domaine du développement de base de données. Il nous a donc semblé opportun de démontrer qu'une solution plus élégante était possible et facilement mettable en place.

Analyse d'impact : Moyen

Afin d'effectuer ces changements, il faudrait supprimer la colonne `count` dans la base de données et modifier les endroits dans le code où cette colonne est utilisée par une fonction permettant de calculer le nombre de cartes présentes.

Il faudrait également utiliser une clé étrangère id reliant `Favourite_cards` à `Decks`. L'impact a été défini comme moyen car une seule classe est impactée mais par la modifications de quatre requêtes sql dans différentes méthodes de celle-ci.

Exemple de fonction :

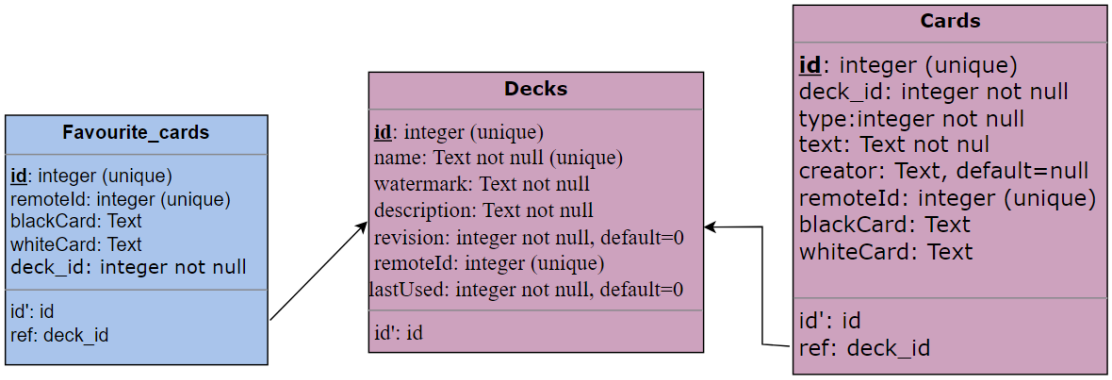


Figure 7: Schéma logique modifié

```
1    private int countCards(int deck_id) {
2        SQLiteDatabase db = getReadableDatabase();
3        db.beginTransaction();
4        try (Cursor cursor = db.rawQuery("SELECT COUNT(*) FROM
     favourite_cards WHERE deck_id=?", new String[]{String.valueOf(
     deckId)})) {
5            if (cursor == null || !cursor.moveToNext()) return 0;
6            else return cursor.getInt(0);
7        } finally {
8            db.endTransaction();
9        }
10   }
```

3.8 Remplacer les colonnes *whites_count* et *blacks_count* par une méthode

Description

Dans le schéma conceptuel (1.5), on peut observer dans la table `Chrome_cast_decks` que le calcul du nombre de cartes de chaque type possible (noires ou blanches) est l'objet de sa propre colonne or, cela n'étant pas nécessaire, nous avons décidé de les remplacer par l'ajout de requêtes sql permettant de compter le nombre d'instances de la table `Cards` où respectivement la valeur de *blackCard* et *whiteCard* est équivalente à "1" (les valeurs de ses colonnes étant déclarées comme des *Text* même si elles sont utilisées comme des booléens pour déterminer le type de l'instance en question de la table `Cards`).

Justification

Le fait de réaliser des opérations sur des valeurs pouvant être extraites d'autres tables et de les entreposer dans une colonne représente un code smell dans le domaine du développement de base de données. Il nous a donc semblé opportun

de démontrer qu'une solution plus élégante était possible et facilement mettable en place.

Analyse d'impact : Moyen

La marche à suivre pour réaliser ces modifications consistera en :

- création de deux requêtes (comme dans le scénario précédent concernant la colonne *cards_count* de *Favourite_decks* mais en en allant chercher les instances de la table `Cards` (ou `Favorite_cards`) où la carte est spécifiée dans la colonne *blackCard* (respectivement *whiteCard*) si la carte est du type voulu pour la méthode correspondante.
- suppression des colonnes blackCount et whiteCount de la table Chrome_cast_decks.
- le remplacement dans le code Java de l'utilisation des valeurs de la colonne blackCard par cette nouvelle requête (respectivement, pour whiteCard).

L'impact est moyen car il y a 4 requêtes à créer et 5 lignes de code Java à modifier.

3.9 Remplacer les colonnes *blackCard* et *whiteCard* de `Cards` par une nouvelle colonne

Description

Comme expliqué dans l'étape un, les colonnes *blackCard* et *whiteCard* de `Cards` déclarées comme étant de type *Text* ne servent qu'à différencier les deux sortes de cartes que l'on peut retrouver dans le jeu. Nous avons donc entrepris de les *remplacer* par une seule colonne de type *booléenne* mais cela n'étant pas possible dû aux limitations de **SQLite** nous avons préféré nous orienter vers un type *numérique* (*int*).

Justification

Cette modification a été mise en avant afin de trouver une solution plus élégante à celle couramment présente dans le projet. Le choix du type *int* pour représenter ce choix binaire pour cette nouvelle colonne tient dans le fait que les valeurs extraites de celles-ci remplacées sont déjà utilisées comme telles dans le code Java (voir 1.2.4).

Analyse d'impact : Moyen

La marche à suivre pour réaliser ces modifications consistera en :

- la création d'un nouvelle colonne isBlack dans la table Cards (à l'aide d'une requête sql alter table)
- l'insertion dans cette colonne des valeurs de la colonne blackCard (castée en booléens).

- la suppression des colonnes blackCard et whiteCard de la table Cards
- le remplacement dans le code Java de l'utilisation des valeurs de la colonne blackCard par celles de la colonne isBlack

L'impact a été évalué moyen à dû aux huit requêtes sql dans méthodes différentes.

3.10 Rendre la clé étrangère deck_id explicite et totale

Description - Justification

En partant de l'hypothèse que chaque carte (instance de la table Cards) est reliée à un seul Deck (instance de la table Decks), nous nous sommes demandés les répercussions qu'entraînerait la modification de la définition de cette clé étrangère standard (voir 1.2.9) en une clé étrangère totale.

Analyse d'impact : Faible

La solution la plus simple serait de créer une requête sql permettant de déclarer la clé étrangère deck_id comme totale. Malheureusement, d'une part, cette clé est implicite il va donc falloir effacer la colonne et la remplacer par la déclaration d'une clé étrangère standard totale faisant référence à l'identifiant de la table Decks et de l'autre l'ajout d'une clé étrangère à l'aide d'une requête de type **alter** n'est tout simplement pas possible avec la base de données actuelle (celle-ci étant exprimée en SQLite).

La marche à suivre recommandée dans ce cas est la suivante (dans la base de données CustomDecksDatabase et le fichier CustomDecksDatabase.java):

- renommer la table Cards en old_cards
- créer une nouvelle table Cards avec la clé étrangère deck_id déclarée explicitement comme totale (à l'aide de deux contraintes ondelete cascade et onupdate cascade ce qui est la solution pour sqlite) et comme une référence à la colonne identifiant de la table Decks.
- insérer les données de old_cards dans Cards.

Les lignes de codes correspondants à ces modifications pourront être ajoutées à partie de la ligne 81 dans la méthode *onCreate* de CustomDecksDatabase.

D'autres requêtes sont présentes dans ce même fichier mais elles ne nécessitent pas de modification car elle ne font que ce servir de la valeur de deck_id comme d'un identifiant pour les instances de la table Cards.

```
1 db.execSQL("ALTER TABLE Cards RENAME TO old_cards");
2 db.execSQL("CREATE TABLE Cards (id INTEGER PRIMARY KEY UNIQUE,
    deck_id INTEGER NOT NULL, type INTEGER NOT NULL, text TEXT NOT
    NULL, creator TEXT DEFAULT NULL, remoteId INTEGER UNIQUE,
    FOREIGN KEY (deck_id)
```

```
3         REFERENCES Deck (id)) ON DELETE CASCADE ON UPDATE CASCADE"
    ;
```

3.11 Autres scénarios envisagés mais non explorés

Dans cette sous-section, nous avons tenu à reprendre tous les scénarios intéressants auxquels nous avons pensé, que ce soit durant la relecture des concepts théoriques vus en cours, durant celle du code, ou encore pendant l'écriture de ce rapport, mais qui n'ont pas été traités et complètement explorés, cela par manque de temps.

Séparer la table Cards en deux sous-tables Black_cards et White_cards

Cette séparation est motivée par la prise en compte des deux types de cartes comme des objets distincts dans le code Java. L'idée générale de ce scénario est basée sur ce qui a été fait pour la table Decks sur le schéma conceptuel (1.5). Nous séparerions la table Cards en deux sous-tables : Black_cards et White_cards. La relation d'héritage observée serait toutefois totale et non partielle, comme c'est le cas pour les Decks. En effet, une carte est soit noire, soit blanche, alors qu'un deck, s'il n'est ni utilisé avec la technologie chrome_cast, ni mis en favori, peut toujours exister. Cette solution serait une alternative au scénario de modification des colonnes blackCard et whiteCard (voir 3.9), qui a été complètement explorée, car l'héritage discuté ici ne permettrait la création de deux nouvelles tables, mais n'aurait pas de colonnes différentes avec Cards, si ce n'est une nouvelle que l'on pourrait nommer typeCard et qui serait soit de type integer de façon similaire à la solution du scénario 3.9 ou soit comme étant de type Text en donnant pour valeur "white" ou "black".

Créer une table utilisateur et faire une référence vers celle-ci depuis `Favourite_decks`

La création d'une table utilisateur a été imaginée lors de l'écriture du scénario 3.4 **Ajout de la colonne *owner* et *name* de `Favourite_decks`**. En effet, dans ce scénario, on décrit l'utilisation de ces deux colonnes de `Favourite_decks` dans le code comme identifiant secondaire. Cela nous a fait penser que le propriétaire d'un deck aurait pu être extrait en créant une nouvelle table `utilisateur` et en remplaçant la colonne *owner* par une clé étrangère vers l'identifiant de la table ainsi créée. La table `User` n'aurait eu comme colonne uniquement *id* comme identifiant primaire et *name* qui est la valeur de *owner*. Nous n'avons pas exploré plus loin ce cas dans le *WhatIf* car la table `User` ainsi créée est très réduite. Cependant, cela rendrait l'évolution plus facile si la notion d'utilisateur devient plus développée pour les prochaines releases de l'application. Pour l'instant, tout ce qui concerne les utilisateurs et leur gestion ne se fait qu'à l'aide de code Java.

4 Étape 4

4.1 Évaluation du Schéma de la Base de Données

Le schéma de la base de données présente quelques inconvénients.

Un inconvénient notable est le manque de documentation complète. L'absence d'explications détaillées pour chaque table et colonne peut entraver la compréhension et la maintenance.

Par exemple, il existe une ambiguïté autour du terme `"deck"`. Son utilisation dans la dénomination des différentes tables crée de la confusion, et une définition claire est nécessaire pour améliorer la compréhension globale des schémas. En effet, la relation entre les tables `decks` est floue, nécessitant des conversions entre elles en fonction des colonnes manquantes et nécessaires à diverses endroits du code.

La classe `BasicCustomDeck` sert de fourre-tout avec une accumulation d'attributs au fil du temps, manquant d'une structure cohérente et étant un indice de manaque de stratégie pré-datant le début de ce projet quand au cahier des charges et à la structure de la base de données à utiliser. Cela pourrait être corrigé en affinant les extensions `CustomDeck` et `FavouriteDeck`, car toutes deux sont des extensions de `BasicCustomDeck`, créant de la redondance et des complications potentielles. Un autre exemple, est la création de la base de données `StarredCardsDatabase` et de son unique table `Cards` qui en plus de porter le même nom que une table déjà présente dans la base `CustomDecksDatabase` ne présente pas de comportement significativement différentes (elle possède moins de colonnes que la table originelle) et pourrait être aisément remplacée par les scénarios que nous avons mis en avant dans la partie trois de ce rapport.

Les incohérences dans les types de données entre les commandes SQL et le code Java, telles que *integer* vs *long*, présentent un risque d'erreurs lié aux limitations de SQlite comme nous l'expliquons . De plus, certains noms de colonnes, comme *watermark*, manquent de clarté et bénéficieraient d'explications détaillées sur leur but et leur utilisation au sein du code comme de l'application.

Un autre point négatif est la présence de manque de connaissances des bonnes pratiques de conception de bases de données tels que: l'utilisation d'aucune clé étrangère explicite dans ses différentes tables ainsi que la présence de colonnes ne servant que de lieu pour entreposer des informations dérivables d'autres tables ou de liaison entre celles-ci (comme pour les colonnes cards_count, whites_count et blacks_count).

4.2 Évaluation du Code de Manipulation de la Base de Données

Le code de manipulation démontre certaines forces et faiblesses. L'utilisation de modèles provenant de la library Sqlite pour Java-singleton 1.2.8 pour les classes de base de données garantit une instance unique, contribuant à la stabilité. La gestion des mises à jour de la base de données témoigne de la prise en compte de l'évolution de l'application.

```
1 public final class CustomDecksDatabase extends SQLiteOpenHelper {
```

Cependant, des exemples de code mort existent, notamment en ce qui concerne les tables temporaires, suggérant une redondance qui pourrait être éliminée pour simplifier le code. De plus, aucune relation entre les tables n'est définie explicitement dans la base de données. La progression de l'application sans une vision initiale cohérente soulève des préoccupations quant à la cohérence de la conception globale.

Le manque de cohérence dans la réflexion quant à l'utilisation des colonnes de la base de données est démontré par l'exemple des colonnes *remote_id*, utilisées à des fins de synchronisation entre l'API et la base de données `CustomDecksDatabase` (voir 1.2.5). Cette colonne montre également que le développeur a oublié que le fait de l'utiliser dans trois tables différentes et possédant les mêmes valeurs constituait en fait une clé étrangère multi-targets (qui n'est pas un type de clé étrangère standard).

Un autre exemple de manque de cohérence est l'ignorance du caractère unique de la colonne *name* pour les instances de la table `Decks`, ayant conduit à la revérification qu'aucune de ces instances ne possédait le même nom à l'aide de code Java dans une méthode dédiée *isNameUnique* (voir 3.3).

4.3 Recommandations pour l'amélioration

Documentation Complète : Fournir une documentation détaillée pour les 3 schémas de la base de données, expliquant la finalité de chaque table et colonne ainsi que la sémantique pour le schéma conceptuel. Tout cela pour faciliter l'évolution de la base de donnée.
Normalisation des Types de Données : Assurer la cohérence des types de données entre les déclarations SQL et le code Java pour éviter d'éventuelles erreurs et garantir la qualité globale du code. Lorsque ce n'est pas possible, implémenter et documenter l'implantation de la solution la plus efficace pour le typage des colonne, leur utilisation dans le code java et la conversion dans le code.
Clarté dans les Noms de Colonnes : Clarifier les noms de colonnes ambigus pour améliorer la compréhension du schéma de la base de données et la qualité générale du code en adoptant des conventions de nommage (cfr.1.2.3).
Consolidation des Définitions de "Deck" : Définir clairement le terme "deck"

pour éviter toute confusion et affiner la structure des classes associées.
Définition des relations entre les tables : Définir dans les requêtes SQL les relations de clés étrangères (implicite ainsi que celles mentionnées dans les scénarios *WhatIf*) entre les tables de la base de donnée.
Se former en SQLite:
Une formation SQLite permettrait de mettre en place ces différentes clés étrangères de façon correcte afin de ne pas créer des clés étrangères non standard (voir pathologiques pour remplacer les colonnes *blacks_count* et *whites_count* par exemple) ainsi que d'autres erreurs liées à la connaissances des spécificités de framework.
Revue de Conception : Envisager une revue de conception globale pour garantir la cohérence de l'architecture globale du système. (Ex: Volere)
Élimination du Code Mort : Supprimer les segments de code redondants ou obsolètes, tels que les tables temporaires, pour simplifier et améliorer la maintenabilité du code.
Instoration de tests : L'ajout de tests de couverture de code permettrait de reprérer plus facilement le code mort car ces tests ne sont pas présents (en tout cas ne sont pas disponibles sur le GitHub). De plus, l'ajout de tests de régression permettrait d'ajouter de la persistance dans l'évolution parallèle entre le code **DDL**, le code **Java** et les **schémas de base de données**.
Wrapper entre le code et la base de donnée : Comme expliqué dans le **WhatIf** 3.6, on peut implémenter une API entre le code et la base de donnée permettant au code de faire abstraction de l'implémentation logique de la base de donnée pour utiliser des méthodes représentant une interaction avec le schéma conceptuel de la base de donnée.

5 Bonus

La version actuelle considérée au moment de la rédaction de ce rapport est la version 5.1.7 .
Les deux autres versions de l'application sont les suivantes.

D'abord, la version 3.2.1 qui se situe chronologiquement et selon le nombre de versions parues au milieu du cycle de vie et d'évolution de cette application.

Ensuite, la version 2.6.8 étant la plus ancienne version disponible sur le répertoire github de ce projet.

Afin de souligner l'évolution représentée aux travers de ces différentes versions, nous nous concentrerons tout d'abord sur les différences majeures de fonctionnalités présentes entre celles-ci. Puis, nous représenterons le schéma conceptuel de chacune. Enfin nous nous concentrerons sur l'aspect de la qualité en prenant en compte les changements réalisés pendant le cycle de vie de cette application que ce soit concernant le code (Java, sql,...) ou la structure du schéma de la base de données (toujours en ne se concentrant que sur la fusion des bases CustomDecks-Database et StarredCardsDatabase).

5.1 Comparaison des fonctionnalités

Version	Fonctionnalité
Decks customisés	
2.6.8	∅ Aucune mention
3.2.1	Intégration de CustomDeck avec un répertoire dédié, création et utilisation d'une base de données, mention dans le changelog
5.1.7	Apparition de fichiers adaptateurs pour la technologie Chrome_cast
Utilisation des decks Card_cast	
2.6.8	Existence dans le répertoire SpareActivities d'un sous-répertoire CardCastDecks sous-entendant que c'est un type de deck particulier

Version	Fonctionnalité
3.2.1	Abandon de CardCast dans la version 3.1.4 (visible dans le changelog.md), récupération des decks CardCart ainsi que disparition des fichier portant sur CardCast (Par exemple, les fichiers présents dans netio/models n'y sont plus et on a pu voir que le répertoire netio a été renommé en api).
5.1.7	∅ Aucune mention ni de changement par rapport à 3.2.1. Il reste des commentaires expliquant la réutilisation pour la technologie ChromeCast.
Gestion des Starred_cards	
2.6.8	Présence d'un répertoire nommé Starred contenant des fichiers spécifiques à la gestion des Starred_decks (StarredDecksManager, StarredDecksActivity).
3.2.1	Disparition de la plupart des fichiers portant sur les Starred_decks (StarredDecksManager, StarredDecksActivity, StarredDecksAdapter). Nous posons comme hypothèse que ces changements sont dus à l'introduction de CustomDecksDatabase.
5.1.7	Introduction de la StarredCardsDatabase et introduction d'une table StarredDecks dans CustomDecksDatabase. Ceci renforce notre hypothèse évoquée en 3.2.1. Que les StarredDecks méritent d'être gérés par la base de données, seulement ils ont décidé de créer une nouvelle base de donnée à la va-vite sans trop de conceptualisation.
Implémentation de la technologie Chrome_cast	
2.6.8	∅ Aucune mention
3.2.1	∅ Aucune mention
5.1.7	Apparition d'un répertoire lié dans le répertoire api afin de mettre en place cette technologie (cfr. Fonctionnalité Card_cast 5.1.7).

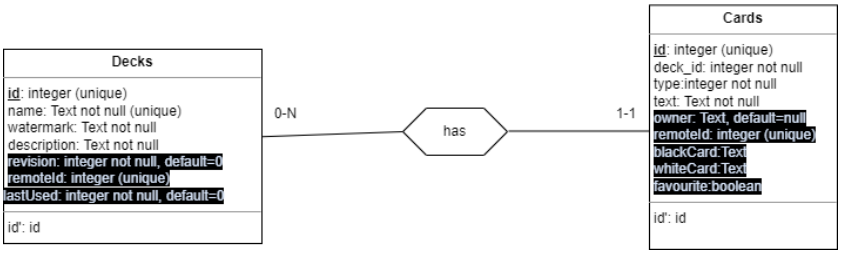
Version	Fonctionnalité
Gestion du mode Overloaded (premium)	
2.6.8	∅ Aucune mention
3.2.1	∅ Aucune mention
5.1.7	Apparition d'un répertoire lié au mode Overloaded à la racine de l'application contenant le code portant sur ce mode, l'introduction d'un chat entre utilisateurs, les demandes d'amis. Apparition également de nombreux fichiers contenant *Sync*. Ils sont utilisés pour gérer la synchronisation des comptes utilisateurs, de leurs decks dès qu'il s'abonnent (et se désabonnent).

5.2 Schémas conceptuels des différentes versions

Version 2.6.8.
Il n'y a PAS d'utilisation de base de données donc il n'y a PAS besoin de schéma conceptuel.

Version 3.2.1.
Présence uniquement des tables `Decks` et `Cards` dans la base de données nouvellement créée `Customdecksdatabase`. Celle-ci, étant mise en place dans le fichier `CustomDecksDatabase.java` aux lignes 39 et 40 de sa méthode *onCreate*. Cette méthode étant la même où l'on avait découvert les requêtes **SQL** permettant la création de `CustomDeckDatabase` dans la version courante 5.1.7 .

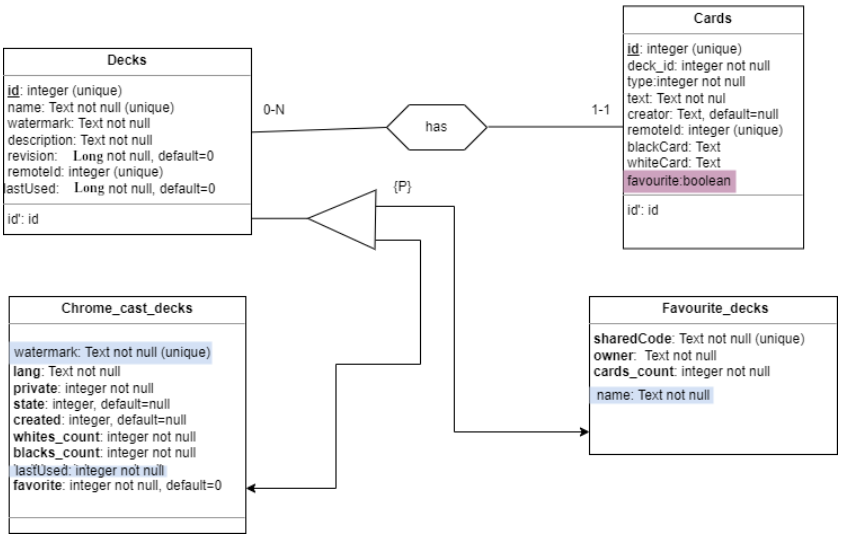
```
1 @Override
2 public void onCreate(SQLDatabase db) {
3     db.execSQL("CREATE TABLE IF NOT EXISTS decks (id INTEGER, name
      TEXT UNIQUE NOT NULL, watermark TEXT NOT NULL, description
     TEXT NOT NULL");
4     db.execSQL("CREATE TABLE IF NOT EXISTS cards (id INTEGER,
     deck_id INTEGER NOT NULL, type INTEGER NOT NULL, text TEXT NOT
     NULL");
5 }
```



Comme on peut l'observer, des colonnes ont été sur-lignées en noire, cela pour représenter qu'elles n'étaient pas encore présentes pour ces tables durant cette version de l'application et qu'elles ont été ajoutées par la suite. On peut de même constater la présence de la clé étrangère *deck_id*.

Version 5.1.7.

Ajout de la technologie **Chromecast**, elle obtient sa propre table dans `CustomDecks Database`. Ajout également de la table `starred_decks`, cependant aucune justification de cet ajout dans changelogs ou en commentaire dans le code. Serait-elle utile pour la synchronisation liée à l'overloaded vu leurs ajouts simultanés ? Ce qui confirmerait notre hypothèse présentée dans l'étape 2.



5.3 Commentaires sur l'évolution de la qualité

Points Positifs

Depuis la première version, l'introduction d'une base de donnée dans le projet a été une amélioration pour la gestion des données. Ensuite, dans la version courante de l'application, nous pouvons voir dans notre sous-schéma logique que toutes les parties du schéma est utilisé dans l'application.

Confirmation des hypothèses négatives

La non conceptualisation et la non-planification de la base de donnée a entraîné des problèmes dans le développement du logiciel. Par exemple, la technologie

`Starred_Cards` qui était géré uniquement par des fichiers Java avant l'introduction de la première base de données (cfr. 2.6.8). Après l'introduction de la base de donnée, il y a eu nécessité d'en introduire une deuxième, une nouvelle table dans la première ainsi que du code Java (cfr. 5.1.7).

Il y a un manque de connaissances par rapport à la base de données et introduction d'incréments "pansements" au lieu de corriger la source de l'erreur. Il y a donc un manque de connaissances par rapport aux bonnes pratiques de programmation pour la gestion de base de données. Par exemple, la clé implicite introduite lors de la création de la base de données en 3.2.1 n'est toujours pas corrigée en 5.1.7. De plus, le code mort lié aux decks card_cast laissé entre 2.6.8 et 3.5.7 a été réutilisé et adapté pour gérer la mise en place de la technologie chrome_cast.