Projet C++ : Paris JO Explorer

Romain CAPRON, Yannick ZHANG - MAIN4

1 Introduction

Dans le cadre de l'un des événements sportifs les plus prestigieux et attendus au monde, les Jeux Olympiques, notre équipe a développé une application unique destinée à enrichir l'expérience des spectateurs, des touristes et des résidents de Paris. Paris JO Explorer est conçue pour être le compagnon indispensable pour naviguer dans l'effervescence des Jeux Olympiques, offrant un accès simplifié et interactif à une multitude d'informations et de ressources.

Cette application, développée avec la puissance et la flexibilité de Qt, présente une interface intuitive qui guide les utilisateurs à travers les divers aspects des Jeux Olympiques. Elle inclut une liste détaillée des épreuves sportives, offrant des informations sur les horaires, les lieux et les disciplines, ainsi que des données contextuelles pour enrichir la compréhension et l'appréciation de chaque événement. Ainsi qu'une carte, afin de mieux se repérer dans Paris.

FIGURE 1 – Logo de notre application (fait par IA)

Pour pouvoir lancer notre application, vous pouvez vous diriger vers notre GitHub. Vous y trouverais aussi les instructions détaillées dans le fichier README pour installer les dépendance nécessaire et les étapes à suivre faire pour compilé et exécuter notre code.

2 Description de l'application

Nous avons souhaité développer une application qui permette de naviguer parmi les différentes épreuves des JO, ainsi que les restaurants. Afin que l'utilisateur puisse savoir les epreuves qui vont avoir lieu, quand elles auront lieu, avec une petite description, ainsi que les transports pour y accéder, et même les restaurants à proximité des épreuves afin de se restaurer facilement. La quasi-intégralité du projet est développée en C++ (plus de 99%), mais nous avons utilisé du JSON pour notre base de données, ainsi que du HTML pour l'affichage de certaines informations. Voyons plus en détail comment s'organise le fonctionnement de l'application.

2.1 Fonctionnement global de notre application

Notre application possède **3 fenêtres principales**, qui concentrent la majorité des fonctionnalités. Une **fenêtre d'accueil**, une **fenêtre d'informations**, une **carte**. Et une fenêtre de dialogue, qui va permettre d'afficher des informations détaillées de chaque lieu. Toutes les informations de la fenetre d'information et de la fenêtre de dialogue viennent **directement** de notre base de données. Cela signifie qu'elles ne sont pas écrites artificiellement, un changement de la base de donnée changera donc l'affichage. Des boutons en bas de l'écran vous permettent de passer d'une fenêtre à l'autre, le bouton le plus à droite permet de quitter l'application.

— **Fenêtre d'accueil :** C'est la fenêtre qui s'affiche normalement lors du lancement de l'application. Elle vous souhaite la bienvenue aux JO de Paris dans plusieurs langues !

FIGURE 2 – Diagramme UML

— **Fenêtre d'informations :** C'est la qui répertorie l'intégralité des lieux (épreuves et restaurants). Cette fenêtre est dotée d'un **défilement vertical**, qui est utile lorsque un grand nombre d'éléments y sont affichés. Nous avons également implémenté une **barre de recherche**, qui permet de faire une recherche soit par le **nom** (d'une épreuve ou d'un restaurant), soit par **l'adresse** du lieu. Un onglet de **filtres** est également présent, il contient 3 options, **Tout** (pas de filtre), **Epreuves** (filtre pour ne voir que les épreuves) et **Restaurants** (filtre pour ne voir que les restaurants). L'intérêt est que la **barre de recherche** continue de fonctionner même lorsqu'un filtre est actif, permettant de trouver encore plus rapidement ce que l'on cherche. Ensuite, les lieux qui défilent sur cette fenêtre sont tous rendus **cliquables**. Cela signifie qu'une fois votre épreuve ou restaurant trouvé, vous

pouvez cliquer dessus pour afficher de plus amples informations. Une fenêtre de dialogue s'ouvre alors.

FIGURE 3 – Diagramme UML

— **Fenêtre détaillée :** C'est la fenêtre de dialogue qui s'ouvre lorsqu'on clique sur un des items de la fenêtre scrollable d'informations. Elle contient des informations différentes pour les **épreuves** et les **restaurants**. Pour les restaurants : un drapeau du pays des spécialités, le nom, l'adresse, la plage horaire, la spécialité, la description, ainsi que les épreuves à proximité de ce restaurant. Pour les épreuces, une image du lieu où se déroule l'épreuve, le nom, l'adresse, la date et l'horaire, le prix du billet, la description, les transports pour s'y rendre, et les restaurants à proximité.

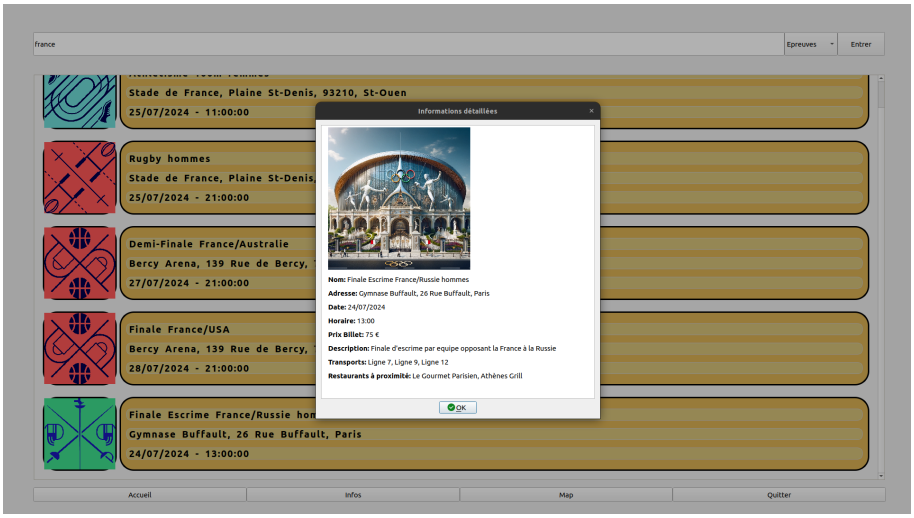

FIGURE 4 – Diagramme UML

— **Carte :** C'est une carte des lieux (restaurants et épreuves). Lorsque vous passez votre curseur de souris sur l'un des lieux, le nom de celui-ci s'affiche. C'est la seule implémentation "artificielle" des noms de notre programme. Cela veut dire que ces noms ne sont pas tirés directement de la base de donnée, mais écrits par nous. (dû à un manque de temps pour le développement, nous aurions évidemment voulu utiliser notre base de donnée et avoir une carte cliquable et plus fonctionnelle)

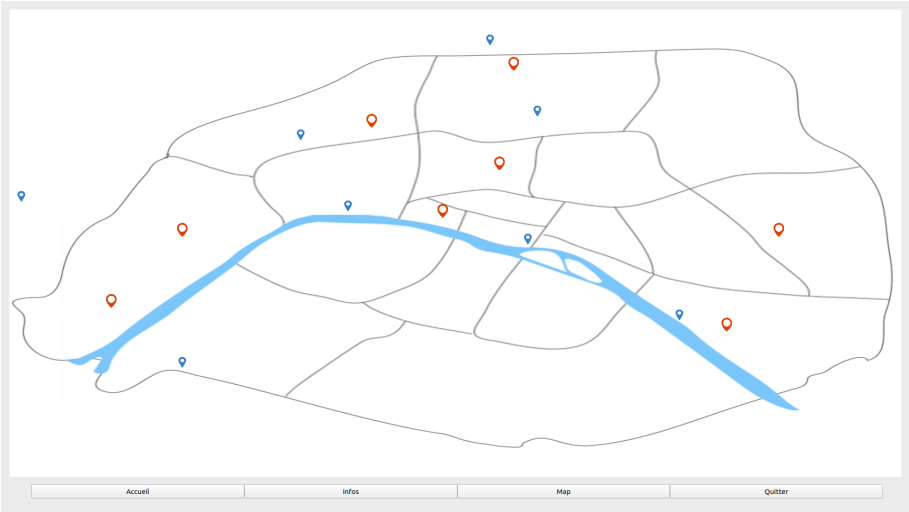

FIGURE 5 – Diagramme UML

2.2 Structuration de notre base de données

Comme dit précédemment, notre application prend toutes ses informations d'une base de donnée, écrite par nous, en JavaScript Object Notation. Il est donc important de noter que la grande majorité des informations affichées à l'écran ne sont pas "artificielles", mais belles et bien mises à jour directement si l'on touche la base de donnée. Cela fait partie de l'enjeu de notre projet, l'intérêt n'est pas dans le fait d'afficher des fenêtres d'informations, mais bien d'avoir une application capable de lire une base de donnée potentiellement mise à jour dans le futur.

Cette base de données est structurée ainsi :

— **"lieux" :** C'est la clé de premier niveau. Elle engloble l'ensemble de la base de donnée, à savoir toutes les épreuves, et tous les restaurants.
— **"Epreuves" :** Une des deux clés de second niveau, elle possède des attributs en commmun avec les restaurants, mais aussi ses attributs propres.


```
"Epreuves": [
    {
        "id": 1,
        "nom": "Athlétisme 100m femmes",
        "adresse": "Stade de France, Plaine St-Denis, 93210, St-Ouen",
        "description": "100m d'athlétisme pour les femmes",
        "image_lieu": ":/images/stade_de_france.jpg",
        "image_epreuve": ":/images/athletisme.jpg",
        "transports": [
            "Ligne 4",
            "Rer B"
        ],
        "horaireDebut": "2024-07-25T11:00:00",
        "prixBillet": 50.00,
        "proximiteRestaurant": [22]
    },
```


FIGURE 6 – Structure JSON "Epreuves"

— **"Restaurants :** Une des deux clés de second niveau, elle possède des attributs en commun avec les épreuves, mais aussi ses attributs propres.

```
"Restaurants": [
    {
        "id": 21,
        "nom": "Le Gourmet Parisien",
        "adresse": "789 Rue de la Gastronomie, Paris",
        "description": "Cuisine française traditionnelle",
        "image": ":/images/francais.jpg",
        "transports": [
            "Ligne 1",
            "Ligne 4",
            "Ligne 7",
            "Ligne 9",
            "Ligne 11",
            "Ligne 12",
            "Ligne 13",
            "Ligne 14"

        ],
        "plageHoraire": "12:00 - 23:00",
        "specialite": "Cuisine française",
        "proximiteEpreuve": [ 5, 6, 10, 11, 20

        ]
    },
```


FIGURE 7 – Structure JSON "Restaurants"

On notera que les attributs **"proximiteRestaurants"** et **"proximiteEpreuves"** ne donnent pas directement une chaine de caractères afin d'avoir les épreuves et noms à proximité, mais bien les **"id"** de ces lieux. Cela permet une plus grande flexibilité quant à la modification des noms des épreuves et des restaurants.

2.3 Diagramme UML et structure du code

Avant d'entrer en détail dans la manière dont notre projet est structuré (classes, méthodes etc..), voici le diagramme UML complet :

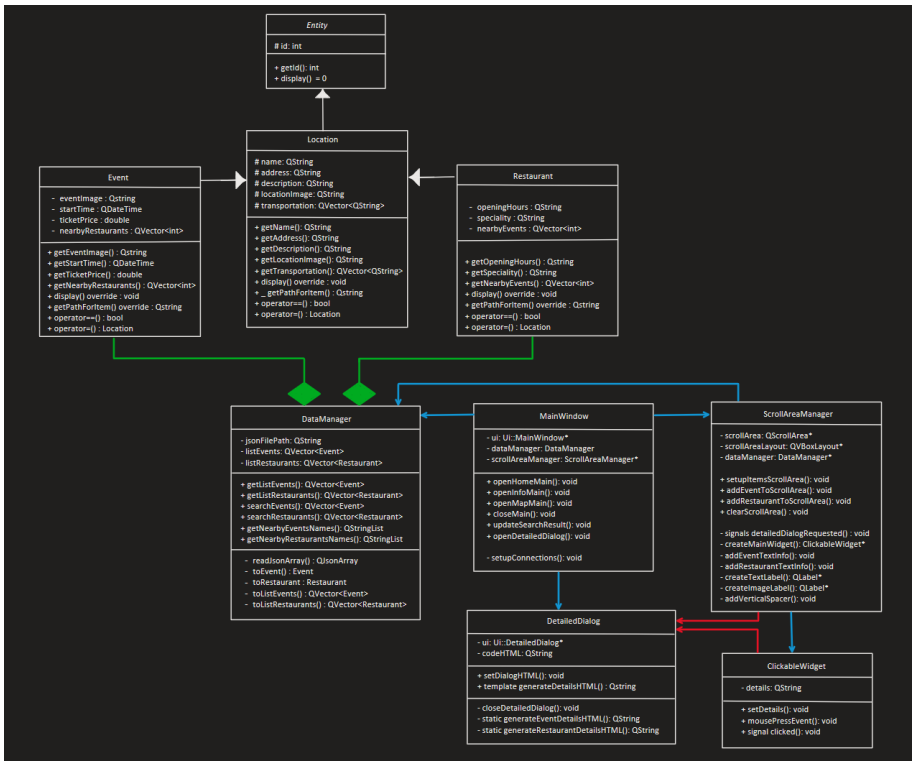

FIGURE 8 – Diagramme UML

Dans ce diagramme, les flèches **blanches** correspondent à des flèches **d'héritage**. Les flèches **vertes** sont des flèches **d'aggrégation**. Les deux flèches **rouges** indiquent qu'un

signal est **émit** vers ce qu'elles pointent, tandis que les flèches **bleues** indiquent une **utilisation** d'une classe par une autre.

Nous avons donc la classe mère **"entity"**, classe abstraite, qui ne possède qu'un **attribut** protégé (id), et deux méthodes dont une virtuelle **display** (virtuelle), et **getId**. Elle possède une classe fille **location**, qui va englober un bien plus grand nombre d'attributs et de méthodes. Cette classe possède une méthode **getPathForItem** qui est virtuelle, et donc être redéfinie dans ses classes filles. Deux surcharges d'opérateurs sont également présents, (==) et (=), ainsi que la majorité des attributs utiles à notre programme. Enfin, **location** possède elle même deux **classes filles**, **Event** et **Restaurant**. Nous avons procédé ainsi car la plupart des attributs peuvent être partagés entre une épreuve et un restaurant, mais certains attributs sont bien spécifiques à l'un ou l'autre, comme la spécialité du restaurant, ou encore le prix du ticket d'une épreuve. De plus, ces deux classes font une utilisation différente de la méthode **"display"**, qui permet d'afficher les attributs de chaque location. (cela est directement lié au fait qu'ils n'aient pas les mêmes attributs à afficher)

- **Classe Event :** c'est cette classe qui caractérise toutes les **épreuves** dans notre application. Elle reprend donc les attributs généraux de la classe **location**, et y ajoute ceux spécifiques aux épreuves. Comme les **eventImage** qui correspondent aux pictogrammes de chaque sport, **startTime**, qui n'est autre que l'horaire et la date de début de l'épreuve, **ticketPrice** qui nous donne le prix du ticket et **nearbyRestaurants**, qui correspond aux restaurants à proximité de l'épreuve. Enfin, comme citées précédemment, les deux méthodes virtuelles getPathForItem et la surcharge d'opérateur == sont redéfinies ici pour s'adapter spécifiquement aux **épreuves**. Sans oublier les autres getters spécifiques **getStartTime getEventImage getTicketPrice et getNearbyRestaurants**.
- **Classe Restaurant :** c'est cette classe qui caractérise tous les **restaurants** dans notre application. Elle reprend tous les attributs généraux de la classe **location**, et y ajoute ceux spécifiques aux restaurants. Comme l'attribut **plageHoraire** ou **spécialité**. Une méthode analogue à **nearbyRestaurants** est également créée pour nous donner les épreuves à proximité des restaurant, elle s'appelle **nearbyEvents**. Toutes les surcharges d'opérateur sont également redéfinis ainsi que les getters spécifiques aux restaurants : **getOpeningHours, getSpecialty, getNearbyEvents**.

Autre classe importante, **dataManager**, c'est celle-ci qui nous permet de gérer les données venant de notre base de données JSON, pour ensuite faire le lien entre le **backend** et le **frontend**. Elle nous est utile pour céer des vecteurs de restaurants ou d'épreuves (QVector pour l'utilisation avec Qt), à partir d'un fichier JSON, pour ensuite les exploiter dans la suite du programme. C'est la méthode **readJsonArray** qui permet de lire le fichier JSON. Les méthodes **toEvent** et **toRestaurant** sont capitales, elles permettent respectivment de "convertir" un objet JSON en un objet **Event** ou **Restaurant**, par la suite exploitable dans les classes **Event** et **Restaurant**. Les méthodes **toListEvents** et **toListRestaurant** créent une liste d'objets **d'Events** ou de **Redstaurants** à partir du fichier JSON. Vis à vis de notre **searchbar**, les deux méthodes **searchEvents** et **searchRestaurants** permettent de rechercher des events ou des restaurants, en fonction de leur nom, ou de leur adresse (ainsi qu'à l'aide du filtre implémenté). Les deux dernières méthodes, **getNearbyEventsNames** et **getNearbyRestaurantsNames** permettent d'obtenir les **noms** des épreuves et restaurants à proximité, à l'aide de leur Id, ce qui permet une grande flexibilité dans le changement des noms de ceux-ci (si nécessaire).

Enfin, les 4 classes restantes sont toutes des classes intrasèquement liées à **Qt** et aux **fenêtres**. À savoir la **MainWindow**, le **ScrollAreaManger**, le **DetailedDialog** et le

ClickableWidget. La classe **MainWindow** est celle qui va contenir toutes les méthodes permettant d'interagir avec l'interface utilisateur. Le **ScrollAreaManager** va gérer tout ce qui est en lien avec le panneau d'informations scrollables. **DetailedDialog** est la classe qui gère la fenêtre de dialogue et toutes les informations qui y sont affichées, tandis que **ClickableWidget** rend cliquable les items présents dans l'interface scrollable, afin justement d'afficher cette fenetre de dialogue.

3 Exploitation des contraintes

Des contraintes ont été établies afin de garantir une qualité minimale et une structure cohérente du code. Il y'a le **diagramme UML** de la partie précédente et les suivantes :

8 classe minimum et 3 niveaux de hiérarchie : Dans le diagramme UML on peux voir qu'on a en tout 9 classe avec chacun leur fonctionnalité. De plus, le projet possède une structure avec trois niveaux de hiérarchie. La classe de base **Entity** agit comme la classe mère, pour la classe **Location** qui elle-même est une classe mère pour **Event** (pour les événements) et **Restaurant** (pour les restaurants). Bien que la classe **Entity** ne semble pas actuellement nécessaire dans l'application, étant donné que l'accent est actuellement mis sur les fonctionnalités basées sur les lieux, elle peut être utile pour des extensions futures. Par exemple, si l'application devait être élargie pour inclure des fonctionnalités impliquant des personnes, **Entity** pourrait servir de classe de base pour des entités plus génériques, et des classes spécifiques pourraient être créées pour des entités spécifiques, comme **People**. En intégrant la classe **People**, on pourrait alors établir une hiérarchie supplémentaire, avec **People** en tant que classe intermédiaire, et des classes plus spécifiques dérivées telles que **Athletes** pour représenter différents athlètes des JO. Ou encore **User** pour créer un système de connexion. Cela **Entity** offre donc une flexibilité et une extensibilité à l'application pour prendre en charge plus de fonctionnalités à l'avenir.

Fonctions virtuelles (pure et normal) Le projet intègre des concepts de polymorphisme avec l'utilisation de fonctions virtuelles pures. Par exemple, la méthode `display()` dans la classe **Entity** est une fonction virtuelle pure, obligeant les classes dérivées telles que **Location**, **Event**, et **Restaurant** à la redéfinir, favorisant ainsi une personnalisation spécifique à chaque type d'entité. De plus on a une autre fonction virtuelle défini dans **Location** : **getPathForItem()** qui nous a permis d'avoir des chemin d'image (différents pour les épreuves et restaurants), et ainsi les afficher sur notre application.

Surcharges d'opérateurs : Le projet présente des surcharges d'opérateurs avec l'implémentation des opérateurs `==` et `=`. Les classes **Location**, **Event**, et **Restaurant** redéfinissent ces opérateurs pour permettre des comparaisons et des affectations spécifiques à chaque type d'entité. Cela améliore la lisibilité et la facilité d'utilisation du code lors de la manipulation d'instances de ces classes.

Deux conteneurs différents de la STL : Le projet utilise aussi divers conteneurs, démontrant une variété dans les choix de structures de données. Les `QVector` sont employés pour stocker les listes d'événements, de restaurants et certains attribut de ces classes, tandis que `QJsonArray` est utilisé pour stocker les informations tirés de notre base de données. Cette diversité dans l'utilisation de conteneurs offre une flexibilité supplémentaire dans la manipulation des données au sein du projet.

Commentaire du code et méthode pas plus de 30 lignes : Le projet respecte la contrainte de maintenir la longueur des méthodes en dessous de 30 lignes (hors commentaires et sauts de lignes), favorisant ainsi la lisibilité, la modularité et la facilité de maintenance du code. Pour garantir la compréhension du code, des commentaires ont été ajoutés de manière judicieuse. L'approche adoptée consiste à fournir des commentaires explicatifs au niveau des méthodes, soulignant leur rôle et leur logique générale, sans alourdir inutilement le code. Pour les méthodes plus complexes, des commentaires

internes ont été ajoutés aux endroits stratégiques pour faciliter une compréhension plus approfondie du fonctionnement.

Tests unitaires et utilisation de github : Le projet intègre des tests unitaires, notamment des tests de fonctions pour les classes, **Event**, et **Restaurant**. Ces tests sont présents dans le (`main`) du code, mais ils ont été commentés dans la version finale de l'application. Et pour le test pour les méthodes des autres classes ont été faites lors de notre implémentation de ces méthodes. En effet on a avait test toute nos fonction au fur et a mesure que notre projet avançait Mais ces tests ont été supprimés dans la version finale. Cette décision vise à éviter toute confusion dans l'interface utilisateur et à garantir une expérience fluide. Vous pourrez les trouver dans des versions plus anciennes de l'historique du commit

Et enfin on a aussi un **outil de build automatique** comme vous pourrez le voir dans les instruction de notre GitHub

4 Point fort de notre implémentation

Nous sommmes fiers de certaines aspects ou fonctionnalités dans notre projet qui, selon nous, apportent un vrai plus à l'utilisation. Les voici :

La struture : Chaque classe possède des fonctionnalités distinctes. comme par exemple **MainWindow** qui est dédier aux fonctionnalités d'affichage des différentes page, puis **ScrollAreaManager** qui se charge du contenue à ajouter dans la zone où l'on affiche les résultat de notre barre de recherhce. Cette structure facilite la compréhension, la maintenance et offre une base robuste pour le développement futur

La fonctionnalité de rendre les widget cliquable : En effet pour notre fenêtre d'informations globale, un grand nombre de données s'affiche, et ces mêmes données changent en fonction de la recherche faite ou encore du filtre appliqué. C'est pourquoi, il n'était pas possible de faire en sorte "d'ajouter" un bouton pour chaque élément (item), et surtout pas à l'aide de l'interface Qt. Nous voulions de plus nous rapprocher d'une utilisation d'un moteur de recherche standard, où l'on ne clique pas sur un bouton, mais bien **directement** sur le lien qui nous intéresse. C'est là où ce widget **ClickableWidget**, prend tout son sens. Il permet de rendre tous les éléments **cliquables**, et peut même être réutilisé dans d'autres cas de figure. Il nous facilite pour ainsi dire le développement de l'application, en nous permettant de rendre cliquable ce dont nous avons besoin afin d'en afficher les détails.

5 Conclusion

Pour conclure, nous avons trouvé ce projet très enrichissant, autant d'un point de vue de la programmmation C++ en elle-même, mais aussi pour les outils comme Qt que nous avons pu découvrir. En effet, c'était une bonne manière de passer en revue l'ensemble des concepts appris dans ce module de C++, en y voyant des intérêts pratiques. De plus, être confronté à un projet où la structure globale des fichiers (headers et source), ainsi que les classes et sous-classes prend une importance capitale, surtout lorsque le projet devient plus fourni. Nous aurions aimé développer une carte avec plus de fonctionnalités, et aussi un calcul d'itinéraire, ainsi qu'une fenêtre spécifique pour acheter des billets. Mais nous sommes sincèrement fiers de l'application que nous avons développé et de son fonctionnement, et pensons que la structure globale de notre programme est cohérente et réfléchie (et également évolutive). Nous espérons que vous avez apprécié l'utilisation de l'application Paris Jo Explorer !