Université Paris Cité

# Conception détaillée
# PROJET L2P1
# Application ISA

| | |
|---|---|
| Références du document: | |
| Version du document: | 0.0.1 |
| Date du document: | 09/04/24 |

| | |
|---|---|
| Validé par : | |
| Validé le : | |
| Soumis le : | 23/04/24 |
| Confidentialité : | UFR Maths-Info / Paris-Cité |

| | | | |
|---|---|---|---|
| Maîtrise d'ouvrage : | Christophe Gnaho | Chef de projet : | Aymeric Letaconnoux |
| Date/Signature : | | Date/Signature : | *Aymeric Letaconnoux* |

# Table des matières

# 1 Introduction

La conception détaillée a pour rôle d'élargir la conception générale en présentant l'ensemble des classes et méthodes qui sont nécessaires au bon fonctionnement de l'application et du module d'exploitation, dont les besoins sont précisés dans le cahier des recettes.

Sa réalisation offrira aux développeurs la possibilité d'implémenter l'application de manière plus aisée, en se limitant essentiellement à la traduction de la conception détaillée.

Le document s'articulera à travers 3 grands axes:

- La présentation des technologies et outils.
- La présentation des packages.
- La présentation de l'ensemble des classes et méthodes (*dont les schémas de parcours et les diagrammes UML*).

## 1.1 Documents de référence

Afin de réaliser la conception détaillée, l'ensemble des fonctionnalités de l'application et du module d'exploitation devront être connus, ces informations sont disponibles dans le cahier des charges[1]. Ces fonctionnalités seront de plus explicitées à l'aide de scénarios présents dans le cahier des recettes[2] qui a pour but de définir les besoins fonctionnels du produit.

## 1.2 Guide de lecture

Afin de consulter le document efficacement, l'ensemble du document devra être lu. En fonction de ces besoins, le maître d'oeuvre pourra consulter la section *Technologies et outils (voir section)* afin de connaître les langages et outils à utiliser par exemple. Il pourra ensuite consulter l'ensemble des sections comme *Les activités (voir section)* pour connaître de manière précise les différentes activités à implémenter. La section *Détails des parcours (voir section)* pourra également l'aider pour la compréhension générale des différentes transitions entre les différentes activités.

## 1.3 Maîtrise d'oeuvre

Le projet sera réalisé par Ramzy Chibani, Vibol Arnaud Sok, Kevin Chen et Aymeric Letaconnoux.

Tous les membres du projet participeront aux tâches de conception et de développement. Pour les tâches de développement, chaque membre travaillera soit de manière individuelle en fonction de la tâche qui lui est attribuée ou en groupe si la tâche demande beaucoup de main d'œuvre.

## 1.4 Maîtrise d'ouvrage

Le projet est commandé par David Janiszek, directeur de l'UFR Mathématiques et Informatique de l'université Paris-Cité Campus Saint-Germain-des-Prés et Christophe Gnaho, encadrant du projet et professeur à l'université Paris-Cité.

# 2 Technologies et outils

Les outils de développement afin de réaliser l'application et le module d'exploitation seront les suivants :

- L'IDE Android Studio sera utilisé durant l'entièreté du développement de l'application.
- Le contrôleur de version SVN sera utilisé durant l'entièreté du projet afin de gérer efficacement l'ajout de nouveau code au fur et à mesure.
- Un logiciel de traitement de texte comme Sublime Text pour écrire les scripts PHP.

Quant aux technologies, les suivantes seront utilisées:

- Les langages de programmation/balisages pour l'application: Java et XML.
- Les langages de programmation/balisages pour le module d'exploitation des données : HTML, CSS, JavaScript et PHP.

De plus, le module d'exploitation et tous les scripts dont il dépend seront stockés sur un serveur web hébergé chez OVH.

# 3 Concepts de base

Pour éviter certaines redondances, certains concepts seront expliqués dans cette section afin de ne pas surcharger le document avec des explications qui seront réitérées dans tout le document :

- Une classe *Singleton*[3] est une classe où le constructeur est déclaré privé pour interdire la création de nouvelles instances. Une instance de la classe est créée de manière unique dans la classe. Cela permettra l'existence d'un seul objet de cette classe.
- Une *View* (vue en Français) peut être résumée à un "widget" visible par l'utilisateur sur un écran. On peut par exemple citer les boutons, les champs de saisie ou les zones de texte.
- Les classes dites *Activités* sont des classes représentant un "écran" de l'application (Par exemple les classes *MainActivity* ou *ChoixModeActivity* représentent chacune un écran précis).
- Les méthodes dites *Listener* sont des méthodes permettant d'écouter certains événements, lorsque l'événement qu'il écoute est déclenché, alors le code associé s'exécute. On peut prendre comme exemple la méthode *OnClickListener* sur un bouton afin de pouvoir exécuter un code précis lorsque ce bouton est appuyé par l'utilisateur.
- La méthode *findViewById* est une méthode permettant de retrouver une vue à l'aide de son identifiant à l'intérieur d'un programme. Par exemple *findViewById(R.id.boutonTest)* où "boutonTest" est l'ID d'un bouton. Les développeurs pourront ensuite intégrer ce bouton dans leurs programmes.
- La classe *Intent* associée à la méthode *startActivity* permet de lancer une nouvelle activité, par exemple pour transiter entre deux écrans.
- La méthode *onClick* qui sera généralement appelée grâce à une méthode *Listener*. Son rôle est de lancer le code présent à l'intérieur de cette méthode lors de l'exécution de la méthode *Listener*. Cette méthode prend en argument une variable de type *View*.

# 4 Structure du logiciel

La structure du logiciel sera organisée en 2 parties:

1. La structure de l'application. (*sous forme de package java*).
2. La structure du module d'exploitation des données.

## 4.1 Structure de l'application

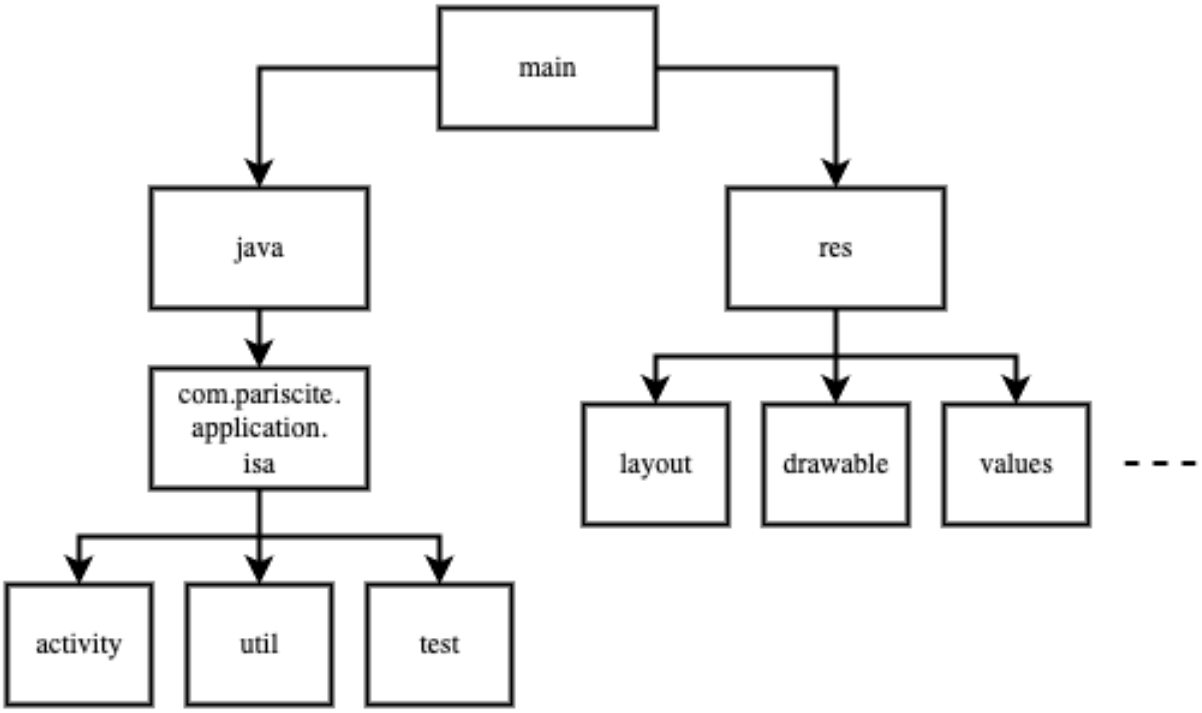


Figure 1: *Structure de l'application mobile*

L'ensemble des fils, petits fils, etc. du noeud *java* représente l'ensemble du code source lié aux fonctionnalités de l'application :

- Le package *activity* contient l'ensemble des classes dites *Activity*.
- Le package *util* contient l'ensemble des classes *Manager* et utilitaires dont les classes *Singleton*.
- Le package *test* contient l'ensemble des classes permettant de réaliser des tests graphiques. (afficher les valeurs par exemple).

L'ensemble des fils, petits fils, etc. du noeud *res* représente l'ensemble du code source lié au visuel de l'application (ce que voit l'utilisateur à l'écran) :

- Le package *layout* permet de gérer l'agencement des différents éléments des classes *Activity*.
- Le package *drawable* permet de gérer les éléments graphiques qui seront vus à l'écran.
- Le package *values* permet de stocker dans des variables les différents textes, couleurs, etc. vus à l'écran.

## 4.2 Structure du module d'exploitation

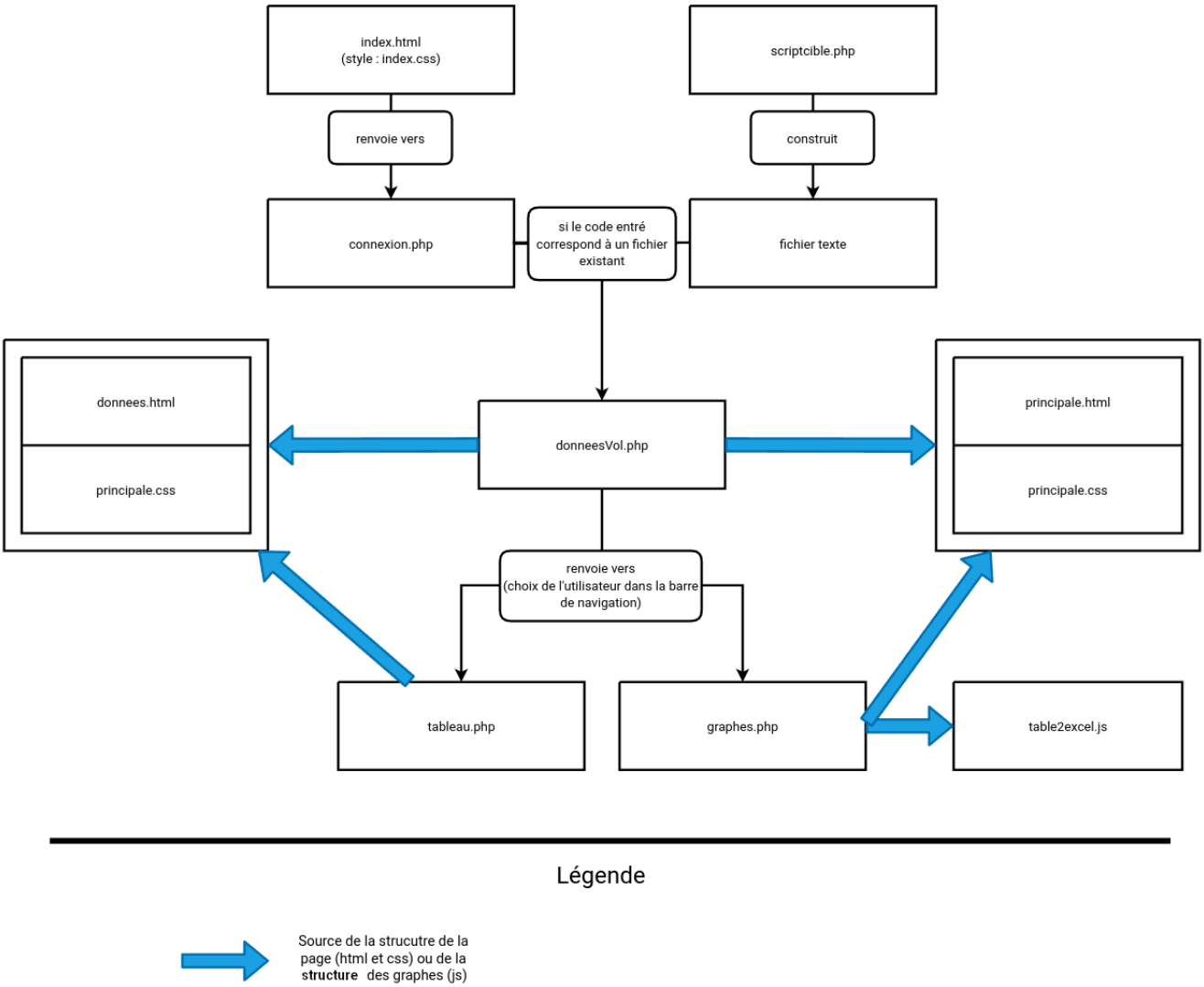


Figure 2: Structure du module d'exploitation

## 4.3 Détails des parcours

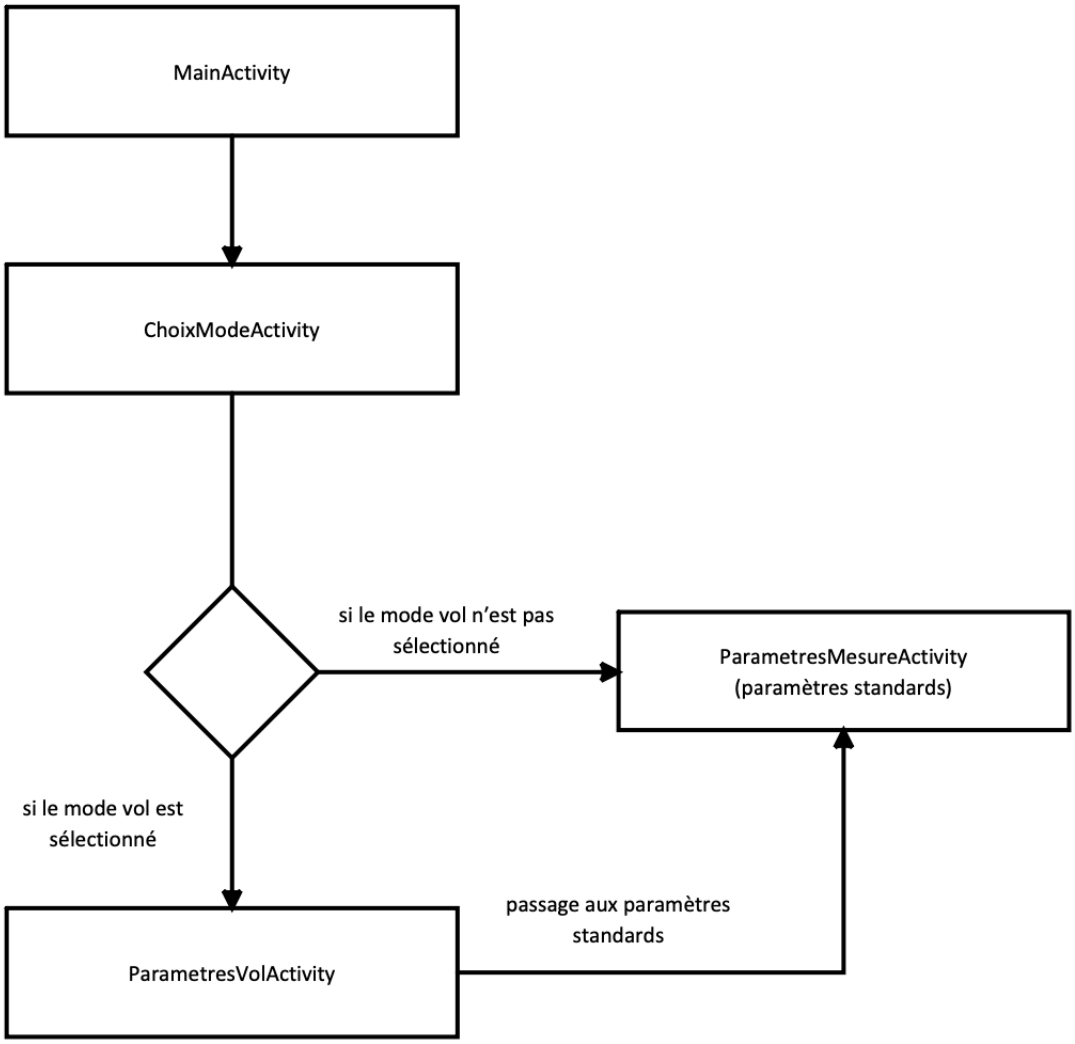


Figure 3: *Schéma du parcours des activités 1/2*

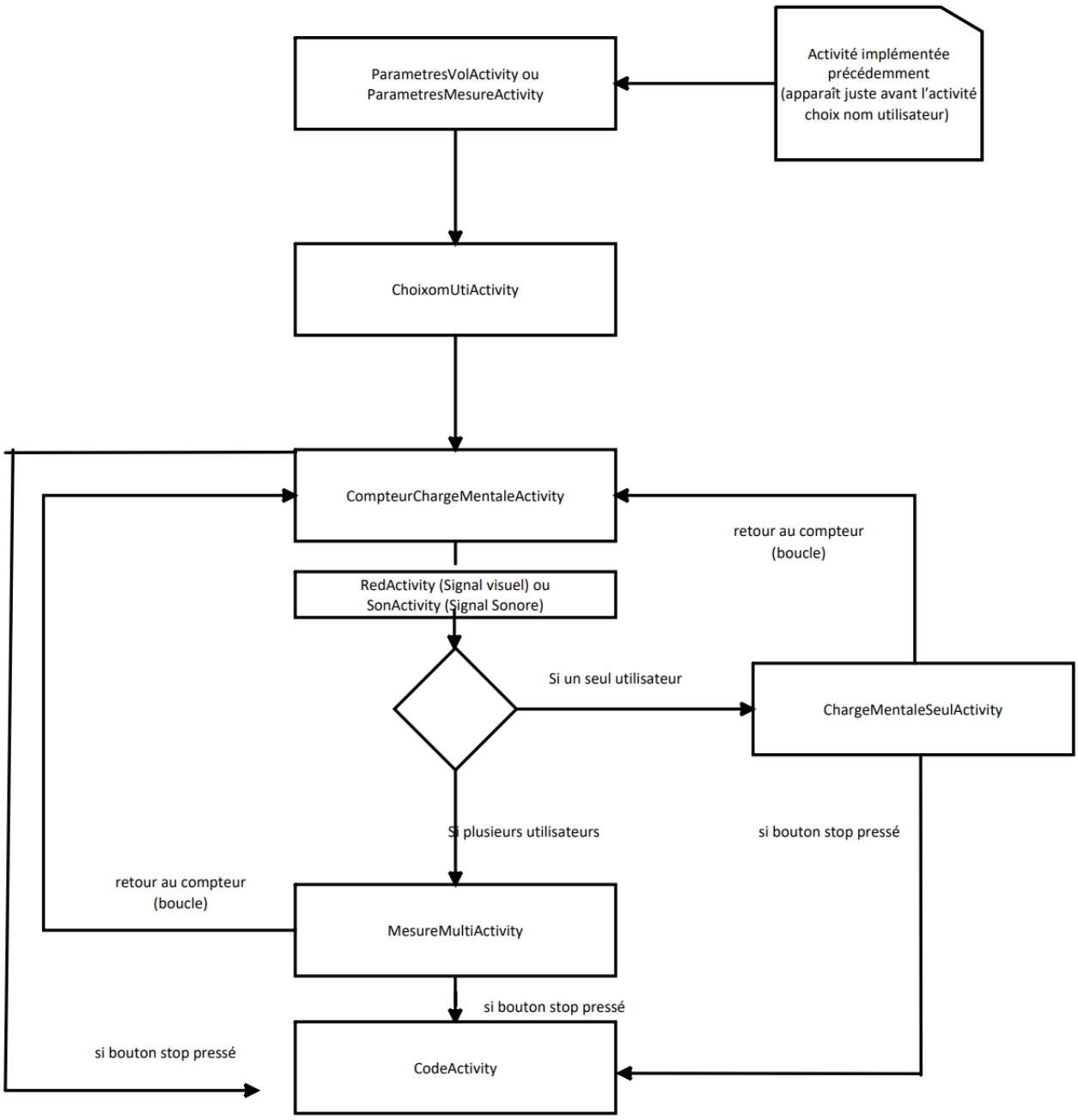


Figure 4: *Schéma du parcours des activités 2/2*

# 5 Classe Singleton et utilitaires

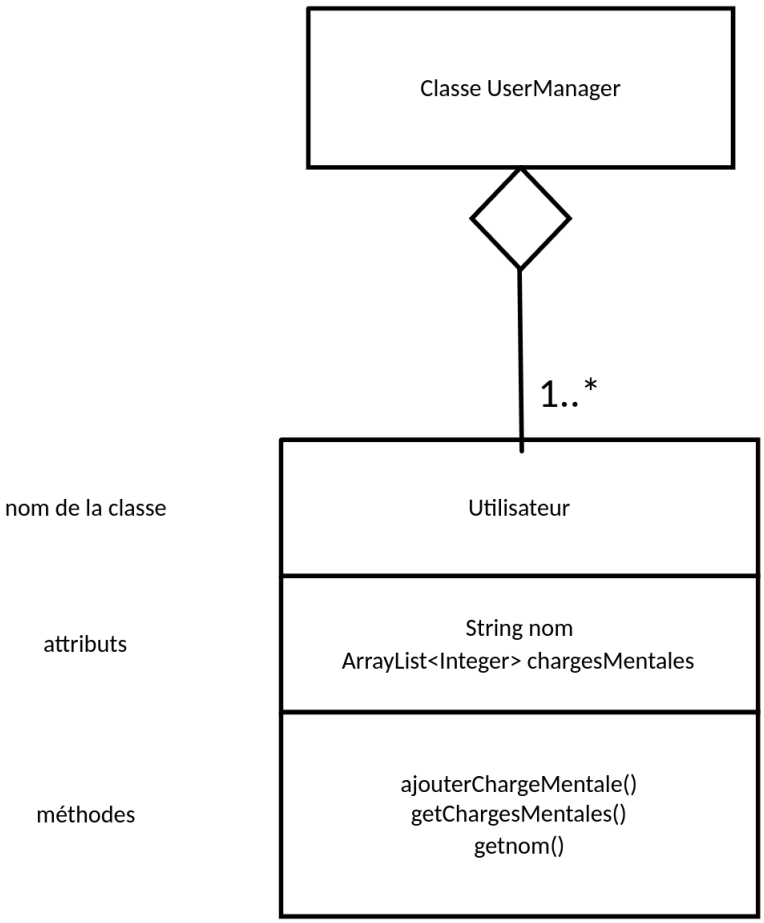


Figure 5: *Diagramme de classe de la classe UserManager*

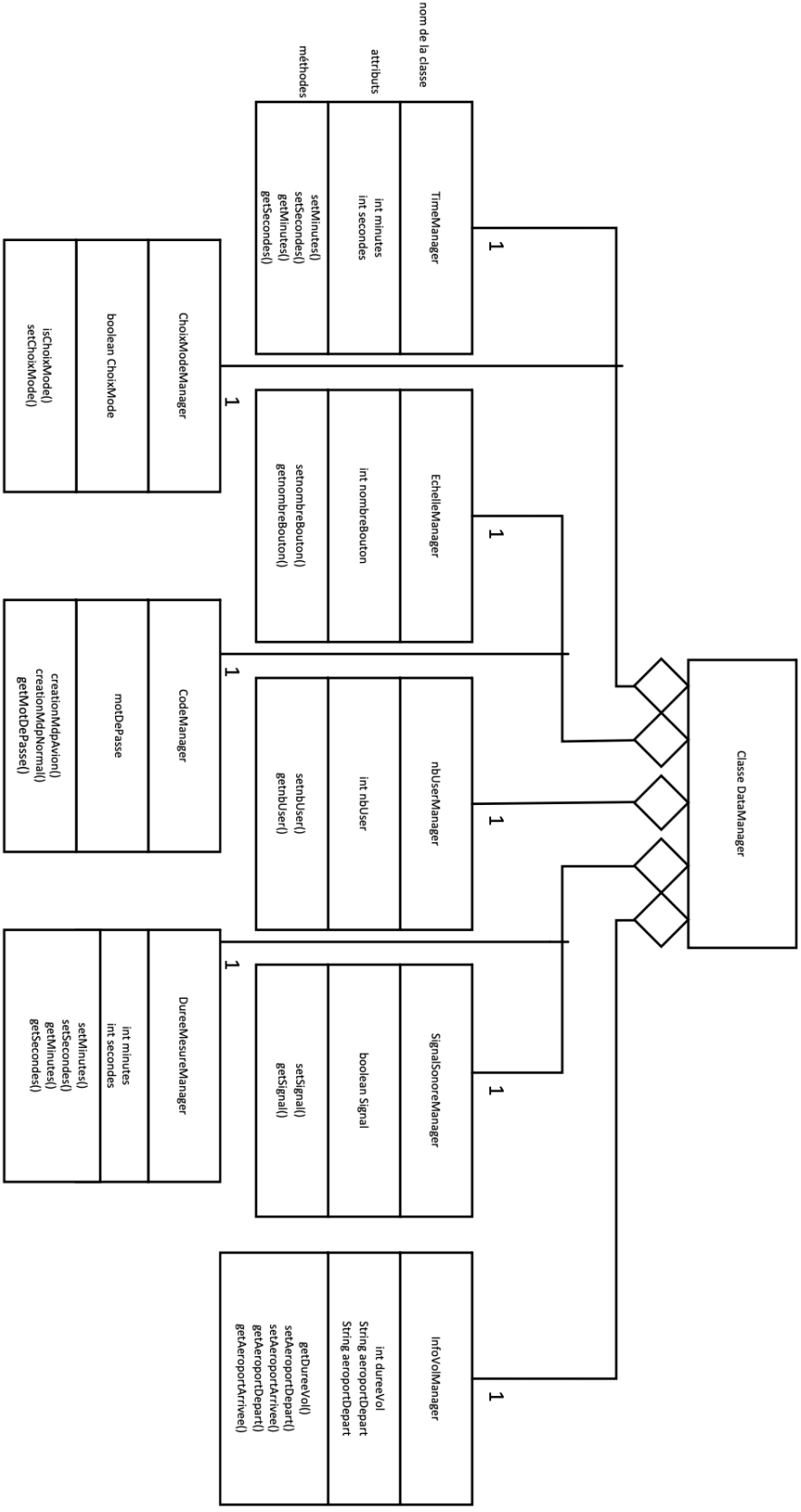


Figure 6: *Diagramme de classe de la classe DataManager*

## 5.1 DataManager

**Description :** La classe *DataManager* est une classe singleton qui permet de gérer et stocker le choix du mode d'expérimentation, les paramètres liés aux informations du vol et les paramètres liés à l'expérimentation (choix du nombre d'utilisateurs, choix de l'échelle de mesure, choix du mode d'expérimentation et intervalle de mesure.

**Objectif :** L'objectif de cette classe est de stocker de façon définitive jusqu'à la fermeture de l'application, toutes les valeurs des différents paramétrages (intervalles de mesure, nombre d'utilisateurs, etc.). Ces informations seront stockées dans un premier temps localement dans différentes activités avant de les stocker dans la classe *DataManager*.

**Attributs**

- L'attribut echelleFinal final de type *EchelleManager*.
- L'attribut nbUserFinal de type *NbUserManager*.
- L'attribut modeFinal de type *SignalSonoreManager*.
- L'attribut infos de type *InfoVolManager*.
- L'attribut intervalleFinal de type *TimeManager*.
- L'attribut choixModeFinal de type *ChoixModeManager*.
- L'attribut codeFinal de type *CodeManager*.
- L'attribut dureeFinal de type *DureeMesureManager*.
- L'attribut instance[1] de type *DataManager*.

**Détails des classes "Manager":**

Voici un court résumé des classes dites "Manager" (citées plus haut):

[1]Cette instance est l'unique instance de la classe DataManager. De ce fait le constructeur est déclaré privée

- *EchelleManager* : La classe contient un attribut de type *int* représentant l'échelle d'expérimentation, un constructeur par défaut et des méthodes assesseurs et mutateurs.

- *NbUserManager* : La classe contient un attribut de type *int* représentant le nombre d'utilisateurs, un constructeur par défaut et des méthodes assesseurs et mutateurs.

- *SignalSonoreManager*; La classe contient un attribut de type *boolean* représentant le choix du mode d'expérimentation, un constructeur par défaut et des méthodes assesseurs et mutateurs.

- *InfoVolManager*: La classe contient un attribut de type *int* et deux attributs de type *String* représentant la durée du vol, l'aéroport de départ et d'arrivée. Un constructeur par défaut et des méthodes assesseurs et mutateurs sont également présent.

- *TimeManager*: La classe contient deux attribut de type *int* représentant le nombre de minutes et de secondes de l'intervalle de mesure, un constructeur par défaut et des méthodes assesseurs et mutateurs.

- *CodeManager*: La classe contient un attribut de type *String* afin de stocker le code d'accès qui sera généré et un constructeur par défaut. Cette classe contient également deux méthodes qui permettent de générer selon le mode d'expérimentation, deux codes d'accès fabriqués à partir des informations de l'expérience. Une méthode assesseur est également présente.

- *DureeMesureManager*: La classe contient deux attributs de type *int* représentant le nombre de minutes et de secondes composant la durée d'une mesure, un constructeur par défaut et des méthodes assesseurs et mutateurs.

**Méthodes:**

La classe *DataManager* possède une méthode *getInstance()* en plus de ses méthodes assesseurs et mutateurs. Elle permet de récupérer l'instance unique de la classe dont l'algorithme est :

```
getInstance()

debut
     /* Entree : Aucun argument en entree */
    /*Sortie : Retourne un element de type DataManager */

    cree une instance de DataManager;
    si ( instance == nul) {
        cree une instance de DataManager;
    }
    retourner Instance:
fin
```

- La classe comporte les méthodes assesseurs et mutateurs pour chaque attribut afin de pouvoir stocker de manière définitive les informations souhaitées grâce à la notion de *Singleton*.

## 5.2 UserManager

**Description :** La classe *UserManager* est une classe singleton qui permet de gérer et stocker la liste des utilisateurs ainsi que la liste des noms d'utilisateurs.

**Objectif:** L'objectif de cette classe est de stocker de façon définitive jusqu'à la fermeture de l'application, la liste d'utilisateurs (chaque utilisateur sera associé à une liste de charge mentale) ainsi que la liste des noms de tous les utilisateurs. Ces informations seront stockées dans un premier temps localement dans différentes activités avant de les stocker dans la classe *UserManager*.

**Attributs**:

- L'attribut utilisateurs final de type *ArrayList (d'objet de type Utilisateur).*
- L'attribut nomUtilisateur de type*ArrayList (d'objet de type String).*
- L'attribut instance[2] de type *DataManager.*

**Méthodes**:

La classe *UserManager* possède une méthode *getInstance()* en plus de ses méthodes accesseurs et mutateurs. Elle permet de récupérer l'instance unique de la classe dont l'algorithme est :

[2]Cette instance est l'unique instance de la classe UserManager. De ce fait le constructeur est déclaré privée

```
getInstance()

debut
     /* Entree : Aucun argument en entree */
    /*Sortie : Retourne un element de type DataManager */

    cree une instance de UserManager;
    si ( instance == nul) {
        cree une instance de UserManager;
    }
    retourner Instance:
fin
```

Les méthodes mutateurs de la classe *UserManager* contrairement à celles de la classe *DataManager* ajoute un élément dans ses listes respectives.

Voici un court résumé des classes dites "Manager" qu'utilise UserManager (citées plus haut):

- Classe utilisateur:

  **Description :** La classe *Utilisateur* est une classe permettant de stocker le nom et les charges mentales d'un utilisateur à l'aide d'une liste.

  **Objectif :** Son objectif est de récupérer des informations sur un utilisateur (tout ceux qui participeront à l'expérimentation), ces informations seront plus tard stockées dans un tableau d'utilisateurs. grâce à la classe Singleton *UserManager*.

  **Attributs:**

  - L'attribut nom de type *String*.
  - L'attribut chargesMentales de type *ArrayList (Integer)*.

  **Les méthodes :** Les méthodes sont les méthodes standards assesseurs et mutateurs permettant de peupler la liste de charges mentales et de retourner cette même liste et le nom de l'utilisateur. La méthode mutateur *ajouterChargeMentale* permet d'ajouter des unités de charge mentale dans la liste.

## 5.3 SauvergardeEtEnvoi

**Description :** Cette classe contient le code permettant de réaliser une requête HTTP de méthode POST vers le serveur web.

**Objectif :** Son objectif est d'envoyer les données de l'expérimentation, c'est-à-dire les charges mentales, les noms des utilisateurs et les informations sur le vol si le mode vol est activé, vers le serveur web, qui les enregistrera dans un fichier. Ce fichier sera plus tard utilisé pour créer les tableaux et les graphes affichés sur le site web.

**Attributs :**

- L'attribut donneesAEnvoyer de type *String*.
- L'attribut urlChaine de type *String*.
- L'attribut context de type *Context*.
- L'attribut donnees de type *DataManager*.
- L'attribut donneesUtilisateur de type *UserManager*.

  L'attribut context permet de récupérer le contexte de l'activité dans laquelle on instancie la classe SauvegardeEtEnvoi, cela sert à renvoyer des messages d'erreurs dans l'activité depuis la classe SauvegardeEtEnvoi, au cas où la requête n'aurait pas aboutie.

**Les méthodes :**

- La méthode envoyerRequettePost : Elle transforme les données à envoyer contenues dans l'attribut donneesAEnvoyer en objet JSON (JavaScript Object Notation), puis envoie cet objet via une requête POST vers le serveur web, via une cible, ici un script PHP interprétant les données reçues.

# 6 Les activités

## 6.1 MainActivity

**Description :**

La classe *MainActivity* représente le point d'entrée principal de l'application. La classe est essentiellement constituée d'un seul bouton où s'applique dessus plusieurs méthodes *Listener*.

**Objectif :**

L'objectif de cette classe est de permettre à l'utilisateur d'être renvoyé vers la fenêtre suivante qui est la classe *ChoixModeActivity* à l'aide d'un bouton Démarrer.

**Attributs:**

- L'attribut demarrer qui est de type *Button*.

**Méthodes**

Cette classe est composée des méthodes suivants:

- La méthode *activiteSuivante* qui permet à l'utilisateur lorsque le bouton *demarrer* est appuyé, de passer à l'activité suivante (*ChoixModeActivity*). Elle ne prend aucun argument et le type de retour est *void*.
- La méthode *onRequestPermissionsResult* qui permet à l'utilisateur d'accepter ou de refuser l'autorisation de reconnaissance vocale. Elle appelle de plus la méthode *makeText* de la classe *Toast* afin d'afficher un pop-up qui indique si l'autorisation a été acceptée ou non.

## 6.2 ChoixModeActivity

**Description :** La classe *ChoixModeActivity* représente la fenêtre permettant d'activer ou non le mode Vol. La classe est constituée d'une zone de texte et de deux boutons pour activer et désactiver le mode.

**Objectif :** Son objectif est de permettre à l'utilisateur de choisir le type de test qu'il veut lancer en proposant deux types de paramétrage : un où l'expérimentation comporte 3 utilisateurs au maximum (le mode Vol) et une autre avec une limite de 10 utilisateurs. Si l'utilisateur choisit le mode Vol, il est redirigé vers le menu des informations liées au vol avant le menu de paramétrage de l'expérimentation, sinon il est directement redirigé vers le menu de paramétrage de l'expérimentation.

**Attributs:**

- Un bouton activer de type *Button*.
- Un bouton desactiver de type *Button*.
- une variable choixModeCourant de type *ChoixModeManager*.
- une variable donnees de type *DataManager*.

**Méthodes:**

Cette classe est composée des méthodes suivants:

- La méthode *activiteModeVol* qui permet à l'utilisateur lorsque le bouton *activer* est appuyé, de passer à l'activité suivante (*ParametresVolActivity*). Elle ne prend aucun argument et le type de retour est *void*. Elle permet également le choix final dans l'unique instance de *DataManager*.
- La méthode *activiteParaMesure* qui permet à l'utilisateur lorsque le bouton *desactiver* est appuyé, de passer à l'activité suivante (*ParametresMesureActivity*). Elle ne prend aucun argument et le type de retour est *void*. Elle permet également le choix final dans l'unique instance de *DataManager*.

## 6.3 ParamètresVolActivity

**Description :** La classe *ParametresVolActivity* représente la fenêtre qui permet à l'utilisateur de rentrer les informations liées au vol si le mode Vol a été activé.

**Objectif :** Cette activité permet de récupérer toutes les informations liées au mode Vol, puis de stocker les différentes informations dans la classe *DataManager*.

**Attributs:**

- Une variable donneesFinales de type *DataManager*.
- Une variable donnees de type *InfoVolManager*.
- Deux roues dureeHeure et dureeMin de type *NumberPickers*.
- Un bouton boutonSuivant de type *Button*.
- Un bouton boutonPrecedant de type *Button*.
- Une variable listeAerodrome qui est un tableau de *String*.
- Une variable listeAerodrome qui est un tableau de *String*.
- Une variable arrayListAerodromes qui est un *ArrayList* de *String*.
- Une variable adapter qui est un *ArrayAdapter* de *String*.
- Une variable afficherAerodromes qui est de type *TextInputLayout*.
- Une variable aerodromes_correspondants qui est de type *AutoCompleteTextView*.
- Une variable afficherAerodromesArrivee qui est de type *TextInputLayout*.
- Une variable aerodromes_correspondantsArrivee qui est de type *AutoCompleteTextView*.
- Un bouton boutonArrivee de type *MaterialButton*.
- Un bouton boutonDepart de type *MaterialButton*.

*La liste des aéroports est accessible grâce à un fichier XML comportant un peu plus de 320000 éléments référençant les aérodromes du monde entier.*

**Méthodes:**

- Une méthode *dureeHeures* qui permet de mettre à jour le *NumberPickers dureeHeure* à chaque interaction de l'utilisateur. Elle ne prend aucun argument et le type de retour est *void*.
- Une méthode *dureeMinutes* qui permet de mettre à jour le *NumberPickers dureeMin* à chaque interaction de l'utilisateur. Elle ne prend aucun argument et le type de retour est *void*.
- Une méthode *activiteSuivante* qui permet que lorsque le bouton *boutonSuivant* est appuyé, de passer à l'activité *ParametresMesureActivity*. Elle ne prend aucun argument et le type de retour est *void*.
- Une méthode *activitePrecedante* qui permet que lorsque le bouton *boutonPrecedant* est appuyé, de retourner à l'activité précédente *(ParametresMesureActivity)*. Elle ne prend aucun argument et le type de retour est *void*.
- Une méthode *boutonValiderDepart* qui permet de valider le choix de l'aérodrome de départ en appuyant sur le bouton *boutonDepart*. Elle ne prend aucun argument et le type de retour est *void*.
- Une méthode *boutonValiderArrivee* qui permet de valider le choix de l'aérodrome d'arrivée en appuyant sur le bouton *boutonArrivee*. Elle ne prend aucun argument et le type de retour est *void*.

## 6.4 ParamètreMesureActivity

**Description :** La classe *ParametresMesuresActivity* représente la fenêtre permettant à l'utilisateur de choisir en fonction de ses besoins la valeur des différents paramétrages.

**Objectif :** L'objectif est de permettre à l'utilisateur de saisir ses préférences pour l'expérimentation afin de prélever les mesures souhaitées dans les bonnes conditions. Ses préférences devront ensuite être stockées dans la classe *DataManager*.

**Attributs:**

- une variable donnees de type *DataManager*.
- une variable intervalleCourant de type *TimeManager*.
- une variable echelleCourante de type *EchelleManager*.
- une variable nbUtiCourant de type *NbUserManager*.
- une variable signalCourant de type *SignalSonoreManager*.
- une variable choixModeCourant de type *ChoixModeManager*.
- une variable dureeMesureCourant de type *DureeMesureManager*.
- deux roues minutes et secondes de type *NumberPicker*.
- deux roues minutesDureeMesure et secondesDureeMesure de type *NumberPicker*.
- Un bouton boutonSuivant de type *Button*.
- Un bouton boutonPrecedant de type *Button*.
- 2 boutons imagés bouton_incrementer_nbUti et bouton_decrementer_nbUti de type *ImageButton*.
- Une zone de texte affichageNbUti de type *TextView*.

- 2 boutons imagés bouton_incrementer_echelle et bouton_decrementer_echelle de type *ImageButton*.
- Une zone de texte affichageEchelle de type *TextView*.
- un interrupteur SwitchSignal de type *Switch*.

**Méthodes:**

- Une méthode *setMinutes* qui permet à chaque interaction avec le *NumberPicker minutes* de mettre à jour la valeur des minutes pour l'intervalle de mesure. Elle ne prend aucun argument et a pour type de retour *void*.
- Une méthode *setSecondes* qui permet à chaque interaction avec le *NumberPicker secondes* de mettre à jour la valeur des secondes pour l'intervalle de mesure. Elle ne prend aucun argument et a pour type de retour *void*.
- Une méthode *setMinutesDureeMesure* qui permet à chaque interaction avec le *NumberPicker minutesDureeMesure* de mettre à jour la valeur des minutes pour la durée d'une mesure. Elle ne prend aucun argument et a pour type de retour *void*.
- Une méthode *setSecondesDureeMesure* qui permet à chaque interaction avec le *NumberPicker secondesDureeMesure* de mettre à jour la valeur des secondes pour la durée d'une mesure. Elle ne prend aucun argument et a pour type de retour *void*.
- Une méthode *incrementEchelle* qui permet d'incrémenter l'échelle de mesure à chaque interaction avec le bouton *bouton_incrementer_echelle* et met à jour le *TextView* associé. Lorsque la valeur est à 10, le bouton n'est plus cliquable. Elle ne prend aucun argument et a pour type de retour *void*.
- Une méthode *decrementEchelle* qui permet de décrémenter l'échelle de mesure à chaque interaction avec le bouton *bouton_decrementer_echelle* et met à jour le *TextView* associé. Lorsque la valeur est à 2, le bouton n'est plus cliquable. Elle ne prend aucun argument et a pour type de retour *void*.

- Une méthode *incrementUtilisateur* qui permet d'incrémenter le nombre d'utilisateur à chaque interaction avec le bouton *bouton_incrementer_nbUti* et met à jour le *TextView* associé. Lorsque la valeur est à 10, le bouton n'est plus cliquable si le mode Vol n'est pas activé. Si le mode Vol est activé, la valeur est limitée à 3. Elle ne prend aucun argument et a pour type de retour *void*.

- Une méthode *decrementUtilisateur* qui permet de décrémenter le nombre d'utilisateur à chaque interaction avec le bouton *bouton_decrementer_nbUti* et met à jour le *TextView* associé. Lorsque la valeur est à 1, le bouton n'est plus cliquable. Elle ne prend aucun argument et a pour type de retour *void*.

- Une méthode *setMinMax* qui permet d'initialiser le maximum et minimum des *NumberPicker minutes*, *secondes*, *minutesDureeMesure*. et *secondesDureeMesure*. Elle ne prend aucun argument et a pour type de retour *void*.

- Une méthode *setSwitchSignal* qui permet de changer l'état du *Switch switchSignal* à chaque interaction avec l'utilisateur. Ce paramètre permet de définir le mode de signal (visuel ou sonore) de l'expérimentation. Elle ne prend aucun argument et a pour type de retour *void*.

- Une méthode *nextActivity* qui permet de passer à la prochaine activité (*ChoixNomUtiActivity* lorsque le bouton *boutonSuivant* est appuyé. Elle ne prend aucun argument et a pour type de retour *void*.

- Une méthode *pastActivity* qui permet de passer à l'activité précédente (*ChoixModeActivity* lorsque le bouton *boutonPrecedant* est appuyé. Elle ne prend aucun argument et a pour type de retour *void*.

- Une méthode *majEtatBoutonEchelle* qui permet de désactiver le bouton *bouton_decrementer_echelle* si la valeur de l'échelle est inférieure à 0. Elle prend en argument un *int* (la valeur courante de l'échelle) et a pour type de retour *void*.

- Une méthode *majEtatBoutonUti* qui permet de désactiver le bouton *bouton_decrementer_nbUti* si la valeur de l'échelle est inférieure à 0. Elle prend en argument un *int* (le nombre d'utilisateur courant) et a pour type de retour *void*.

## 6.5 ChoixNomUtiActivity

**Description :** La classe *ChoixNomUtiActivity* représente la fenêtre qui permettra à l'utilisateur de choisir le nom/pseudonyme des différents utilisateurs qui participeront à l'expérimentation. Elle est constitué d'un objet de classe *TextView*, d'un champ de saisi de classe *EditText* et d'un bouton pour valider le choix du nom.

**Objectif:** Son objectif est de permettre à l'utilisateur de pouvoir choisir tous les noms de tous les utilisateurs participant à l'expérimentation. À la fin des ajouts, la classe va également gérer le stockage permenant de ces noms dans une liste de nom dans la classe *UserManager*.

**Attributs:**

- L'attribut donnees de type *UserManager*.
- L'attribut currentUserNumber de type*int*.
- L'attribut totalUser de type *int*.
- L'attribut nextButton de type *Button*.
- L'attribut boutonPrecedant de type *Button*.
- L'attribut userNumberTextView de type *TextView*.
- L'attribut userNameEditText de type *EditText*.
- L'attribut builder de type *AlertDialog.Builder*: Pour pouvoir afficher le pop-up afin de quitter l'expérimentation.

**Méthodes:**

- La méthode *updateCurrentUserNumber* qui permet d'actualiser le *TextView userNumberTextView* (par exemple afficher Utilisateur 1, Utilisateur 2 et ainsi de suite en fonction du nombre d'utilisateurs). Elle ne prend aucun argument et a pour type de retour *void*.

- La méthode *pastUser* qui permet lorsque le bouton *boutonPrecedant* est appuyé, de permettre à l'utilisateur de retourner sur l'utilisateur précédent afin de pouvoir redéfinir le nom associé souhaité.

- La méthode *nextUser* qui permet, lorsque le bouton *nextButton* est appuyé de stocker le nom d'utilisateur rentré par l'utilisateur et de créer l'utilisateur associé à ce nom. Lorsque l'ensemble des utilisateurs possèdent un nom, un pop-up est affiché à l'écran afin de confirmer le début de l'expérimentation. Si l'utilisateur confirme alors l'activité *RedActivity* ou *SonActivity* (ces activités sont liées au mode de signal choisi) est alors affichée. Elle ne prend aucun argument et a pour type de retour *void*.

- La méthode *clearText* qui permet lorsque l'utilisateur appuie sur l'objet de type *EditText*, de faire disparaître le texte qui était initialement présent afin de le signaler qu'il est en mode écriture. Elle ne prend aucun argument et a pour type de retour *void*.

## 6.6 ChargeMentaleSeulActivity

**Description :** Cette activité permet à l'utilisateur de rentrer sa charge mentale, elle ne s'affiche que si l'expérimentation est réalisé par une seule personne.

**Objectif :** Cette fenêtre sert à récupérer la charge mentale de l'utilisateur à l'aide de boutons, ou par la voix à l'aide du bouton prévu pour.

**Attributs :**

- utilisateurListe de type *UserManager* : Sert à récupérer la liste d'utilisateurs.
- utilisateur de type *ArrayList(Utilisateur)* : La liste d'utilisateurs.
- donneesEchelle de type *DataManager* : Sert à récupérer l'échelle de mesure.
- echelle de type *EchelleManager* : L'échelle de mesure.
- echelle-valeur de type *int* : Pour une utilisation locale de la valeur.
- buttonContainer de type *LinearLayout* : Conteneur pour les boutons dynamique, ils seront affichés dedans.
- bouton_stop de type *Button*.
- boutonMicro de type *ImageButton* : pour réaliser la mesure par la voix.
- Reco_Vocale de type *SpeechRecognizer*.
- vocalAction de type *Intent*
- nomUtilisateur de type *TextView* : Afficher le nom de l'utilisateur à l'écran.
- context de type *Contexte*.
- valeurChargeMentale de type *int* : Pour stocker la charge mentale de l'utilisateur courant lors de la mesure par la voix.
- temps de type *long*.
- temps_restant de type *long*.

- mCountDownTimer de type *CountDownTimer*.
- builder de type *AlertDialog.Builder*.
- compteur_en_cours de type *boolean*.

**Les méthodes :**

- La méthode *startTimer* permet de créer un compteur d'une durée déterminée. À la fin du compteur, l'écoute vocale grâce à *reco_Vocale* est désactivée, l'ensemble des charges mentales est ajouté puis l'activité *CompteurChargeMentale* est affichée. Elle ne prend aucun argument et a pour type de retour *void*.
- La méthode *boutonMesure* permet de créer dynamiquement une série de boutons qui permettra à l'utilisateur en appuyant sur l'un des boutons de mesurer la charge mentale. Le nombre de boutons est défini pendant l'activité *ParametresMesuresActivity* par l'utilisateur. Elle ne prend aucun argument et a pour type de retour *void*.
- La méthode *boutonVoix* permet l'activation de la reconnaissance vocale. Lorsque l'utilisateur appuie sur le bouton *boutonMicro* et le maintient, alors la reconnaissance vocale est activée, et désactivée dans le cas contraire. Elle ne prend aucun argument et a pour type de retour *void*.
- La méthode *reconnaisanceVocale* qui permet de traiter les données reçues lors de la reconnaissance vocale grâce à la méthode *onResults*. Dans cette méthode, seul le dernier mot énoncé (qui est la valeur de la charge mentale) est récupéré. Elle est ensuite ajoutée dans le *ArrayList* de charges mentales de l'utilisateur en question. Si la valeur n'est pas reconnue ou non comprise dans l'échelle de mesure, alors l'utilisateur doit la répéter. Elle ne prend aucun argument et a pour type de retour *void*.
- La méthode *boutonStop* qui permet à l'utilisateur de quitter l'expérimentation lorsqu'il appuie sur le bouton *bouton_stop*. Un pop-up sera affiché à l'écran à l'aide de l'attribut *builder*, si l'utilisateur appuie sur "Confirmer" alors il sera redirigé vers *CodeActivity*, sinon il continuera l'expérimentation. Elle ne prend aucun argument et a pour type de retour *void*.

## 6.7 MesureMultiActivity

**Description:**

La classe *MesureMultiActivity* représente la fenêtre qui permettra à l'utilisateur de choisir sa charge mentale et celles des différents utilisateurs qui participeront à l'expérimentation lors d'une expérience à plusieurs utilisateurs. Elle est constituée d'un objet de classe *TextView* afin d'afficher le nom de la personne mesurée, d'une série de bouton dynamique et d'un bouton pour arrêter la mesure. Un bouton est également présent pour permettre à l'utilisateur, si le bouton est maintenu de dire sa charge mentale à l'oral.

**Objectif:**

L'objectif de cette classe est de permettre au propriétaire du téléphone de mesurer la charge mentale dont le nom est affiché sur l'écran. Cette charge mentale sera ensuite stockée dans la liste de charges mentales associées à l'utilisateur mesuré.

**Attributs:**

- L'attribut utilisateurs de type *ArrayList (de Integer)*.
- L'attribut utilisateurActuelIndex de type *int*.
- L'attribut listeUtilisateurs de type *UserManager*.
- L'attribut MAX_ECHELLE_MESURE de type *int (et qui est de plus statique et finale)*.
- L'attribut layoutMesures de type *LinearLayout*.
- L'attribut nomUtilisateurTextView de type *TextView*.
- L'attribut bouton_stop de type *Button*.
- L'attribut boutonMicro de type *ImageButton*.
- Reco_Vocale de type *SpeechRecognizer*.

- contexte de type *Contexte*.
- valeurChargeMentale de type *int* : Pour stocker la charge mentale de l'utilisateur courant lors de la mesure par la voix.
- isListening de type *boolean*: Pour stocker l'état de l'écouteur de son de la reconaissance vocale.
- temps de type *long*.
- temps_restant de type *long*.
- mCountDownTimer de type *CountDownTimer*.
- builder de type *AlertDialog.Builder*.
- compteur_en_cours de type *boolean*.

**Méthodes:**

- La méthode *startTimer* permet de créer un compteur d'une durée déterminée. À la fin du compteur, l'écoute vocale grâce à *reco_Vocale* est désactivée, l'ensemble des charges mentales est ajouté puis l'activité *CompteurChargeMentale* est affichée. Si l'utilisateur ne répond pas dans le délai imparti pour lui-même alors sa charge mentale est mise au maximum, sinon la valeur 0 est mise pour les autres utilisateurs. Elle prend argument une variable de type *int* (l'index) et a pour type de retour *void*.
- La méthode *boutonStop* qui permet à l'utilisateur de quitter l'expérimentation lorsqu'il appuie sur le bouton *bouton_stop*. Un pop-up sera affiché à l'écran à l'aide de l'attribut *builder*, si l'utilisateur appuie sur "Confirmer" alors il sera redirigé vers *CodeActivity*, sinon il continuera l'expérimentation. Elle ne prend aucun argument et a pour type de retour *void*.
- La méthode *boutonVoix* permet l'activation de la reconnaissance vocale. Lorsque l'utilisateur appuie sur le bouton *boutonMicro* et le maintient, alors la reconnaissance vocale est activée, et désactivée dans le cas contraire. Elle ne prend aucun argument et a pour type de retour *void*.

- La méthode *reconnaisanceVocale* qui permet de traiter les données reçues lors de la reconnaissance vocale grâce à la méthode *onResults*. Dans cette méthode, seul le dernier mot énoncé (qui est la valeur de la charge mentale) est récupéré. Elle est ensuite ajoutée dans le *ArrayList* de charges mentales de l'utilisateur en question. Si la valeur n'est pas reconnue ou non comprise dans l'échelle de mesure, alors l'utilisateur doit la répéter. Elle ne prend aucun argument et a pour type de retour *void*.

- La méthode *startListening* permet de passer la valeur de *isListening* à vrai et lance la reconnaissance vocale. Elle est utilisée dans la méthode *boutonVoix*.

- La méthode *stopListening* permet de passer la valeur de *isListening* à faux et arrête la reconnaissance vocale. Elle est utilisée dans la méthode *boutonVoix*.

- La méthode *mesurerChargeMentale* permet à l'utilisateur courant, de mesurer la charge mentale de l'ensemble des utilisateurs en l'appelant récursivement. Tant que *utilisateurActuelIndex* n'est pas égal au nombre d'utilisateurs, alors la méthode est continuée à être appelée en incrémentant *utilisateurActuelIndex*. Après cela, la méthode permet de passer à l'activité *CompteurChargeMentale-Activity*. Cette méthode permet de plus de créer les boutons de mesure de manière dynamique. Elle prend en argument une variable de type *Utilisateur* et a pour type de retour *void*.

## 6.8 CompteurChargeMentaleActivity

**Description :** Cette activité affiche un compteur de temps entre chaque mesure de charge mentale.

**Objectif :** Cette fenêtre permet à l'utilisateur de savoir le temps restant avant la prochaine mesure.

**Attributs :**

- donnees de type *DataManager* : Pour récupérer l'intervalle de temps.
- donnee_temps de type *TimeManager* : L'intervalle de temps.
- compteur de type *TextView* : Pour afficher le compteur.
- mCountDownTimer de type *CountDownTimer* : Le compteur.
- bouton_stop de type *Button* : Bouton stop.
- temps de type *long* : Temps affiché sur le compteur au départ.
- temps_restant de type *long* : Temps restant.
- compteur_en_cours de type *boolean* : Pour vérifier si le compteur est encore en cours.
- builder de type *AlertDialog.Builder*: Pour pouvoir afficher le pop-up afin de quitter l'expérimentation.

**Les méthodes :**

- La méthode *boutonStop* qui permet à l'utilisateur de quitter l'expérimentation lorsqu'il appuie sur le bouton *bouton_stop*. Un pop-up sera affiché à l'écran à l'aide de l'attribut *builder*, si l'utilisateur appuie sur "Confirmer" alors il sera redirigé vers *CodeActivity*, sinon il continuera l'expérimentation. Elle ne prend aucun argument et a pour type de retour *void*.

- La méthode *startTimer* permet de créer un compteur d'une durée déterminée. À la fin du compteur, l'utilisateur est redirigé vers une des deux activités liées au signal (*RedActivity* ou *SonActivity*). Elle ne prend aucun argument et a pour type de retour *void*.

- La méthode *updateCountDownText* qui permet de mettre à jour le *TextView compteur* afin d'indiquer à l'utilisateur le temps qu'il lui reste avant la prochaine mesure. Elle ne prend aucun argument et a pour type de retour *void*.

## 6.9 ParamètresActivity

**Description :** Cette activité permet d'afficher les données des utilisateurs stockées dans la classe *DataManager*. Elle est essentiellement composée d'objets de type *TextView*.

**Objectif :** L'objectif de cette classe est de nous permettre de vérifier que les données contenues dans *DataManager* sont bien enregistrées et cohérentes suite aux différents paramétrages. Elle n'a pas vocation à rester dans l'application finale.

**Attributs:**

Cette classe est composée des attributs suivants:

- Une variable donnes de type *DataManager*.
- Une variable infos de type *InfoVolManager*.
- Une variable nb de type *NbUserManager*.
- Une variable time de type *TimeManager*.
- Une variable echelle de type *EchelleManager*.
- Une variable signal de type *SignalSonoreManager*.
- Une variable modeVol de type *ChoixModeVolManager*.

Les autres attributs sont des variables de type *TextView* qui servent essentiellement à afficher à l'écran les différentes valeurs stockées dans la classe *DataManager*.

**Méthodes:**

Les méthodes utilisées sont les suivantes:

- La méthode *findViewById* qui sert à instancier les différents objets de type *TextView*.
- La méthode *getInstance* appelée sur un objet de classe *DataManager* afin de pouvoir récupérer les données.
- La méthode *setText* qui prend en argument un objet de type *String* et qui sera appelée sur un objet de type *TextView* afin de changer sa valeur affichée.
- L'ensemble des méthodes assesseurs de la classe *DataManager* afin de pouvoir récupérer les différentes données.

## 6.10 RedActivity

**Description :** Cette activité permet d'afficher la fenêtre liée au signal visuel. Un écran de couleur "rouge" est affiché pendant une durée déterminée avant de passer à l'activité *CompteurChargeMentaleActivity*.

**Objectif :** L'activité *RedActivity* a pour but de respecter la méthode "ISA", en affichant un signal visuel à l'utilisateur avant de pouvoir mesurer la charge mentale.

**Attributs:** Pour cette classe il n'y a pas d'attribut.

**Méthodes:**

Les méthodes utilisées sont les suivantes:

- La méthode *passerALaProchaineActivite* qui permet de passer à la prochaine activité, *MesureMultiActivity* ou *ChargeMentaleSeuleActivity* en fonction du nombre d'utilisateurs. Cette méthode est composée de deux objets de types *Intent* afin de pouvoir lancer les activités. Elle ne prend aucun argument et a pour type de retour *void*.
- La méthode *run* de la classe *Handler* qui permet d'exécuter une tâche en différé. Dans notre cas, de passer à la prochaine activité après un certain temps. Elle ne prend aucun argument et a pour type de retour *void*.

## 6.11 SonActivity

**Description :** Cette activité permet d'afficher la fenêtre liée au signal sonore. Un signal sonore choisi au préalable est émis pendant une durée déterminée avant de passer à l'activité *CompteurChargeMentaleActivity*.

**Objectif :** L'activité *SonActivity* a pour but de respecter la méthode "ISA", en émettant un signal sonore à l'utilisateur avant de pouvoir mesurer la charge mentale.

**Attributs:**

Les attributs de l'activité *SonActivity* sont les suivants :

- mediaPlayer de type *MediaPlayer*.

**Méthodes:**

Les méthodes utilisées sont les suivantes:

- La méthode *passerALaProchaineActivite* qui permet de passer à la prochaine activité, *MesureMultiActivity* ou *ChargeMentaleSeuleActivity* en fonction du nombre d'utilisateurs. Cette méthode est composée de deux objets de types *Intent* afin de pouvoir lancer les activités. Elle ne prend aucun argument et a pour type de retour *void*.
- La méthode *onDestroy* qui permet de libérer les ressources du lecteur audio. Elle ne prend aucun argument et a pour type de retour *void*.
- Les méthodes *create* et *start* de la classe *MediaPlayer* qui permet de créer un lecteur audio associé à un son et qui permet de lancer l'enregistrement audio.
- La méthode *run* de la classe *Handler* qui permet d'exécuter une tâche en différé. Dans notre cas, de passer à la prochaine activité après un certain temps, et en plus de cela arrêter l'enregistrement audio. Elle ne prend aucun argument et a pour type de retour *void*.

## 6.12 CodeActivity

**Description :** Cette activité permet d'afficher la fenêtre liée au code d'accès. À la fin de l'expérimentation lorsque l'utilisateur a décidé définitivement d'arrêter les mesures, le code d'accès lui est présenté.

**Objectif :** L'activité *CodeActivity* a pour objectif d'afficher un code d'accès qui permettra ensuite aux utilisateurs de le rentrer sur le module d'exploitation afin d'accéder à leurs analyses de données (graphiques etc.).

**Attributs:**

Les attributs de l'activité *SonActivity* sont les suivants :

- vueCode de type *TextView*.
- code de type *CodeManager*.
- date de type *LocalDateTime*.

**Méthodes:** Cette activité ne possède pas de méthodes.

# 7 Scripts PHP pour le module d'exploitation des données

## 7.1 Script PHP cible

**Description :** Ce script est la cible de la requête POST, il se trouve sur le serveur web et permet de traiter les données reçues.

**Objectif :** Son objectif est d'interpréter les données reçues, ces données commencent toujours par un code, que le script utilise pour créer un fichier (ie le nom du fichier est le code), si le fichier existe déjà il écrit dedans, sinon il le créé. Ce système de code permet de regrouper les données de plusieurs utilisateurs différents mais participant à la même expérimentation sur un même fichier. Après avoir créé le fichier, le script rentre les données reçues dedans, c'est-à-dire les noms, les charges mentales et les informations liées au vol si le mode vol est activé durant l'expérimentation.

## 7.2 tableau

**Description :** Ce script permet d'afficher des tableaux HTML sur le module.

**Objectif :** Son objectif est de sortir les données brutes des fichiers textes contenant les résultats des expérimentations, et de les mettre dans des tableaux HTML que les personnes qui vont les analyser pourront consulter et comprendre facilement.

**Fonctions :** Ce script comporte plusieurs fonctions :

- *initialisation*, elle permet de générer un tableau d'indices contenant les données brutes de l'expérimentation, ce tableau servira plus tard à rentrer les données dans un tableau HTML plus facilement. Elle prend en entrée une chaîne de caractère qui est le code correspondant à l'expérimentation et le chemin d'accès vers le fichier textes contenant les résultats de l'expérimentation.

- *noms*, elle retourne un tableau d'indices avec dans chaque cases le nom d'un des participants à l'expérimentation, il servira pour placer les bonnes valeurs au bons endroits dans le tableau issu de *creationTableauTriHTML*. Elle prend en entrée un tableau semblable à celui que génère la fonction initilisation.

- *tri*, cette fonction n'est pas utile pour le moment

- *creationTableauTriHTML*, cette fonction permet de générer le code HTML d'un tableau trié, c'est-à-dire que les données sont regroupées par utilisateurs, par exemple sur une ligne on aura les charges mentales liées à un seul utilisateur, la charge mentale qu'il s'est donné, et les charges mentales que les autres lui ont donné. Elle prend en entrée un tableau de noms et un tableau de valeurs.

- *creationTableauBaseHTML*, cette fonction permet de générer le code HTML d'un tableau de base, c'est-à-dire que les données sont affichées telles qu'elles sont reçues par le serveur. Elle prend en entrée un tableau de noms et un tableau de valeurs.

## 7.3 Connexion

**Description :** Le script permet de récupérer le code saisi par l'utilisateur et le redirige vers la page suivante si le fichier avec ce code existe.

**Objectif :** L'objectif est de récupérer le code saisi par l'utilisateur dans la page *index.html*, vérifie si un fichier texte avec ce nom existe.

Si c'est le cas, alors la page de l'expérimentation s'affiche, sinon cela affiche une erreur.

**Fonctions :** Ce script ne comporte qu'une fonction :

- La fonction principale vérifie si le champ a été saisi, récupère le code, l'enregistre comme un cookie (qui dure 24h) et renvoie l'utilisateur vers '*donneesVol.php*' si le code existe, sinon cela affiche une erreur (le code n'a pas été saisi ou qu'il n'existe pas).

## 7.4 DonneesVol

**Description :** Le script permet d'afficher les données du vol si le mode *vol* a été activé durant l'expérimentation.

**Objectif :** : L'objectif est de vérifier si le mode vol a été activé durant l'expérimentation, si c'est le cas alors la fonction récupère les données du vol et les affiche.

**Fonctions :** Ce script ne comporte qu'une fonction :

- La fonction principale récupère le code saisi précédemment après qu'il a été sauvegardé en cookie.

  Elle ouvre ensuite le fichier texte qui porte ce nom, vérifie si le mode " *Vol*" a été activé en récupérant la première lettre du nom de notre fichier.

Elle sépare la première ligne du fichier avec un *explode()*, à l'aide de *str_replace* remplace les balises spécifiques dans *index.html* par les éléments de la ligne. Si le mode vol est désactivé, alors elle affiche 'OFF' et 'none'.

## 7.5 Graphiques

**Description :** Le script permet de générer et d'afficher des graphiques sur le module.

**Objectif :** Son objectif est d'extraire les données des utilisateurs participant à l'expérience depuis un fichier texte, puis de les transformer en données JSON lisibles en JavaScript puis d'enfin les afficher dans des graphiques à l'aide de Google Chart.

**Fonctions :** Ce script comporte plusieurs fonctions :

- *initialisation*, cette fonction prend en entrée le nom et l'emplacement du fichier texte avec les données qu'on souhaite récupérer. Elle retourne ensuite un tableau en utilisant "\n" comme séparateur avec la fonction *explode()*.
- *noms*, cette fonction prend en entrée le tableau obtenu à partir de la fonction *init()*, pour chaque élément de ce tableau, elle divise la chaîne par le délimiteur "-" avec la fonction *explode()*. Elle prend ensuite la première partie du résultat et la divise par le délimiteur "." avec *explode()* pour obtenir le nom sans extension. Elle ajoute le nom obtenu dans un tableau qu'elle retourne à la fin.

### 7.5.1 JavaScript

- *drawChart* (premier graphique), cette fonction est responsable du dessin du premier graphique, elle commence par récupérer les données JSON des noms et des moyennes PHP (encodées précédemment) dans les variables JavaScript *noms* et *moyennes*.

  Elle crée ensuite une nouvelle instance de *google.visualization.DataTable()* qui va stocker les données du graphique, puis elle ajoute deux colonnes à la *DataTable* : une pour les noms (de type '*string*') et une pour les moyennes (de type '*number*'), à l'aide d'une boucle, elle parcourt les tableaux *noms* et *moyennes* et ajoute chaque paire nom-moyenne à la *DataTable*.

- *drawChart* (deuxième graphique), cette fonction est responsable du dessin du deuxième graphique. Elle commence par intégrer les données JSON PHP dans le script JavaScript en les assignant aux variables JavaScript *noms*, *valeurs* et *nombre_valeurs*.

  Ensuite, elle crée une nouvelle instance de *google.visualization.DataTable()* qui va stocker les données du graphique, elle ajoute une colonne pour les valeurs de la mesure (axe X) et une colonne pour chaque nom d'utilisateur (chaque série de données).

  À l'aide de deux boucles imbriquées, elle parcourt les valeurs stockées dans *valeurs*, les divise en fonction des points décimaux et les ajoute à la *DataTable*, ligne par ligne, pour chaque mesure.

# 8 Références et sources

[1]**Le lien permettant de télécharger et consulter le cahier des charges:** https://forge.ens.math-info.univ-paris5.fr/attachments/download/10942/Cahier_des_charges_PROJET_L2P1%20VERSION%202.0.0.pdf

[2] **Le lien permettant de télécharger et consulter le cahier des recettes:** https://forge.ens.math-info.univ-paris5.fr/attachments/download/11217/Cahier_des_recettes_PROJET_L2P1_VERSION_1.01.pdf

[3] **Classe singleton :**
https://www.baeldung.com/java-singleton

**Documentation PHP :** https://www.php.net/manual/fr/index.php
https://www.baeldung.com/java-singleton

**Téléchargement des données pour Excel:** https://youtu.be/vWbMRlUvre8?si=Aiy3OIepsdOri1Nu