VINCI QUIZ
Réalisation

PAGE DE SERVICE

Référence : Vinci Quiz
Plan de classement : analyse-conception-quiz
Niveau de confidentialité : confidential

Mises à jour

Version	Date	Auteur	Description du changement
1.0.0	12-04-2016	Julian CUCHERAT	Création Networking Inc.
1.1.0	10-12-2022	Nathan PONSON	MAJ

Validation

Version	Date	Nom	Rôle
1.0.0	18-12-2022	Delphine TALARON	Direction Technique Vinci Quiz Project

Diffusion

Version	Date	Nom	Rôle
1.0.0	18-12-2022	All	SLAM Networking Inc.

OBJET DU DOCUMENT

Ce document décrit l'implémentation Java du projet Vinci Thermo Green.
Ce projet vise à produire une application réalise l'implémentation du diagramme des classes métier et du diagramme de séquence objet présenté dans la documentation d'analyse conception.

Sommaire

1 ARCHITECTURE

Conformément à l'analyse conception, la réalisation de l'application est structurée en trois couches selon une structure qui ressemble à un design-pattern MVC (Model View Controler) mais qui en réalité ne l'est pas vraiment. Cependant cette structure du code permet d'envisager dans une version ultérieure une évolution vers un modèle réellement MVC.

Le modèle MVC fonctionne selon le principe illustré par le schéma ci-dessous[1] :

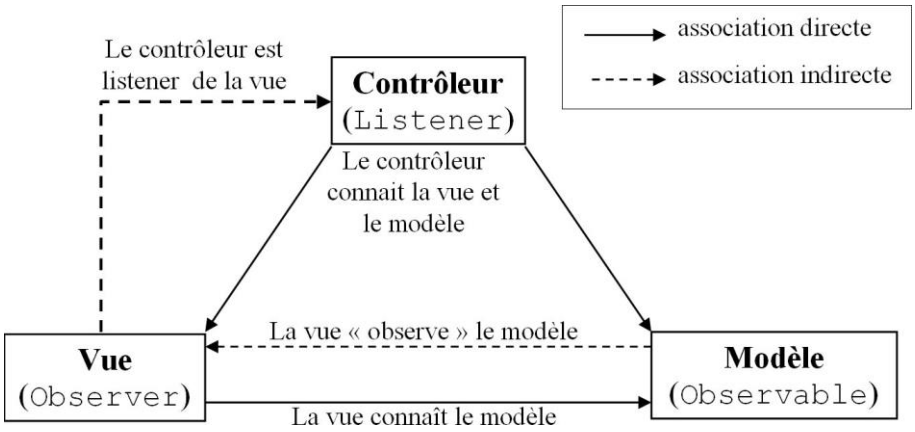

Figure 1 : principe de fonctionnement du modèle MVC

La conception de la v.1.0.0 de l'application Vinci Quiz prévoit la mise en œuvre d'un contrôleur. Ce contrôleur représente non pas le "listener" au sens MVC du terme mais le Data Access Object (DAO)[2] d'une architecture n-tiers.
Cependant, l'utilisation de la bibliothèque graphique Swing permet d'écouter la vue au sens propre du design-pattern MVC. Cela pourra être abordé lors d'une version ultérieure.

L'utilisation d'un DAO permet de s'abstraire de la façon dont les données sont stockées au niveau des objets métier. Ainsi, le changement du mode de stockage ne remet pas en cause le reste de l'application. Seules les classes dites "techniques" seront à modifier.

Les objets en mémoire vive sont liés à des données persistantes (stockées en base de données, dans des fichiers, dans des annuaires, etc…). Le modèle DAO regroupe les accès aux données persistantes dans des classes techniques spécifiques, plutôt que de les disperser. Il s'agit surtout de ne pas écrire ces accès dans les classes "métier", qui ne seront modifiées que si les règles de gestion métier changent.

Ce modèle complète le modèle MVC (Modèle - Vue - Contrôleur), qui préconise de séparer dans des classes les différentes problématiques :

- Des "vues" (charte graphique, ergonomie)
- Du "modèle" (cœur du métier)
- Des "contrôleurs" (tout le reste : l'enchaînement des vues, les autorisations d'accès, etc…)

Ci-dessous est présenté le diagramme des composants mettant en application le modèle MVC :

[1] http://www.infres.enst.fr/~hudry/coursJava/interSwing/boutons5.html
[2] https://fr.wikipedia.org/wiki/Objet_d'acc%C3%A8s_aux_donn%C3%A9es

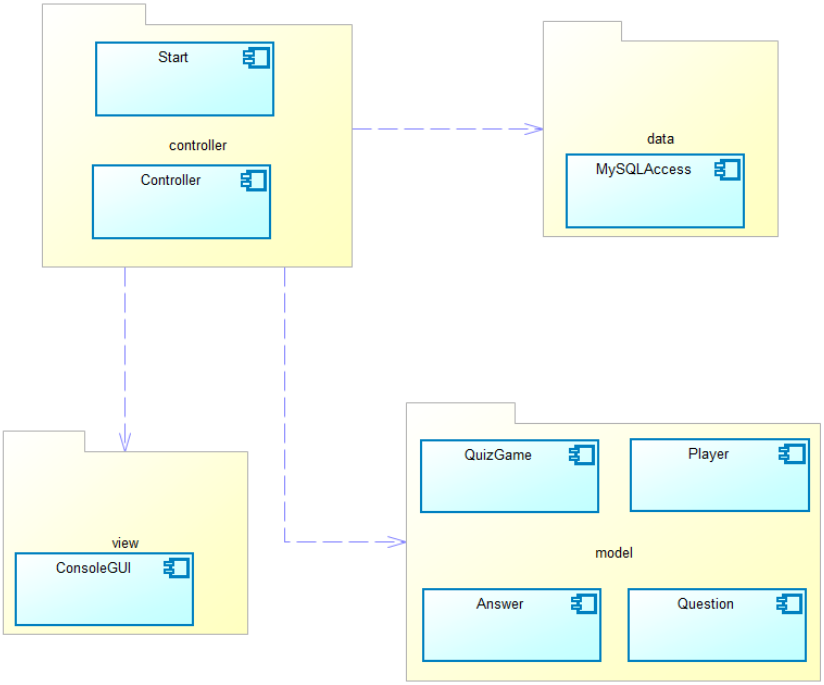

Figure 2 : diagramme des composants de l'application

Dans ce schéma, on observe clairement le modèle MVC. Dans le package controller, la classe Start est le lancement de l'application, elle contient le main. Ce dernier va instancier un objet controller qui va, comme son nom l'indique, contrôler l'application :

```
public static void main(String[] args) throws FileNotFoundException,
ClassNotFoundException, ParseException, IOException, SQLException {
    //controller instantiation
    Controller theController = new Controller();
}
```

Ce contrôleur est celui qui « voit tout », c'est-à-dire que c'est par lui que vont passer les différents composants de l'application pour communiquer entre eux. Il faut pour cela créer des constructeurs pour chaque classe qui attendent un objet controller en paramètre. Lors de son initialisation, le contrôleur va donc instancier la plupart des objets en passant comme paramètre, lui :

```
public Controller(){

}
```

2 Implementation des classes metiers

L'analyse a permis de modéliser les classes métiers selon le diagramme ci-dessous (non documenté, cf. document d'analyse-conception) :

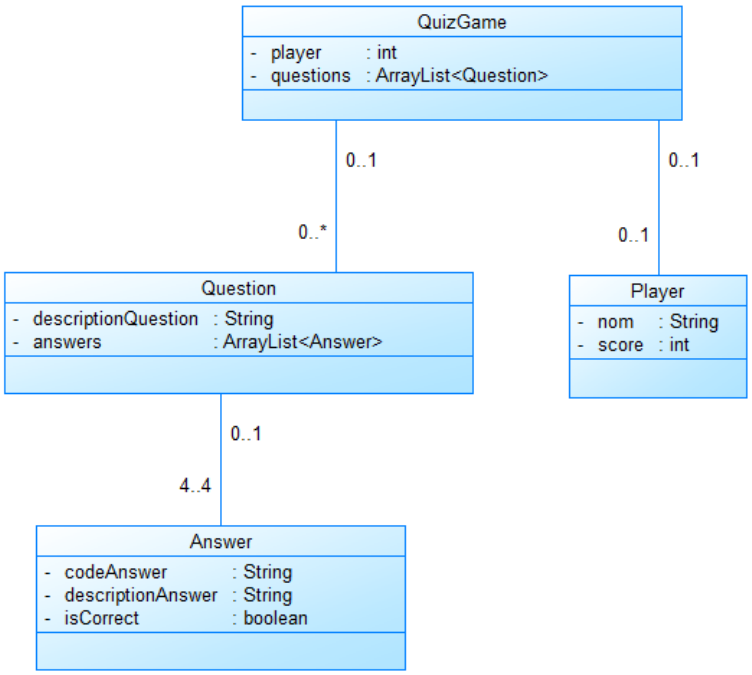

Figure 3 : diagramme des classes métiers

L'application lit une base de données qui contient les questions et réponses du quiz. La classe "QuizGame" stocke les questions et réponses de la partie, ainsi que le joueur.

```
public class QuizGame {

   private Player myPlayer;

    private ArrayList<Question> questions;
    private ArrayList<Integer> listeIdQuestions;

    private Controller monController;

    public QuizGame(Controller aController, Player player, ArrayList<Question>
questions) {
         this.monController = aController;
        this.myPlayer = player;
        this.questions = questions;
    }

   public Player getMyPlayer() {
    return myPlayer;
   }

   public void setMyPlayer(Player myPlayer) {
    this.myPlayer = myPlayer;
   }

   public ArrayList<Question> getQuestions() {
        return questions;
    }

    public void setQuestions(ArrayList<Question> questions) {
        this.questions = questions;
    }
```

```
    public Boolean isCorrectThisAnswer(Question question, int idAnswer) {
        if (question.getAnswers().get(idAnswer).getIsCorrect()) {
            this.myPlayer.setMyScore(this.myPlayer.getMyScore() + 10);
        return true;
        } else {
            return false;
        }
    }
}
```

3 Acces aux donnees

3.1 SQLite[3]

4 Gerer une collection d'objet[4]

5 Interface Homme – Machine

L'ihm est la partie interface. C'est ce que voit l'utilisateur. Il lui permet de communiquer avec l'application, via des boutons pour notre cas.

Il va varier en fonction de ce qui est demandé initialement. Mais il sera dans tous les cas en android basé sur différents layouts en fonction de l'étape en question. Pour notre le layout des résultats n'est pas le même que le layout pour les questions.

Sous Android Studio il est possible de constituer cette interface graphique en codant sans vraiment regarder mais également pas à pas via un menu dédié avec une sélection de plusieurs boutons ou label différents par exemple. Après avoir placé nos différents ou composants actionnables il faudra ensuite les relier à des actions (en les appelant avec leurs noms) concrètes pour qu'ils affichent ou effectuent les actions souhaitées.

Cette interface sera contenue dans des objets appeler des vues. Dans ces mêmes vues ont retrouves l'appel de nos différents composants et de ces attributs comme c'est le cas avec notre button :

Au final pour afficher une fenêtre avec des simples questions derrière il y a forcément 2 fichiers : Un fichier XML pour la description d'interface (layout) Un fichier JAVA pour les actions (instructions)

5.1 Common Intents[6]

Un `Intent` vous permet de démarrer une activité dans une autre application en décrivant une action simple que vous souhaitez effectuer (telle que "afficher une carte" ou "prendre une photo") dans un `Intent` objet. Ce type d'intention est appelé intention implicite , car il ne spécifie pas le composant d'application à démarrer, mais spécifie à la place une action et fournit des données avec lesquelles effectuer l'action.

Lorsque vous appelez `startActivity()` ou `startActivityForResult()` et que vous lui transmettez une intention implicite, le système résout l'intention en une application qui peut gérer l'intention et démarre son correspondant Activity. Si plusieurs applications peuvent gérer l'intention, le système présente à l'utilisateur une boîte de dialogue pour choisir l'application à utiliser.

Cette page décrit plusieurs intentions implicites que vous pouvez utiliser pour effectuer des actions courantes, organisées par type d'application qui gère l'intention. Chaque

[3] https://devstory.net/10433/android-sqlite-database

[4] https://openclassrooms.com/courses/apprenez-a-programmer-en-java/les-collections-d-objets
http://www.jmdoudoux.fr/java/dej/chap-collections.html

[6] https://developer.android.com/guide/components/intents-common

section montre également comment vous pouvez créer un filtre d'intention pour annoncer la capacité de votre application à effectuer la même action.

```
Intent intent = new Intent(CategoryActivity.this, QuizActivity.class);
startActivity(intent);
```

5.2 PARCELABLE[7]

Dans Android, si nous voulons passer des objets aux activités sans avoir besoin de communiquer avec différents composants de votre application, des composants système ou d'autres applications installées sur le téléphone, il faut utiliser Parcelable. Il vous aidera à transmettre des données entre ces composants.

Voici une implémentation typique :

```
public class MyParcelable implements Parcelable {
    // Vous pouvez inclure des types de données de colis
    private int mData;
    chaîne privée mName ;

    // Nous pouvons également inclure des objets Parcelable enfants. Supposons que
MySubParcel est un tel Parcelable :
    private MySubParcelable mInfo ;

    // C'est ici que vous écrivez les valeurs que vous souhaitez enregistrer dans
le `Parcel`.
    // La classe `Parcel` a des méthodes définies pour vous aider à enregistrer
toutes vos valeurs.
    // Notez qu'il n'y a que des méthodes définies pour des valeurs simples, des
listes et d'autres objets Parcelable.
    // Vous devrez peut-être créer plusieurs classes Parcelable pour envoyer les
données souhaitées.
    @Override
    public void writeToParcel(Parcel out, int flags) {
        out.writeInt(mData);
        out.writeString(mName);
        out.writeParcelable(mInfo, flags);
    }

    // En utilisant la variable `in`, nous pouvons récupérer les valeurs
    // que nous avons écrites à l'origine dans la `Parcel`. Ce constructeur est
généralement
    // privé afin que seul le champ `CREATOR` puisse y accéder.
    private MyParcelable(Parcel in) {
        mData = in.readInt();
        mName = in.readString();
        mInfo = in.readParcelable(MySubParcelable.class.getClassLoader());
    }

    public MyParcelable() {
        // Actions normales effectuées par la classe, puisqu'il s'agit toujours
d'un objet normal !
    }

    // Dans la grande majorité des cas, vous pouvez simplement renvoyer 0 pour
cela.
    // Il y a des cas où vous devez utiliser la constante
`CONTENTS_FILE_DESCRIPTOR`
    // Mais cela sort du cadre de ce tutoriel
```

[7] https://medium.com/@hossamrakha0/what-is-parcelable-and-why-we-need-it-574d9e376741

```
    @Override
    public int describeContents() {
        return 0;
    }

    // Après avoir implémenté l'interface `Parcelable`, nous devons créer la
    constante // `Parcelable.Creator<MyParcelable> CREATOR` pour notre classe ;
    // Remarquez comment notre classe est spécifiée comme type.
    public static final Parcelable.Creator<MyParcelable> CREATOR
            = new Parcelable.Creator<MyParcelable>() {

        // Cela appelle simplement notre nouveau constructeur (typiquement privé)
  et
        // transmet le `Parcel` non trié, puis renvoie le nouvel objet !
        @Override
        public MyParcelable createFromParcel(Parcel in) {
            return new MyParcelable(in);
        }

        // Nous avons juste besoin de copier ceci et de changer le type pour qu'il
  corresponde à notre classe.
        @Override
        public MyParcelable[] newArray(int size) {
            return new MyParcelable[size] ;
        }
    } ;
}
```

Le controller devant être passer entre les activités, doit donc implémenter `Parcelable` :

```
public class Controller implements Parcelable {

    private QuizGame laGame;


    public Controller() {
       this.laGame = null;

    }


    protected Controller(Parcel in) {
        laGame = in.readParcelable(QuizGame.class.getClassLoader());
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeParcelable(laGame, flags);
    }

    @Override
    public int describeContents() {
        return 0;
    }

    public static final Creator<Controller> CREATOR = new Creator<Controller>() {
        @Override
        public Controller createFromParcel(Parcel in) {
            return new Controller(in);
        }
```

```
        @Override
        public Controller[] newArray(int size) {
            return new Controller[size];
        }
    };


    public QuizGame getLaGame() {
        return laGame;
    }
    public void setLaGame(QuizGame laGame) {
        this.laGame = laGame;
    }

}
```

Nous pouvons donc par la suite l'ajouter à un Intent pour le passer entre les activités :

```
        Intent intent = new Intent(CategoryActivity.this, QuizActivity.class);

        monController.setLaGame(new QuizGame(player, quizQuestions));

        intent.putExtra("monController", monController); // la clé, la valeur
        startActivity(intent);
```

6 APK - DEPLOIEMENT ET SIGNATURE[8]

6.1 LE PACKAGE APK RUNNABLE

Android Package (ou APK, pour Android Package Kit) est un format de fichiers pour le système d'exploitation Android. Un APK1 (ex. : « nomfich.apk ») est une collection de fichiers pour Android (« package ») assemblés et compressés sous la forme d'une archive au format ZIP2. L'ensemble constitue un « paquet ».

6.1.1 GENERATION D'UN APK FILE SIGNE ASSISTEE PAR ANDROID STUDIO

Afin de générer un fichier .apk signé, il faut cliquer sur Build > Generate Signed Bundle or APK et sélectionner "APK".

Par la suite, il faut sélectionner le fichier de clé permettant la signature et entrer les informations.

[8] https://fr.wikipedia.org/wiki/APK_(format_de_fichier)

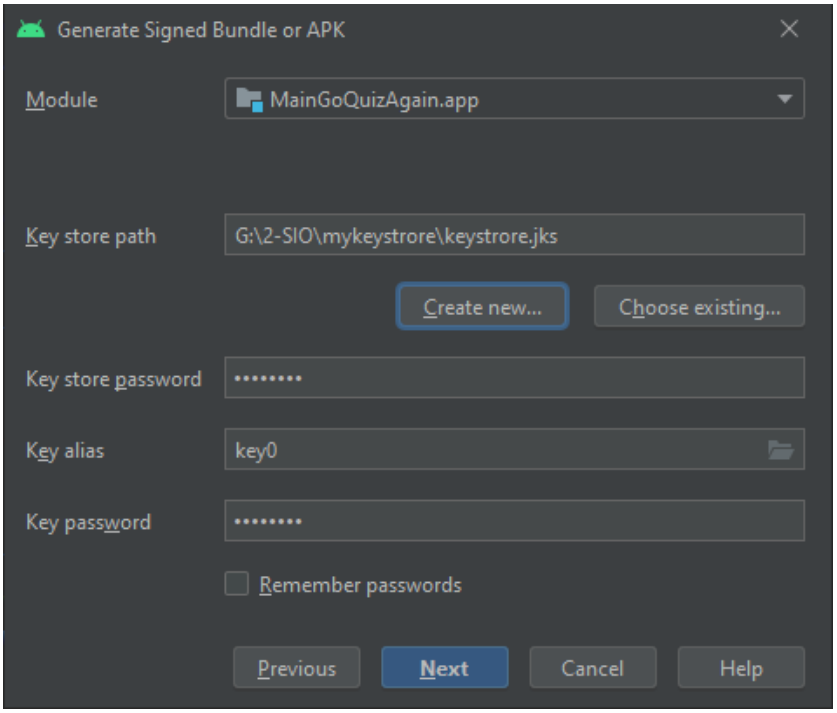

Figure 4 : signature du fichier APK

Pour finir, il faut sélectionner "release" afin de générer le fichier apk signé.

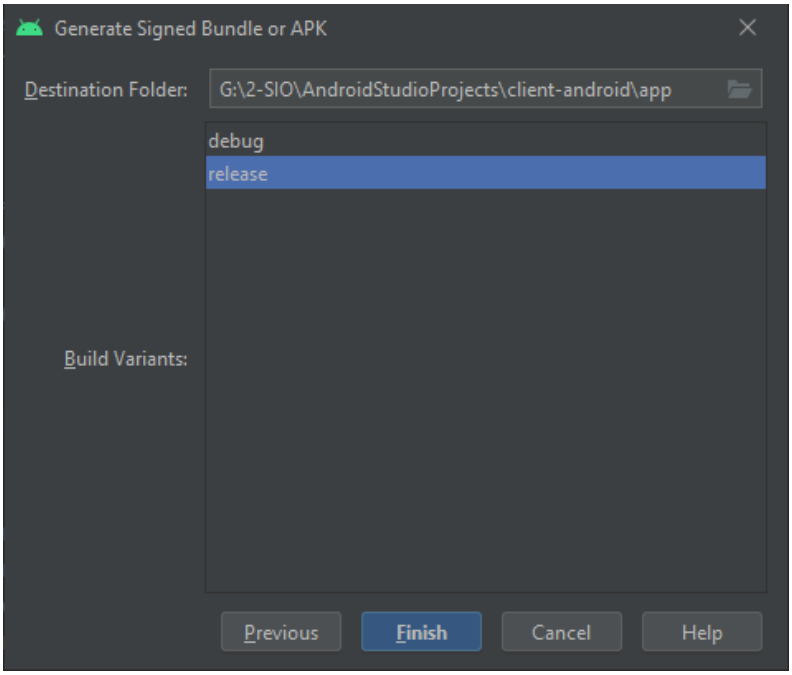

Figure 5 : génération du fichier APK

7 TABLE DES ILLUSTRATIONS