Développement d'une application pour "Jouer à la RO"

Documentation

Clients :

Pierre Fouilhoux pierre.fouilhoux@lipn.fr
Lucas Létocart lucas.letocart@lipn.univ-paris13.fr

Equipe de suivie :

Thierry Hamon thierry.hamon@univ-paris13.fr
Sophie Toulouse sophie.toulouse@lipn.univ-paris13.fr

Équipe de développement (Groupe 2) :

Nicolas Trirayoute Soutthiboune nicolas.soutthiboune@edu.univ-paris13.fr
Dino Avril dino.avril@edu.univ-paris13.fr
Fatima Bellafkih fatima.bellafkih@edu.univ-paris13.fr
Manina Meng manina.meng@edu.univ-paris13.fr
Camille Pedron camille.pedron@edu.univ-paris13.fr

Formation Ingénieur Informatique 2ème année - 12/04/2021 - Version 1.0

Introduction

Ce document a pour vocation de décrire l'application du projet "Jouer à la RO[1]". Elle contiendra entre autres les instructions d'installation, d'utilisation ou encore une description de la conception. Il s'adresse tantôt aux clients, tantôt aux utilisateurs de l'application, tantôt à la future équipe de développement qui pourra reprendre le projet.
Différents termes techniques, notamment en anglais, seront utilisés et expliqués au fur et à mesure.

I-Présentation du projet

Dans le cadre de notre formation d'ingénieur informatique à Sup Galilée, nous participons à l'élaboration du projet de développement d'une application smartphone "**Jouer à la RO**" (pour l'instant pour Android principalement). Le jeu s'insère dans le cadre d'une simulation de gestion d'une ville de manière ludique et en apportant la compréhension des aspects de la RO. Le but initial de ce projet était d'implémenter deux jeux en rapport avec la RO dans ce cadre, au final un seul est implémenté. Ce projet pourra être repris les années suivantes. L'application est implémentée sous le framework **Flutter[2]** et avec le gestionnaire de version **GitHub** et des fichiers **JSON[3]** pour le stockage des données (représentera notre base de données). Pour la modélisation et résolution des problèmes du jeu nous utiliserons le langage de programmation **Julia**.

Les jeux sont en fait des problèmes de la RO. Ils sont au préalable modélisés et des instances de ceux-ci sont résolus en Julia pour connaître la solution optimale de chaque instance. L'ensemble de ces informations sont ensuite stockés dans l'application pour pouvoir modéliser les problèmes sous forme de jeux et comparer la performance du joueur avec la solution optimale trouvée en Julia.

[1] RO: Recherche Opérationnelle, l'ensemble des méthodes et techniques rationnelles orientées vers la recherche du meilleur choix.
[2] Framework: Environnement de développement, le framework Flutter utilise notamment le langage de programmation Dart
[3] JSON: format de fichier textuel pour le stockage des données

I.1-Patron de conception utilisé

Le développement de l'application sur le Framework Flutter a été fait sous Androïd et associé à la notion de modèle-vue-contrôleur.

L'approche MVC nous a permis de décomposer notre programme en trois parties:

• La vue constitue la couche graphique, intégrant les boutons, les zones de textes, les images. Elle interagit avec le modèle afin de pouvoir afficher les informations venant de la base de données.

Pour cela, nous avons créé une classe "Data"[4] pour chaque modèle qui garde en mémoire les informations dont la vue a besoin. Par exemple, pour le premier jeu il y a une classe qui va servir à stocker le nombre de clients et leur gain pour chaque niveau de jeu. Les valeurs stockées dans la classe data ne sont pas amenées à changer pendant un niveau. Ce sont en fait les valeurs initiales du niveau (nombre de service, gain pour chaque service, nombre d'immeuble, coût des étages, valeur de la solution optimale, etc)

• Le modèle permet d'effectuer les calculs et les opérations sur la data. Il sauvegarde les données à un instant T. En interrogeant le modèle, on pourra alors connaître les attributions des clients aux immeubles à l'aide d'une matrice et récupérer le score actuel.

• Le contrôleur agit comme une liaison entre la vue et le modèle. Chaque interactions entre chaque élément a été faite via le contrôleur ce qui rend la vue et le modèle à jour en même temps..

Ce découpage permet de simplifier le développement en isolant le code métier (le modèle) de l'interface graphique (vue) et de leurs interactions (contrôleur). Ainsi la modification de l'un n'impose pas la réécriture des autres, optimisant ainsi la réalisation et la maintenance du code.

I.2-Les différentes parties du projet

Le projet est séparé en trois grandes parties:

- L'analyse et la conception du problème ainsi que de l'architecture de l'application.
- La modélisation des problèmes de la RO, leur résolution ainsi que la génération de plusieurs de leurs instances.
- Le développement de l'application mobile ainsi que l'intégration des différentes instances générées dans la partie modélisation des problèmes.

et une dernière partie plus mineure en terme de temps de travail:

- Le déploiement de l'application sous format APK afin d'être utilisé.

[4] Data: signifie données en français

II-Mode d'emploi

II.1-Installer l'environnement de travail pour coder l'application

Pour cette étape nous allons suivre les étapes pour faire l'installation sur Windows données dans la documentation de Flutter. Pour plus de détails et concernant l'installation sur d'autres systèmes d'exploitation se référer sur ce lien :
https://flutter.dev/docs/get-started/install.

II.1.a-Installer Flutter et Dart

Il faut commencer par télécharger la dernière version de Flutter SDK et extraire le ZIP à l'endroit voulu. Dart se trouve également dans ce même SDK.
Dans le dossier qui a été extrait, aller dans le dossier "bin", ouvrir une invite de commande (ou PowerShell), et lancer la commande "./flutter doctor". Cette commande permet de voir ce qu'il reste à installer pour pouvoir utiliser flutter. Il devrait normalement afficher qu'il reste à installer Android Studio, Android SDK, et brancher un téléphone.

II.1.b-Installer Android Studio

Il faut maintenant télécharger et installer la dernière version d'Android Studio et suivre les étapes du "Android Studio Setup Wizard".

II.2-Compiler l'application

Il y a deux manières de tester l'application. On peut soit la compiler directement sur un Smartphone, soit dans un émulateur. Dans tous les cas, il faut faire des configurations pour les utiliser.

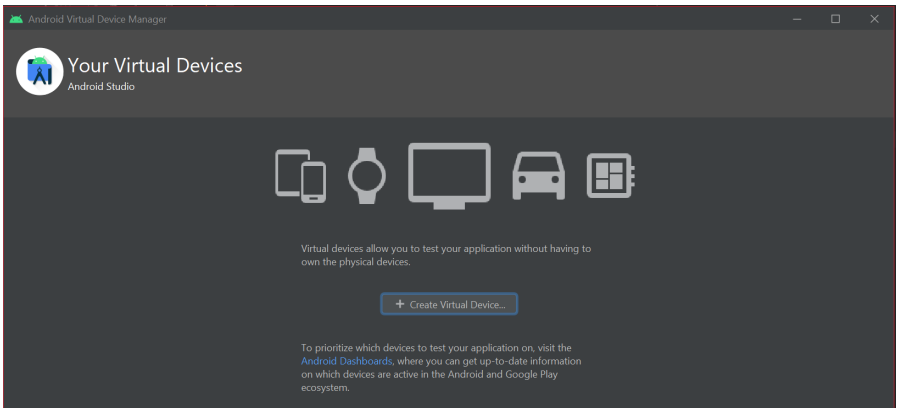
II.2.a-Sur téléphone portable

Si vous choisissez d'utiliser directement un Smartphone, il faut débloquer les options pour développeurs dans les paramètres, dans cette option autoriser le débogage USB, enfin (si vous êtes sur Windows) installer les drivers nécessaires pour faire le liens entre le téléphone et l'ordinateur.
Au moment de brancher le Smartphone à l'ordinateur il faudra accepter la demande d'autorisation sur le téléphone.
Il ne reste plus qu'à lancer la compilation, et l'application s'installera sur le Smartphone où vous pourrez la tester.

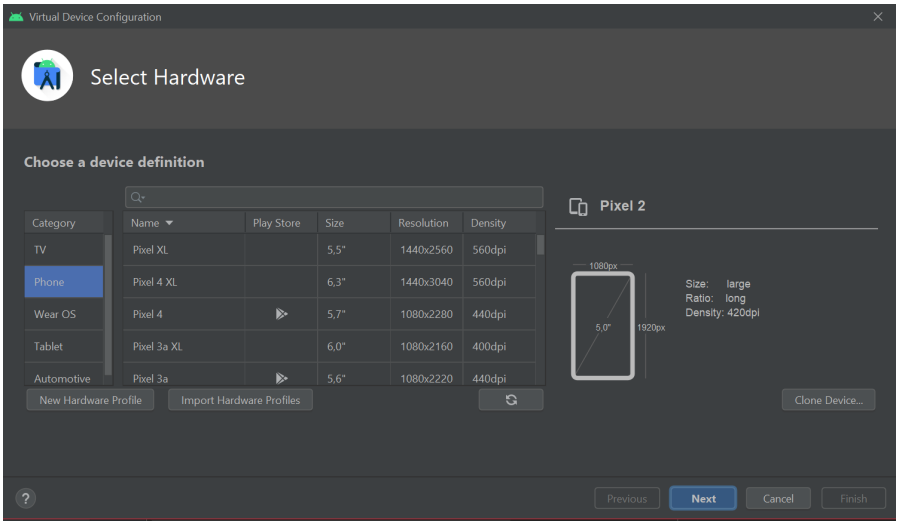
II.2.b-Sur Emulateur

Si vous préférez utiliser un émulateur sur Android Studio, il faudra commencer par choisir votre émulateur. Dans la barre d'outil d'Android, il y a une icône (entourée en rouge) :

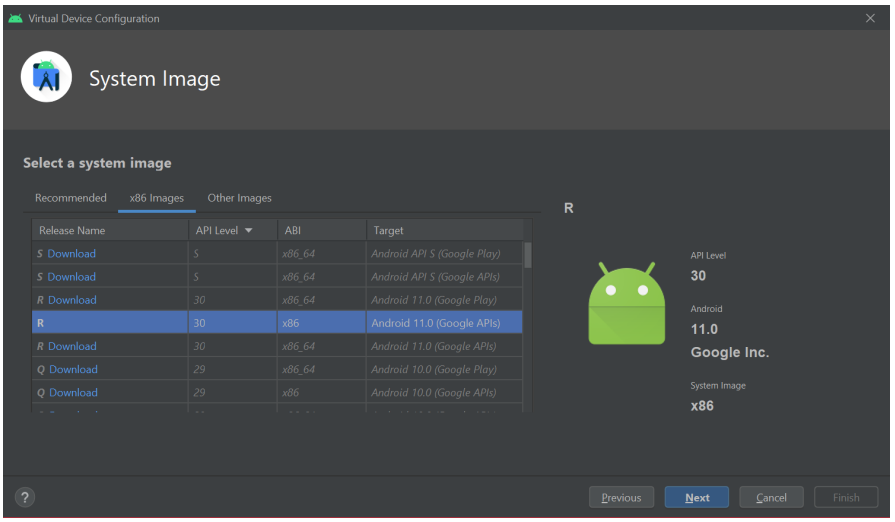
Lorsque vous cliquez sur cette icône, cela ouvre la fenêtre suivante. Cliquez sur + Create Virtual Device :



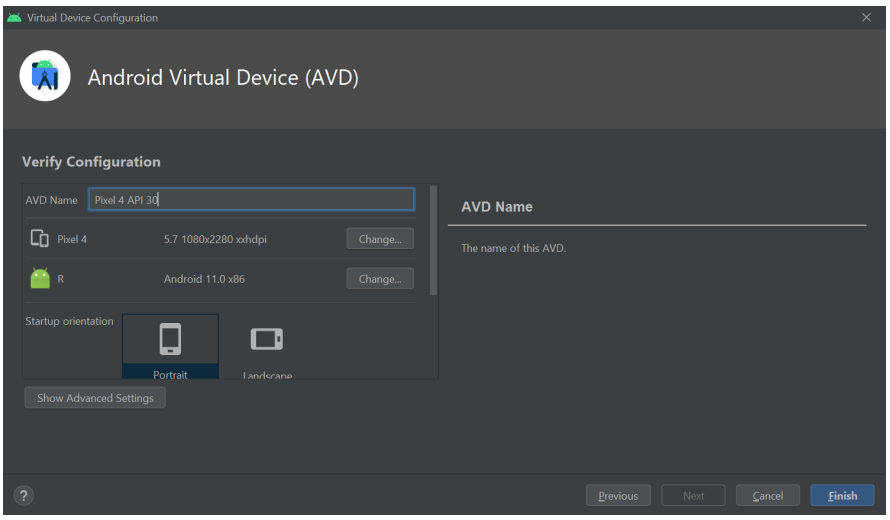
Sélectionnez le modèle de téléphone de votre choix :



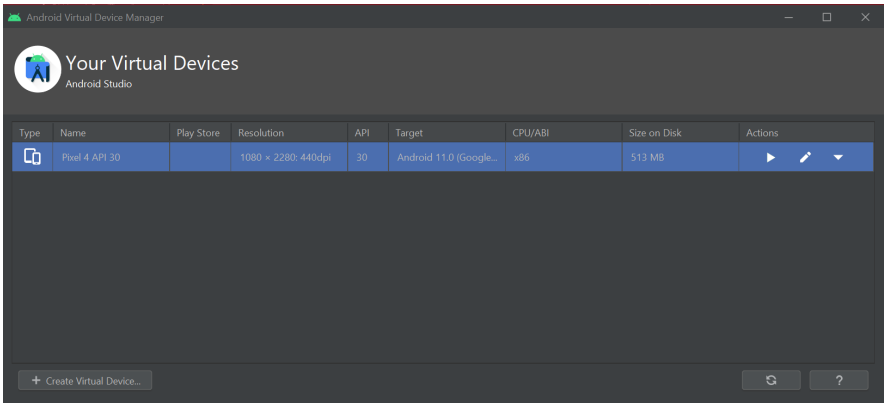
Cependant, si vous êtes sous Windows, vous aurez accès uniquement à des appareils Android. Une fois l'appareil choisi, appuyez sur Next. Vous devez télécharger une image système. Nous avons utilisé l'image système R pour notre part :



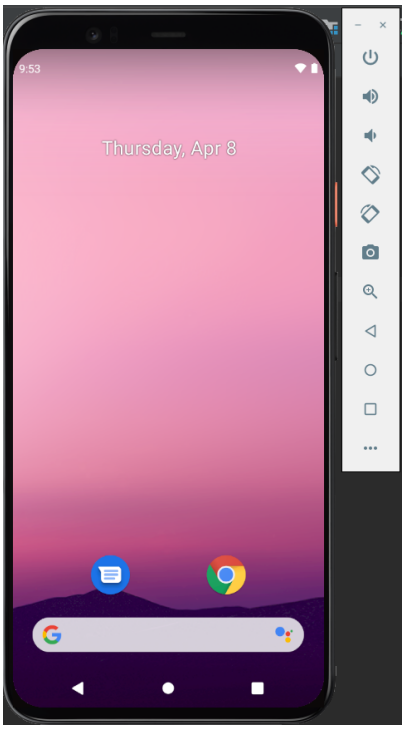
Il ne vous reste plus qu'à choisir le nom de votre émulateur (ou le laisser tel quel) et à cliquer sur Finish :



Vous avez maintenant une liste d'émulateurs. Pour démarrer votre émulateur cliquez sur l'action play et fermez cette fenêtre :

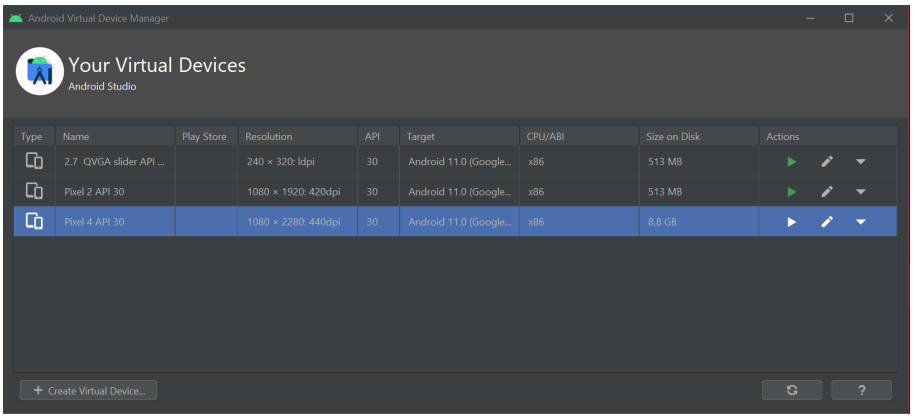


L’émulateur va s’ouvrir dans une autre fenêtre :



Avant de compiler l’application, vérifiez que l'émulateur est bien sélectionné :

Cela peut prendre un peu de temps (de l’ordre de maximum 1 min) avant que l'émulateur ne soit reconnu. Vous pourrez par la suite créer d’autres appareils virtuels et avoir une liste d’appareil afin de tester l’application sur différentes tailles de téléphones :



II.3-Modifier le nom et l’icône de l’application

Pour changer le nom et l’icône de l’application nous avons suivi ce tutoriel:
https://medium.com/@vaibhavi.rana99/change-application-name-and-icon-in-flutter-bebbec297c57
Pour modifier l’icône nous avons utilisé la partie “Additional Step: Change the icon manually” en utilisant le lien suivant pour créer l'icône de base:
https://romannurik.github.io/AndroidAssetStudio/
et en l’ajoutant au projet à l’aide de ce tutoriel:
https://developer.android.com/studio/write/image-asset-studio

II.4-Installation outils de résolutions et création d'instances de problèmes

II.4.a-Installer Julia/Jump

Cette partie indique comment installer Julia ainsi que les librairies nécessaires sur windows.
Pour pouvoir faire fonctionner les outils de résolution d'instances, il faut commencer par installer la dernière version stable de Julia, sur leur site officiel :
https://julialang.org/downloads/

Il faut ensuite installer les 2 librairies nécessaires à la résolution, JuMP et GLPK.
Après avoir lancé Julia une invite de commande s'ouvre.
Appuyer sur la touche] dans l'invite de commande pour entrer en mode pkg, puis entrer les commandes:
add GLPK
add JuMP@0.19

Pour ouvrir le notebook (endroit où l'on code), il faut entrer deux commandes dans un nouveau terminal Julia :
using IJulia
notebook()

Un onglet devrait s'ouvrir dans le navigateur par défaut, il suffit alors d'ouvrir le notebook à partir de cette fenêtre.

II.4.b-Création d'instances

Le jeu contient essentiellement des instances du problème créées à la main.

Nous avons cependant créé un outil pour créer des instances automatiquement:

Le programme de création automatique d'instances est codé en C. Il vous faut l'environnement nécessaire à la compilation et exécution de programme en C (généralement Linux).
Pour le compiler, un Makefile[5] est à votre disposition. Une fois dans le dossier contenant le fichier Makefile La commande make (ou make all) lance la compilation de l'exécutable. La commande make clear permet d'effacer les fichiers objets, la commande make fclear supprime aussi l'exécutable. Il peut aussi être intéressant de compiler à la main pour ajouter des options permettant de modifier certaines constantes:

gcc -o buildings.exe game1.c utils.c -D ARG1=VALUE1 -D ARG2=VALUE2

[5]Makefile: fichier de compilation automatique du langage de programmation C.

Il est aussi possible de changer ces constantes directement dans le fichier const_game1.h, fichier dans lequel les constantes sont définies et expliquées.

L'exécutable quant à lui demande entre 1 et 3 arguments. Le 1er argument est obligatoire, il permet de donner un numéro lors de la création des fichiers. Le deuxième argument est optionnel et permet de choisir le nombre d'immeubles dans l'instance créée. Il peut ne pas être saisi ou être ignoré en mettant sa valeur à -1. Le 3e argument permet de choisir le nombre de clients et est facultatif.

Le programme crée 2 fichier:

- Le fichier Buiding_X permet de lancer facilement le solveur d'instance, en copiant le contenu du fichier dans le notebook() Julia contenant le solveur.
- Le fichier Building_X.json pour créer une instance dans le format de stockage pour l'application téléphone.

III-Déploiement sur téléphone sans passer par le Play Store

Notez que si vous avez compilé l'application sur votre téléphone, l'application y est déjà installée. On explique ici comment installer l'application sur un autre téléphone sans passer par Android Studio.

REMARQUE IMPORTANTE: cette méthode ne fonctionne que si vous compilez sur un réel téléphone et non pas un émulateur. Via émulateur vous aurez une APK mais qui ne s'installe pas correctement sur téléphone.

A chaque fois que vous compilez le projet, une APK (*Android Package Kit,* ce qui permet d'installer l'application) se crée dans le projet. Celle ci se trouve à l'endroit suivant dans le projet :
[PATH]\PlayingOR\build\app\outputs\flutter-apk
où [PATH] correspond à l'emplacement où se trouve le projet.

Pour l'installer sur votre téléphone il suffit de télécharger l'APK dessus, et autoriser l'installation sur votre téléphone. Attention, cela peut ne pas fonctionner sur tous les téléphones. Certains peuvent avoir besoin de passer par une application intermédiaire comme les appareils iOS.

Dans le cas des smartphones iOS, il faut d'abord compiler le projet sur Xcode (équivalent d'Android Studio) sur un ordinateur Mac pour récupérer l'IPA (*iOS App Store Package*, équivalent d'APK sur iOS), et enfin passer par des applications intermédiaires pour l'installer. Au moment de la rédaction de ce document, n'ayant pas d'ordinateur MAC, nous n'avons compilé le projet que pour les appareils Androids.

Pour cette version de l'application nous n'avons pas essayé de le mettre sur le Playstore ou l'AppStore, pour le faire vous pouvez suivre les tutoriels ci-dessous:
https://flutter.dev/docs/deployment/android
https://flutter.dev/docs/deployment/ios
(cela implique l'achat d'un compte développeur)

IV-Vocation de l'application

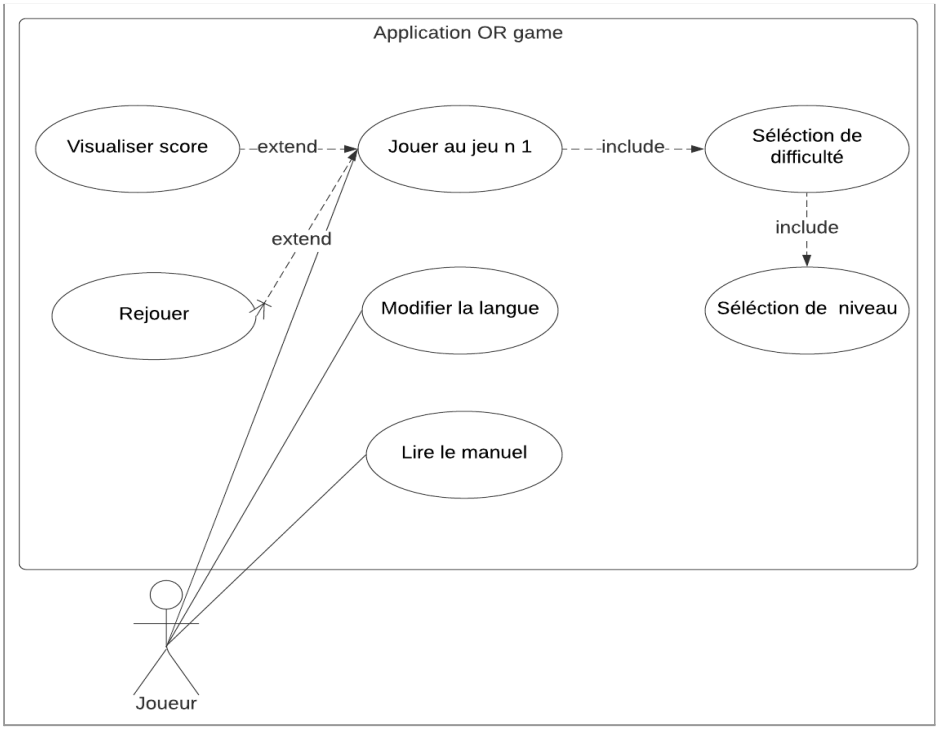
L'application consiste à réunir plusieurs jeux éducatifs en lien avec la recherche opérationnelle tout en gardant le cadre de la gestion d'une ville: Les deux jeux prévus initialement sont:

- Un jeu de construction d'immeubles où le joueur a plusieurs requêtes de la part de clients voulant occuper de nouveaux bâtiments commerciaux. Cependant chaque étage d'un immeuble coûte de l'argent à construire et plus un immeuble est haut plus il est cher. Le joueur doit maximiser l'argent obtenu en choisissant quelles requêtes attribuer ou non à quel immeuble. Ce jeu a un lien avec plusieurs problèmes de la RO comme le problème du sac à dos.
- Un jeu de tracé de trajet de bus où le joueur doit choisir un chemin entre plusieurs points reliés entre eux où certains doivent être obligatoirement dans le chemin et d'autres non. Le but est de faire un chemin de distance parcourue minimal, ce jeu est relié à des problèmes de recherche de plus court chemin dans un graphe ou encore au connue problème du voyageur de commerce.

Nous avons réussi à modéliser les deux problèmes liés au Jeu de construction des immeubles et la ligne de Bus ainsi que la génération de leurs instances et le développement du premier jeu sous Flutter.

IV.1-Cas d'utilisation

Le diagramme de cas d'utilisation représente les relations fonctionnelles entre le joueur et les différents jeux Il donne une description cohérente de toutes les vues[6] que l'on peut avoir dans notre application.



[6]une vue: interface visuelle entre le joueur et l'application

D’après le diagramme décrit ci-dessus, l’utilisateur peut choisir de:
-jouer aux jeu de construction des immeuble qui est le jeu numéro 1
-choisir de jouer à une difficulté et à un niveau bien précis
-de rejouer
-de modifier les paramètres tels que la modification de la langue
-Il peut également lire le manuel afin de comprendre le principe du jeu.

En jouant, l’utilisateur est capable de visualiser son score à chaque essai et de recommencer une nouvelle partie.

Les données de l’application sont stockées et manipulées via des fichiers JSON internes à l’application. (Le joueur n'interagit pas avec ces données ce qui explique le fait qu'ils ne sont pas présents au niveau du diagramme).

On rappelle que "include" signifie que ce cas d'utilisation est obligatoirement réalisé si celui auquel il est relié est réalisé et que "extend" signifie que ce cas d'utilisation peut être réalisé pendant celui auquel il est relié mais ce n'est pas une obligation.

Description des cas d’utilisation les plus prioritaires:

Cas d’utilisation “Jouer au jeu n°1”

Le joueur peut:
- Déplacer un service avec son doigt, et le mettre dans un bâtiment.
- Visualiser son score et le valider en cliquant sur le bouton de vérification.
- Demander des indices et des conseils en cliquant sur le bouton des indices.
- Changer la langue afin de mieux comprendre les règles du jeu.
- Relire le manuel en cas de besoin.

Cas d’utilisation "Sélection de difficulté”

Le joueur peut :
- Choisir la difficulté du jeu auquel il veut jouer, parmi une liste de difficultés disponibles en fonction de son avancement dans le jeu.
- Revenir au menu principal.

Cas d’utilisation "Sélection du niveau”

Le joueur peut :
- Choisir le niveau associé au jeu et à la difficulté précédemment choisie parmi une liste de niveaux disponibles en fonction de son avancement dans le jeu.
- Revenir au menu principal.

Cas d’utilisation "Modifier la langue”

Le joueur peut :

- Modifier la langue du jeu dans n’importe quelle page où il se retrouve, il a le choix entre anglais et français pour l’instant.

Cas d’utilisation "Lire le manuel”

Le joueur peut :

- Lire et relire le manuel dans n’importe quelle page ou il se retrouve.

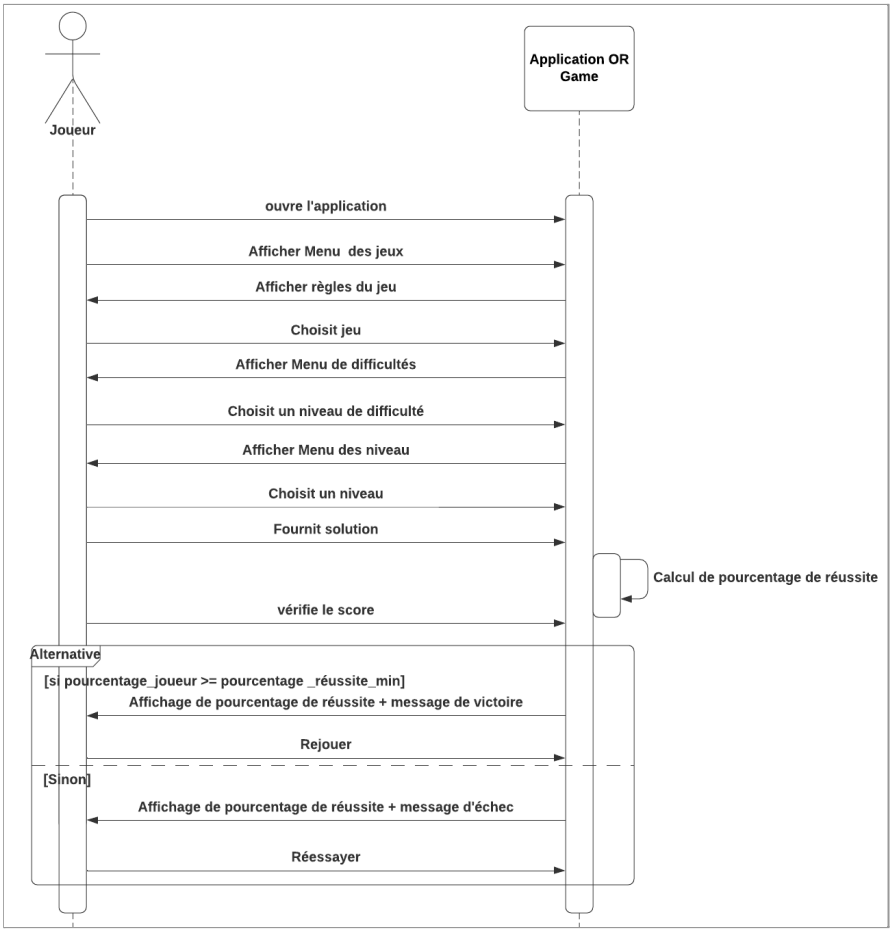
Cas d’utilisation "Visualiser le score”

Le joueur peut :

- Visualiser son score et le valider en cliquant sur un bouton de vérification.Il pourra par la suite rejouer une nouvelle partie ou réessayer sur la même partie.

IV.2-Diagramme de séquence

Ce diagramme de séquence représente un scénario typique d’une utilisation de l’application ainsi que des interactions entre les différents acteurs:



Scénario :

1. Le joueur ouvre l'application et choisit un jeu sur le menu.
2. Les règles peuvent être lues sur chaque page au niveau du manuel.
3. Pour pouvoir jouer, le joueur choisit une difficultée ainsi qu'un niveau.
4. Le joueur fournit une solution en jouant une partie.
5. Au niveau de l'application, un pourcentage de réussite sera calculé en fonction de l'instance jouée.
6. Le joueur vérifie son score.
7. S'il réussit, le pourcentage sera affiché avec un message de réussite.
8. Le joueur pourra rejouer une nouvelle partie.
9. Sinon, le pourcentage sera également affiché mais cette fois avec un message d'échec.
10. Le joueur pourra réessayer sur la même partie.

V-Architecture globale

Informations préalables:
Nous avons décidé d'écrire le code ainsi que les commentaires de code en anglais pour une question d'uniformisation du vocabulaire et de la langue car nous utilisons des composants pré-créés de flutter qui ont des noms en anglais.

Nous allons utiliser deux termes par la suite *BuildingConstruction* et *BusLine*, ceux-sont les noms que nous avons choisi pour le premier et le deuxième jeu, ils sont utilisés pour nommer des fichiers de code, des classes et des fonctions.

V.1-Modélisation des problèmes

Pour modéliser et résoudre les problèmes, nous avons choisi d'utiliser les librairies JuMP et GLPK en julia, car ce sont les outils que nous avons découverts en formation d'ingénieur informatique deuxième année pour résoudre des problèmes linéaires en nombres entiers (PLNE) qui sont les types de problèmes que nous avons modélisé avec nos jeux.
Pour les deux jeux, le problème est résolu par une fonction *resolve*, qui prend différents arguments en paramètre en fonction du jeu.
Le problème est modélisé par une fonction objectif (une formule à minimiser ou maximiser), des variables et des contraintes, qui sont accessibles à travers le modèle appelé m dans le code.

V.1.a-Modélisation du problème *BuildingConstruction*

Pour BuildingsConstruction, la fonction resolve prends plusieurs arguments :

- buildings_number, qui est le nombre de bâtiments disponibles pour l'instance à résoudre
- clients_number, qui est le nombre de clients disponibles
- client_floors, qui est la liste du nombre d'étages demandé par chaque client
- client_gains, qui est la liste des gains apportés par chaque clients, dans le même ordre que la liste client_floors
- building_max_size, qui est la taille maximum des immeubles. Elle est égale à la somme des étages pris par tous les clients, et est la longueur de la liste suivante
- floor_prices, qui est la liste des coûts de chaque étage, du plus bas au plus haut.

Pour représenter ce problème nous utilisons 3 variables:
La première est un tableau en deux dimensions nommé *A*, contenant des variable binaires, qui nous permet d'affecter un client à un immeuble : si le client i est affecté à l'immeuble k, alors *A[i, k]* = 1, sinon *A[i, k]* = 0.
La deuxième variable est aussi un tableau en deux dimension *B* contenant des variables binaires, qui nous permet de savoir pour chaque immeuble quels sont ses étages utilisés :
Si l'étage k de l'immeuble i est utilisé, alors *B[k, i]* = 1, sinon *B[k, i]* = 0.

Ces deux matrices binaires nous permettent de calculer plus facilement la fonction objectif.
Le but de ce jeu étant de maximiser des gains, la fonction objectif de ce problème est donc le maximum de ces derniers, qui sont calculés par la somme des gains de chaque client retenus moins le coût de création des immeubles, qui est égale à la somme du coût de chaque étage utilisé.

On utilise un 3e tableau de variables *B_bis* qui va nous permettre de relier les deux derniers tableaux grâce aux contraintes. C'est un tableau en une dimension qui nous indique pour chaque immeuble combien d'étages sont utilisés, cette variable est nécessaire de manière pratique dans le code Julia.
Pour faire le lien entre le tableau *B_bis* et la matrice *B*, on utilise une contrainte indiquant que pour un immeuble k, la somme des étages utilisés (de la matrice *B*) est égale au nombre d'étages utilisés pour ce même immeuble k dans le tableau *B_bis*.
Pour le lien entre *B_bis* et A, on ajoute une contrainte permettant, pour chaque immeuble k, d'avoir le nombre d'étages utilisés (*B_bis*) égal à la somme des étages demandés par les clients présents dans le même immeuble k (A).

Deux autres contraintes ont été ajoutées par souci de cohérence : la première est une contrainte empêchant à un client d'être dans deux bâtiments différents et la deuxième est une contrainte obligeant les immeubles à se construire de bas en haut, et de ne pas utiliser un étage alors que celui du dessous n'est pas utilisé.

V.1.b-Modélisation du problème *BusLine*

La fonction résolve de ce problème prend 3 paramètres :

- V, qui est une liste de couple de réels, qui sont les coordonnées des stations disponibles. La fonction va également créer une matrice distance à partir de cette liste, permettant d'obtenir les distances entre chaques stations
- G, qui est une matrice binaire nxn, où n est le nombre de stations, dans laquelle G[i, j] = 1 si une rue relie la station et la station j, 0 sinon.
- S, qui est une liste d'entiers permettant de connaître le type de chaque station.
- Si S[i] = 0 alors i est une station optionnelle; si S[i] = 1 alors alors i est une station obligatoire; si S[i] = 2, i est la station de départ et si S[i] = 3, i est la station finale.

Nous avons choisi de résoudre le problème comme un problème de graphe. V représente la liste des sommets du graphe et G est la matrice d'adjacence de ce dernier.

Pour modéliser le problème, nous utilisons deux matrices de taille nxn. La première est une matrice binaire R servant pour le calcul de la fonction objectif : si R[i, j] = 1, alors dans la solution finale l'arête qui va de i à j est utilisée, sinon R[i, j] = 0.
Le but du jeu étant de faire le chemin le plus court, la fonction objectif est ici le minimum de la somme des distances de toutes les arêtes utilisées.

Pour créer un chemin, on utilise les contraintes suivantes :

- Si le sommet v est un sommet de Steiner (si S[v] = 0), alors si une arête entre dans v, une arête doit aussi en sortir, sinon aucune arête ne peut en sortir.
- Si le sommet v est un sommet obligatoire (si S[v] = 1), alors une arête doit entrer dans v et une arête doit aussi en sortir
- Si le sommet v est le sommet de départ (si S[v] = 2), alors une arête doit en sortir et aucune arête ne doit y entrer.
- Si le sommet v est le sommet d'arrivée (si S[v] = 3), alors une arête doit y entrer et aucune arête ne doit en sortir.

De plus, pour des soucis de cohérence, nous avons ajouté deux contraintes, une permettant d'empêcher de passer plusieurs fois par le même sommet et une autre indiquant qu'on ne peut utiliser que des arêtes existantes dans la matrice d'adjacence (on ne peut pas créer des rues).

Cependant cette modélisation n'est pas encore suffisante : des sous-tours peuvent se former dans certains cas autour du chemin principal. Autrement dit, plusieurs chemins non connectés entre eux peuvent se former du moment qu'ils forment un cycle. Pour résoudre ce problème nous avons utilisé une deuxième matrice de taille nxn, permettant de représenter des flots.

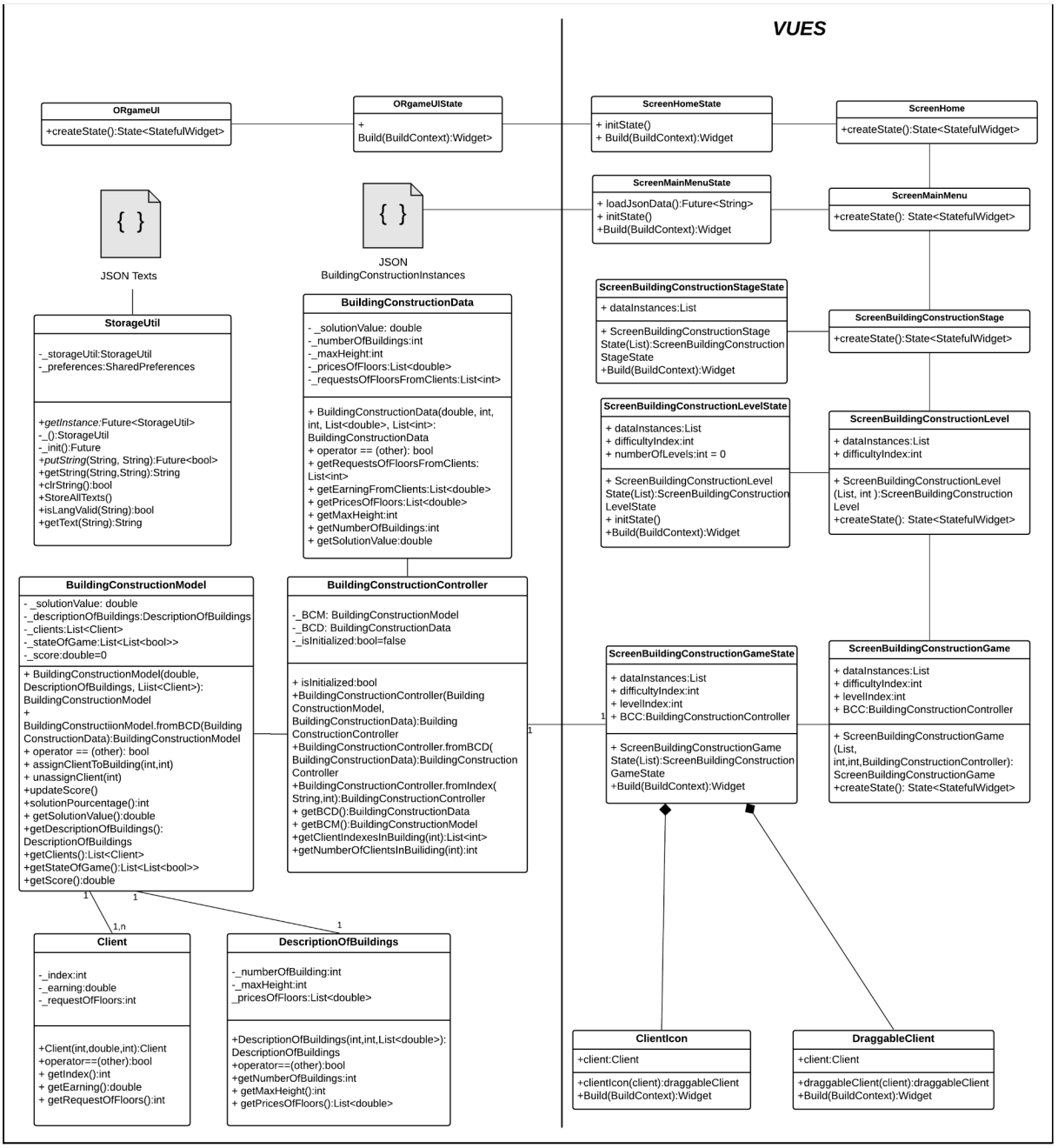
Mais nous ne connaissons pas par avance la taille du chemin optimal, nous avons donc utilisé un flot qui commence avec comme valeur le nombre de sommets obligatoire et qui diminue à chaque fois qu'il passe par un de ces derniers.

Des sous-tours peuvent donc toujours se former avec des sommets de Steiner, mais cela n'arrivera pas car cela ne sera jamais optimal. Pour faire fonctionner cette solutions, nous utilisons les contraintes ci-dessous :

- Si une arête n'est pas utilisée, alors elle ne peut pas porter de flot
- Si un sommet est un sommet de Steiner, alors le flot entrant doit être égal au flot sortant
- Si un sommet est un sommet obligatoire, alors le flot sortant doit être égal au flot entrant - 1
- Si un sommet est le sommet de départ, alors le flot sortant doit être égal au nombre de sommet obligatoires + 1
- Si un sommet est le somme d'arrivée, alors le flot entrant doit être égal à 1.

V.2-Diagramme de classes

Le diagramme de classes représenté dans la figure suivante décrit la version finale des associations entre les classes et ceci afin de déterminer les dépendances entre les différentes classes au niveau du code.



La suite détaillera les rôles et utilisations des classes utilisées :

- Comme nous utilisons des Widget, la partie View comprend plusieurs classes de type *StatefulWidget* afin d'attribuer un état à une *Widget.* Chaque classe est associée à une classe *State* qui représente son état et fait appel aux différentes fonctions ainsi qu' à la méthode *build* pour afficher la vue et ses composantes.

- Il existe des routes entre les différents Widgets de la vue. Elles sont utilisées notamment à l'aide des boutons qui redirigent vers la bonne vue-widget une fois

appuyés. C'est pourquoi les liens de ces classes ne sont pas orientés et n'ont pas d'arité mais sont de simples traits.

- La classe *ORgameUIState* représente la classe générale de l'application vu qu'elle est liée directement à la page d'accueil.

- Le reste du diagramme présente les classes utilisées pour modéliser le jeu, c'est une architecture classique "Modèle Vue Contrôleur", avec une classe "*Data*" en plus qui contient les informations par rapport au jeu contenu dans les fichiers *JSON* de stockage des données (elle sert d'interface pour contenir les données primaires d'une instance).

- La relation de composition entre la classe *ScreenBuildingConstructionGameState* et les deux classes *ClientIcon* et *DraggableClient* est décrite du fait que ces deux classes font partie de l'interface et décrivent respectivement les rectangles déplaçables et l'apparence d'un rectangle sur l'écran. Cela est lié principalement aux données des instances ce qui explique la création des classes et non seulement des widgets à afficher.

- La classe *StorageUtil* permet de stocker et récupérer des données dans l'application ce qui explique sa relation avec le fichier *JSON texts.Json* qui contient les données des champs textuelles de l'application.

- Pour les instances des jeux, la relation entre le fichier *JSON BuildingConstructionInstances.Json* et la classe *ScreenMainMenu* explique qu'au niveau de cette dernière, les données des instances du jeu ont été chargé à partir de ce fichier *JSON* et pourront être récupéré par la suite sur l'interface du jeu en fonction du choix de l'utilisateur effectué sur les Menus des niveaux et des difficultés.

V.3-Modèle

Nous avons dans le code de notre application deux modèles, un pour chacun des jeux initialement prévus. L'application n'en utilise qu'une seule actuellement, celle pour le jeu de construction d'immeuble. L'autre modèle est implémenté pour une utilisation future. La suite va décrire les modèles.
Ces modèles utilisent une autre classe nommée avec le mot *Data*. Elle représente une classe contenant les informations primaires de départ d'une instance du jeu.
Elle sert d'interface supplémentaire de communication entre les différentes classes du programme. Par exemple, elle peut permettre de définir plusieurs méthodes pour obtenir les données d'une instance (par lecture de fichier ou d'une base de données locale ou distante) sans avoir à toucher au modèle des jeux.
Remarque général: le caractère "_" est souvent utilisé devant les noms d'attributs. Dans le langage de programmation Dart cela signifie que c'est un attribut privé de la classe. La suite du document va spécifier les classes, leur objectif, attributs et méthodes.

V.3.a-*BuildingConstruction*

BuildingConstructionModel

Objectif : Représente le modèle du jeu de construction d'immeuble
Attributs :

- double _solutionValue: la valeur de la solution optimale (la valeur à trouver)
- *DescriptionOfBuildings* _descriptionOfBuildings: contient différentes informations par rapport à l'instance.
- List<Client> _clients: liste de *Client*, chaque *Client* contient deux informations: le nombre d'étages demandé et le gain qu'il rapporte.
- List<List<bool>> _stateOfGame: matrice de booléen représentant l'état du jeu.
- double _score: représente le score du joueur.

Constructeurs :

- BuildingConstructionModel(double solution, DescriptionOfBuildings description, List<Client> clients) :
  constructeur de base, initialise l'attribut _stateOfGame.
- BuildingConstructionModel.fromBCD(BuildingConstructionData BCD):
  constructeur utilisant la classe data correspondante

Méthodes:

- void assignClientToBuilding(int i, int j): change l'état du jeu pour que la demande du client soit assignée au bâtiment j.
- void unassignClient(int i): change l'état du jeu pour que la demande du client ne soit assignée à aucun bâtiment.
- void updateScore(): met à jour le score selon l'état du jeu.
- int solutionPercentage(): calcul le pourcetage de réussite du joueur selon son score.

DescriptionOfBuildings
Attributs :

- int _numberOfBuildings: nombre d'emplacement de bâtiments.
- int _maxHeight: hauteur maximum des bâtiments.
- List<double> _pricesOfFloors: liste du prix de chaque étage.

Client
Attributs :

- int _index: indice représentant le numéro du client (utile pour la vue).
- double _earning: le gain que rapporte le client.
- int _requestOfFloors: le nombre d'étages que demande le client.

BuildingConstructionData

Objectif : Représente les données de départ d'une instance du jeu de construction d'immeuble
Attributs :

- double _solutionValue: la valeur de la solution optimale (la valeur à trouver)
- int _numberOfBuildings: nombre d'emplacement de bâtiments.
- int _maxHeight: hauteur maximum des bâtiments.
- List<double> _pricesOfFloors: liste du prix de chaque étage.
- List<double> _earningsFromClients: liste des gains que rapporte chaque client.
- List<int> _requestOfFloors: liste des nombres d'étages que demande chaque client.

V.3.b-*BusLine*

BusLineModel

Objectif : Représente le modèle du jeu de tracé de trajet de bus
Attributs :

- double _solutionValue: la valeur de la solution optimale (la valeur à trouver)
- int _numberOfStations: le nombre de stations.
- List<Coordinates> _stationsCoordinates: liste des coordonnées des stations.
- List<int> _typeOfStations: liste du type de chaque station.
- List<List<bool>> _adjacencyMatric: matrice d'adjacence du graphe.
- List<List<bool>> _stateOfGame: matrice représentant l'état du jeu.
- double _totalDistance: représente la distance totalel du trajet du joueur.

Constructeurs :

- BusLineModel(double solution, int numberofstations, List<Coordinates> coords, List<int> typeofstations, List<List<bool>> adjacencymatrix) :
  constructeur de base, initialise l'attribut _stateOfGame.
- BusLineModel.fromBLD(BusLineData BLD):
  constructeur utilisant la classe data correspondante

Méthodes:

- void createLine(int i, int j):change l'état du jeu pour créer une ligne qui va de i vers j.
- void eraseLine(int i, int j):change l'état du jeu pour effacer une ligne qui va de i vers j.
- double calculateDistance(int i, int j): calcule la distance entre deux sommets i et j du graphe.
- double calculateTotalDistance(): calcule la distance totale du chemin tracé selon l'état du jeu.
- void updateTotalDistance(): met à jour la distance totale.
- int solutionPercentage(): calcule le pourcentage de réussite du joueur

à implémenter:

- une méthode permettant de vérifier qu'un chemin (état du jeu) est une solution réalisable et donc peut être valider. Cette vérification peut être faite par la vue plutôt que par le modèle. (faire en sorte que le joueur ne peut tracer qu'une solution réalisable en le forçant à commencer par le point de départ, à finir par le point d'arriver et sans retourner sur un point déjà visité).

BusLineData

Objectif : Représente les données de départ d'une instance du jeu de tracé de trajet de bus
Attributs :

- double _solutionValue: la valeur de la solution optimale (la valeur à trouver)
- int _numberOfStations: le nombre de stations.
- List<Coordinates> _stationsCoordinates: liste des coordonnées des stations.
- List<int> _typeOfStations: liste du type de chaque station.
- List<List<bool>> _adjacencyMatric: matrice d'adjacence du graphe.
- double _totalDistance: représente la distance totalel du trajet du joueur.

Coordinates

Objectif : Représente les coordonnées dans le plan euclidien
Attributs :

- double _x: coordonnée x.
- double _y: coordonnée y.

V.4-Vue

La vue de l'application correspond à l'interface utilisateur. Celle-ci est implémentée en Dart avec le Framework Flutter au même titre que le modèle. Voici l'interface finale du projet :

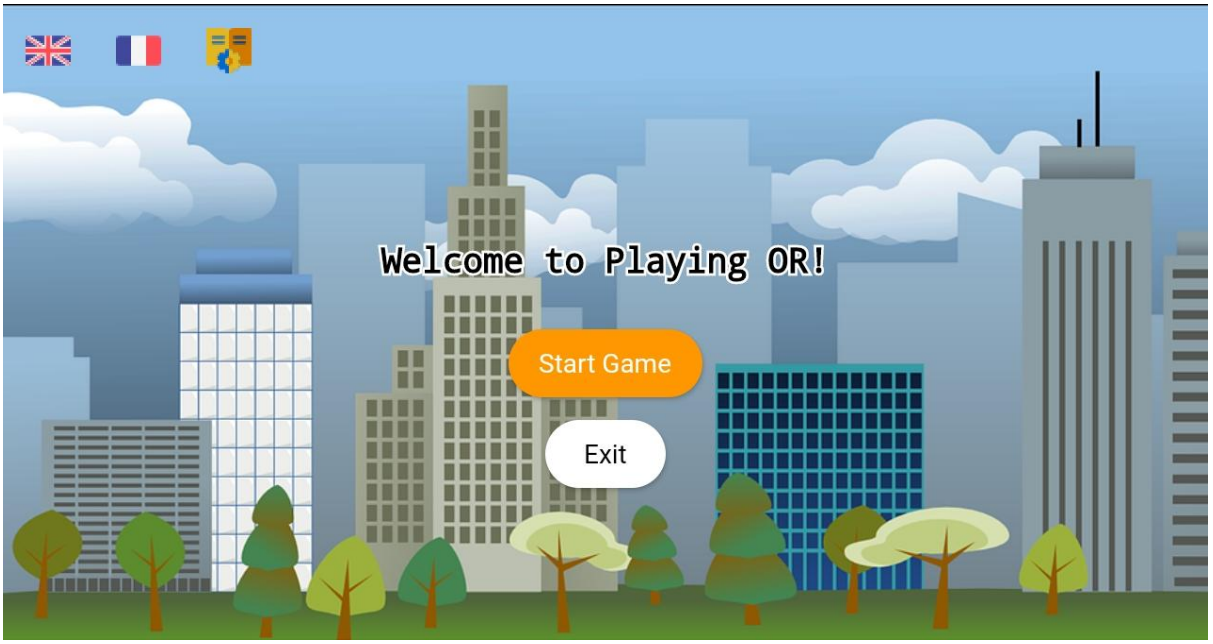


Figure - Page d'accueil de l'application OR game

Le framework flutter utilise des *Widgets*. Ce sont des composants déjà créés en Flutter tels que les boutons ou le texte dont on peut modifier le style et le contenu. Un *Widget* peut en contenir un autre (Exemple : un bouton peut contenir du texte). Aussi, et ce afin de mieux structurer l'application, nous avons principalement utilisé des *Widgets* de type *StatefulWidget* (c'est-à-dire en terme informatique "héritant de" *StatefulWidget*). Les *Widgets* que nous avons créés contiennent un grand nombre d'autres *Widgets* afin de structurer et d'embellir l'écran de jeu.

Tout d'abord, il convient d'expliquer la différence entre *StatelessWidget* et *StatefulWidget*. Un *Widget StatefulWidget*, possède, comme son nom l'indique, un état (*state*). Un état permet de mettre à jour l'affichage lorsque l'utilisateur va faire une action. Cela permettra par exemple de mettre à jour les textes dans la bonne langue lorsque l'utilisateur cliquera sur les icônes de langue. Cependant, lorsque l'affichage n'a pas besoin d'être modifié lors de l'action d'un utilisateur, il convient, et ce pour une raison d'optimisation et de simplicité du code, d'utiliser un *StatelessWidget* (A noter que les boutons sont implémentés autrement et n'ont pas besoin d'état). Cependant, notre application nécessitant une mise à jour de l'affichage dans absolument tous les écrans (due à la mise à jour de l'affichage des langues), tous nos *Widgets* de vues sont des *StatefulWidget*.

Voici un exemple de changement d'état au niveau de la *Widget ScreenMainMenu*.
En cliquant sur l'un des drapeaux en haut gauche de l'interface on sera capable de mettre à jour l'état de cette *Widget* en changeant la langue de tous les champs textuels contenus dans la *Widget*. Cela est réalisé en faisant appel aux données stockées au niveau du fichier *JSON texts.json*.

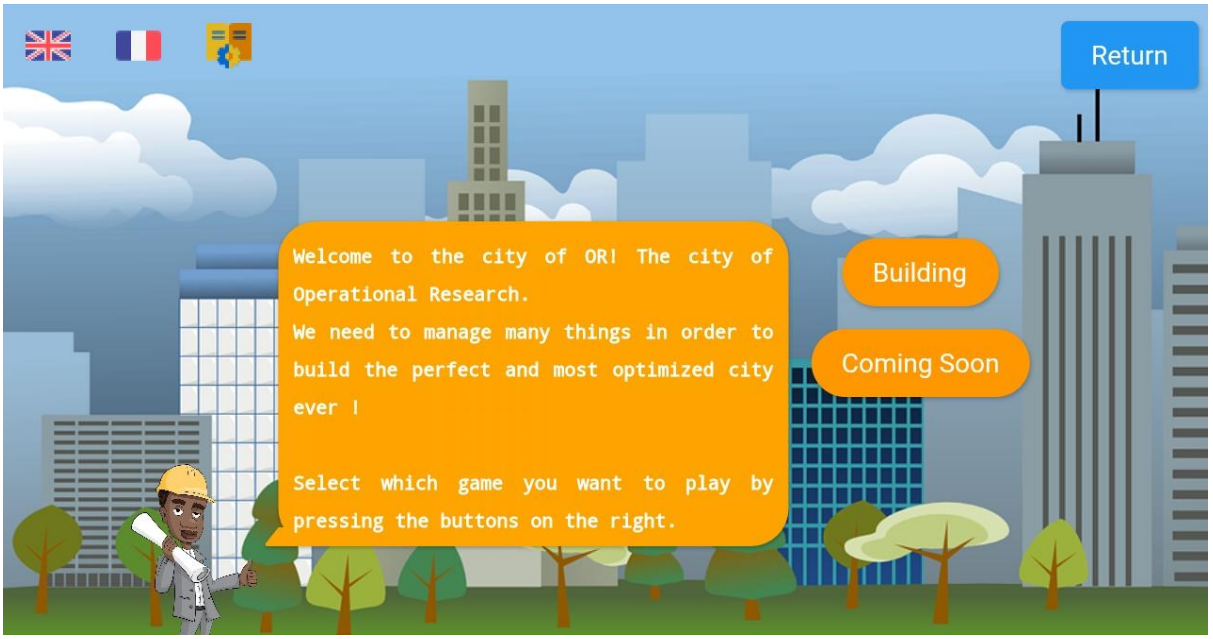
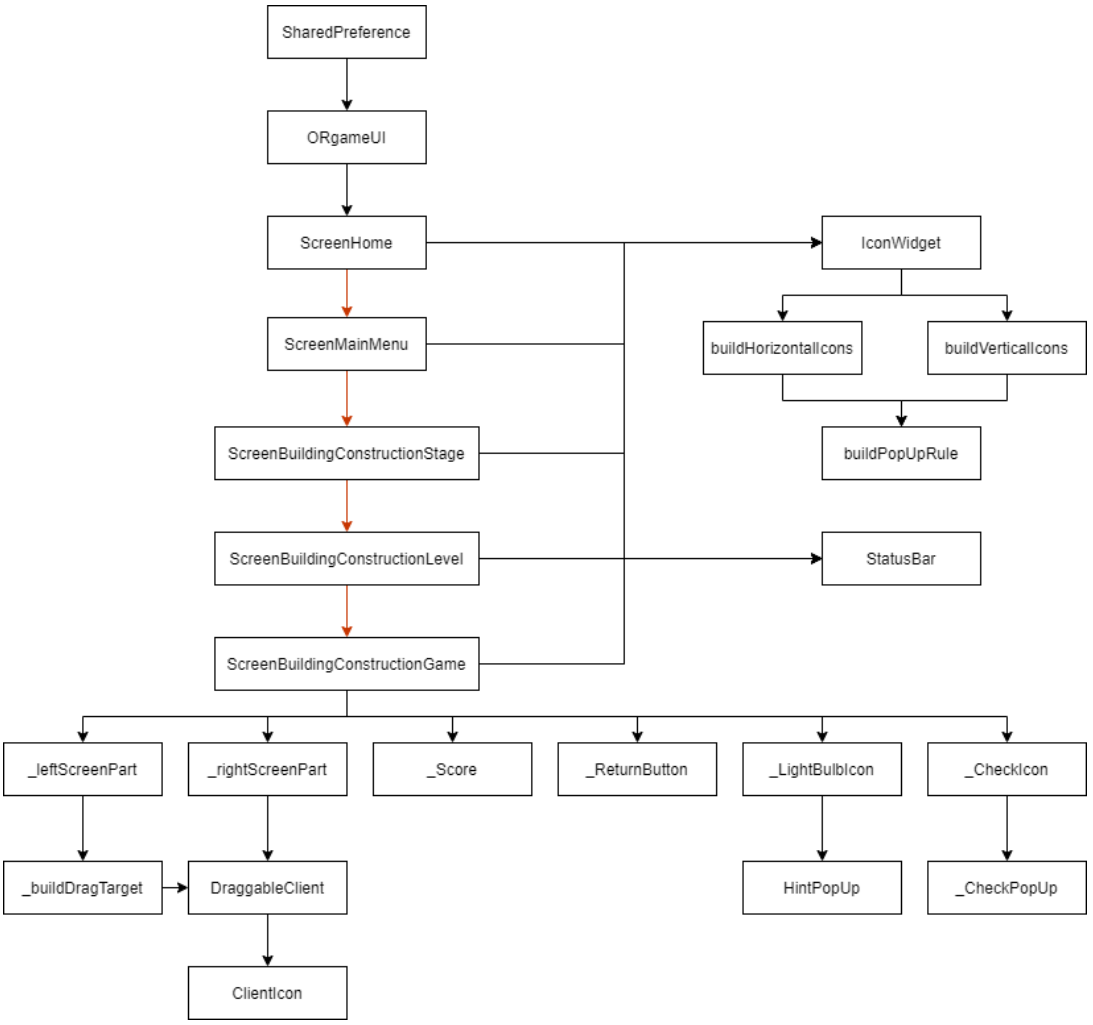


Figure - la vue par défaut, tous les textes sont en langue anglaise

Figure - la vue après avoir cliqué sur le drapeau français

V.4.a-Arbre de Widget simplifié

La vue de l'application est donc composée de Widgets que nous avons implémentés. Voici un Arbre de Widget dans lequel nous avons mis uniquement les Widget que nous avons créés :



Sur ce diagramme, vous pouvez voir qu'il y a des flèches oranges et des flèches noires. Les flèches noires signifient qu'un *Widget* est compris dans un autre. Les flèches oranges signifient qu'il existe une route entre les deux. Une route est tout simplement le passage d'un écran à l'autre. Ici, les écrans sont des *Widgets* à part entière. Ce sont tous les *Widgets* dont le nom commence par *Screen*. L'action qui nous permet de passer d'un écran à l'autre est la pression sur un bouton dans le jeu.

Le *Widget SharedPreference* est celui contenu dans la classe *main.dart*. Celui-ci définit juste le nom et la couleur de l'application et contient le *Widget ORgameUI* qui, lui, définit l'application comme horizontale et est un reste de l'utilisation du *Framework Flame* qui n'est maintenant plus utilisé dans notre application.

Dans la suite de ce rapport, nous allons donner les caractéristiques de chaque *Widget* de cet arbre.

V.4.b-Widgets Génériques

Voici plusieurs Widgets génériques que nous utilisons plusieurs fois dans différents écrans:

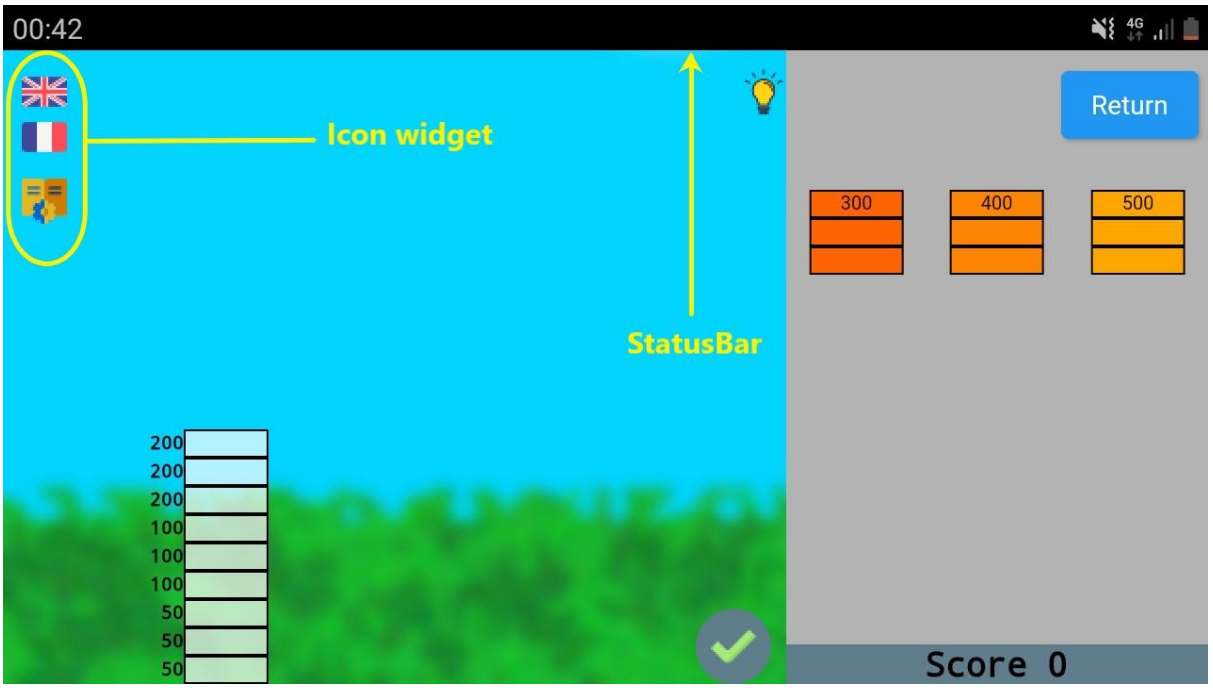


Figure - Les *Widgets :IconWidgets et StatusBar* affichés dans la vue.

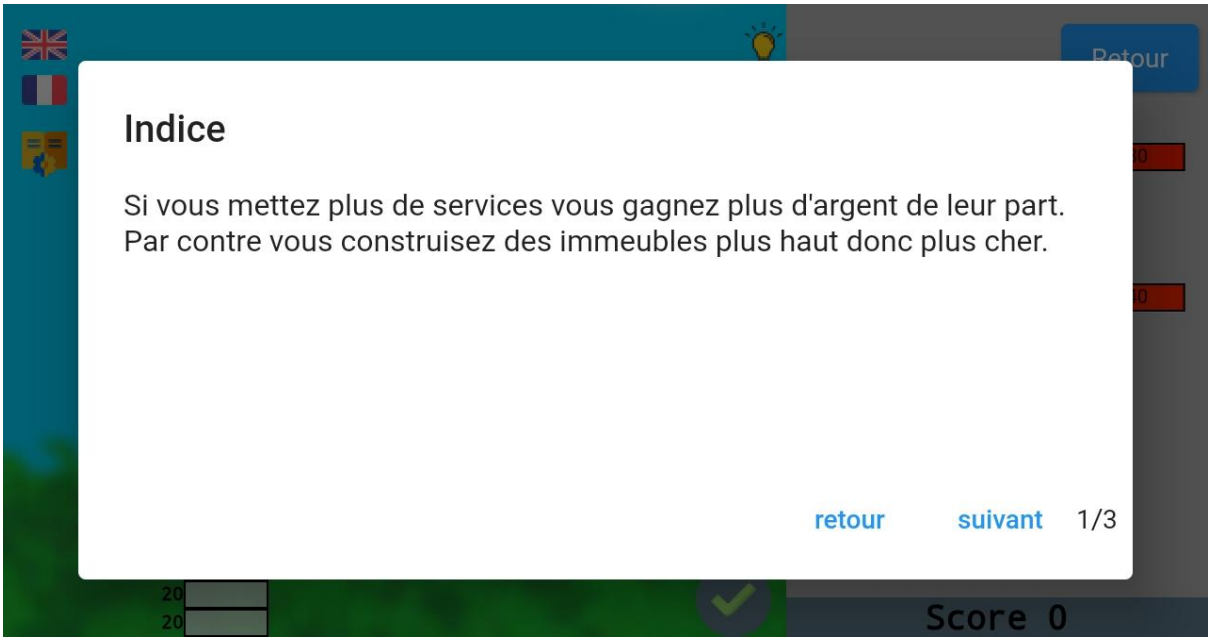


Figure - Une *Widget PopUp* qui représente un indice du jeu

IconWidget

Objectif: Icônes d'options (drapeau anglais/français, manuel d'instructions)
Attributs:

- Axis direction: indique la direction des icônes (affichées en verticale ou horizontale)
- State parent: l'état du Widget parent de IconWidget, permet de mettre à jour l'état du parent quand un bouton d'IconWidget est pressé.

StatusBar

Objectif: barre noire rectangulaire pour que la barre de statut du téléphone ne se superpose pas sur les vues de l'application.

PopUp

Objectif: une fenêtre qui affiche plusieurs pages de textes explicatifs.
Attributs:

- String title: titre de la PopUp
- int maxpages: total du nombre de pages à afficher
- List<String> listOfTexts: liste des textes à afficher sur chaque page.

V.4.c-ScreenHome

Ce fichier regroupe les éléments nécessaires à l'affichage du tout premier écran, celui de bienvenue.

ScreenHome

Type : Classe de type StatefulWidget
Objectif : Créer l'état ScreenHomeState
Constructeur: ScreenHome()

ScreenHomeState

Type : Classe de type State<ScreenHome>
Objectif : Écran de bienvenue permettant de démarrer ou de quitter l'application
Constructeur : ScreenHomeState()

V.4.d-ScreenMainMenu

Ce fichier regroupe les éléments nécessaires à l'affichage de l'écran de choix de jeu. Cet écran possède également une bulle d'explication du jeu.

ScreenMainMenu

Type : Classe de type StatefulWidget
Objectif : Créer l'état ScreenMainMenuState
Constructeur: ScreenMainMenu()

ScreenMainMenuState

Type : Classe de type State<ScreenMainMenu>
Objectif : Écran de sélection du jeu et d'explication du contexte
Constructeur : ScreenMainMenuState()
Attribut:

- List BuildingConstructionDataInstances: données JSON des instances du jeu construction d'immeubles

V.4.e-ScreenBuildingConstructionStage

Ce fichier regroupe les éléments nécessaires pour faire l'écran de sélection de la difficulté de jeu qui correspond à l'interface suivante :

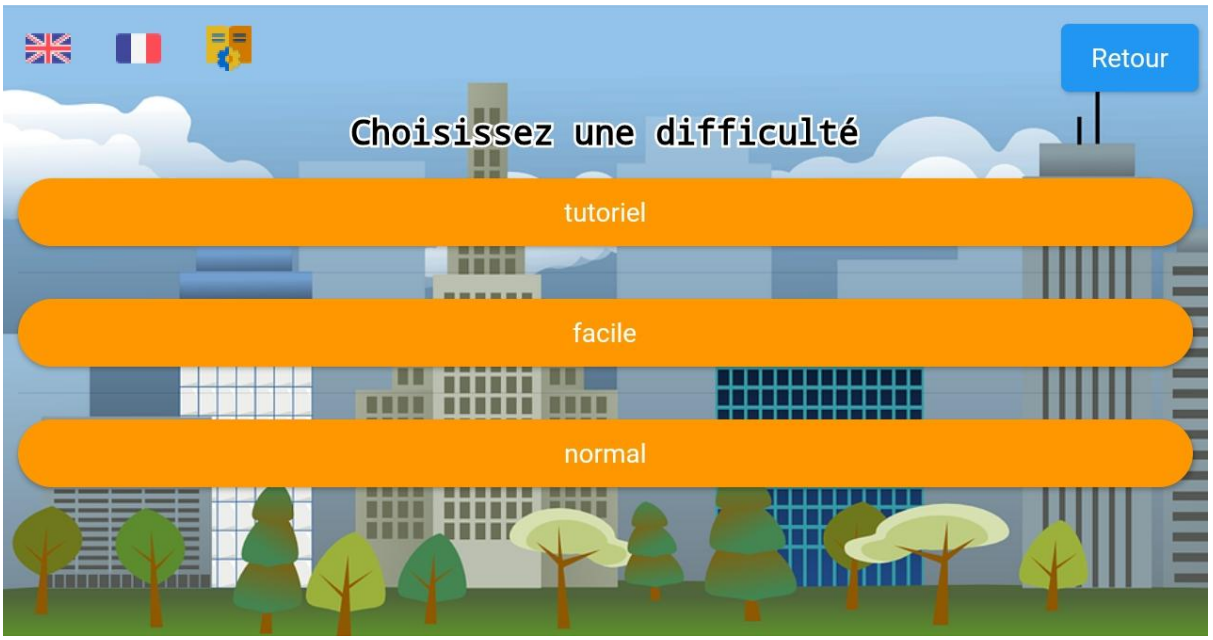


Figure - La vue *ScreenBuildingConstructionStage*

ScreenBuildingConstructionStage

Type : Classe de type StatefulWidget
Objectif : Créer l'état ScreenBuildingConstructionStageState
Constructeur : ScreenBuildingConstructionStage()
Attribut:

- List dataInstances: l'ensemble des instances du jeu construction d'immeubles

ScreenBuildingConstructionStageState

Type : Classe de type State<ScreenBuildingConstructionStage>
Objectif : Écran de sélection de la difficulté du jeu de construction d'immeuble
Constructeur : ScreenBuildingConstructionStageState()
Attribut:

- List dataInstances: l'ensemble des instances du jeu construction d'immeubles.

V.4.f-ScreenBuildingConstructionLevel

Ce fichier regroupe les éléments nécessaires pour faire l'écran de sélection du niveau de jeu qui correspond à l'interface suivante :

Figure - La vue *ScreenBuildingConstructionLevel*

ScreenBuildingConstructionLevel

Type : Classe de type StatefulWidget
Objectif : Créer l'état ScreenBuildingConstructionLevelState
Constructeur : ScreenBuildingConstructionLevel(String difficulty)
Attribut:

- List dataInstances: l'ensemble des instances du jeu construction d'immeubles
- int difficultyIndex: indice indiquant à quel difficulté nous sommes

ScreenBuildingConstructionLevelState

Type : Classe de type State<ScreenBuildingConstructionLevel>
Objectif : Écran de sélection du niveau du jeu de construction d'immeuble
Constructeur : ScreenBuildingConstructionLevelState(String difficulty)
Attributs :

- List dataInstances: l'ensemble des instances du jeu BuildingConstruction
- int difficultyIndex: indice indiquant à quel difficulté nous sommes
- int numberOfLevels: nombre de niveaux à afficher

V.4.g-ScreenBuildingConstructionGame

Ce fichier regroupe les éléments nécessaires pour faire l'écran du jeu de construction d'immeubles. Cet écran étant particulièrement fourni, il a été plus subdivisé en sous-widgets que les autres.
Voici à quoi consiste cette *Widget* comme vue :

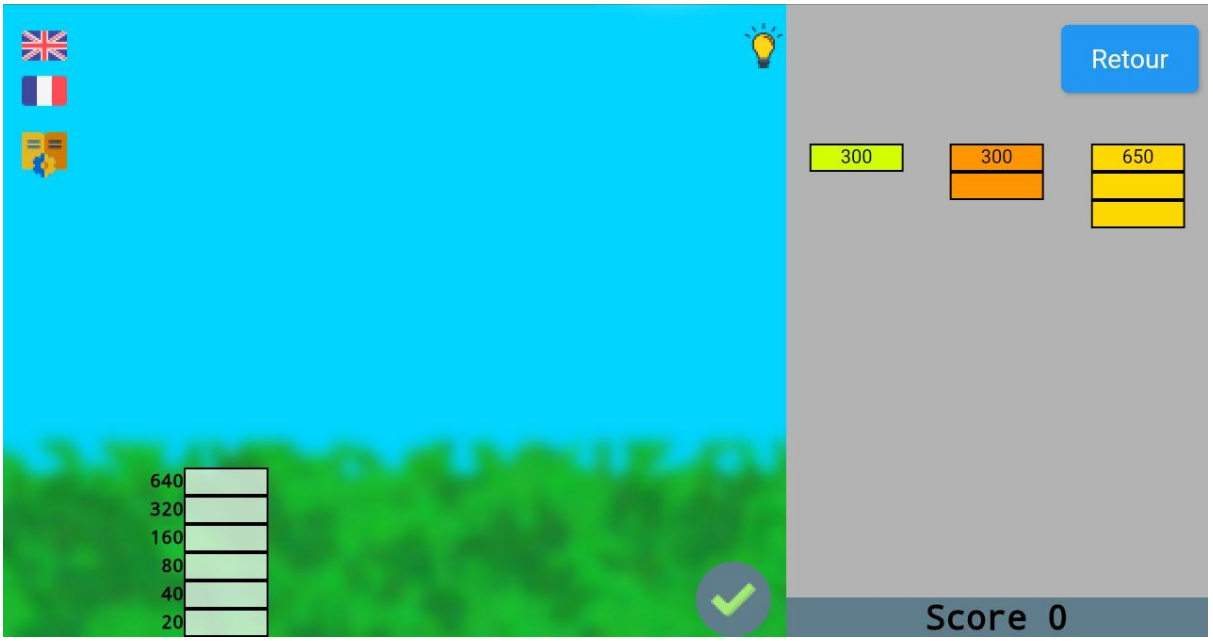


Figure - La vue *ScreenBuildingConstructionGame*

ScreenBuildingConstructionGame

Type : Classe de type StatefulWidget
Objectif : Créer l'état BuildingConstructionGameState
Constructeur : ScreenBuildingConstructionGame(String difficulty, int level, BuildingConstructionController BCC)
Attributs :

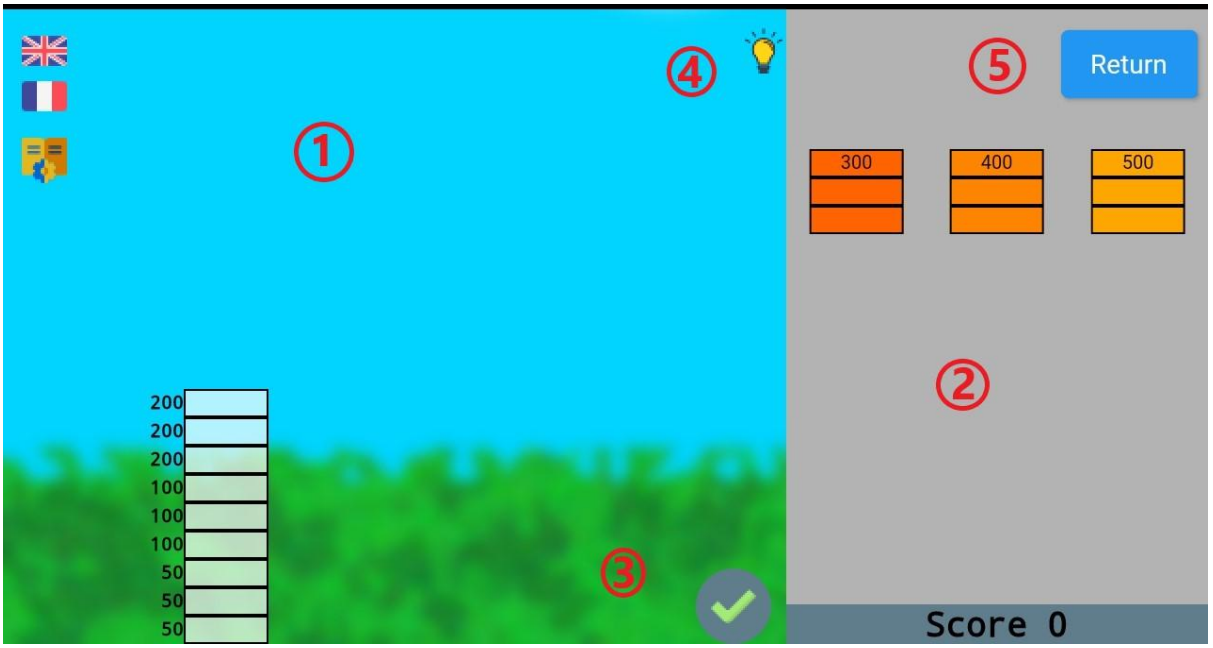
- List dataInstances: l'ensemble des instances du jeu construction d'immeubles
- int difficultyIndex: indice indiquant à quel difficulté nous sommes
- int levelIndex: indice indiquant à quel difficulté nous sommes
- BuildingConstructionController BCC : Contrôleur du jeu de construction d'immeubles

BuildingConstructionGameState

Type : Classe de type State<ScreenBuildingConstructionGame>
Objectif : Écran pour jouer au jeu de construction d'immeuble
Constructeur : BuildingConstructionGameState(String difficulty, int level, BuildingConstructionController BCC)
Attributs :

- List dataInstances: l'ensemble des instances du jeu construction d'immeubles
- int difficultyIndex: indice indiquant à quel difficulté nous sommes
- int levelIndex: indice indiquant à quel difficulté nous sommes
- BuildingConstructionController BCC : Contrôleur du jeu de construction d'immeubles

Ces figures représentent les widgets renvoyés par les méthodes listées ci-dessous :



1. _leftScreenPart

Type : Fonction de BuildingConstructionGameState
Type de retour : Widget
Objectif : partie gauche de l'écran

2. _rightScreenPart

Type : Fonction de BuildingConstructionGameState
Type de retour : Widget
Objectif : partie droite de l'écran

3. _CheckIcon

Type : Fonction de BuildingConstructionGameState
Type de retour : Widget

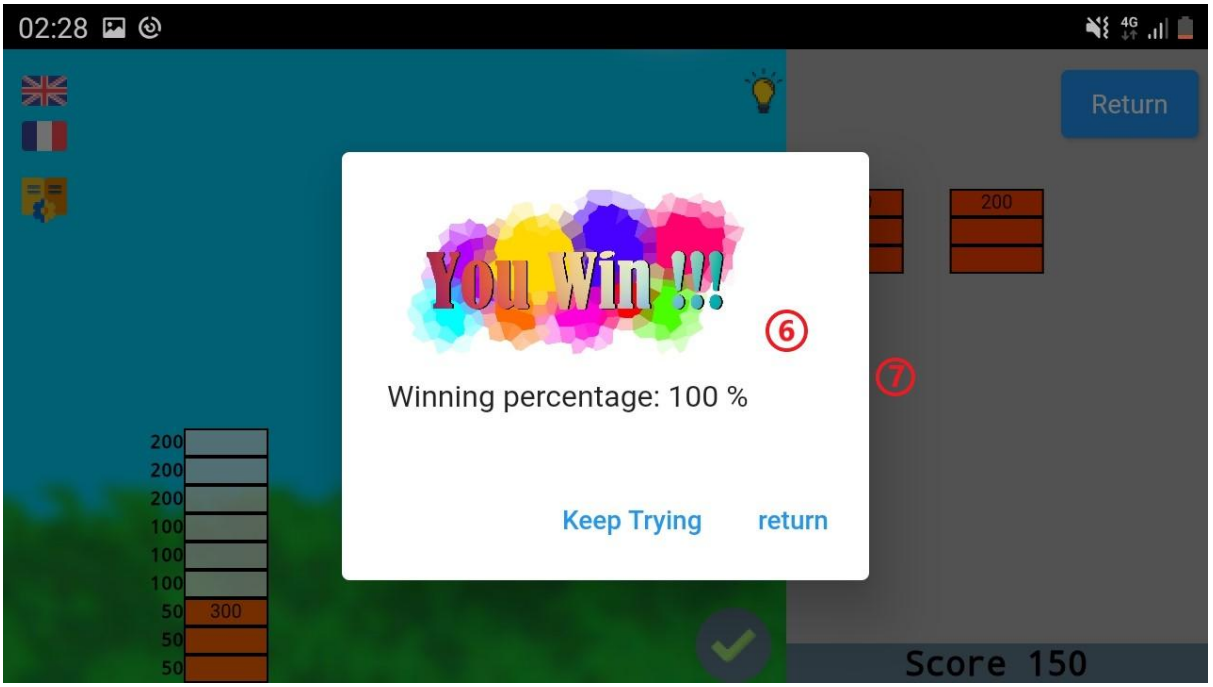
Objectif : icône de validation

4. _LightBulbIcon

Type : Fonction de BuildingConstructionGameState
Type de retour : Widget
Objectif : icône d'ampoule pour afficher des indices

5. _ReturnButton

Type : Fonction de BuildingConstructionGameState
Type de retour : Widget
Objectif : bouton de retour



6. _Score

Type : Fonction de BuildingConstructionGameState
Type de retour : Widget
Objectif : représente le score

7. _CheckPopUp

Type : Fonction de BuildingConstructionGameState
Type de retour : Widget
Objectif : PopUp spéciale montrant le pourcentage de réussite

D'autres *Widgets* :

_buildDragTarget

Type : Fonction de BuildingConstructionGameState
Type de retour : Widget DragTarget<Client>

Objectif : Construire la zone des immeubles où on peut déposer les services

DraggableClient

Type : Classe de type StatelessWidget
Objectif : Créer des services déplaçable
Constructeur : DraggableClient(Client client)
Attribut : Client client : Objet du modèle du jeu
Type de retour : Widget Draggable<client>

ClientIcon

Type : Classe de type StateLessWidget
Objectif : Créer l'apparence d'un service sur l'écran selon un Client donnée
Constructeur : ClientIcon(Client client)
Attribut : Client client : Objet du modèle du jeu
Type de retour : Widget Align qui représente un service

colorRatio

Type : Fonction de Client_Icon
Objectif : Donner une couleur au service selon son ratio gain/nombre d'étage
Type de retour : Color

V.5-Contrôleur

Comme pour les modèles nous avons deux contrôleurs, un pour chaque jeu. Seul celui pour le jeu de construction d'immeubles est utilisé actuellement.

BuildingConstructionController

Objectif : contrôleur du jeu de construction d'immeubles
Attributs :

- BuildingConstructionModel _BCM: modèle du jeu.
- BuildingConstructionData _BCD: données primaires de départ de l'instance du jeu.

Constructeurs :

- BuildingConstructionController (BuildingConstructionModel BCM, BuildingConstructionData BCD): constructeur de base.
- BuildingConstructionController.fromBCD(BuildingConstructionData BCD): constructeur utilisant la classe data correspondante

Méthodes:

- List<int> getClientsIndexesInBuilding(int buildingIndex): renvoie la liste des indices des clients qui sont dans le bâtiment donné en paramètre (sous forme d'indice).
- int getNumberOfClientsInBuilding(int buildginIndex): renvoie le nombre de clients qui sont dans le bâtiment donné en paramètre (sous forme d'indice).

BusLineController

Objectif : contrôleur du jeu de tracé de trajet de bus
Attributs :

- BuildingConstructionModel _BLM: modèle du jeu.
- BuildingConstructionData _BLD: données primaires de départ de l'instance du jeu.

Constructeurs :

- BusLineController(BusLineModel BLM, BusLineData BLD):constructeur de base.
- BusLineController.fromBLD(BusLineData BLD): constructeur utilisant la classe data correspondante.

A implémenter:

- bool isValidSolution(): doit vérifier si l'état du jeu du modèle est une solution valide.

V.6-Test

Pour effectuer les tests sur la vue, cela se fait directement en lançant l'application et en identifiant les problèmes en les voyant.
Pour les modèles, les classes *data* et les contrôleurs nous avons effectué des tests unitaires dans deux fichiers, un pour chaque jeu: BuildingConstructionUnitTests et BusLineUnitTests. Ils testent les fonctions basiques et actions des différentes classes modèle, contrôleur et *data* (création, méthodes, comparaison d'égalité, …).

V.7-Récupération des données et stockage de paramètres

Cette partie fournit quelques détails sur la façon dont nous récupérons des données et les stockons pour être utilisées.

Le JSON est un format textuel de représentation des données. Nous l'avons choisi pour stocker à la fois les instances des problèmes du jeu et les textes à afficher.
Le format JSON pour stocker les textes affichés est le suivant:

```
[
  {
    "textName": "welcomeText",
    "EN": "Welcome to Playing OR!",
    "FR": "Jouons à la RO!"
  },
  {
    "textName": "startGameButtonText",
    "EN": "Start Game",
    "FR": "Jouer"
  },
  ...
]
```

C'est une liste d'éléments ayant trois sous-éléments:

- textName: un identifiant sous la forme d'un nom permettant d'identifier le texte
- EN: le texte en anglais
- FR: le texte en français

Ensuite pour stocker et récupérer ces textes nous utilisons la classe suivante:

StorageUtil

Objectif: classe utilitaire permettant de stocker et récupérer des données.
Source d'inspiration du code:
https://fluttercorner.com/how-to-store-and-get-data-from-shared-preferences-in-flutter/
Méthodes principales utilisées:

- Future<StorageUtil> getInstance() : récupère l'instance du stockage.
- Future<bool> putString(String key, String value) : stocke la valeur *value* à la clé *key*.
- String getString(String key, {String defValue = ''}) : récupère la valeur stockée à la clé donnée.
- storeAllTexts() : lis le fichier texts.json et stocke tous les textes qui seront affichés dans l'application.

Fonctions hors-classe:

- bool isLangValid(String lang) : vérifie si la langue donnée en paramètre est valide.
- String getText(String textName): renvoie le texte stockée correspondant à l'identifiant *textName* dans la langue courante

En plus de cela, au lancement nous stockons la langue courante qui change si l'utilisateur appuie sur un bouton et nous stockons tous les textes.

En jouant sur le nom des clés de stockage avec le nom des langues durant le stockage et la récupération depuis le stockage nous récupérons les textes dans la bonne langue. Pour ajouter un texte il suffit de le rajouter à la suite du fichier texts.json dans le même format et pour rajouter une langue il suffit de rajouter la traduction de chaque texte dans le fichier texts.json et de modifier quelques lignes dans la classe *StorageUtil*.

Pour le stockage des instances nous le faisons un peu différemment, voici le format du fichier JSON des instances du problème du jeu BuildingConstructionInstances.json:

```
[
  {
    "difficultyEN": "tutorial",
    "difficultyFR": "tutoriel",
    "instances":
    [
      {
        "solutionValue":450.0,
        "numberOfBuildings":1,
        "maxHeight":9,
        "pricesOfFloors":[50, 50, 50, 100, 100, 100, 200, 200, 200],
        "earningsFromClients":[300, 400, 500],
        "requestsOfFloorsFromClients":[3, 3, 3]
      },
        ...
    ]
  },
  {
    "difficultyEN": "easy",
    "difficultyFR": "facile",
    "instances":
    [
      {
        "solutionValue":180.0,
        "numberOfBuildings":1,
        "maxHeight":11,
        "pricesOfFloors":[20,20,40,40,80,80,80,100,100,100,100],
        "earningsFromClients":[50,300,30,40,200],
        "requestsOfFloorsFromClients":[2,4,1,1,3]
      },
        ...
    ]
  },
  ...
]
```

Nous avons une liste d'éléments contenant trois sous-éléments:

- difficultyEN: le nom de la difficulté en anglais
- difficultyFR: le nom de la difficulté en français
- instances: une liste d'instance qui contiennent les informations par rapport à une instance du problème.

Lors de la création des différentes vues, nous stockons toutes ces données dans une variable qui est passée de vue en vue et qui crée les différents boutons de choix de niveau et de difficulté selon la taille et l'ordre dans ce fichier JSON. Attention l'ordre dans lequel les instances sont mis dans "instances" défini l'ordre des niveaux. En jouant sur les indices et les noms qui sont récupérés lorsque nous appuyons sur les boutons qui dirigent vers la vue suivante nous savons quelle instance utiliser lorsque nous arrivons dans un niveau.

V.7.b Particularité de Flutter/Dart pour le chargement de fichiers (textes)

Cette partie est essentiellement à l'attention de la future équipe qui récupérera ce projet.

Pour lire un fichier et récupérer des données de celui-ci, ce n'est pas aussi simple que dans d'autres langages. Les fonctions Dart permettant de lire un fichier texte comme du JSON sont des méthodes utilisant les mots-clés *await*, *async*, *Future.*

Pour simplifier, les fonctions contenant *Future* comme celle permettant de charger des fichiers JSON, ne sont pas forcément exécutées lorsqu'elles sont appelées. Pour être sûr qu'elles s'exécutent il faut mettre devant leur appel le mot *await*.
(par exemple si vous utilisez une fonction *loadJsonData()* qui lit un fichier json et met ses données dans une variable il ne faut pas juste appeler *loadJsonData()* dans le code mais *await loadJsonData()*)
Cependant si le mot *await* se trouve dans une fonction, il faut que celle-ci ait le mot clé *async*. Or si faire ceci dans certaines fonctions prédéfinies de flutter n'est pas possible.

De manière pratique voici un des problèmes que nous avons rencontrés:
-Nous voulions charger un fichier JSON lorsque nous arrivions sur une vue (un Widget) qui utilise ces données.
-Nous voulions donc mettre la fonction de chargement du fichier dans la fonction *build* du Widget, fonction *build* qui renvoie le Widget et ses éléments.
-Nous devions donc mettre un *await* à l'intérieur ce qui obligeait la fonction *build* à être asynchrone (avec le mot clé *async*) ce qui était impossible pour la fonction *build* qui renvoyait un *Widget* et non pas un *Future*<*Widget*>.
-Pour s'en sortir nous devions:
Soit charger le JSON dans le main avant d'appeler les méthodes qui crée la vue et stocker les données quelque part (ce que nous avons fait pour les textes de l'application).
Soit charger le JSON lorsque dans la vue précédente nous appuyons sur le bouton qui mène à la vue qui utilisera le JSON, car la méthode *OnPressed()* d'un bouton, que nous utilisons pour se diriger de vue en vue, peut être rendu asynchrone avec *async*.

Ce fonctionnement nous a demandé quelque temps à comprendre. Nous conseillons la future équipe qui reprendra ce projet faire attention à cette partie du code et à se documenter sur ce fonctionnement, quelques liens utiles:
https://dart.dev/codelabs/async-await
https://neeraj-s-sharma.medium.com/flutter-read-json-file-stored-in-asset-folder-721b86762c22
(le lien vers le tutoriel principal que nous avions utilisé n'est plus disponible semble-t-il:
https://flutter-tutorial.com/flutter-read-local-json-file-from-assets)