Rapport de projet 2 - LO21

Membres

- Justine POUGET
- Axel CAMPAÑA
- Gabriel SANTAMARIA
- Pierre ROUSSEL
- Yicheng WAMG

Structuration et organisation du projet

Le projet sera hébergé sur GitHub et sera suivi via la fonctionnalité `Project`.
Le projet sera divisé en `Issues` qui seront assignées aux membres du groupe.

Les issues

Chaque issue suivera le workflow suivant avec ces différents statut:

- `To do` – Ce statut défini la liste des tâches toujours à réaliser
- `In Progress` – Ce statut défini la liste des tâches en cours de réalisation
- `Staging` – Ce statut défini la liste des tâches qui sont réalisées et tester spécifiquement lors de l'implémentation
- `Done` – Ce statut défini la liste des tâches finies qui sont testés et considérés comme fiable

Dans le projet, chaque tâche pourra être priorisé grace à un attribut dans l'issue. Ils pourront aussi être prioriser par le ordre d'apparition dans les `Backlogs`.
Un suivi des issues qui sont assignées sera possible grâce à la view `My issue`.
La branche `main` sera la branche principale du projet et servira pour le code considéré comme fiable. La branche `staging` sera celle où les tâche sont réalisées mais pas entièrement testées. Les autres branches seront les branches de developpement et porteront comme nom la clé du ticket (à voir si réalisable).

Lien vers le projet -> schotten-totten

Les vues

Pour voir l'ensemble des tâches, il est possible de consulter le tableau Board qui donne une vue d'ensemble de l'avancement du projet.
De même, la vue Backlog donne une version d'ensmble des tâches à réaliser. Cette vue à pour objectif principal de prévoir tout en priorisant les tâches à réaliser en fonction des besoins à un instant t.
La vue My issues permet de voir les tâches qui nous sont assignées. Seules les tâches ayant le statut `To do`, `In Progress`, `In Review` ou `Staging` sont

affichées.
La vue Staging et Done permettent de voir l'ensemble des tâches qui sont considérées comme finies (même si non testé hors developpement).

Code guidelines

L'entièreté des codes guidelines se trouvent dans le fichier `.clang-format` à la racine du projet. Avant de faire une `Pull Request`, vérifiez que votre code est conformes aux guidelines.

Nous avons mis à la disposition de tous un script `Powershell` qui se trouve aussi à la racine du projet (`run-clang.ps1`) pour vous permettre de formatter le code que vous venez d'écrire avant de le publier.

- Dans tout le projet, on essaiera au plus que possible d'utiliser les possibilités du C++ moderne en utilisant des pointeurs intelligents et les conteneurs disponibles dans la STL.

Conception et architecture du projet

Le projet `git`

Nous avons mis en place une repository avec `Git` pour faciliter les contributions entre tous les membres du projet. Le projet est hébergé sur Github, à l'adresse suivante: https://github.com/lo21-p23-project/lo21-project.

Grâce à Github, nous avons mis en place un système de gestion des tâches basées sur des `Issues`. Pour chaque tâche à faire:

- Elle est d'abord décrite de manière précise dans un ticket
- Elle est ensuite assignée à une personne du projet souhaitant la prendre en charge
- Elle est finalement mise dans le backlog

Cela nous permet de garder un oeil sur les tâches qu'il nous reste à faire et celles que nous avons déjà accomplies.

Les Github Actions

Grâce à Github Action, nous avons pu mettre en place un flow de validation des pull request de chacune des branches qui sont crées, avant de les mettre dans la branche principale.

clang-format L'une de ces deux actions est le formattage du code, pour assurer un code qui soit homogène et que les conventions soient les même pour tous. Nous avons aussi développé un petit script Powershell pour permettre à tout le monde de formatter le code avant de le push sur le dépôt.

L'action `clang-format` s'exécute à chaque pull request dans la branche `staging` ou `main`.

continuous integration (build) Pour être sûr que les codes qui seront merge dans la branche principale ou de staging ne soient pas défectueux, nous avons aussi mis en place un système de `CI` pour valider que le code qui a été publié compile bien sur les plateformes principales (linux, windows et macos).

Cette action a été quelque peu difficile à mettre en place, au vu de la complexité de l'installation et de la compilation de projets qui utilisent le framework Qt. Elle n'est pour le moment pas encore fonctionnelle, mais nous espérons pouvoir la mettre en place avant la fin du projet.

Choix du design pattern

Le projet a été designé en utilisant deux design-pattern. Le premier, qui nous sert à bien utiliser Qt et séparer les mondes de *l'UI* et du *backend* est le MVC: vous trouverez à la racine source du projet (*src/*) trois dossiers: **Model**, **Controller** et **View**.

- Dans **Controller** se trouve le code qui nous permet de faire l'interface entre notre modèle et notre vue. Les classes qui y sont présentes sont en grande majorité statiques, et ne servent qu'à faire le lien entre les deux mondes.

- Dans **View** se trouve l'entièreté du code d'affichage du projet qui utilise le framework *Qt* en version *6.5.0*. Tout le code spécifique à *Qt* (ou qui utilise des objets *Qt*) se trouve dans *src/View*.

- Dans **Model** se trouve le code du backend. Toute la logique du code (sur laquelle s'appuient les vues) est contenu dans le dossier *src/Model*.

Côté frontend

La partie frontend du projet utilise le design pattern Observer puisque le framework Qt repose entièrment sur ce design pattern.

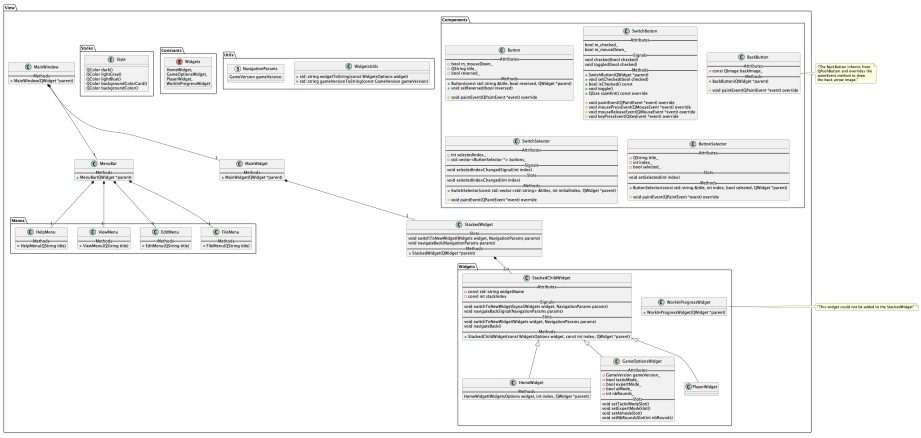
Le répertoire View se décompose lui-même en plusieurs modules:
- Les components: Ce sont les composants de base de l'application, ils sont réutilisables et peuvent être appelés dans n'importe quelle vue et/ou autre component. Ils permettent de garder un design cohérent à travers toute l'application.
- Les widgets: Ce sont les vues de l'application, elles sont composées de components et de widgets.
- Les constantes: Ce sont les constantes de la partie frontend. Elles permettent par exemple de simplifier la logique de navigation entre les vues.
- Les utils: Ce sont des fonctions utilitaires qui permettent d'effectuer des actions récurrentes dans l'application. - Les styles: Ce sont les couleurs et polices utilisées dans l'application. Tout comme les components, les styles permettent

de garder un design cohérent à travers toute l'application.
- Les composants de base: A la racine du dossier View se trouvent les composants de base de l'application. Ils correspondent à la logique principale de la partie frontend de l'application.

Logique de création des vues
Afin de créer les vues, il suffit d'utiliser une `QMainWindow` qui sera la fenêtre principale de l'application. Afin de permettre à l'application de montrer différentes vues, il faut aussi créer un `MainWidget` qui permettera de contenir les différentes vues à afficher et à cacher en fonction du workflow de l'application.

Logique de navigation entre différentes vues
Bien que le modèle précédent nous permettent de créer une pseudo logique de navigation entre les vues, il est cependant très peu pratique à utiliser et à maintenir. Dans l'objectif de palier à cela, nous avons utilisé un `QStackedWidget` qui permet de gérer la navigation entre les différentes vues de manière plus simple et plus maintenable. La classe `StackedWidget` possède des slots (`switchToNewWidget()` et `navigateBack()`) qui reste à l'écoute des signaux des vues qu'il contient.
Aussi toute vue, héritera de la classe `StackedChildWidget` qui possède des slots (`switchToNewWidget()` et `navigateBack()`) et des signaux (`switchToNewWidgetSignal()` et `navigateBackSignal()`) permettant de naviguer entre les différentes vues. Cette classe possède aussi un attribut `widgetName` qui permet d'identifier la vue et un attribut `stackIndex` qui permet de définir l'index de la vue dans la `QStackedWidget`.
Afin de garder une flexibilité dans le code, la `StackedWidget` créera la vue et l'ajoutera à la `QStackedWidget` lors de l'appel du slot `switchToNewWidget()`. Lors de l'appel du slot `navigateBack()`, la `StackedWidget` supprimera la vue de la `QStackedWidget` et la détruira.
Par défaut, la `StackedChildWidget` possède un bouton de retour qui permet de naviguer vers la vue précédente.

Pour plus d'informations, il est possible de consulter le diagramme de classe frontend.



Côté backend

Nous avons pris le choix d'avoir un backend qui utilise le design pattern Observer. Pour la mise en place de ce design pattern, vous trouverez deux classes:

- `EventManager`: une classe qui s'occupe de gérer la levée des évènements dans le code
- `ISubscriber`: une classe qui décrit le fonctionnement des objets qui s'abonnent à certains évènements

Fonctionnement et utilisation de ces deux classes

Il s'agit ici de décrire le fonctionnement de ces deux classes, et de montrer comment les utiliser.

Utilisation

L'utilisation de ces deux classes est assez directe: soit `A` et `B` deux classes, et supposons que nous souhaitons que `B` puisse recevoir des messages de la part de `A`.

A titre d'exemple, les messages que s'échangent `A` et `B` sont des entiers.

Tout d'abord, nous devons permettre à `B` de souscrire à des évènements:

```
using namespace Model::Shotten;

class B : ISubscriber<int> {
  /* ISubscriber est une classe virtuelle définie dans Model::Shotten */

public:
    void trigger(int message) {
        std::cout << "Message reçu: " << message << std::endl;
    }
```

```
    void trigger() {
        std::cout << "Evenement recu sans message" << std::endl;
    }
}
```

Ensuite, nous devons ajouter à `A` la possibilité d'envoyer des messages à `B`. Pour se faire, il nous suffit d'ajouter un attribu à `A` qui gère les messages envoyés vers `B`:

```
using namespace Model::Shotten;

class A {
public:
    EventManager<int> bEventManager;
}
```

Pour faire en sorte que `B` souscrive à `bEventManager`, un simple:

```
    auto bInstance = std::make_shared<B>();
    bEventManager->subscribe("name_of_the_event", bInstance);
```

Ensuite, pour que `B` reçoive un message lorsque l'évènement `name_of_the_event` est trigger, il suffit de faire:

```
    bEventManager->call("name_of_the_event", 12);
    /* cet appel à call fait que B::trigger(12) est appelé */
```

Fonctionnement

`EventManager` maintient une collection d'abonnés, qui sont tous des instances d'objets qui implémentent l'interface `ISubscriber`. `EventManager` est générique, ce qui lui permet de gérer différents type de données de message.

Voici une courte description de ce que fait chaque méthode dans `EventManager`:

- `subscribe`: Cette méthode ajoute un nouvel abonné à un événement spécifique. Chaque événement est identifié par une chaîne de caractères, et un abonné peut s'abonner à autant d'événements qu'il le souhaite.
- `unsubscribe`: Cette méthode retire un abonné d'un événement spécifique. Si l'abonné n'est pas dans la liste pour cet événement, rien ne se passe.
- `call`: Ces méthodes sont utilisées pour déclencher un événement. Lorsqu'un événement est déclenché, tous les abonnés à cet événement sont notifiés et leur méthode trigger est appelée. Il y a deux versions de cette méthode, une qui prend une donnée de type T qui est passée aux abonnés, et une autre qui ne prend rien.

`ISubscriber` est une interface pour les objets qui peuvent s'abonner aux événements gérés par un `EventManager`. Un objet qui implémente `ISubscriber` doit fournir une méthode `trigger`, qui sera appelée par `EventManager` lorsqu'un événement auquel l'objet est abonné est déclenché.

La méthode `trigger` a deux formes: une qui prend un argument de type `TriggerType` (le type de donnée pour les messages) et une qui n'en prend pas.

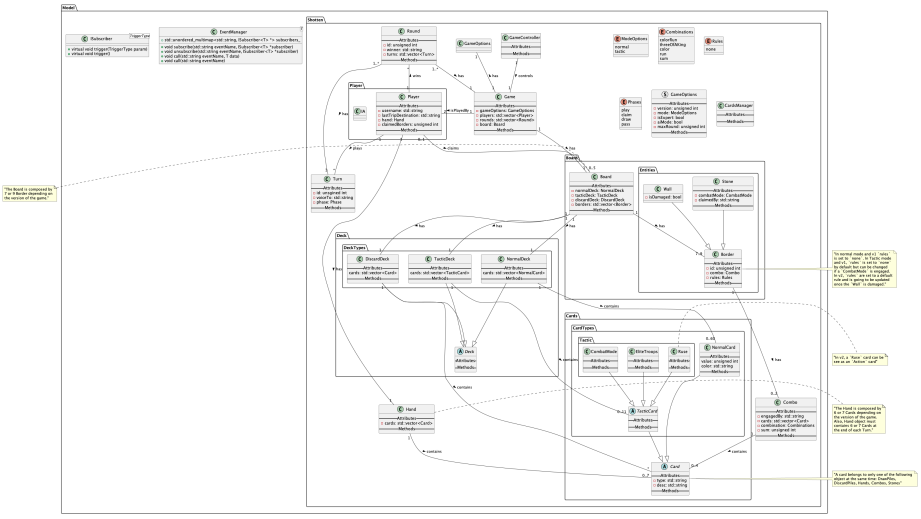
Pourquoi utiliser cette architecture?
L'Observer nous permet de mettre une couche d'abstraction supplémentaire sur nos objets et de ne plus trop se préoccuper de la manière dont les objets sont communiquent des données entre eux.

A partir du moment où l'on a besoin d'une communication, il nous suffit de mettre en place un canal en utilisant les classes `ISubscriber` et `EventManager`, et le tour est joué!

De plus, le jeu vidéo étant un domaine beaucoup basé sur les évènements, mettre en place un tel système paraît presque naturel.

A noter qu'un tel système pour le *Shotten-Totten* n'est pas entièrement requis. Néanmoins, le découpage que nous avons fait nous permets de pouvoir changer les Vues sans impacter le code des Modèles, ou encore de changer la logique interne du modèle, sans avoir à toucher au Vues.

Pour plus d'informations, il est possible de consulter le diagramme de classe backend.



Wireframe

-> Lien

Diagramme de classe

-> Frontend
-> Backend

Partie commune du rapport

Liste des tâches mise à jour:

Se référer à la section **Structuration et organisation du projet** pour plus d'informations quant à l'utilisation et à lecture du projet.
Globalement, toutes les tâches ayant le statut `In Review`, `Staging` ou `Done` sont à considérer comme terminées.
A noter que les tickets sont assignés à la personne en charge de la tâche. Les estimations de temps de travail ne sont pas reportées par soucis d'estimation complexe...

Affectation des tâches

Se référer à la section **Structuration et organisation du projet** pour plus d'informations quant à l'utilisation et à lecture du projet.
Il est possible de facilement voir les affectations des tâches en se rendant sur Board et en ajoutant le fitre `assignee:` et en selectionnant l'un des usernames proposés: col-roussel (Pierre ROUSSEL), gabyfle (Gabriel SANTAMARIA), AxelCMPN (Axel CAMPAÑA), Justinepg (Justine POUGET), Yichengaz57 (Yicheng WANG).

Etat d'avancement du projet par rapport au précédent rapport

Par rapport au précédent rapport, nous avons pu avancer sur les points suivants: - Mise en place final de l'outils projet pour la gestion des tâches - Mise en place des GitHub Actions (cf. Les Github Actions) - Mise en place de l'environnement de développement (CLion, vcpkg, CMake, Qt) - Refonte de l'architecture coté backend - Mise en place d'une architecture MVC - Mise en place de l'architecture coté frontend - Implémentation des premières classes du modèle - Implémentation des premières vues (HomeWidget, GameOptionsWidget, PlayerWidget) - Design des premières vues (HomeWidget, GameOptionsWidget, PlayerWidget) - Discussion par rapport à la vue du GameWidget

Le rythme de travail est relativement soutenu à raison de 1 à 3 réunions par semaine. Le temps de travail est estimé à 6h jusqu'à 15h par semaine par personne.

Cohésion du groupe

Il est relativement complqué de faire des réunion avec l'équipe complète. Seul Gabriel et Pierre sont disponibles pour toutes les réunions. Justine et Yicheng demande régulièrement des nouvelles quant à l'avanement du projet. Ils sont aussi en demande de tâches à réaliser mais il est relativement complexe de donner des tâches sans avoir une connaisance de l'architecture du projet et des choix qui ont été faits.
Axel quant à lui ne répond plus ni ne lit les messages envoyés sur le groupe

depuis le 2 avril. Il est donc impossible de savoir si il est toujours dans le projet ou non.

Lien vers le projet -> schotten-totten

Note:

- Merci de ne pas sauvegarder les filtres modifier pour ne pas impacter les autres utilisateurs.
- En cas de problème pour accéder au projet, merci de contacter Pierre ROUSSEL (col-roussel) ou Gabriel SANTAMARIA (gabyfle) pour obtenir les droits d'accès.