Hypercool Reader

IMAGE APPLI

|  |  |
| --- | --- |
| Auteur | Murat Bayrakci |
| Date de début de projet | 13.03.2017 |
| Date de fin de projet | 11.05.2017 |
| Date de reddition du rapport |  |
| Modification |  |
| Version | Dernière Version |

Table des matières

[1 Introduction 3](#_Toc467491237)

[2 Documentation de développement 4](#_Toc467491238)

[2.1 Explications détaillées du projet 4](#_Toc467491239)

[2.2 Diagramme des cas d'utilisation 6](#_Toc467491240)

[2.3 Architecture du système 7](#_Toc467491241)

[2.4 Arborescence des fichiers 7](#_Toc467491242)

[2.5 Définition des conventions applicables 7](#_Toc467491243)

[2.6 Planning de livraison global 8](#_Toc467491244)

[2.7 Base de données 8](#_Toc467491245)

[2.8 Flux de Navigation 8](#_Toc467491246)

[3 Réalisation des cas d'utilisation 8](#_Toc467491247)

[3.1 Cas d'utilisation n°... ou nommé .... 8](#_Toc467491248)

[3.1.1 Scénario 8](#_Toc467491249)

[3.1.2 Maquettes 8](#_Toc467491250)

[3.1.3 Analyse du scénario 8](#_Toc467491251)

[3.1.3.1 Algorithme ou Structogramme 8](#_Toc467491252)

[3.1.3.2 Explications détaillées 8](#_Toc467491253)

[3.1.4 Le planning de livraison 8](#_Toc467491254)

[3.1.5 La phase de programmation 9](#_Toc467491255)

[3.1.6 La phase de tests 9](#_Toc467491256)

[3.2 Cas d'utilisation suivant°... ou nommé .... 9](#_Toc467491257)

[3.2.1 … 9](#_Toc467491258)

[4 Mode d'emploi utilisateur 10](#_Toc467491259)

[5 Problèmes rencontrés et solutions 10](#_Toc467491260)

[6 Conclusions 13](#_Toc467491261)

[7 Annexes 13](#_Toc467491262)

[7.1 Journal de bord 13](#_Toc467491263)

[7.2 Cahier des charges 13](#_Toc467491264)

[7.3 Code source 13](#_Toc467491265)

[7.4 Références 14](#_Toc467491266)

# Introduction

Le 13.03.2017 marquait le début de ce projet. J’avais 2 mois pour réaliser ce qui m’était demandé, et répondre au cahier des charges. Je suis très intéressé par la programmation, et j’avais initialement demandé un sujet portant sur le développement. Durant notre cursus au Cpln, nous avons appris, aux travers des différents modules, les bases de la programmation en C#. Depuis cette année, nous avons commencé en atelier, à travailler sur du Java pour développer des applications Android. Les 2 langages étaient assez proches et ça nous a permis de nous adapter facilement.

J’ai reçu un cahier des charges me demandant de réaliser une application mobile, orientée pour les appareils fonctionnant avec le système d’exploitation Android.

Le but principal de l’application était simple : Afficher l’horaire des classes depuis un téléphone mobile Android. Je devais faire attention à la lisibilité et à l’ergonomie de l’affichage car actuellement, la lisibilité de l’horaire disponible sur le site internet du Cpln n’est pas satisfaisante pour les appareils mobiles.

Pour développer, je disposais des postes du Cpln ayant l’environnement de développement, Android Studio. J’ai utilisé cet utilitaire depuis le début de l’année. Les données de l’horaire des classes m’étaient envoyées depuis un site crée par un professeur de l’école. J’avais également le droit d’amener mon appareil Android personnel pour pouvoir effectuer des tests.

Au-delà du fait de rendre un bon projet pour passer l’année et ainsi terminer ma formation, j’étais motivé à produire une application de qualité, pour qu’elle puisse éventuellement être utilisée par les élèves des prochaines années.

# Documentation de développement

## Explications détaillées du projet

Pour ce projet, j’ai reçu une liste de fonctionnalités principales qui devaient être présentes dans l’application. J’ai également une deuxième liste de choses à faire si le temps le permet.

Comme introduit au chapitre précédent, le but principal de mon application est d’afficher l’horaire des classes présentes au Cpln. 2 vues différentes me sont demandées. Tout d’abord, lors de l’ouverture de l’application, afficher l’horaire d’une classe uniquement pour un seul jour. En revanche, la deuxième vue doit permettre de visualiser l’horaire de la semaine (du lundi au vendredi). J’ai reçu un exemple de design graphique pour la vue jour dans le cahier des charges. Je pouvais m’en inspirer ou bien créer quelque chose de totalement différent. Pour la vue semaine, je n’ai pas reçu d’exemple, et c’est moi qui devais proposer un design au professeur.

Au lancement de l’application, le jour actuel doit automatiquement être inséré dans le champ date. Ceci permet d’éviter à l’utilisateur de le taper manuellement à chaque ouverture.

Egalement au lancement de l’application, le nom de la dernière classe que l’utilisateur a recherché doit être inséré dans le champ classe.  
Ces deux étapes permettent de lancer la recherche et d’afficher l’horaire directement après l’ouverture de l’application.

Dans la vue jour, quatre boutons doivent être présentes pour modifier la date. Les deux boutons situés aux extrémités permettront d’ajouter/diminuer la date d’une semaine, et les deux boutons au milieu d’ajouter/diminuer un jour. Dans la seconde vue, seulement deux boutons seront présents, chacun modifiant d’une semaine la date.

L’utilisateur doit pouvoir garder un historique de toutes les recherches effectuées. A chaque appui sur le bouton rechercher, l’application doit enregistrer le nom de la classe pour pouvoir l’afficher par la suite. L’utilisateur doit pouvoir accéder à cette liste à l’appui d’un bouton historique.

Un menu doit être présent pour pouvoir changer de vue et également effacer les recherches déjà effectuées.

Des contrôles de saisies doivent être effectués pour ne pas laisser à l’utilisateur la possibilité d’entrer des classes vides ou non existantes dans le champ classe. Pareil avec le champ date qui doit uniquement laisser les dates valides. En cas d’erreurs, des messages clairs et précis doivent signaler le problème à l’utilisateur.

L’application doit s’adapter à la taille de l’écran de l’utilisateur et proposer un affichage correct dans toutes les circonstances.

Les points suivants sont des objectifs secondaires présents dans le cahier des charges, ou ajoutés par le professeur.

Proposer la liste de toutes les classes disponibles lorsque l’utilisateur est en train d’entrer une classe dans le champ classe. Convenu avec le professeur.

Prévoir un bouton qui va afficher un calendrier. L’utilisateur pourra alors choisir plus facilement la date désirée. Convenu avec le professeur.

Pouvoir naviguer d’un jour à l’autre (vue jour) ou d’une semaine à l’autre (vue semaine) avec un balayage de l’écran de gauche à droite et inversement.

Gérer efficacement la rotation de l’écran. Les boutons doivent d’adapter au nouveau mode (portrait/paysage) et l’affichage de l’horaire ne doit pas être de qualité inférieure.

Stocker les données reçues dans une base de données. En cas d’inaccessibilité à internet, utiliser les données de cette base pour tout de même afficher quelque chose à l’utilisateur.

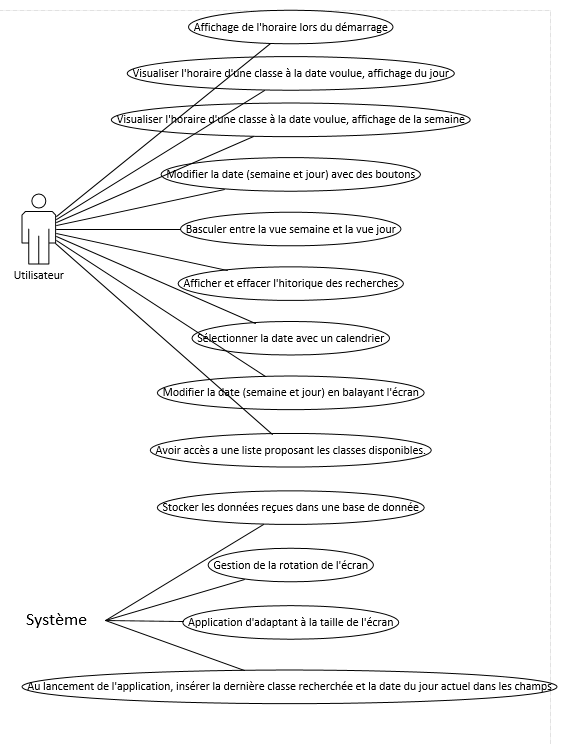
Certains noms de classe renvoient plusieurs id de plusieurs classes. Par exemple, 3m3i va nous renvoyer l’id des classes 3m3i2, 3m3i3 et 3m3i1. Lorsque cela se produit, avertir l’utilisateur et lui proposer de choisir parmi la liste des classes renvoyées. Convenu avec le professeur.

Il s’agit d’une explication détaillé du projet.

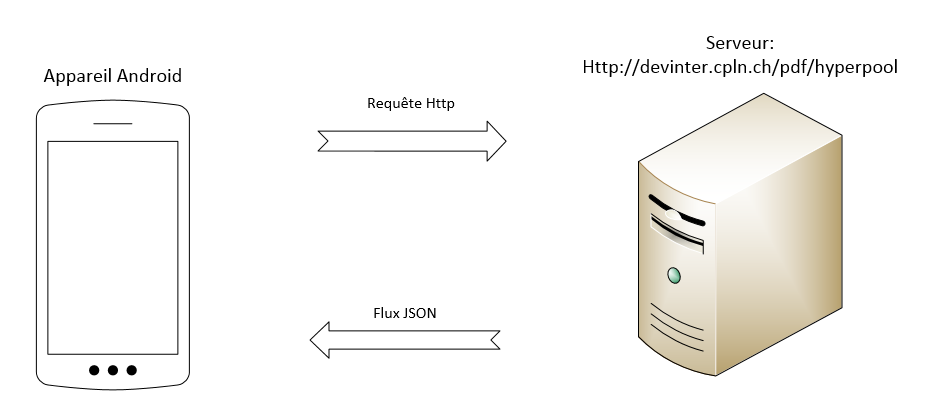
Cela définira les objectifs que le programme doit remplir.

Cette explication est basée sur le cahier des charges.

## Diagramme des cas d'utilisation



## Architecture du système



1. L’application (appareil Android) va faire une requête au serveur avec les éléments demandés (id des classes, horaire des classes…)
2. Le serveur va renvoyer les informations demandées sous format JSON.

## Arborescence des fichiers

Représentation de la structure des fichiers et de leurs emplacements.

## Définition des conventions applicables

Convention de nommage des variables :

|  |  |
| --- | --- |
| Type | Convention |
| String | strExemple |
| Int | iExemple |
| Array | arrayExemple[] |
| Arraylist | AlistExemple |
| Float | flExemple |
| Boolean | bExemple |
| File | fExemple |
| Bouton | btnExemple |
| TextView | tvExemple |
| LinearLayout | llExemple |
| AutoCompleteTextView(Actv) | actvExemple |
| EditText | etExemple |

Convention de nommage des activités :

Les noms des activités commencent par Activite\_

Ex : Activite\_VueSemaine

Convention de nommage des fonctions :

Première lettre des mots en majuscule.

Ex :

public void ExempleAfficher() {}

Si le premier mot est « get » ou « set », alors premier mot en minuscule, et les suivants en majuscules.  
Ex:

public String getDerniereClasseRecherchee(){}

public void setDate(){}

Cette rubrique permet aussi la définition des standards graphiques applicables.

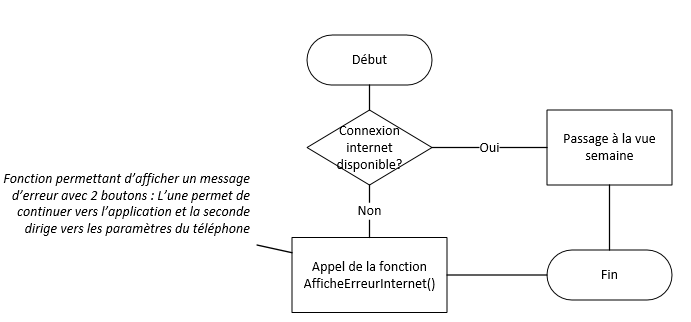
## Planning de livraison global

Planning en annexe.

## Flux de Navigation

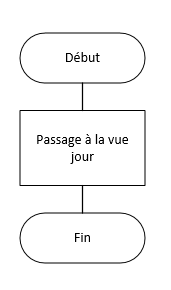
Vue Jour

Lors du clic sur le bouton « Vue Semaine » depuis la vue jour.



Vue Semaine

Lors du clic sur le bouton « Vue Jour » depuis la vue semaine.



# 

# Réalisation des cas d'utilisation

## Cas d'utilisation « Affichage de l’horaire du jour lors du démarrage »

Ce cas d’utilisation est utilisé au démarrage de l’application. Il va servir à afficher l’horaire du jour actuel. La dernière classe recherchée va être insérée dans le champ classe. Ça permet à l’utilisateur de visualiser l’horaire utile à lui dès le démarrage de l’application.

### Scénario

Décrire les différentes étapes par des phrases concises :

1. L’utilisateur ouvre l’application.
2. Le système vérifie la connexion internet.
3. Le système remplit le champ classe avec la dernière classe recherchée
4. Le système remplit le champ date avec la date du jour actuel.
5. Le système met dans des variables le contenu des champs, vérification des saisies.
6. Le système affiche le nom du jour de la semaine.
7. Le système établit une connexion avec le serveur et lance les requêtes (Réception id, et liste de toute les classes)
8. Le système effectue des vérifications des données reçues.
9. Le système lance la requête pour avoir l’horaire de la classe avec l’id de la classe.
10. Le système met le nom de la classe recherchée dans l’historique.
11. Le système lit l’historique pour mettre à jour le tableau contenant l’historique.
12. Le système met les données de l’horaire reçues dans des variables.
13. Le système tri les branches de manière chronologique à leurs heures de début.
14. Le système affiche les données.

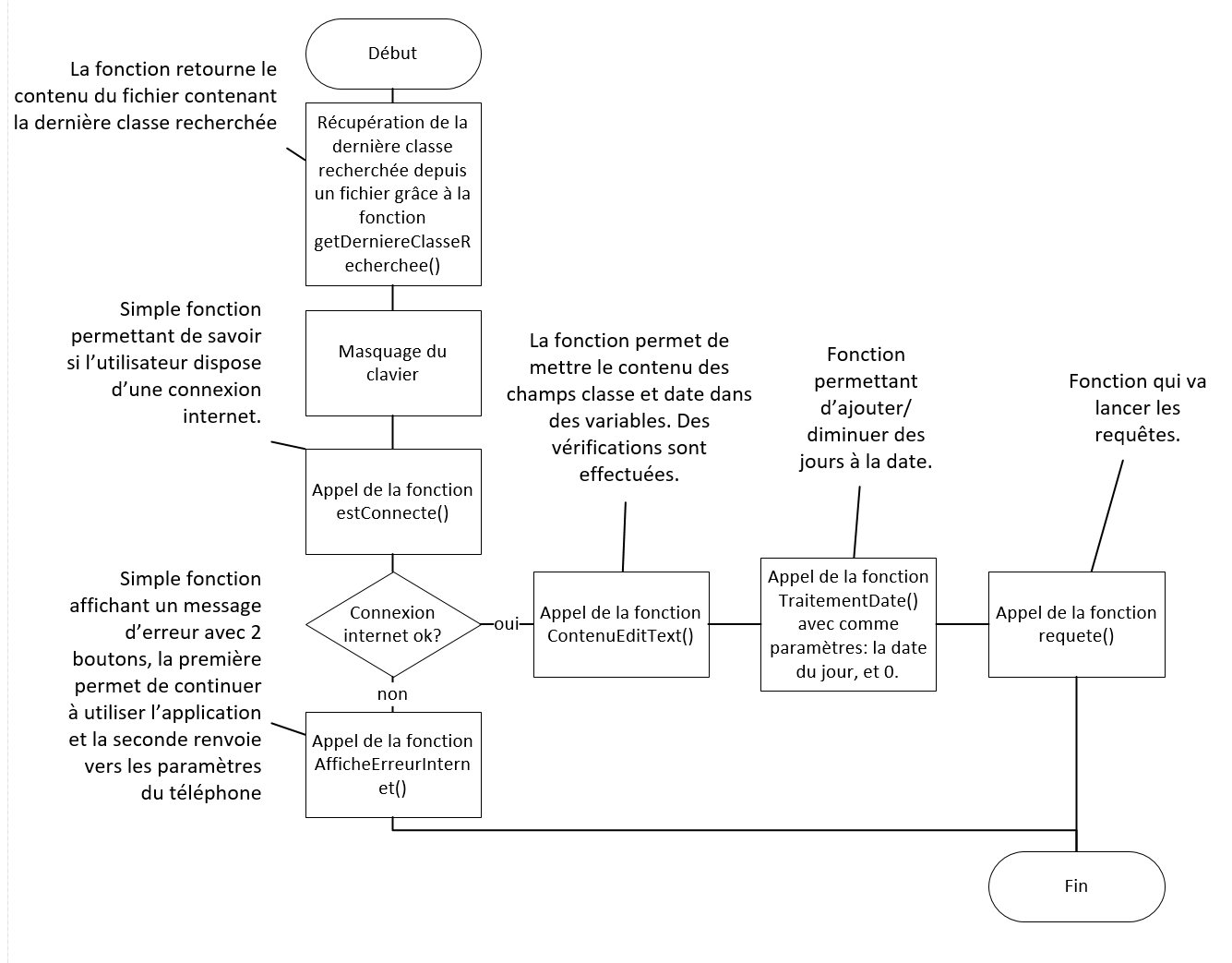
### Maquettes

### 

### Analyse du scénario

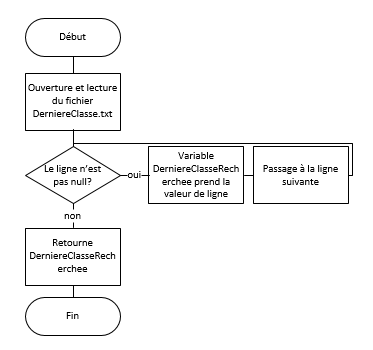
#### Algorithme ou Structogramme

Lors du onCreate() :

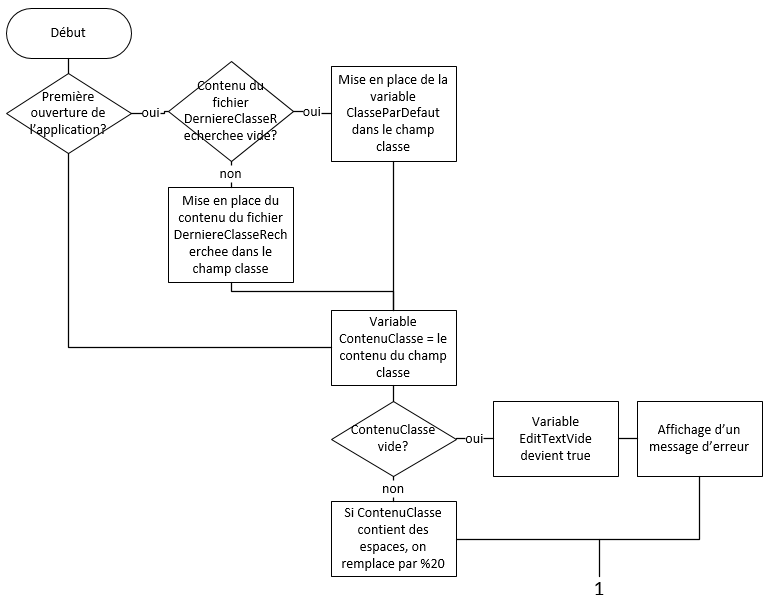


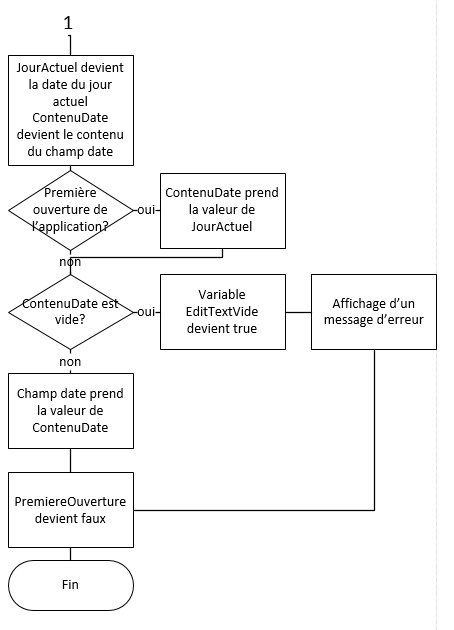
#### 

Fonction getDerniereClasseRecherchee() :

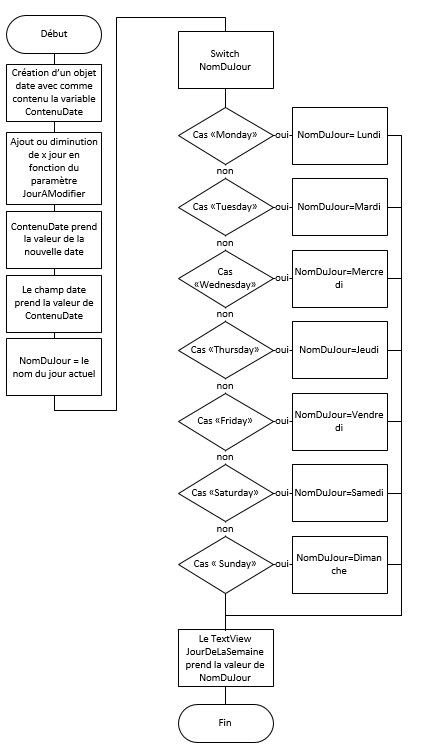


Fonction ContenuEditText() :

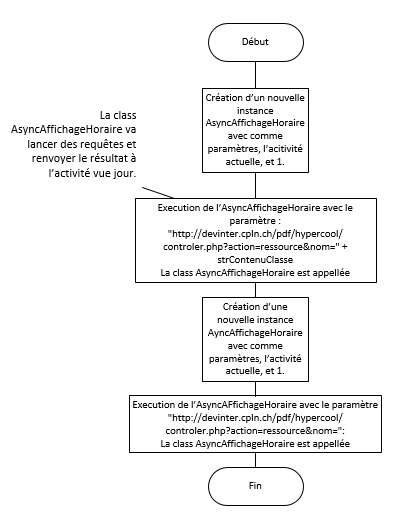




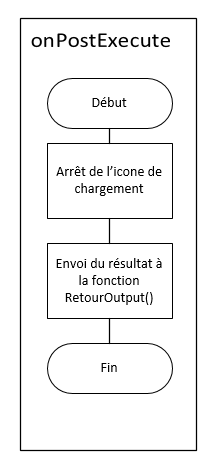
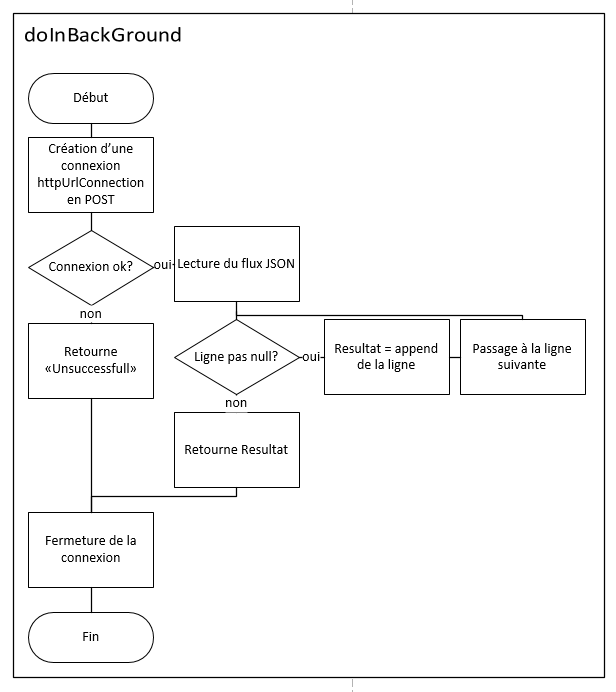
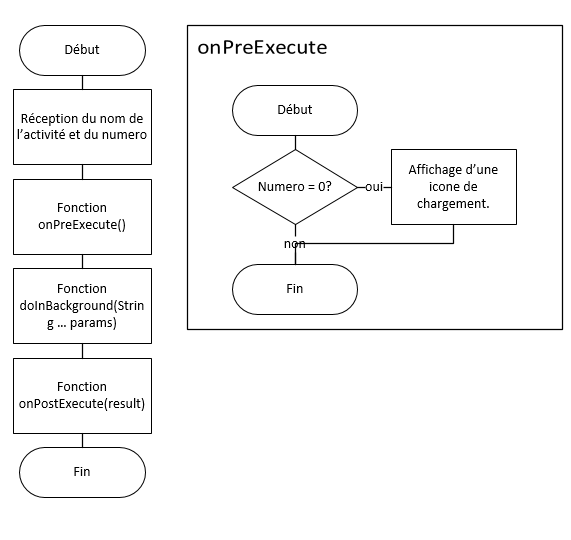
Fonction TraitementDate() :



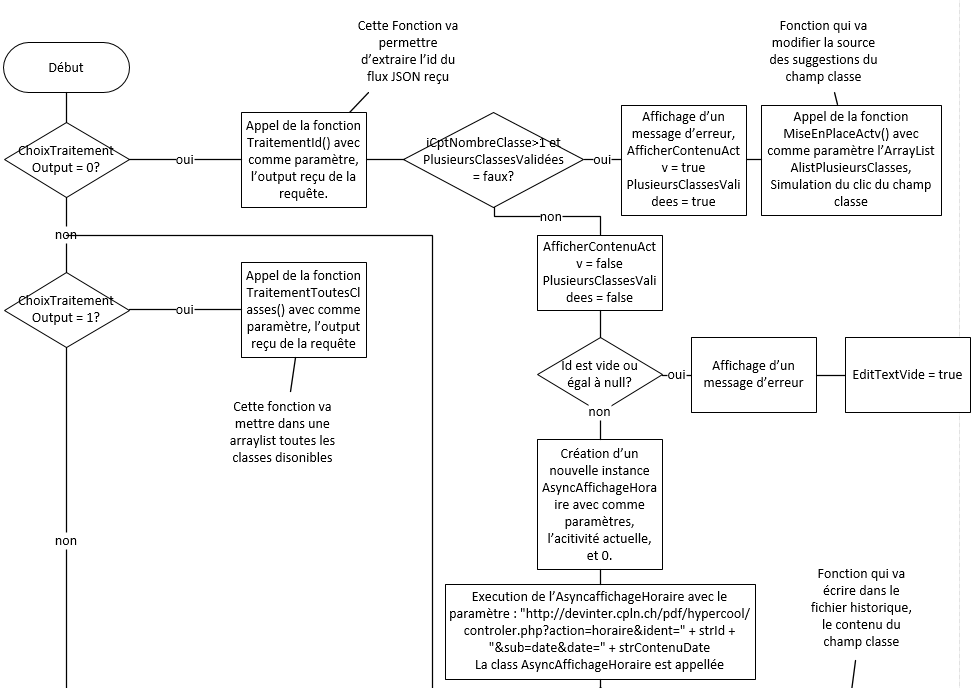
Fonction Requete() :

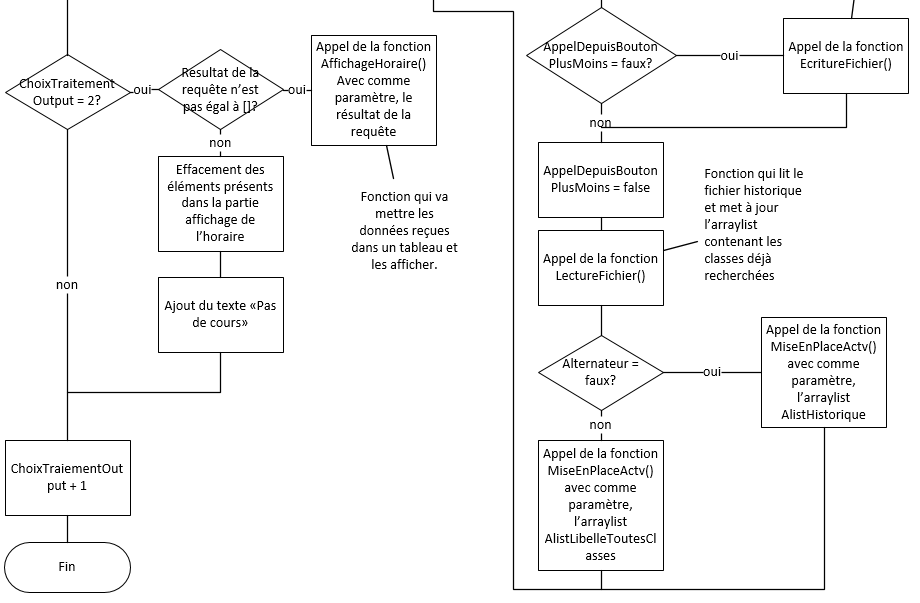


Class AsyncAffichageHoraire() :

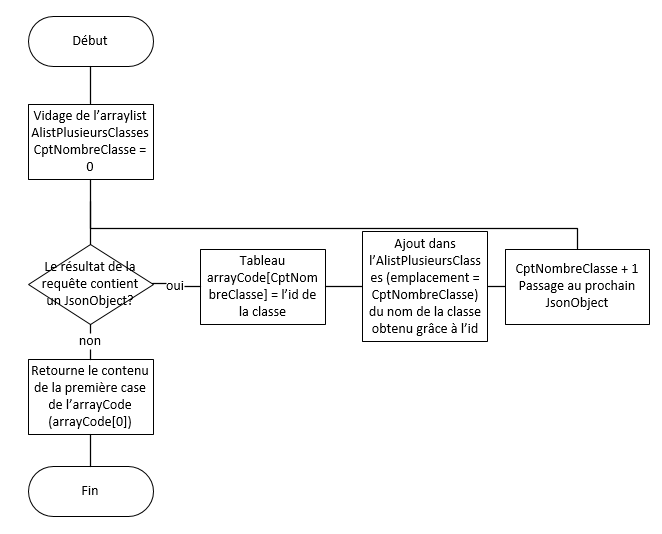


Fonction RetourOutput() :

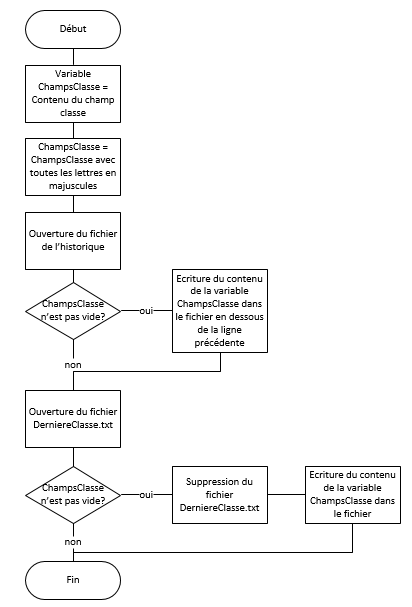




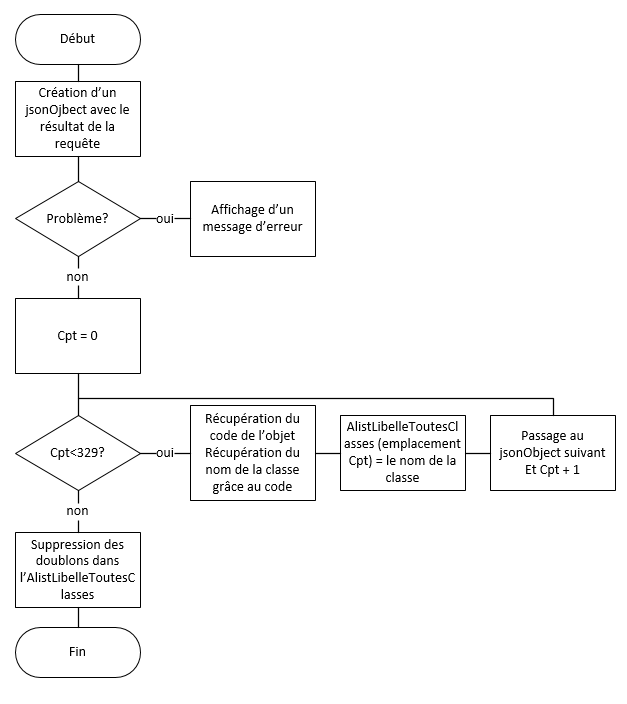
Fonction TraitementId() :



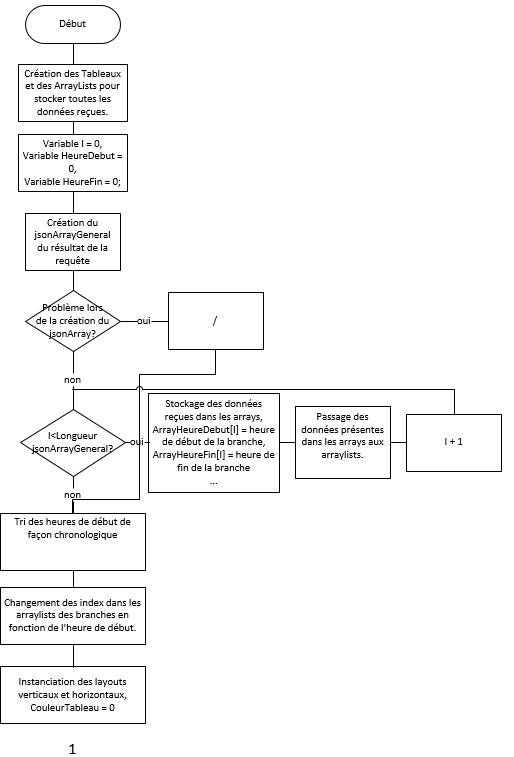
Fonction EcritureFichier() :

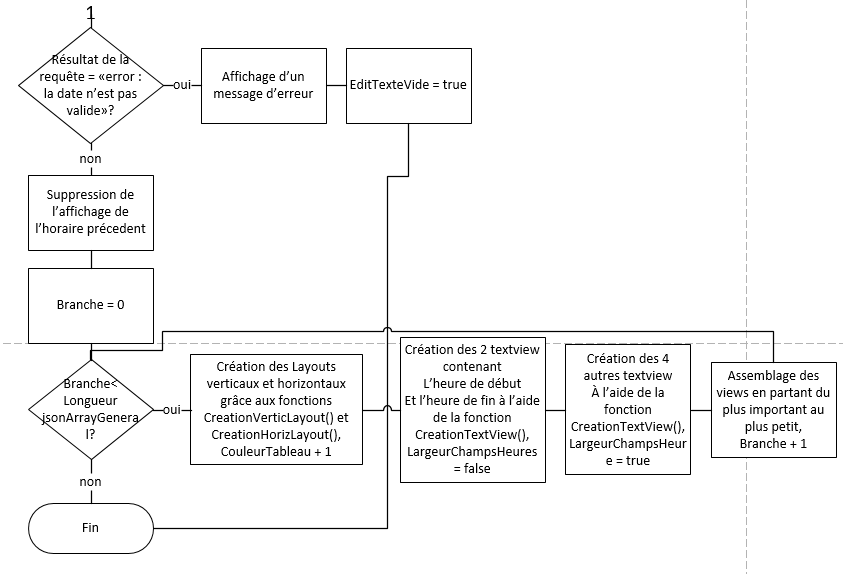


Fonction TraitementToutesClasses() :

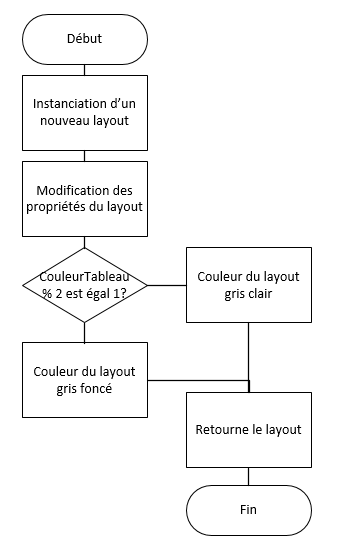


Fonction AffichageHoraire() :

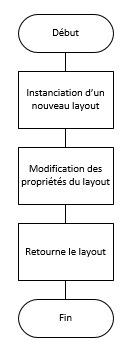




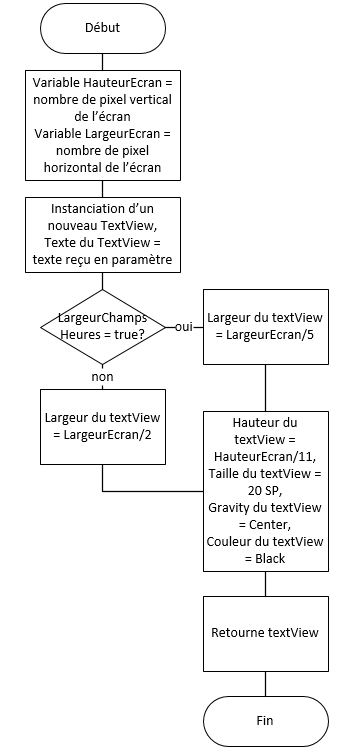
Fonction CreationVerticLayout() :



Fonction CreationHorizLayout() :



Fonction CreationTextView() :



#### Explications détaillées

Tout d’abord, on récupère le nom de la dernière classe recherchée que l’on place dans une variable.

On masque le clavier qui, normalement apparait automatiquement lors de l’ouverture de l’application. Dans ce cas-là, c’est inutile car l’utilisateur veut directement voir l’affichage de l’horaire et n’a pas besoin de taper le nom d’une classe.

Pour vérifier la connexion internet, on fait appel à la fonction estConnecte(). Elle retourne true si la connexion est ok et false dans le second cas. Si l’utilisateur n’as pas de connexion internet, alors on fait appel à la fonction AfficheErreurInternet(). Cette fonction affiche un message avec 2 options possibles à l’utilisateur. La première permet de continuer à utiliser l’application et la seconde, d’accéder aux paramètres de l’appareil pour vérifier sa connexion.

Si tout est ok, On va passer à la fonction ContenuEditText(). Si c’est la première fois que l’utilisateur ouvre l’application, alors on va vérifier le contenu de la variable ayant cherché la dernière classe recherchée dans le fichier. Si elle est vide, alors on attribue une classe par défaut au champ classe. Normalement, la variable contenant la dernière classe recherchée, ne doit pas être vide, c’est uniquement possible par exemple lorsque l’utilisateur utilise la toute première fois l’application, ou bien que le fichier contenant cette information a été supprimé.

Le champ classe va prendre le contenu de cette variable lorsqu’elle contient quelque chose. Après ces étapes, la fonction va vérifier que le contenu du champ ne soit pas vide. (Cette partie est plus utilisé après la première ouverture.). La variable bEditTextVide passe à true si c’est le cas et un message d’erreur apparait. Si le champ n’est pas vide, il faut remplacer les espaces par des %20. Sans faire cela, il est impossible de récupérer les id des classes contenant un ou des espaces. Si on fait une requête manuellement sur le site hypercool, on voit que les espaces sont automatiquement remplacés par des %20. Lors de mes requêtes dans l’application, je suis tout de même obligé de passer par cette étape, le site ne le fait pas automatiquement. Pour s’occuper de la date, on crée un calendrier pour obtenir le jour actuel. On récupère aussi le contenu du champ date que l’on met dans une variable strContenuDate. Si c’est la première ouverture, alors la variable strContenuDate prend la valeur du jour actuel.

On doit de nouveau tester si la variable est vide ou pas. bEditTexteVide devient true et un message apparait comme avant en cas d’erreur. Autrement le contenu du champ date devient la valeur de strContenuDate. Tout à la fin de la fonction, bPremiereOuverture passe à false.

La fonction TraitementDate() est appelée. Elle sert normalement à ajouter/diminuer des jours à la date, mais elle permet également d’afficher le nom du jour actuel dans une TextView. C’est pour cela qu’on envoie le paramètre 0, la date ne va pas être modifiée mais le nom du jour va être affiché.

Finalement on peut lancer la fonction Requete(). 2 requêtes sont lancées ici. La première va récupérer le code de la classe. Pour cela il fallait obtenir le nom de la classe que l’on a eu précédemment. La deuxième requête est lancée au même temps que la première. Celle-ci va renvoyer la liste de toutes les ressources que contient le site Hypercool. Les classes, professeurs, salles, cours… sont renvoyées. Seules les classes nous intéressent, en obtenant toutes les classes disponibles, on peut enregistrer leurs noms pour les proposer à l’utilisateur sous forme de liste. Les requêtes sont effectuées dans une class séparée. Il n’est pas autorisé de lancer des requêtes HttpUrlConnection dans l’activité principal. En passant par cette class séparée, on peut rendre la tâche asynchrone. Ça nous permet d’afficher une icône de chargement et de ne pas faire « freeze » l’écran principal.

Cette classe Asynctask, avant de lancer les requêtes, fait différentes choses. Tout d’abord on obtient le nom de l’activité et un code (passé en paramètre). Dans le onPreExecute(), on crée l’icône de chargement si le numéro est égal à 0. Après cela, les requêtes sont exécutées dans le doInBackground(), grâce à HttpUrlConnection qu’on a paramétré. On va ensuite mettre dans une variable les données reçues. On fait passer cette variable dans le onPostExecute(). L’icône de chargement est de toute façon supprimée ici peu importe le code. Le résultat (données reçues de la requête) va être envoyé à la fonction RetourOutput, située dans l’activité principale.

La fonction sert principalement à aiguiller le résultat de la requête. Par défaut, iChoixTraitementOutput est égal à 0. Lorsque la première requête qu’on a envoyée (pour avoir l’id de la classe) se termine, les données reçues vont passer par le if(iChoixTraitementOutput==0) Tout à la fin de la fonction, iChoixTraitementOutput est incrémentée. Ça signifie que les données de la seconde requête qu’on a envoyée vont passer par un autre chemin. Pour avoir l’id de la classe, il faut traiter le flux JSON reçu pour prendre uniquement ce qui nous intéresse. C’est là qu’intervient la fonction TraitementId(). On va créer un jsonObject avec le résultat reçu, ça nous permet de récupérer uniquement l’id de la classe. Si plusieurs id sont renvoyées, alors on va compter le nombre et mettre dans l’AlistPlusieursClasses le nom des classes correspondent aux ids. Cette situation est possible lorsque l’on recherche le nom d’une classe qui renvoie plusieurs classes. Par exemple : 3m3i va renvoyer les données des classes 3m3i1, 3m3i2 et 3m3i3. La fonction retourne l’id de la classe qui va être utilisé par la requête suivante. La variable strId contient dorénavant l’id d’une classe. Il faut maintenant vérifier le nombre de classes reçues et ainsi agir en conséquence. Pour que ça soit plus agréable pour l’utilisateur, lors de l’ouverture de l’application, si la dernière classe recherchée, renvoie plusieurs classes, alors on ne va pas demander de choisir la bonne. Comme ça, on peut directement lancer la recherche. C’est pour cela que dans le if qui va détecter si le nombre d’id est supérieur à 1, il y a également la condition « && !bPlusieursClasseValidees ». Cette variable est true lors du démarrage. Dans notre cas, on passe automatiquement dans le else() car l’utilisateur vient d’ouvrir l’application. Dans le else(), on va mettre à false les variables bAfficherContenuActv et bPlusieursClasseValidees. Une vérification est faite de l’id. Si elle vaut null ou bien elle est vide, alors ça signifie que la classe n’existe pas. On affiche alors un message d’erreur et bEditTexteVide passe à vide (variable qui permettra de bloquer l’appui sur les boutons + et – lorsqu’un champ est erroné). Une fois ce test passé, on peut lancer la dernière requête avec l’id et la date. Cette fois ci, le code qu’on envoie comme paramètre de l’Asynctask équivaut 1 et va donc afficher l’icône de chargement. Il reste encore à enregistrer le contenu du champ classe dans l’historique\*. Vu que la variable bAppelDepuisBoutonPlusMoins est faux, on fait appel à la fonction EcritureFichier(). Cette fonction ouvre un fichier texte, et met le contenu du champ classe (à moins qu’il soit vide) en majuscule en dessous des lignes précédentes. Ça permet d’éviter les redondances (sans ça, 3m3i2 et 3M3I2 vont être enregistrées séparément). En parallèle, le contenu du champ classe est également enregistré dans le fichier « DerniereClasse.txt ». Après cette fonction, c’est LectureFichier() qui est appelée. Elle va ouvrir le fichier contenant l’historique et mettre chaque ligne dans l’arraylist AlistHistorique. Etant donné qu’il peut y avoir des doublons, SuppressionDoublons() va supprimer toutes les items de l’AlistHistorique ayant le même nom. Une fois ces étapes accomplies, une dernière fonction est appelée. MiseEnPlaceActv() va, comme son nom l’indique, alimenter en suggestions le champ classe. Je ne m’attarde pas dessus car il sera expliqué plus en détail dans le chapitre « afficher la liste des classes disponibles ». Avant d’avoir eu recours aux fonctions EcritureFichier() et LectureFichier(), on a lancé la dernière requête. Le résultat de cette requête va passer par RetourOutput() mais cette fois ci, dans le if(iChoixTraitementOutput ==2). Les deux premières requêtes que l’on a envoyées sont déjà passées par cette fonction et ont ainsi incrémenter 2 fois la variable iChoixTraitementOutput. Les données reçues contiennent l’horaire de la classe à la date choisie. Une dernière vérification s’impose. Si le résultat est égal à « [] » , alors la classe n’a pas cours et on informe l’utilisateur. Autrement, on peut appeler la fonction AffichageHoraire() avec comme paramètre le résultat.

C’est ici que l’on va stocker les données dans des tableaux et des arraylists pour les afficher à l’écran. Tout d’abord, les tableaux et les arraylists sont créés pour chaque type d’information que l’on va recevoir (Heure début, Heure fin, salle…). ArrayCalculHeure et AlistCalculHeure sont également crée pour pouvoir trier les branches chronologiquement par rapport à leurs heures de début. Une boucle de type for va faire le même nombre de tour que le nombre de branches reçues. Dans cette boucle, on va mettre les données de l’horaire dans les tableaux.

Des jsonObject et jsonArray sont créés pour pouvoir accéder au éléments dont on a besoins.

Ex : ArrayHeureDebut[i] = jsonObjectGeneral.getString("heureDebut");

Il se peut que certaines branches ne possèdent pas de salle (par exemple maitrise de classe), on met alors « / » dans le tableau. Les salles contenant des informations inutiles (par exemple B306-23455) sont découpées pour que seule la partie avant le tiret soit gardée. Pareil avec les professeurs ou seul le nom est gardé. Lorsque toutes les données ont été enregistrées dans les tableaux, on transfert ces données dans des arrayslists. Ça permettra de les trier plus facilement.

Etat Actuel :

Le numéro détermine la branche.

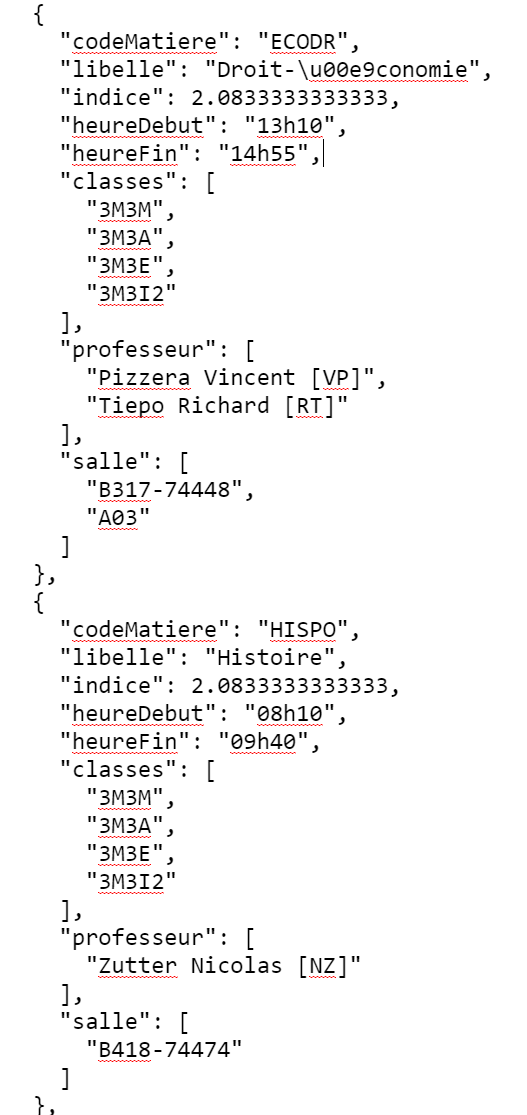
AlistLibelle.get(0) = Allemand

AlistProf.get(0) = Ramseyer  
AlistSalle.get(0) = b410  
AlistLibelle.get(1) = Droit-Economie

AlistProf.get(1) = Pizzera

AlistSalle.get(1) = B317  
….

Les branches envoyées par hypercool ne sont pas dans l’ordre, c’est à moi de trouver un moyen pour savoir quelles branches commencent avant les autres.



Voici une partie des informations de l’horaire de la classe 3m3i2 à la date 03.05.2017. On voit que l’histoire est envoyée après le droit alors qu’il commence à 08h10.

Pour pouvoir les mettre dans l’ordre. J’ai initialement pensé à trier l’heure du début en comparant l’heure puis les minutes, mais j’ai trouvé un moyen plus simple. J’additionne simplement l’heure de début avec l’heure de fin.

Branche reçues du flux JSON :

Allemand : 14h55 -> 16h25 = 14 + 16 = 30.

Economie : 13h10 -> 14h55 = 13 + 14 = 27.

Histoire : 08h10 à 09h40 -> 8 + 9 = 17.

Anglais : 09h40 -> 11h30 = 9 + 11 = 20.

Il me suffit alors de trier du plus petit au plus grand le résultat du calcul. AlistCalcul contient ce résultat. Comme avant, l’item 0 représente la première branche, l’item 2, la deuxième …

Après avoir effectué ce tri, je change l’index des autres Arraylists(AlistSalle, AlistLibelle…) pour qu’il soit dans l’ordre chronologique à l’heure de début.

Exemple :

Avant :

AlistLibelle.get(0) = Allemand

AlistHeureDebut.get(0) = 09h40

AlistCalcul.get(0) = 20   
AlistLibelle.get(1) = Droit-Economie  
AlistHeureDebut.get(1) = 13h10

AlistCalcul.get(1) = 27  
AlistLibelle.get(2) = Histoire

AlistHeureDebut.get(2) = 08h10

AlistCalcul.get(2) = 17  
  
Après :

AlistLibelle.get(0) = Histoire

AlistHeureDebut.get(0) = 08h10

AlistCalcul.get(0) = 17   
AlistLibelle.get(1) = Allemand  
AlistHeureDebut.get(1) = 09h40

AlistCalcul.get(1) = 20  
AlistLibelle.get(2) = Droit-Economie

AlistHeureDebut.get(2) = 13h10

AlistCalcul.get(2) = 27

Une fois les données triées et dans l’ordre, il ne reste plus qu’à les afficher. Pour réaliser cette tâche, j’ai décidé de créer la grille qui va contenir les données dynamiquement. Je me sers du layout « LlGeneral » comme base pour pouvoir ajouter les autres views par-dessus. J’instancie alors les 3 layout verticaux et le layout horizontal pour pouvoir les utiliser plus tard. A cette étape, une dernière vérification est nécessaire. Certaines dates sont valides lors de la première vérification dans la fonction ContenuEditText(). Par exemple 30-04-9999. Si on tape cette date sur hypercool, une erreur est envoyée. Si « error : la date n’est pas valide » est égal au résultat de la requête, alors l’affichage n’est pas lancé. Autrement, on continue. Dans une boucle, qui, comme avant, tourne le même nombre de fois que le nombre de branches reçues, je crée à l’aide des fonctions CreationVerticLayout() et CreationHorizLayout(), 1 layout horizontal et 3 layout verticaux. 6 textview sont également crées en faisant appel à la fonction CreationTextView() avec comme paramètres les données à afficher. La variable bLargeurChampsHeures permet de modifier la largeur des textviews (la colonne de gauche est plus petite que les autres). iCouleurTableau va permettre d’alterner la couleur du tableau. Une fois tous les textviews et layouts crées, je peux assembler le tout. J’ajoute le layout horizontal au LlGeneral, puis les layouts verticaux au layout horizontal. Les textview vont s’ajouter aux layouts verticaux.

### Le planning de livraison

Citez dans le tableau ci-dessus, les étapes du scénario du cas d'utilisation en cours et dans quels délais elles seront réalisées. Il est aussi possible, si vous avez assez de place, d'insérer ici un diagramme de Gantt qui permettra d'ordonner ces différentes tâches dans le temps.

|  |  |
| --- | --- |
| Tâche | délais |
| *Réaliser la structure de la base de données* | 4jours |
|  |  |
|  |  |

### La phase de programmation

Lancement des 2 premières requêtes.

AsyncAffichageHoraire asyncAffichageHoraire0 = new AsyncAffichageHoraire(MainActivity.this, 1);  
asyncAffichageHoraire0.delegate = (AsyncReponse) this;  
asyncAffichageHoraire0.execute("http://devinter.cpln.ch/pdf/hypercool/controler.php?action=ressource&nom=" + strContenuClasse);  
AsyncAffichageHoraire asyncAffichageHoraire2 = new AsyncAffichageHoraire(MainActivity.this, 1);  
asyncAffichageHoraire2.delegate = (AsyncReponse) this;  
asyncAffichageHoraire2.execute("http://devinter.cpln.ch/pdf/hypercool/controler.php?action=ressource&nom=");

Ecriture dans le fichier pour l’historique.

File fChemin = getBaseContext().getFilesDir();  
File fFichier = new File(fChemin, "storage.txt");  
Writer writer;  
// Si c'est vide, on ne met pas dans le fichier. Utile par exemple lors du changement d'orientation si le champ était vide.  
if (!"".equals(strChampsClasse)) {  
 try {  
 writer = new BufferedWriter(new FileWriter(fFichier, true));  
 writer.append(strChampsClasse + "\n");  
 writer.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

Traitement de la requête 2, on remplit AlistLibelleToutesClasses avec le nom de toutes les classes disponibles.

try {  
 reader = new JSONObject(strOutputlisteclasse);  
 Iterator iterator = reader.keys();  
 for (int iCpt = 0; iCpt < 329; iCpt++) {  
 strCode = (String) iterator.next();  
 JSONObject jsonObject = reader.getJSONObject(strCode);  
 AlistLibelleToutesClasses.add(iCpt, jsonObject.getString("nom"));  
 }  
} catch (JSONException e) {  
 e.printStackTrace();  
 // Si cette requete ne répond pas, ca signifie que hypercool n'est pas disponible.  
 Toast.*makeText*(MainActivity.this, "Hypercool hors-ligne, impossible de récupérer les données.", Toast.*LENGTH\_SHORT*).show();  
}

Remplissage des tableaux avec les données reçues.

JSONObject jsonObjectGeneral = jsonArrayGeneral.getJSONObject(i);  
ArrayHeureDebut[i] = jsonObjectGeneral.getString("heureDebut");  
ArrayHeureFin[i] = jsonObjectGeneral.getString("heureFin");  
ArrayLibelle[i] = jsonObjectGeneral.getString("libelle");

Passage des données des tableaux aux arraylists

AlistHeureDebutComplet.add(i, ArrayHeureDebut[i]);  
AlistHeureFinComplet.add(i, ArrayHeureDebut[i]);  
AlistProfesseur.add(i, ArrayProfesseur[i]);  
AlistArraySalle.add(i, ArraySalle[i]);  
AlistLibelle.add(i, ArrayLibelle[i]);

Tri des branches en fonction de l’addition de l’heure de début et de l’heure de fin, et changement d’ordre des données.

// Tri et basculement en fonction du bon ordre des données.  
JSONArray jsonArrayGeneral = null;  
try {  
 jsonArrayGeneral = new JSONArray(strOutput);  
 Collections.sort(AListCalcul); // Tri du calcul càd HeureDebut + HeureFin.  
  
 for (int iBranche = 0; iBranche < jsonArrayGeneral.length(); iBranche++) {  
  
 for (int i10 = 0; i10 < jsonArrayGeneral.length(); i10++) {  
 if (AlistCalculHeure.get(i10).equals(AListCalcul.get(iBranche))) {  
 AlistLibelle.add(iBranche, ArrayLibelle[i10]);  
 AlistHeureDebutComplet.add(iBranche, ArrayHeureDebutComplet[i10]);  
 AlistHeureFinComplet.add(iBranche, ArrayHeureFinComplet[i10]);  
 AlistProfesseur.add(iBranche, ArrayProfesseur[i10]);  
 AlistArraySalle.add(iBranche, ArraySalle[i10]);  
 }  
 }  
 }  
  
} catch (JSONException e) {  
 e.printStackTrace();  
}

Affichage des résultats

// Creation de l'affichage.  
llGeneral.removeAllViews();  
for (int iBranche = 0; iBranche < jsonArrayGeneral.length(); iBranche++) {  
  
 llHoriz = CreationLayoutHoriz();  
 llVert1 = CreationLayoutVertic();  
 llVert2 = CreationLayoutVertic();  
 llVert3 = CreationLayoutVertic();  
 iCouleurTableau++;  
 TextView tv1 = CreationTextView(AlistHeureDebutComplet.get(iBranche));  
 TextView tv2 = CreationTextView(AlistHeureFinComplet.get(iBranche));  
 bLargeurChampsHeures = false;  
 TextView tv3 = CreationTextView(AlistLibelle.get(iBranche));  
 TextView tv4 = CreationTextView(AlistProfesseur.get(iBranche));  
 TextView tv5 = CreationTextView(AlistArraySalle.get(iBranche));  
 TextView tv6 = CreationTextView(" ");  
 bLargeurChampsHeures = true;  
 llGeneral.addView(llHoriz);  
 llHoriz.addView(llVert1);  
 llHoriz.addView(llVert2);  
 llHoriz.addView(llVert3);  
 llVert1.addView(tv1);  
 llVert1.addView(tv2);  
 llVert2.addView(tv3);  
 llVert2.addView(tv4);  
 llVert3.addView(tv5);  
 llVert3.addView(tv6);

### La phase de tests

## Cas d'utilisation « Affichage de l’horaire d’un jour avec classe et date choisie par l’utilisateur »

Une fois l’application ouvert, l’horaire de la dernière classe recherchée avec la date actuelle, est déjà affiché. Pour ce cas d’utilisation, l’utilisateur doit pouvoir modifier la classe et la date voulue. Après avoir fait cela, il appuie sur le bouton Rechercher pour faire apparaitre l’horaire désiré.

Une grosse partie de la réalisation est similaire au use case précédent. Certaines vérifications

### Scénario

1. L’utilisateur tape le nom de la classe voulue dans le champ classe.
2. L’utilisateur tape le nom de la date voulue dans le champ date.
3. L’utilisateur appuie sur le bouton rechercher.
4. Le système vérifie la connexion internet.
5. Le système met dans des variables le contenu des champs, vérification des saisies.
6. Le système affiche le nom du jour de la semaine.
7. Le système établit une connexion avec le serveur et lance les requêtes (Réception id, et liste de toute les classes)
8. Le système effectue des vérifications des données reçues.
9. Le système lance la requête pour avoir l’horaire de la classe avec l’id de la classe.
10. Le système met le nom de la classe recherchée dans l’historique.
11. Le système lit l’historique pour mettre à jour le tableau contenant l’historique.
12. Le système met les données de l’horaire reçues dans des variables.
13. Le système tri les branches de manière chronologique à leurs heures de début.
14. Le système affiche les données.

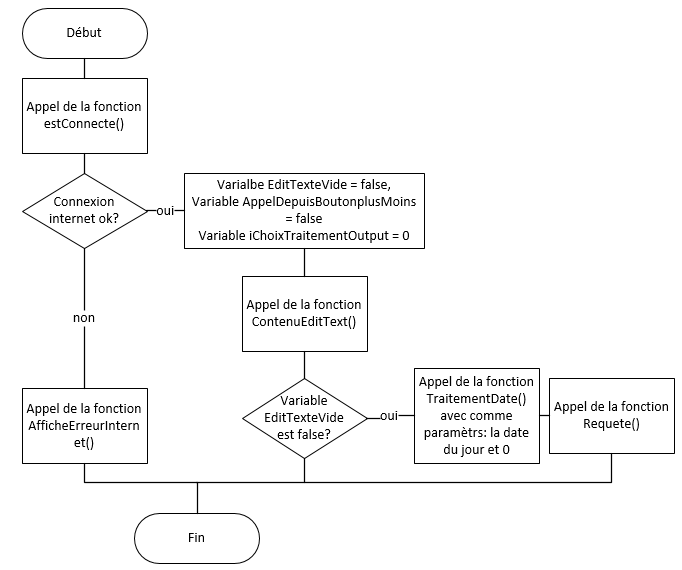
### Marquettes

Même maquette que le cas précédent

### Analyse du scénario

#### Algorithme

Lors de l’appui sur le bouton Rechercher :



#### Explications détaillées

Le développement est très similaire au cas précédent. Avant on mettait la dernière classe recherchée et la date actuel dans les champs. Maintenant, c’est l’utilisateur qui choisit le contenu désiré. Lors du passage dans la fonction ContenuEditText(), la variable bPremiereOuverture est dorénavant égal à false. C’est grâce à ça que cette fois ci, on met le contenu des champs dans les variables (et non plus la dernière classe recherchée et la date actuelle). La variable bAppelDepuisBoutonPlusMoins est égal à false. Elle va servir à savoir si on doit mettre le contenu du champ classe dans l’historique. Les boutons permettant de changer la date de jour/semaine, ont cette variable à true, et cela permet de ne pas enregistrer la classe. bPlusieursClasseValidees passe aussi à true. C’est utilisé pour afficher la liste des classes retournées lorsqu’un nom de classe retourne plusieurs classes (3m3i retourne 3m3i1, 3m3i2, 3m3i3). Elle était false lors du démarrage pour ne pas faire apparaitre la liste, et ainsi ne pas gêner l’utilisateur. IChoixTraitementOutput est égal à 0 pour permettre à nouveau d’aiguiller nos 3 requêtes. Après la fonction ContenuEditText(), un test est effectué pour savoir si un des deux champs est vide. Cette opération n’était pas nécessaire au démarrage de l’application étant donné que les deux champs étaient automatiquement remplis. Une fois ce test passé, on effectue comme avant la fonction TraitementDate() pour afficher le nom du jour. Finalement, la requête est lancée.

La requête s’effectue de la même manière. Il y a désormais un changement dans la fonction RetourOutput(). Si une seule id est renvoyée, alors le programme suit le même acheminement que le cas d’utilisation précédent. Si par contre, plusieurs classes sont reçues, alors on va passer par la condition qui vérifiait le nombre de classes reçues. Cette fois ci, bPlusieursClasseValidees est false, alors qu’au démarrage ce n’était pas le cas. Dans cette condition, on va avertir l’utilisateur que plusieurs classes ont été trouvées. Grâce à MiseEnPlaceActv(), on va changer la source des suggestions du champ. Maintenant les noms des classes trouvées sont suggérés. Pour forcer l’utilisateur à choisir une classe présente dans la liste, on masque le clavier et on déroule la liste des propositions du champ.

Toute la partie traitement des données et affichage est similaire au cas précédent.

### Le planning de livraison

### La phase de programmation

Ouverture de cette condition grâce au bPlusieursClasseValidees.

if (iCptNombreClasse > 1 && !bPlusieursClasseValidees) {  
 Toast.makeText(MainActivity.this, iCptNombreClasse + " classes trouvées, veuillez choisir la bonne classe", Toast.LENGTH\_LONG).show();  
 bAfficherContenuActv = true;  
 MiseEnPlaceActv(AlistPlusieursClasses);  
 bPlusieursClasseValidees = true;  
 AutoCompleteTextView actvClasse = (AutoCompleteTextView) findViewById(R.id.ActvClasse);  
 actvClasse.performClick();  
   
}

### La phase de tests

## Cas d'utilisation « Affichage de l’horaire de la semaine directement au passage à la vue semaine »

Dès que l’utilisateur veut voir l’horaire de la semaine, il appuie sur le menu, puis « Vue Semaine ». L’horaire de la dernière classe recherchée doit directement s’afficher sans que l’utilisateur ait à taper quelque chose.

### Scénario

Décrire les différentes étapes par des phrases concises :

1. L’utilisateur appuie sur le bouton « Vue Semaine » depuis l’activité vue jour.
2. Le système masque le clavier
3. Le système passe l’application en mode paysage
4. Le système remplit le champ classe avec la dernière classe recherchée
5. Le système remplit le champ date avec la date du jour actuel.
6. Le système met dans des variables le contenu des champs, vérification des saisies.
7. Le système récupère les dates de la semaine
8. Le système établit une connexion avec le serveur et lance les requêtes (Réception id, et liste de toute les classes)
9. Le système effectue des vérifications des données reçues.
10. Le système lance la requête pour avoir l’horaire de la classe avec l’id de la classe.
11. Le système met le nom de la classe recherchée dans l’historique.
12. Le système lit l’historique pour mettre à jour le tableau contenant l’historique.
13. Le système met les données de l’horaire reçues dans des variables.
14. Le système tri les branches de manière chronologique à leurs heures de début.
15. Le système affiche l’horaire.

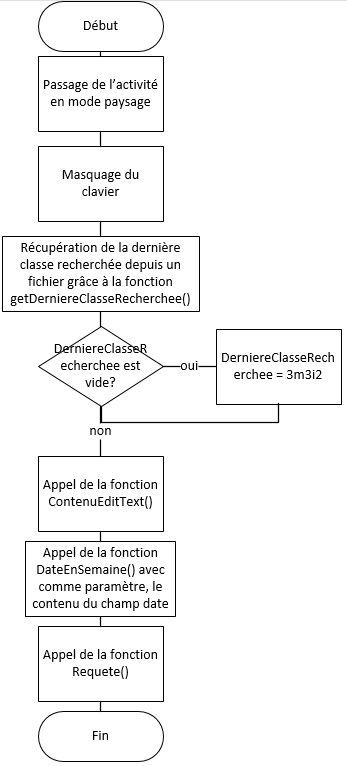
### Maquettes

Insérer des maquettes expliquant comment le scénario retenu pourra être mis en œuvre au niveau de l’interface homme-machine.

### Analyse du scénario

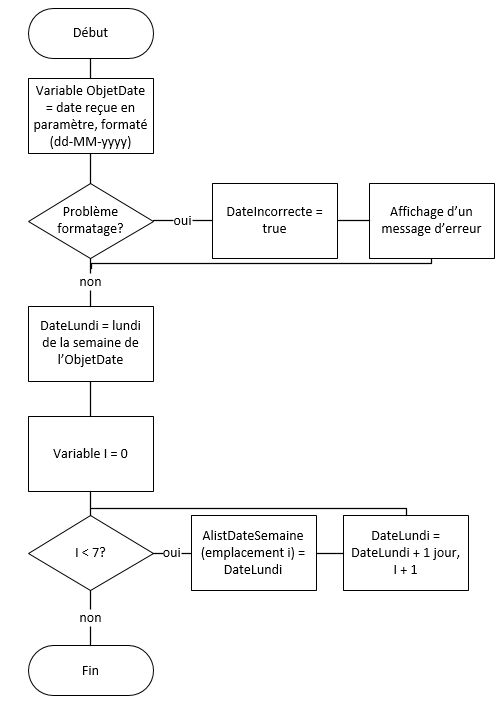
#### Algorithme ou Structogramme

Lors de la création de l’activité :



Fonction ContenuEditText() : Pareil que celui de la vue jour.

Fonction DateEnSemaine() :



Fonction Requete() : Pareil que celui de la vue jour.

Class Asynctask : Pareil que celui de la vue jour.

Fonction RetourOutput() :

#### Explications détaillées

Le déroulement de la vue semaine ressemble à la vue jour.

Tout d’abord, l’application passe en mode paysage. Cette opération est effectuée pour gagner de la place pour pouvoir afficher les données. En mode portrait, la place n’est pas suffisante pour pouvoir afficher les données de la semaine entière. Le clavier est à nouveau masqué. On fait de nouveau appel à la fonction getDerniereClasseRecherchee(). Si la valeur reçu par le fichier est null, alors on attribut en dur une valeur. Cette situation n’est pas censé arriver, mais ça permet de ne pas faire crash l’application si ça se produit. ContenuEditText() va mettre la dernière classe recherchée et la date du jour dans les champs. On fait ensuite appel à la fonction DateEnSemaine(strContenuDate). Elle permet de mettre dans un tableau toutes les dates de la semaine en commençant depuis le lundi. Ces dates vont être utilisées lors des requêtes. Finalement, on appelle Requete(). Comme dans le cas d’utilisation précédent, elle exécute les mêmes requêtes et renvoie les résultats dans la fonction RetourOutput(). Dans cette fonction, il y a de nouveau la vérification pour savoir si plusieurs classes sont reçues. Lors du démarrage, bPlusieursClasseValidees est false, et donc on ne passe pas par ce chemin. Dans le else(), on fait cette fois-ci 5 requêtes. La première requête va retourner l’horaire pour le lundi, la suivante le mardi, et ainsi de suite jusqu’au vendredi. Après avoir fait ces requêtes, on écrit dans le fichier historique et on met à jour les suggestions du champ classe. A chaque fois que l’on reçoit les résultats des requêtes, on place les données reçues dans un tableau. Une fois que toutes les requêtes ont été exécutées (iChoixTraitementOutput==6), on fait appel à la fonction AffichageHoraire en envoyant en paramètre, les données des 5 jours de la semaine.

Dans cette fonction, on va créer les Tableaux et les arraylists pour contenir toutes nos données. Cette fois-ci, ils sont en deux dimensions et vont stocker les données de cette manière :

ArrayLibelle[Le numero du jour] [Le numero de la branche] = Allemand

Lundi = 0

Mardi = 0

…

Cette fois on a 2 boucles. La première va faire le même nombre de tour qu’il y a de branches reçues. Et la deuxième va faire le nombre de jour, donc 5.

Dans ces boucles on va remplir nos tableaux avec les données.

Ex :

**arrayLibelle[iJour][iNbBranche] = jsonObjectGeneral.getString(«libelle »)**

Tout d’abord, on commence par remplir le lundi. Les données de la première branche seront stocké dans l’emplacement [0][0]. La deuxième [0][1] et ainsi de suite. Une fois que l’on a fait le tour de toutes les branches du jour, on change de jour. Les données de la première branche du mardi seront stocké cette fois-ci dans l’emplacement [1][0].

A la fin des boucles, on transfert de nouveau les données des tableaux dans des Arrayslists pour pouvoir les trier plus facilement.

Une fois toutes les données placées dans des tableaux, on va trier les tableaux afin d’avoir les branches de façon chronologique à leurs heures de début. La même méthode est utilisée que dans la vue jour. On additionne l’heure de début avec l’heure de fin, et on tri du plus petit au plus grand.

Avec ces données, nous pouvons passer à l’affichage.

La méthode actuelle que j’ai utilisé n’est pas la plus optimisée. J’ai décidé de créer le tableau en dur dans le xml. Chaque case du tableau contient un LinearLayout Vertical, et 2 TextView. J’ai alors copié/collé les cases en changeant l’id à chaque fois. C’est une méthode pas très propre à faire mais au moins, je pouvais exactement personnaliser comme je voulais toutes les cases. Lors de la création de l’affichage dans la vue jour, j’ai eu beaucoup de mal à réaliser exactement ce que je voulais faire au niveau design en créant dynamiquement les layouts. J’ai alors décidé de ne pas utiliser cette méthode pour l’affichage de la vue semaine. En revanche, j’aurais pu essayer de le faire à la fin du projet si le temps le permettait.

La tableau arrayPeriodes est créé et contient l’heure de début des périodes (exemple : 08h10,08h55…)

J’instancie un tableau à 2 dimensions (llGeneral) de type LinearLayout. Pareil avec les tableaux tvGeneralLibelle, et tvGeneralSalle.

Grâce à des boucles, je vais mettre chaque Views des cases (du xml) dans les tableaux correspondants (utilisation de findViewById).

L’id de ces views ont tous la même structure ce qui me permet de les mettre facilement dans les tableaux à deux dimensions avec des boucles.

Exemple :

ll\_Mardi\_14h55

tv\_Mardi\_14h55\_libelle

tv\_Mardi\_14h55\_salle

ll\_Mercredi\_14h55

tv\_Mercredi\_14h55\_libelle

tv\_Mercredi\_14h55\_salle

Après avoir fait cela, je vide le contenu de toutes ces views grâce à deux boucles. Sinon, les données vont rester dans les cases et vont entrer en conflit avec les nouvelles données d’une nouvelle requête. Je colore également les cases de midi d’une couleur différente.

La fonction RemplissageCases() va permettre de mettre les données à afficher dans les TextView, d’ajouter une couleur aux branches, de couper les libelles trop long et de centrer le contenu des cases. Pour faire cela, elle a besoins de l’heure de début de la branche et de savoir quelles cases elle doit remplir.

Afin de donner ces informations, une boucle est utilisée pour les lui envoyer. Le principe c’est qu’on va tester si l’heure de début de la branche est égal à une période présente dans le tableau arrayPeriodes[]. Si elle est égale, on envoie le contenu du compteur de la boucle à la fonction RemplissageCases(). Le compteur fait 19 tours et correspond aux cases du tableau. (0 = 07h30 – 08h10, 1 = 08h10 – 08h55). Ce chiffre permettra à la fonction de savoir quelles cases elle doit remplir.

Cette opération est effectuée pour chaque branche de chaque jour avec 2 boucles supplémentaires.

La fonction récupère ces informations et va les afficher à l’écran. Elle se sert du tableau arrayPeriodes[] pour calculer le nombre de période que dure la branche.

Par exemple :

Allemand : 14h55 – 16h25

Index de 14h55 dans le tableau = 9

Index de 16h25 dans le tableau = 11

11-9 = 2 périodes.

Si la branche ne dure qu’une seule période, alors on va afficher le code matière et la salle.

Si la branche dure plus de 2 périodes, alors on va afficher le libelle, la salle et le professeur. Si le libelle est trop long (plus de 15 caractères) alors on va le découper en 2 parties et les afficher dans 2 textviews différents. Les libellées des matières reçues sont enregistrées dans le tableau libelleBranche[]. Cela va permettre d’attribuer une couleur différente pour chacune des branches. On a juste à tester si le libelle reçu est déjà présent dans le tableau, et si c’est le cas, on attribue la même couleur.

Il reste encore les heures commençant à des heures non-conventionnelles. Certaines classes ont cours, par exemple à 08h30 au lieu de 08h10. Cela crée un problème avec ma fonction de test actuel. Les heures non-conventionnelles ne sont pas présent dans le tableau arrayPeriodes[], et donc je ne peux pas les repérer. Pour m’occuper de ces cas-là, je crée manuellement un tableau qui va contenir toutes les heures possibles.

Exemple :

Heure de base : 08h10

arrayPeriodeEntre[1][0] = 08h15

arrayPeriodeEntre[1][1] = 08h20

…

Heure de base : 08h55

arrayPeriodeEntre[2][0] = 09h00

arrayPeriodeEntre[2][1] = 09h05

Grace à ce nouveau tableau, je peux effectuer le test similaire au précédent. Si l’heure de début de la branche est égale à une des périodes présentes dans le tableau arrayPeriodeEntre, alors on peut envoyer le numéro de la case qu’elle doit remplir à la fonction RemplissageCases2(). Quatre boucles for composent ce test car il faut passer par les deux dimensions du tableau, pour toutes les branches de chaque jours. L’heure de début n’est pas envoyé à la fonction RemplissageCases2() car avec la méthode utilisée auparavant, il m’est impossible de savoir le nombre de périodes que durent la branche. Je n’ai pas de tableau ayant toutes les heures non conventionnelles. arrayPeriodeEntre[] ne peux pas être utilisé car il contient les heures à intervalle de 5 minutes. Par manque de temps, je n’ai pas pu me pencher d’avantage sur ce problème. Un autre moyen de faire serait par exemple de calculer la différence en minute entre l’heure de début et l’heure de fin de la branche, et ainsi essayer d’en déduire le nombre de période. RemplissageCases2() fonctionne de manière similaire à la première fonction. Elle vérifie le nom du libelle pour savoir si elle doit mettre la même couleur, et affiche les données d’une période cette fois ci, entre les cases du tableau.

Un des problèmes qui se pose avec ma méthode pour afficher les données est que, lorsque le libelle d’une branche est trop long, le textview (contenant le libelle) passe sur 2 lignes et décale tout le tableau. Je devais trouver des moyens pour diminuer la taille et gagner de la place. Tout d’abord j’ai décidé d’afficher le code matière au lieu du libelle lorsque la branche dure une seule période. De cette manière, lorsque la branche dure plus d’un période, je découpe le libelle en 2 parties quand c’est trop long. J’enlève aussi les majuscules après la première lettre. La taille du texte est également diminuée pour être de 13 DP.

### Le planning de livraison

Citez dans le tableau ci-dessus, les étapes du scénario du cas d'utilisation en cours et dans quels délais elles seront réalisées. Il est aussi possible, si vous avez assez de place, d'insérer ici un diagramme de Gantt qui permettra d'ordonner ces différentes tâches dans le temps.

|  |  |
| --- | --- |
| Tâche | délais |
| *Réaliser la structure de la base de données* | 4jours |
|  |  |
|  |  |

### La phase de programmation

Lancement des 5 requêtes

// Si tout est bon, lancement des requêtes.  
AsyncAffichageHoraire[] asyncGeneral = new AsyncAffichageHoraire[8];  
for (int i = 0; i < 5; i++) {  
 asyncGeneral[i] = new AsyncAffichageHoraire(Activite\_VueSemaine.this, 0);  
 asyncGeneral[i].delegate = (AsyncReponse) this;  
 asyncGeneral[i].execute("http://devinter.cpln.ch/pdf/hypercool/controler.php?action=horaire&ident=" + strId + "&sub=date&date=" + AlistDateSemaine.get(i));  
  
}

Ajout des données dans les tableaux (une partie)

for (int iJour = 0; iJour < 5; iJour++) {  
  
  
 JSONArray jsonArraygeneral2 = new JSONArray(output[iJour]);  
 for (int iNbBranche = 0; iNbBranche < jsonArraygeneral2.length(); iNbBranche++) {  
  
 JSONArray jsonArrayGeneral = new JSONArray(output[iJour]);  
  
  
 JSONObject jsonObjectGeneral = jsonArrayGeneral.getJSONObject(iNbBranche);  
 arrayLibelle[iJour][iNbBranche] = jsonObjectGeneral.getString("libelle");  
 arrayLibelle[iJour][iNbBranche] = arrayLibelle[iJour][iNbBranche].substring(0, 1).toUpperCase() + arrayLibelle[iJour][iNbBranche].substring(1).toLowerCase();  
 arrayCodeMatiere[iJour][iNbBranche] = jsonObjectGeneral.getString("codeMatiere");  
 arrayHeureDebut[iJour][iNbBranche] = jsonObjectGeneral.getString("heureDebut");  
 arrayHeureFin[iJour][iNbBranche] = jsonObjectGeneral.getString("heureFin");

Passage des données des tableaux aux arraylists

alistGeneralProf[iJour].add(iNbBranche, arrayProf[iJour][iNbBranche]);  
alistGeneralSalle[iJour].add(iNbBranche, arraySalle[iJour][iNbBranche]);  
alistGeneralLibelle[iJour].add(iNbBranche, arrayLibelle[iJour][iNbBranche]);  
alistGeneralHeureDebutComplet[iJour].add(iNbBranche, arrayHeureDebut[iJour][iNbBranche]);  
alistGeneralHeureFinComplet[iJour].add(iNbBranche, arrayHeureFinComplet[iJour][iNbBranche]);  
alistCalcul[iJour].add(iNbBranche, arrayCalcul[iJour][iNbBranche]);  
alistGeneralCodeMatiere[iJour].add(iNbBranche, arrayCodeMatiere[iJour][iNbBranche]);

Changement de place des données

for (int iJour = 0; iJour < 5; iJour++) {  
 for (int i20 = 0; i20 < NombreInfoRecu[iJour]; i20++) {  
 for (int i10 = 0; i10 < NombreInfoRecu[iJour]; i10++) {  
  
 if (alistCalculHeure[iJour].get(i10).equals(alistCalcul[iJour].get(i20))) {  
 alistGeneralLibelle[iJour].add(i20, arrayLibelle[iJour][i10]);  
 alistGeneralHeureDebutComplet[iJour].add(i20, arrayHeureDebutComplet[iJour][i10]);  
 alistGeneralHeureFinComplet[iJour].add(i20, arrayHeureFinComplet[iJour][i10]);  
 alistGeneralProf[iJour].add(i20, arrayProf[iJour][i10]);  
 alistGeneralSalle[iJour].add(i20, arraySalle[iJour][i10]);  
 alistGeneralCodeMatiere[iJour].add(i20, arrayCodeMatiere[iJour][i10]);  
 }  
 }  
 }  
}

Ajout des views Libelle dans le tableau.

TextView[][] tvGeneralLibelle = new TextView[200][200];  
for (int i = 0; i < 5; i++) {  
 for (int i2 = 0; i2 < 19; i2++) {  
 String Idtv = "tv\_" + arrayJourSemaine[i] + "\_" + arrayPeriodes[i2] + "\_libelle";  
 int resID = getResources().getIdentifier(Idtv, "id", getPackageName());  
 tvGeneralLibelle[i][i2] = (TextView) findViewById(resID);  
 tvGeneralLibelle[i][i2].setTextSize(TypedValue.COMPLEX\_UNIT\_DIP, 13);  
 }  
}

Envoi des paramètres à la fonction RemplissageCases()

for (int iJour = 0; iJour < 5; iJour++) {  
 for (int iBranche = 0; iBranche < NombreInfoRecu[iJour]; iBranche++) {  
 for (int iCpt = 0; iCpt < 19; iCpt++) {  
 if (alistGeneralHeureDebutComplet[iJour].get(iBranche).equals(arrayPeriodes[iCpt])) {  
 int iNumeroll = iCpt;  
 String strHeureDebut = arrayPeriodes[iCpt];  
 RemplissageCases(iJour, iBranche, iNumeroll, strHeureDebut, llGeneral, arrayPeriodes, alistGeneralHeureFinComplet, tvGeneralLibelle, tvGeneralSalle, alistGeneralLibelle, alistGeneralSalle, alistGeneralProf, alistGeneralCodeMatiere, arrayCouleurs);  
 }  
 }  
 }  
}

Envoi des paramètres à la fonction RemplissageCases2()

for(int iJour=0;iJour<5;iJour++) {  
 for (int iBranche = 0; iBranche < NombreInfoRecu[iJour]; iBranche++) {  
 for (int i2 = 0; i2 < 19; i2++) { // le nombre de périodes au total.  
 for (int i = 0; i < 8; i++) { // 8 = le nombre de la seconde dimension du tableau.  
 if(alistGeneralHeureDebutComplet[iJour].get(iBranche).equals(arrayPeriodeEntre[i2][i])){  
 int iNumeroll = i2;  
 RemplissageCases2(iJour, iBranche, iNumeroll,tvGeneralLibelle,tvGeneralSalle,alistGeneralLibelle,alistGeneralSalle,alistGeneralCodeMatiere,arrayCouleurs);  
 }  
 }  
 }  
 }  
}

Une partie de la fonction RemplissageCase()

int iNumeroll = iNumll;  
int iNombreCharMax = 15; // APrès ce nombre de caractère, on va coupé le libelle.  
llGeneral[iJour][iNumeroll].setBackgroundColor(arrayCouleurs[val2]);  
int iIndex1 = 0;  
int iIndex2 = 0;  
int iIndexFinal = 0;  
iIndex1 = Arrays.asList(arrayPeriode).indexOf(strHeureDebut);  
String strHeureFin = AlistHeureFin[iJour].get(i100);  
iIndex2 = Arrays.asList(arrayPeriode).indexOf(strHeureFin);  
iIndexFinal = iIndex2 - iIndex1;  
tvLibelle[iJour][iNumeroll].setText(AlistCodeMatiere[iJour].get(i100));  
tvSalle[iJour][iNumeroll].setText(AlistSalle[iJour].get(i100));  
int iLength = AlistLibelle[iJour].get(i100).length();  
String strPartie1 = "";  
String strPartie2 = "";  
if (iLength > iNombreCharMax) {  
 strPartie1 = AlistLibelle[iJour].get(i100).substring(0, iNombreCharMax);  
 strPartie2 = AlistLibelle[iJour].get(i100).substring(iNombreCharMax);  
}  
  
for (int i = 1; i < iIndexFinal; i++) {  
 llGeneral[iJour][iNumeroll + i].setBackgroundColor(arrayCouleurs[val2]);  
  
}  
  
if (iIndexFinal == 2) {  
 if (iLength > iNombreCharMax) {  
  
 tvLibelle[iJour][iNumeroll].setText(strPartie1 + "-");  
 tvSalle[iJour][iNumeroll].setText(strPartie2);  
 tvLibelle[iJour][iNumeroll + 1].setText(AlistSalle[iJour].get(i100));  
 tvSalle[iJour][iNumeroll + 1].setText(AlistProf[iJour].get(i100));  
 } else {  
 tvLibelle[iJour][iNumeroll].setText(AlistLibelle[iJour].get(i100));  
 tvLibelle[iJour][iNumeroll + 1].setText(AlistProf[iJour].get(i100));  
 }  
}

RemplissageCases2()

tvSalle[iJour][iNumll].setText(AlistCodeMatiere[iJour].get(i100));  
tvLibelle[iJour][iNumll+1].setText(AlistSalle[iJour].get(i100));  
tvSalle[iJour][iNumll].setBackgroundColor(arrayCouleurs[val2]);  
tvLibelle[iJour][iNumll+1].setBackgroundColor(arrayCouleurs[val2]);

### La phase de tests

Expliquer les tests réalisés et le protocole de test. De plus faite TOUJOURS tester votre application par un utilisateur « lambda », il fera des manipulations que vous ne ferez jamais et ce sera bien ce type d’utilisateur qui utilisera votre application. Noter qui était le testeur et ce qu’il a découvert comme problème ou disfonctionnement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test à effectuer** | | **Résultat**  **escompté** | **Résultat**  **obtenu** | **Constatation** |
| Généralités | | | | |
|  |  |  |  |  |
|  |  |  |  |

## Cas d'utilisation « Affichage de l’horaire d’une semaine avec classe et date choisie par l’utilisateur »

Lors de l’ouverture de la vue semaine, la recherche est automatiquement exécutée. L’utilisateur doit pouvoir choisir la classe et la date qu’il veut, puis à l’appui du bouton Rechercher, le programme doit afficher l’horaire de la semaine correspondent.

### Scénario

Décrire les différentes étapes par des phrases concises :

1. L’utilisateur ….
2. le système…

### Maquettes

Insérer des maquettes expliquant comment le scénario retenu pourra être mis en œuvre au niveau de l’interface homme-machine.

### Analyse du scénario

#### Algorithme ou Structogramme

Sur la base du scénario, identifiez les « méthodes » à réaliser.

#### Explications détaillées

Sur la base du structogramme ou de l’algorithme, insérer l’explication utile qui détaille le scénario au niveau de la programmation.

### Le planning de livraison

Citez dans le tableau ci-dessus, les étapes du scénario du cas d'utilisation en cours et dans quels délais elles seront réalisées. Il est aussi possible, si vous avez assez de place, d'insérer ici un diagramme de Gantt qui permettra d'ordonner ces différentes tâches dans le temps.

|  |  |
| --- | --- |
| Tâche | délais |
| *Réaliser la structure de la base de données* | 4jours |
|  |  |
|  |  |

### La phase de programmation

Insérer le code produit commenté en courrier new 9pt encadré. Contentez-vous des parties cruciales…

### La phase de tests

Expliquer les tests réalisés et le protocole de test. De plus faite TOUJOURS tester votre application par un utilisateur « lambda », il fera des manipulations que vous ne ferez jamais et ce sera bien ce type d’utilisateur qui utilisera votre application. Noter qui était le testeur et ce qu’il a découvert comme problème ou disfonctionnement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test à effectuer** | | **Résultat**  **escompté** | **Résultat**  **obtenu** | **Constatation** |
| Généralités | | | | |
|  |  |  |  |  |
|  |  |  |  |

## Cas d'utilisation « Sélecteur de date »

En appuyant sur le bouton calendrier, l’application doit afficher un sélecteur de date pour que l’utilisateur puisse choisir plus facilement la date voulue. Une fois la date choisie, on la mettra dans le champ date. Ce bouton est présent dans les deux activités.

### Scénario

Scénario Vue Jour :

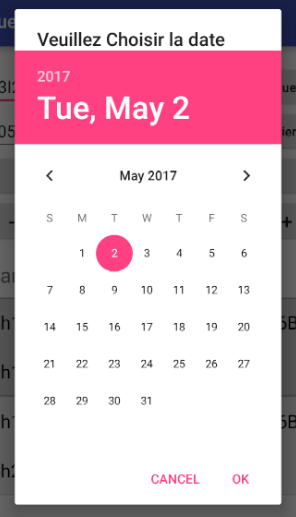
1. L’utilisateur appuie sur le bouton calendrier.
2. L’application affiche un sélecteur de date.
3. L’utilisateur choisit la date voulue et appuie sur « Ok »
4. L’application met la date choisie dans le champ date.

Scénario Vue Semaine :

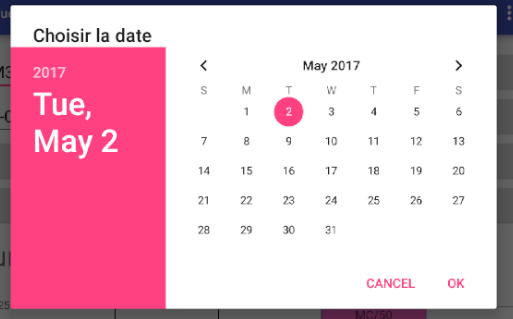
Même Scénario que la vue jour.

### Marquettes

Vue jour :



Vue Semaine :

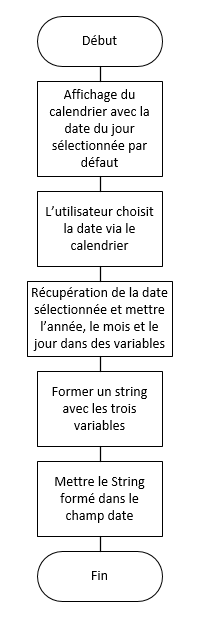


### Analyse du scénario

#### Algorithme

Vue Jour :

Lors du clic sur le bouton Calendrier.



Vue Semaine :

Même algorithme que la vue jour.

#### Explications détaillées

Activité jour :   
Dans le onCreate() de l’activité, on met un listener de clic sur le bouton Rechercher.

Lors du clic sur le bouton on affiche un DatePickerDialog. La date du jour actuel est sélectionnée par défaut. Une fois que l’utilisateur a choisi la date, on place la date choisie dans des variables. Avec ces variables Jour, Mois et Année, on peut créer un string formaté de la bonne manière (séparé par des tirets) qu’on placera dans le champ date.

Activité semaine :

Même explication que la vue jour.

### Le planning de livraison

Vue jour :

|  |  |
| --- | --- |
| Tâche | délais |
| Afficher le calendrier lors du clic sur le bouton | 1 période |
| Mettre la date sélectionnée dans le champ date | 1 période |

Vue semaine :

|  |  |
| --- | --- |
| Tâche | délais |
| Copier /coller du code présent dans la vue jour en modifiant l’id du bouton Calendrier | 0 |

### La phase de programmation

Vue jour :

Création du string qui va être placé dans le champ classe. Le numéro du mois commençait par 0, alors on doit incrémenter de 1.

String strDateComplete = String.*valueOf*(iJour) + "-" + String.*valueOf*(iMois + 1) + "-" + String.*valueOf*(iAnnee);

Vue semaine :

Même Programmation que la vue jour.

### La phase de tests

Vue Jour :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N° | Description | Actions à tester | Résultats attendus | Statut |
| 1. | Vérifier que la date par défaut du datepicker est la date du jour actuel. | Appuyer sur le bouton Calendrier | Date du jour actuel par défaut |  |
| 2 | Mise en place de la date choisie via le datepicker dans le champ date. | Appuyer sur le bouton Calendrier, puis choisir une date et appuyer sur « Ok » | Date sélectionnée dans le champ date. |  |

Vue semaine :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N° | Description | Actions à tester | Résultats attendus | Statut |
| 1. | Vérifier que la date par défaut du datepicker est la date du jour actuel. | Appuyer sur le bouton Calendrier | Date du jour actuel par défaut |  |
| 2 | Mise en place de la date choisie via le datepicker dans le champ date. | Appuyer sur le bouton Calendrier, puis choisir une date et appuyer sur « Ok » | Date sélectionnée dans le champ date. |  |

## Cas d'utilisation « Basculer entre la vue semaine et la vue jour »

Laisser la possibilité à l’utilisateur de changer de vue. Le bouton pour effectuer cette action se trouve dans le menu en haut à droite.

### Scénario

Vue jour :

1. L’utilisateur appuie sur le bouton menu
2. L’utilisateur appuie sur le bouton « Vue semaine »
3. Le système vérifie si la connexion internet est disponible
4. Le système passe à la vue semaine

Vue semaine :

1. L’utilisateur appuie sur le bouton menu
2. L’utilisateur appuie sur le bouton « Vue jour »
3. Le système passe à la vue jour.

### Maquettes

Vue jour :

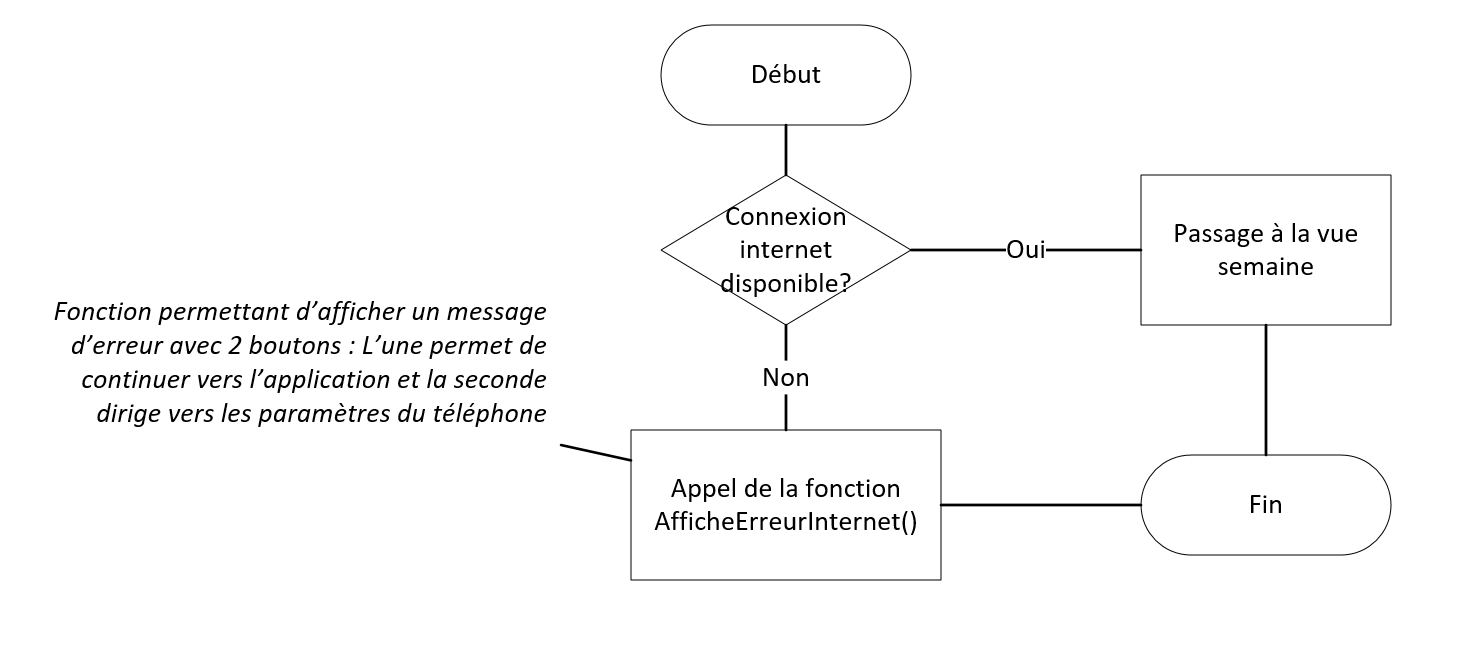
Vue semaine :

### Analyse du scénario

#### Algorithme ou Structogramme

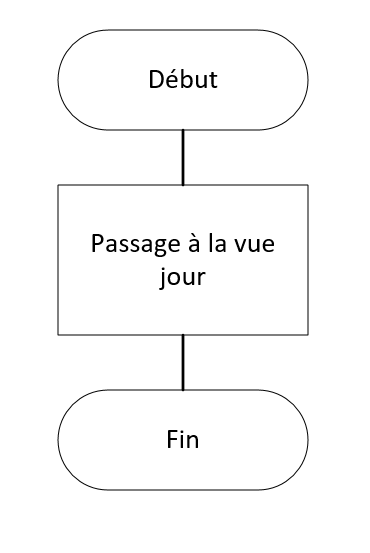
Vue jour :

Fonction VueSemaine()



Vue semaine :

Fonction VueJour()



#### Explications détaillées

Vue jour :

Lorsque l’utilisateur appuie sur le bouton « Vue semaine » depuis le menu, on va vérifier que l’utilisateur ait bien une connexion internet. Si c’est le cas, alors on passe à la vue Semaine. Autrement un message d’erreur permettant d’’accéder aux paramètres apparait.

Vue Semaine :

La vérification internet n’est pas nécessaire ici et le bouton « Vue jour » permet directement de passer à la vue jour.

### Le planning de livraison

Vue jour :

|  |  |
| --- | --- |
| Tâche | délais |
| Créer les fichiers xml nécessaires aux menus | 1 période |
| Créer les fonctions pour changer de vue | 1 période |

Vue semaine :

|  |  |
| --- | --- |
| Tâche | délais |
| Copier/Coller du fichier xml en modifiant le libelle du choix (« Vue jour » au lieu de « Vue semaine »). Ajout du code pour mettre en place le menu et du passage à l’activité vu jour. (très similaire au précèdent) | 0 période |

### La phase de programmation

Vue jour :

Création du menu avec le fichier menu.xml

public boolean onCreateOptionsMenu(Menu menu){

getMenuInflater().inflate(R.menu.menu, menu);

return true;

}

Menu.xml

Contient les 2 options possibles du menu, la deuxième option est documenté dans le chapitre « lol »

<menu xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto">

<item

android:id="@+id/action\_changer\_vue"

android:title="Vue semaine"

app:showAsAction="never" /

<item

android:id="@+id/action\_effacer"

android:title="Effacer les recherches"

app:showAsAction="never" />

</menu>

Lancement de la fonction VueSemaine() lorsque que l’utilisateur appuie sur la bouton « Vue Semaine »

public boolean onOptionsItemSelected(MenuItem item){

switch (item.getItemId()){

case R.id.action\_changer\_vue:

VueSemaine();

return true;

…

Vue Semaine :

Création du menu avec le fichier menu2.xml. Ce fichier-là affiche « Vue jour » dans le menu alors que menu.xml affiche « Vue Semaine ».

public boolean onCreateOptionsMenu(Menu menu){

getMenuInflater().inflate(R.menu.menu2, menu);

return true;

}

### La phase de tests

Vue jour :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N° | Description | Actions à tester | Résultats attendus | Statut |
| 1. | Passage à la Vue semaine. | Appuyer sur le menu, puis « Vue semaine » | Passage à la vue semaine |  |
| 2 | Tentative de passer à la vue semaine en n’ayant pas de connexion internet. | Activer le mode avion, appuyer sur le menu puis « Vue Semaine » | Message d’erreur |  |

Vue Semaine :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N° | Description | Actions à tester | Résultats attendus | Statut |
| 1. | Passage à la Vue jour. | Appuyer sur le menu, puis « Vue jour » | Passage à la vue semaine |  |

## Cas d'utilisation « Afficher l’historique des recherches »

L’utilisateur doit pouvoir accéder à une liste des classes déjà recherchées. A chaque requête que l’utilisateur fait depuis le bouton rechercher, le contenu du champ de la classe est inscrit dans un fichier. Les deux activités se partagent le fichier et peuvent y accéder.

### Scénario

Vue jour :

1. L’utilisateur appuie sur le bouton historique.
2. Le système met à jour l’actv avec le tableau contenant le l’historique des recherches
3. Le système affiche l’actv.
4. Le système change le texte du bouton.

Vue semaine :

Même scénario que la vue jour.

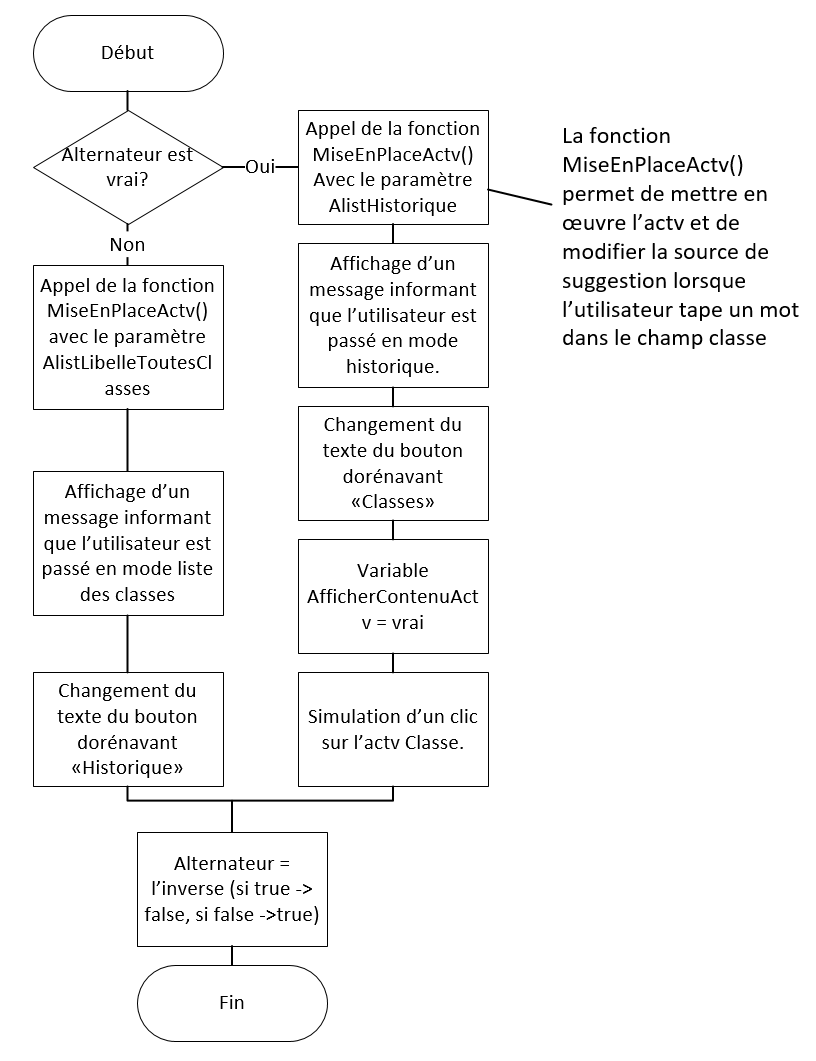
### Maquettes

Insérer des maquettes expliquant comment le scénario retenu pourra être mis en œuvre au niveau de l’interface homme-machine.

### Analyse du scénario

#### Algorithme ou Structogramme

Vue jour :



Vue Semaine :

Même Algorithme que la vue Jour.

#### Explications détaillées

Tout d’abord, dès la création de l’activité, la variable bAlternateur est égale à true. Cette variable sert à déterminer quel Arraylist on va utiliser pour alimenter l’actv. Quand l’utilisateur va cliquer la première fois sur le bouton historique, la source de l’actv va être modifier pour suggérer la liste des classes déjà recherchées. Un message d’information va apparaitre pour notifier l’utilisateur du changement de mode. Le texte du bouton va être modifier pour devenir « Classes ». L’utilisateur, en appuyant sur ce bouton, s’attend à une liste des classes effectuées. Pour cela, la variable AfficherContenuActv va devenir true. Ça permet, au clic du champ classe, d’afficher tout le contenu du tableau qui alimente l’actv. Il ne reste plus qu’à simuler un clic sur le champ pour ainsi afficher la liste.

Si au clic du bouton, bAlernateur est égal à false, alors ça signifie que l’utilisateur est actuellement en mode historique. Dans ce cas, l’actv va être alimenté par le tableau contenant la liste de toutes les classes disponibles. Comme avant, le bouton change de texte et un message informatif apparait.

Après toutes ces étapes, bAlternateur va devenir l’inverse de ce qu’il était. Ça permet, lors du prochain appui sur le bouton, de passer par l’autre chemin, et d’alterner le mode d’affichage.

Vue Semaine :

Même explication que la vue jour.

### Le planning de livraison

|  |  |
| --- | --- |
| Tâche | délais |
| Création du bouton alternateur et du contenu du listener du clic. | 2 Périodes. |

### La phase de programmation

### La phase de tests

## Cas d'utilisation « Application d’adaptant à la taille de l’écran »

L’application doit être adapté pour toutes tailles de terminaux. Les boutons, et les champs doivent et surtout l’affichage de l’horaire d’adapteront à la taille pour être agréable à utiliser pour l’utilisateur.

### Scénario

Vue jour :

1. L’utilisateur ouvre l’application
2. Le système adapte les tailles des boutons, champs et de l’affichage des données

Vue Semaine :

Même scénario que la vue jour.

### Maquettes

Insérer des maquettes expliquant comment le scénario retenu pourra être mis en œuvre au niveau de l’interface homme-machine.

### Analyse du scénario

#### Algorithme ou Structogramme

Sur la base du scénario, identifiez les « méthodes » à réaliser.

#### Explications détaillées

Sur la base du structogramme ou de l’algorithme, insérer l’explication utile qui détaille le scénario au niveau de la programmation.

### Le planning de livraison

Citez dans le tableau ci-dessus, les étapes du scénario du cas d'utilisation en cours et dans quels délais elles seront réalisées. Il est aussi possible, si vous avez assez de place, d'insérer ici un diagramme de Gantt qui permettra d'ordonner ces différentes tâches dans le temps.

|  |  |
| --- | --- |
| Tâche | délais |
| *Réaliser la structure de la base de données* | 4jours |
|  |  |
|  |  |

### La phase de programmation

Insérer le code produit commenté en courrier new 9pt encadré. Contentez-vous des parties cruciales…

### La phase de tests

Expliquer les tests réalisés et le protocole de test. De plus faite TOUJOURS tester votre application par un utilisateur « lambda », il fera des manipulations que vous ne ferez jamais et ce sera bien ce type d’utilisateur qui utilisera votre application. Noter qui était le testeur et ce qu’il a découvert comme problème ou disfonctionnement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test à effectuer** | | **Résultat**  **escompté** | **Résultat**  **obtenu** | **Constatation** |
| Généralités | | | | |
|  |  |  |  |  |
|  |  |  |  |

## Cas d'utilisation n°... ou nommé ....

Description des objectifs du cas traité.

### Scénario

Décrire les différentes étapes par des phrases concises :

1. L’utilisateur ….
2. le système…

### Maquettes

Insérer des maquettes expliquant comment le scénario retenu pourra être mis en œuvre au niveau de l’interface homme-machine.

### Analyse du scénario

#### Algorithme ou Structogramme

Sur la base du scénario, identifiez les « méthodes » à réaliser.

#### Explications détaillées

Sur la base du structogramme ou de l’algorithme, insérer l’explication utile qui détaille le scénario au niveau de la programmation.

### Le planning de livraison

Citez dans le tableau ci-dessus, les étapes du scénario du cas d'utilisation en cours et dans quels délais elles seront réalisées. Il est aussi possible, si vous avez assez de place, d'insérer ici un diagramme de Gantt qui permettra d'ordonner ces différentes tâches dans le temps.

|  |  |
| --- | --- |
| Tâche | délais |
| *Réaliser la structure de la base de données* | 4jours |
|  |  |
|  |  |

### La phase de programmation

Insérer le code produit commenté en courrier new 9pt encadré. Contentez-vous des parties cruciales…

### La phase de tests

Expliquer les tests réalisés et le protocole de test. De plus faite TOUJOURS tester votre application par un utilisateur « lambda », il fera des manipulations que vous ne ferez jamais et ce sera bien ce type d’utilisateur qui utilisera votre application. Noter qui était le testeur et ce qu’il a découvert comme problème ou disfonctionnement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test à effectuer** | | **Résultat**  **escompté** | **Résultat**  **obtenu** | **Constatation** |
| Généralités | | | | |
|  |  |  |  |  |
|  |  |  |  |

# Mode d'emploi utilisateur

Mode d’emploi utilisateur en Annexe.

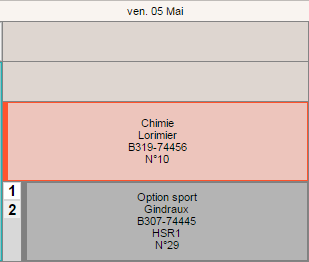
Expliquer écran après écran l’installation et l’utilisation de votre produit.

# Problèmes rencontrés et solutions

* Branches pas affichées :

Lorsqu’il y a plusieurs branches au même moment (souvent dû aux options), impossibilité d’afficher ces branches.

Le site du Cpln propose 2 boutons pour choisir quelle branche on aimerait afficher.



J’aurais pu essayer de reproduire cette méthode mais j’ai à faire à un gros problème. En effet, dans les données JSON reçues, les branches qui sont au même moment ne sont pas renvoyées. Il m’est donc impossible de savoir s’il y a cours à ces périodes-là.

Solution : La seule solution serait de modifier le site Hypercool pour qu’il renvoie les branches ayant la même heure de début. Dans ce cas-là, je pourrais agir.

* Crash lorsque branche « maitrise de classe » :

Mon programme crashait quand la branche « maitrise de classe » devait être affiché. Cette branche, contrairement aux autres, ne disposait pas de salle. Dans ma fonction, j’essayais de manipuler des données vides, ce qui entrainait un crash.

Solution : Ajout de try/catch et si la salle est vide alors on attribut « / » à la valeur de la salle.

* Conflit avec les Id :

Lors de la création du xml de la seconde activité, j’ai bêtement copié/collé les éléments. Je me suis rendu compte que ça créait des conflits dans le code.

Solution : Changer les id pour qu’ils soient uniques.

* Barre de chargement après le freeze :

Dans la vue semaine, pendant que l’application lançait les requêtes, l’écran se bloquait (4-5 secondes) comme si l’application avait crash. Pour éviter cet effet, j’ai mis en place une barre de chargement. En testant, j’ai vu que la barre s’affichait uniquement après que les requêtes soient terminées. L’effet freeze était toujours présent.

Solution : Ne plus utiliser .get pour faire passer le résultat de l’asynctask à l’activité vue semaine. A cause du get, la tâche n’est plus asynchrone.

* Freeze de l’application :

En faisant mes requêtes dans la vue semaine, je me suis rendu compte que l’application se bloquait durant quelques secondes. J’ai appris plus tard que c’était à cause du .get() lors de l’appel de la fonction Asynctask. Ce .get permettait de prendre la valeur de retour de la requête, mais la tâche ne devenait plus asynchrone, ce qui provoquait ce blocage. Après plusieurs recherches sur internet, j’ai trouvé une manière pour faire passer les valeurs reçues de la requête à l’activité principale.

Solution : Création d’une interface contenant la fonction qui va recevoir le résultat, et modification du OnPost() de la classe Asynctask.

* Affichage après 2 clics :

Après avoir mis en place mes requêtes de manière asynchrone, j’ai repéré un petit problème assez dérangeant. Il fallait cliquer 2 fois sur le bouton Rechercher pour que l’affichage ait lieu. J’’ai perdu quelques périodes avant de trouver la solution. Avant, je lançais les 3 requêtes en même temps. La requête nous renvoyant les données de l’horaire d’une classe avait besoins de l’id de la classe pour fonctionner. Et vu que les requêtes étaient envoyées en même temps, l’id était par conséquent null.

Solution : Lancer la requête après la fin de la première requête (celle qui va obtenir l’id de la classe).

* Certaines périodes commencent à des heures inhabituelles, par exemple 08h30 au lieu de 08h10. -> j’en parle dans les use case
* Libelles trop long -> j’en parle dans le use cases.
* Classes avec espaces :

En effectuant des tests, je me suis rapidement rendu compte que les classes contenant un espace ne renvoyaient aucunes données. J’ai donc essayé d’accéder à l’horaire de ces classes directement depuis un navigateur internet. En recherchant l’id d’une classe avec espace, j’ai vu que celui-ci était remplacé par « %20 ».

Solution : Tester le nom de la classe présent dans le champ classe, avant de le mettre dans une variable. Si il contient un espace, le remplacer par « %20 », sinon ne rien faire.

* Nom d’une classe vide/erroné dans l’historique :

Quand l’utilisateur recherchait avec une classe erronée ou vide, le nom de cette classe était tout de même écrit dans le fichier historique. Le problème était que lors du clic sur le bouton rechercher, on écrivait automatiquement dans le fichier sans faire de vérification.

Solution : Les fonctions EcritureFichier(), LectureFichier(), et MiseEnPlaceActv() se lancent désormais après la dernière requête. Lors du lancement de la requête, toutes les vérifications sont déjà effectuées, ca qui nous garantit que la classe est juste et non vide.

* Orientation rstart activité, ajout d’une ligne dans le manifest ca ne fait pas bug, mais juste la date. -> dans le use case
* Boutons avancer/reculer date mettant n’importe quel nom dans l’historique :

A l’appui des boutons pour changer la date d’une semaine/jour, le contenu du champ classe était écrit dans le fichier historique. Ca posait des problèmes car si l’utilisateur lançait une recherche, puis modifiait le contenu du champ classe (sans appuyer sur le bouton rechercher) et appuyait sur les boutons, alors le contenu (pouvant être vide ou erroné) était stocké dans l’historique.

Solution : Si la requête est effectuée depuis les boutons permettant de changer la date, on n’enregistre plus le contenu du champ dans le fichier. Le nom de la classe a normalement déjà été stocké lors de l’appui sur le bouton Rechercher.

* Impossibilité de manipuler les données reçues avec des tableaux :

Une fois les données de l’horaire reçues, il fallait les stocker dans des variables. J’ai tout d’abord essayé avec des tableaux. J’arrivais à stocker les données dans mon tableau, mais il m’était impossible de trier chronologiquement le contenu. Je ne sais toujours pas si c’est le programme qui n’autorise pas une telle opération ou bien une erreur de ma part. J’ai essayé la même opération avec des arraylists et cette fois-ci tout marchait comme parfaitement.

Solution : Stocker les données reçues dans des arraylists plutôt que dans des tableaux.

* Ecriture des noms à la suite dans le fichier historique :

A la première version de l’historique, j’écrivais simplement les noms des classes dans le fichier. Je me suis rendu compte que de base, toutes les classes étaient écrites les uns à la suite des autres. Je devais alors, écrire le nom des classes ligne par ligne pour les utiliser plus facilement.

Solution : Changement de la fonction pour que l’écriture se fasse ligne par ligne.

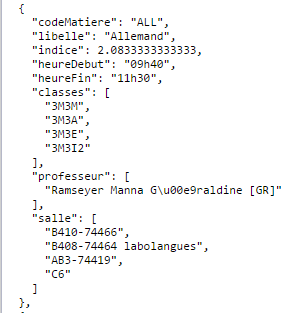
* Crash lorsque la salle ne contenait pas « -« :

Durant mes tests, certaines classes provoquaient un crash de l’application. Le point commun de ses classes étaient qu’elle disposait d’une branche qui avait un nom de salle sans tiret (par exemple : « Maladière 4 »). Pour simplifier l’affichage des salles, une fonction supprimait tout ce qui était après un tiret. (B106B-74487 devenait B106B). Maladière 4 ne contenant pas de tiret, arrêtait la fonction mettant les données reçues dans des arraylists. Les tableaux n’étaient donc pas totalement remplis ce qui provoquait ce crash lors de l’affichage.

Solution : Ajouter un try/catch et si la salle ne contient pas de tiret, alors ne rien faire.

* Réception de plusieurs salles pour une branche :

En vérifiant les données JSON reçues, je me suis aperçu que certaines branches avaient plusieurs salles.



Je ne savais donc pas quelle salle était la bonne. Au début j’ai cru que c’était par ordre chronologique, les salles de début d’année d’abord puis les salles actuelles. En vérifiant avec plusieurs classes, j’ai vu que ce n’était pas le cas. Après discussion avec le professeur, il m’a laissé le choix. Soit j’affiche toutes les salles reçues à la suite (ex : B410, B408, AB3, C6) ou je prends juste la première. J’ai décidé de choisir la deuxième option car sinon ça risquait de créer une chaîne de caractère trop long à afficher.

Solution ; Dans l’état actuel, je ne peux pas choisir la bonne salle. Une solution serait de modifier Hypercool pour qu’il envoie uniquement la salle actuelle.

* Asynctask crash à la deuxième requête.

En exécutant mes requêtes cette fois-ci, de manière asynchrone, j’ai remarqué que l’application crashait lors de l’appui sur le bouton Rechercher. C’était bizarre car la requête dans le OnCreate() fonctionnait correctement. Après plusieurs recherches, j’ai trouvé qu’il n’était pas possible d’utiliser la même instance de l’Asynctask plusieurs fois.

Solution : Créer une nouvelle instance de l’Asynctask avant de l’exécuter.

* L’histo, je voulais d’abord faire un spinner puis j’ai changé pour avoir dans l’actv.

Dans ce chapitre, on recensera les problèmes rencontrés et les solutions appliquées pour les résoudre.

Si des problèmes n’ont pas trouvés de solution, ils devront aussi être notés.

Le but de ce chapitre est de concentrer les problèmes et leurs solutions de manière à ce que le lecteur puisse directement avoir accès aux solutions ou descriptions des problèmes sans devoir lire tout le rapport ou le journal de travail.

# Conclusions

Un paragraphe permettant de donner votre avis sur le projet. Citez les réussites et les points faibles. Citez les éléments de capitalisation.

Ce paragraphe se remplira à la fin du projet lors de la remise du document.

# Annexes

## Journal de bord

## Cahier des charges

## Code source

## Planning

## Références

Inscrivez ici les ressources utilisées dans votre projet selon les conventions ci-dessous

Ressource imprimée : titre, auteur, lieu d'édition, année de parution, isbn. Noter les chapitres ou pages concernées.

Ressource en ligne : titre auteur, date de parution, URI ou même plus précisément de manière à ce que le lecteur puisse aisément retrouver l’information.

Mettre pourquoi j’ai vue jour et vue semaine à chaque use case.

Tableau et arraylist quand je mets la donnés

Nomenplature des Oui /non dans les diagrammes.

AlistToutesCLasses, 319 mauvaiss

Erreur dans le lecture fichier avec le int i = 0qui ne doit pas être dans la boucle.

2 fois même bout de code

Je me rends compte qu’en écrivant le rapport, il y a plein de petite choses que je peux simplifier.

Les méthodes altérnatives on met ca ou

Metéhodologie

J’ai pas besoin de faire un tableau d’asynctask

Arraylists et tableaux

Schéma basculement

Plusieurs choix possible dans l’explication détaillé