Conception et programmation Objets Avancées
Université Clermont-Auvergne
IUT Info 2A

Clément TORTI
Florent BECOUZE
G7
Année 2018/2019

Documentation Code :

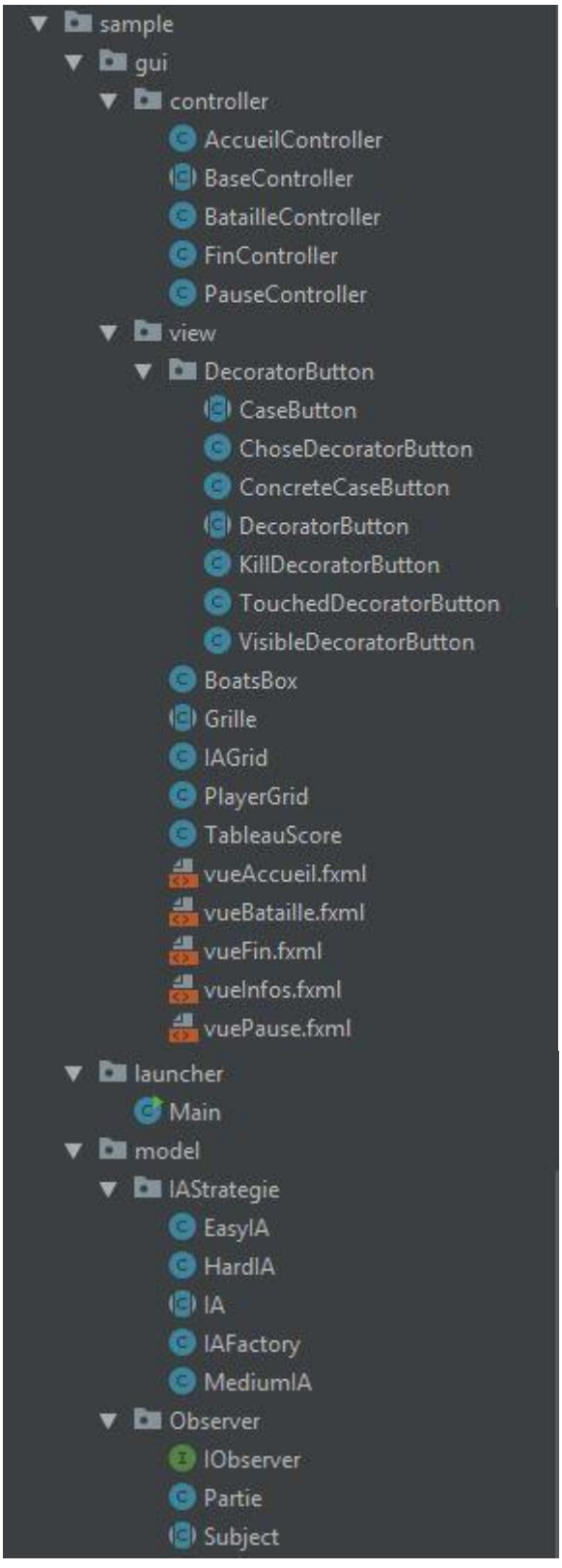
Bataille-navale

I/ Présentation du projet

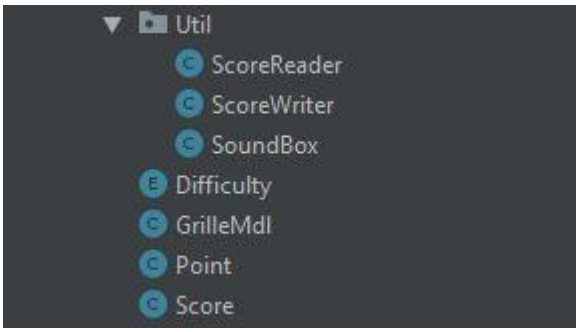
II/ Plan d'exécution

III/ Fonctionnement des classes

(1. sample)
- 1.1. gui
  - 1.1.1. controller
    - 1.1.1.1. AccueilController
    - 1.1.1.2. BaseController
    - 1.1.1.3. BatailleController
    - 1.1.1.4. FinController
    - 1.1.1.5. PauseController
  - 1.1.2. view
    - 1.1.2.1. DecoratorButton
      - 1.1.2.1.1. CaseButton
      - 1.1.2.1.2. ChoseDecoratorButton
      - 1.1.2.1.3. ConcreteCaseButton
      - 1.1.2.1.4. DecoratorButton
      - 1.1.2.1.5. KillDecoratorButton
      - 1.1.2.1.6. TouchedDecoratorButton
      - 1.1.2.1.7. VisibleDecoratorButton
    - 1.1.2.2. BoatsBox
    - 1.1.2.3. Grille
    - 1.1.2.4. IAGrid
    - 1.1.2.5. PlayerGrid
    - 1.1.2.6. vueAccueil.fxml
    - 1.1.2.7. vueBataille.fxml
    - 1.1.2.8. vueFin.fxml
    - 1.1.2.9. vueInfos.fxml
    - 1.1.2.10. vuePause.fxml
- 1.2. launcher
  - 1.2.1. Main
- 1.3. model
  - 1.3.1. IAStrategie
    - 1.3.1.1. EasyIA
    - 1.3.1.2. HardIA
    - 1.3.1.3. IA
    - 1.3.1.4. IAFactory
    - 1.3.1.5. MediumIA
  - 1.3.2. Observer
    - 1.3.2.1. IObserver
    - 1.3.2.2. Partie
    - 1.3.2.3. Subject



1.3.3. Util
    1.3.3.1. ScoreReader
    1.3.3.2. ScoreWriter
    1.3.3.3. SoundBox
1.3.4. Difficulty
1.3.5. GrilleMdl
1.3.6. Point
1.3.7. Score



IV/ Patrons de conception

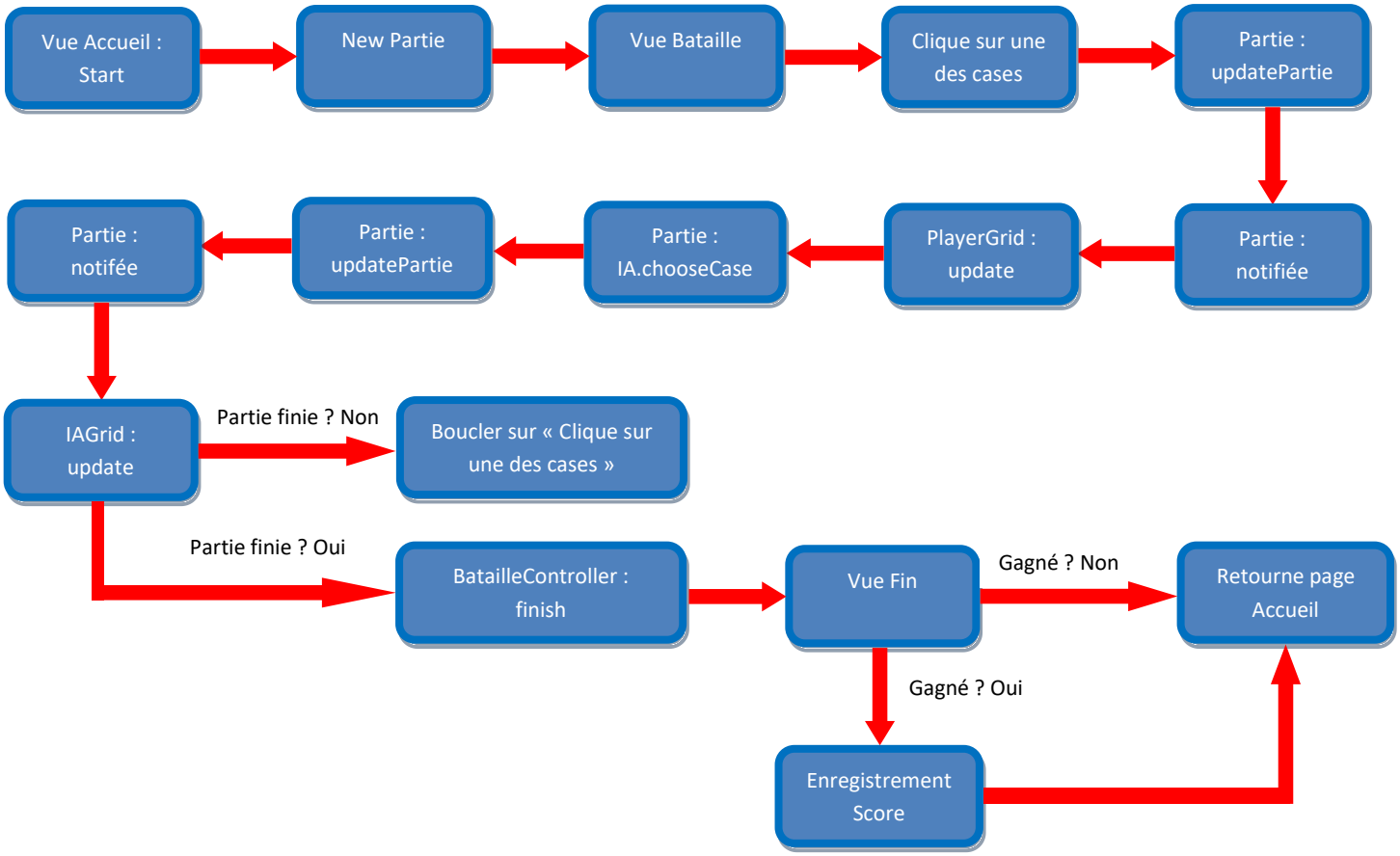
I/ Présentation du projet

Ce projet est un programme visant à recoder le jeu « Bataille Navale ». Un joueur affronte une IA dont il peut choisir le niveau de difficulté. A la fin de la partie, le score du joueur est enregistré s'il a gagné et sera visible sur le tableau des scores de la page d'accueil.

Voir PDF UML.

Voir PDF Cas d'utilisation.

II/ Plan d'exécution



III/ Fonctionnement des classes

(1. sample)

C'est le dossier global du projet, dans lequel on a une structure MVC : dossier « gui » pour les packages « controller » et « view » et le package « model »

1.1. gui (graphical user interface)

C'est le dossier qui possède les packages « controller » et « view ». Dans le dossier, il y a tout ce qui concerne le côté graphique de l'application.

1.1.1. controller

C'est le package permettant de rassembler tous les contrôleurs des différentes vues. Chaque contrôleur hérite de la classe abstraite BaseController.

1.1.1.1. AccueilController

C'est le contrôleur d'accueil, c'est-à-dire, celui qui va être lancé en premier par la classe Main. A l'initialisation du contrôleur :
- On lance la musique d'accueil
- On crée un ensemble de RadioButton, pour permettre le choix de la difficulté de l'IA
- On récupère dans le fichier (représentant notre base de données), les différents scores et on les mets dans la ListView de la vue.

private void start(ActionEvent actionEvent) throws Exception :

- Représente l'action de cliquer sur le bouton « Start » de la vue « vueAccueil » :
    - Elle créée, en fonction du RadioButton choisit, une partie avec la difficulté choisie.
    - Elle change de « scene » (sans changer de fenêtre) en allant sur la vue de bataille, en passant la partie en paramètre.

private void infos(ActionEvent actionEvent) throws Exception :

- Représente l'action de cliquer sur le bouton « Infos » de la vue « vueAccueil ».
- Elle ouvre une nouvelle fenêtre (« stage ») avec la vue d'informations des règles du jeu.

1.1.1.2. BaseController

C'est la classe abstraite qui est la classe mère de toutes les classes de contrôleurs. Elle permet :
- De « garder en mémoire » le « stage » dans lequel on se trouve, c'est-à-dire, la fenêtre actuelle dans laquelle on est

- De pouvoir changer la « scene » de cette fenêtre, c'est-à-dire la vue affichée.
- On garde en mémoire la partie actuelle (duree, nbCoup, winner, les 2 grilles, …).

public void changeScene(String source, BaseController destController) throws IOException :

- Permet de changer de vue sans changer de fenêtre. On modifie donc pour cela la « scene » du « stage » (la vue de la fenêtre). On modifie aussi le contrôleur puisque c'est le contrôleur de la nouvelle vue qui doit prendre le relais.

public void openStage(String source, int width, int height) throws IOException :

- Même que **changeScene**, mais ouvre une nouvelle fenêtre.

1.1.1.3. BatailleController

- C'est la classe contrôleur de la vue « vueBataille ». A l'initialisation :
  On lance la musique du jeu, puis on met à jour la vue (grilles, bateaux, …) en fonction de la partie en cours.
- On établit une connexion entre la partie et les composants Grille. Ainsi à chaque tour la partie est modifiée et les Grille sont averties pour se mettre à jour.
- On établit une connexion entre les messages de la partie et les 2 ListView.
- On finit par initialiser le timer (le chrono du jeu). Pour se faire, on définit une tâche que le timer doit réaliser : on incrémente le compteur de « duree » d'une Partie et on lance une méthode permettant d'afficher le nouveau compteur (compteur modifié). Pour finir, on attache au timer, la tâche, qu'il devra réaliser en boucle.

private void pause(ActionEvent actionEvent) throws Exception :

- Permet d'arrêter le timer et la musique de jeu. Et on change de « scene » (et de contrôleur) pour aller sur la vue de pause.

private void finish() :

- Appelée quand la partie est terminée (gagnée ou perdue). Elle permet d'arrêter le timer et de changer de « scene » (et de contrôleur) pour aller sur la vue de fin (avec le résultat de la partie).

private void update() :

- Appelée lorsque la partie notifie le contrôleur.
- Appelle **finish** si la partie est finie.
- Met à jour la bonne « BoatsBox », si un bateau vient d'être détruit.

1.1.1.4. FinController

Cette classe représente le contrôleur de la vue « vueFin ». Lors de l'initialisation :
- On regarde qui a gagné.
- On affiche le message correspondant avec la durée de la partie et le nombre de coups du joueur.

private void accueil(ActionEvent actionEvent) throws Exception :

- Coupe le son du jeu et on remet le son de la page d'accueil.
- Si le joueur a gagné (et a rempli son nom), on l'enregistre son score dans le fichier texte qui nous sert de base de données.
- Pour finir, on change de « scene » (et de contrôleur) pour aller sur la vue d'accueil.

1.1.1.5. PauseController

- C'est le contrôleur en charge de la vue « vuePause ». Pour initialiser :
  On change juste la musique du jeu pour la musique d'accueil.

private void resume(ActionEvent actionEvent) throws IOException :

- Permet de revenir au jeu, en arrêtant la musique d'accueil et en changeant la « scene » (et de contrôleur) pour aller sur la vue du jeu.

private void quit(ActionEvent actionEvent) throws IOException :

- Permet de revenir à la page d'accueil.
- On arrête, ici aussi, la musique d'accueil.
- On change de « scene » (et de contrôleur) pour aller sur la vue d'accueil.

1.1.2. view

C'est le package permettant de rassembler tous ce qui est en rapport avec la vue (ce que voit l'utilisateur).

1.1.2.1. DecoratorButton

Ce dossier permet de gérer le design des boutons (ceux représentant les cases de la bataille) en fonction de leurs états, c'est-à-dire, s'ils représentent une case vide, une case avec un bateau dessous ou si le bateau est coulé.

- Ces boutons suivent le patron Décorateur :
  Un bouton possède un StackPane d'icônes, qui représentent son état.

Couches successives d'icônes

- Un bouton (case d'une Grille) pouvant être à la fois découvert, touché et coulé, nous pouvons le décorer successivement, ce qui aura pour effet d'ajouter des icônes dans son StackPane.

1.1.2.1.1. CaseButton

C'est la super classe abstraite de nos boutons.

protected abstract void addIcon() :

- Méthode abstraite permettant à tous les décorateurs concrets, d'ajouter une icône au StackPane.

1.1.2.1.2. ChoseDecoratorButton

- Concrétisation du **DecoratorButton**, qui implémente **addIcon()**.
- Correspond à une case découverte mais vide.
- Ajoute un cercle noir.

1.1.2.1.3. ConcreteCaseButton

- Concrétisation du **CaseButton**.

1.1.2.1.4. DecoratorButton

- Hérite de **CaseButton**.
- Prend un bouton en paramètre (constructeur), copie son StackPane d'icônes et appel **addIcon()**.

1.1.2.1.5. KillDecoratorButton

- Concrétisation du **DecoratorButton**, qui implémente **addIcon()**.
- Correspond à une case découverte avec un bateau détruit.
- Ajoute un GIF d'explosion.

1.1.2.1.6. TouchedDecoratorButton

- Concrétisation du **DecoratorButton**, qui implémente **addIcon()**.
- Correspond à une case découverte avec un bateau touché.
- Ajoute une image de bateau rouge.

1.1.2.1.7. VisibleDecoratorButton

- Concrétisation du **DecoratorButton**, qui implémente **addIcon()**.
- Correspond à une case des bateaux du joueur.
- Ajoute une image de bateau.

1.1.2.2. BoatsBox

- Hérite de HBox.
- Contient une liste d'ImageView, représentant les bateaux en jeux.

public void destroyBoat() :

- Change l'image de l'un des bateaux de la liste, en un bateau détruit.

1.1.2.3. Grille

- Classe abstraite qui hérite de GridPane et implémente IObserver.
- A l'initialisation, on crée toutes les colonnes et lignes (lettres, chiffres et boutons).
- Le constructeur de Grille respecte le patron « **Patron de méthode** ». En effet, la création d'un bouton standard est effectuée par la méthode abstraite **configureStandardButton()**. Un bouton type dépendra donc de la classe fille instanciée (Player ou IA).

public abstract void configureButtons(GrilleMdl grille) :

- Méthode qui met à jour les boutons de la Grille, en se calquant sur la Grille passée en paramètre.
- L'implémentation varie d'une fille à l'autre (choix des décorateurs).

public void update() :

- Méthode appelée par notification de la partie.
- Récupère toutes les informations sur le bouton cliqué (sa position, son état).
- En fonction de l'état du bouton, on fait appel au décorateur adapté.
- On supprime l'ancien bouton de « buttonList » et on le remplace par le nouveau bouton décoré.

1.1.2.4. IAGrid

- Hérite de Grille.
- Ajoute aux boutons standards, une action appelée durant le clique : **buttonClicked(int x, int y)**.

public abstract void configureButtons(GrilleMdl grille) :

- Met à jour la Grille. Commence par les cases découvertes, puis par les bateaux touchés pour finir avec les bateaux coulés.

public void update() :

- S'assure que la super méthode **update()** est appelée au bon moment, c'est-à-dire quand c'est la Grille IA qui est modifiée.

1.1.2.5. PlayerGrid

- Hérite de Grille.

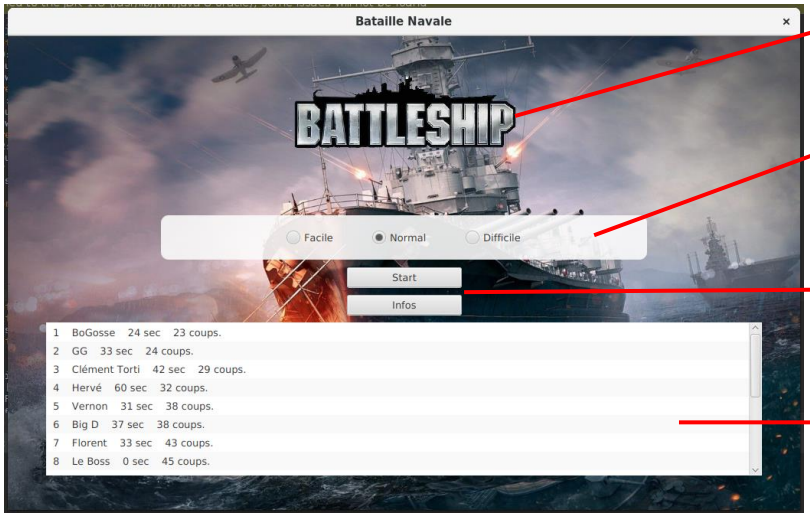
public abstract void configureButtons(GrilleMdl grille) :

- Met à jour la Grille. Commence par tous les bateaux du joueur, puis les cases découvertes, puis les bateaux touchés, pour finir avec les bateaux coulés.

public void update() :

- S'assure que la super méthode **update()** est appelée au bon moment, c'est-à-dire quand c'est la Grille Player qui est modifiée.

1.1.2.6. vueAccueil.fxml



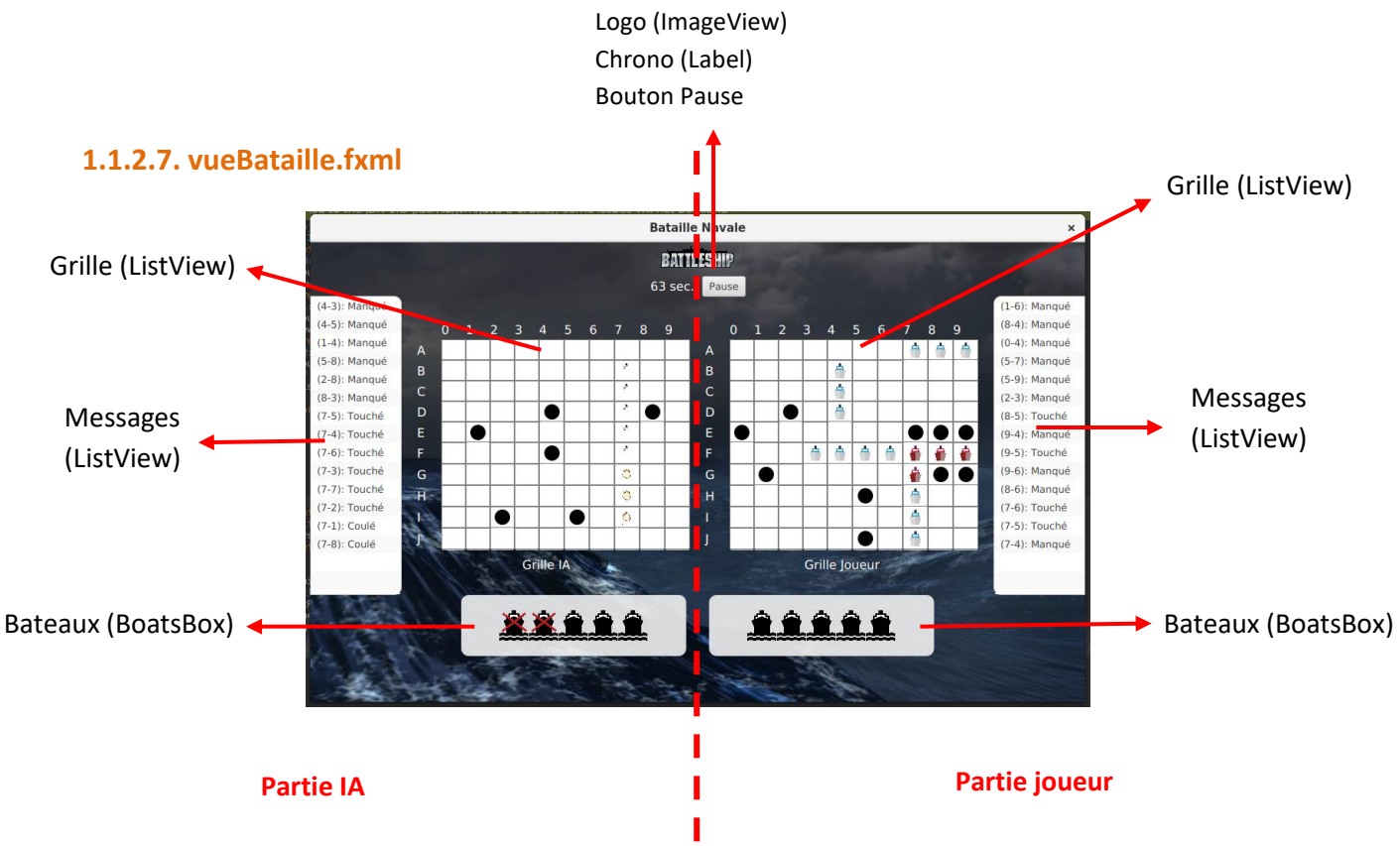
Logo (ImageView)

Boutons de difficultés (HBox -> RadioButton)

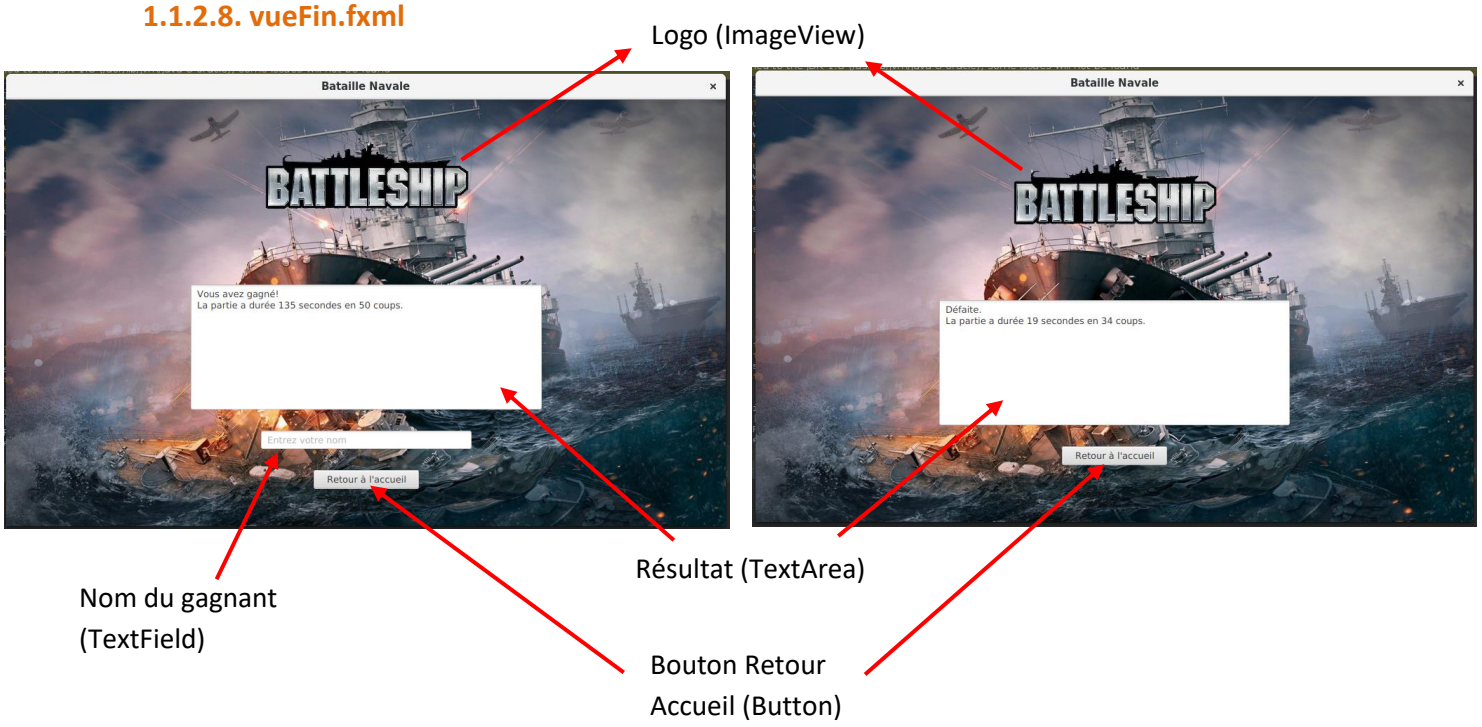
Boutons « Start » et « Infos » (Button)

Liste des scores (ListView)

1.1.2.7. vueBataille.fxml

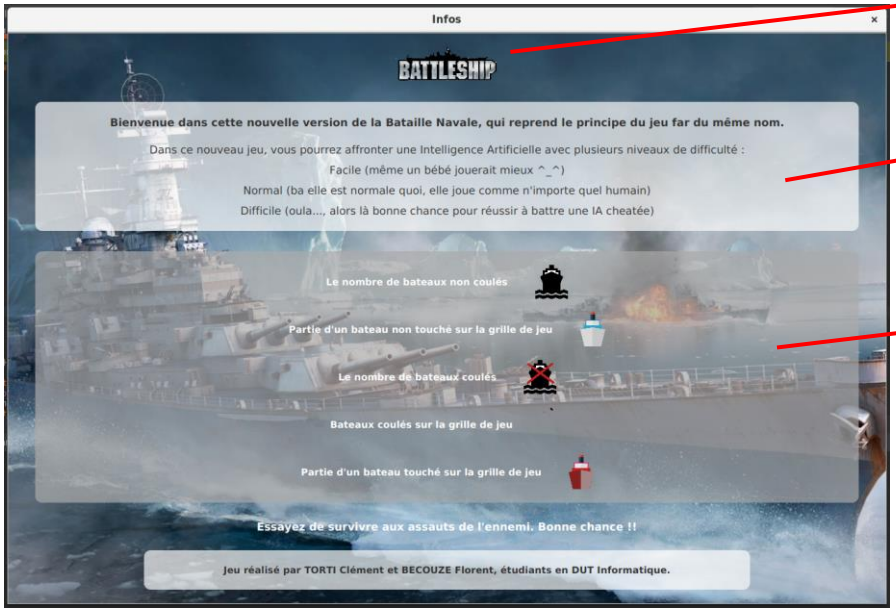


1.1.2.8. vueFin.fxml



Remarque : si le joueur a perdu, on supprime le « TextField » où normalement on doit donner notre nom pour enregistrer notre performance.

1.1.2.9. vueInfos.fxml

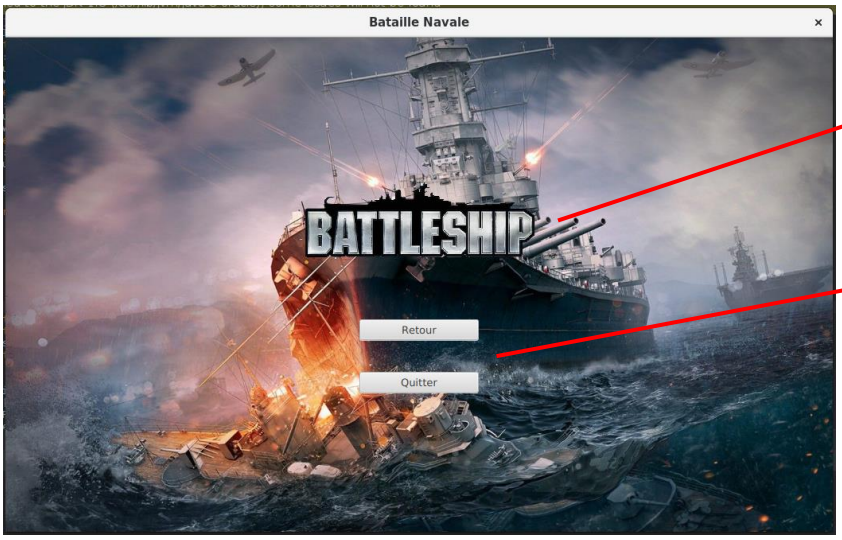


Logo (ImageView)

Description du jeu (VBox + Label)

Description des images (VBox + Hbox + Label + ImageView)

1.1.2.10. vuePause.fxml



Logo (ImageView)

Boutons « Retour » et « Quitter » (Button)

1.2. launcher

C'est le package permettant de rassembler la classe « Main », celle qui va être lancée au tout début du programme.

1.2.1. Main

public void start(Stage primaryStage) throws Exception :

- Première méthode exécutée. Charge le **AccueilController** en lui passant le « stage » et créée la fenêtre.

1.3. model

C'est le package permettant de rassembler tous les classes métiers, utiles.

1.3.1. IAStrategie

Ce dossier contient toutes les classes liées à l'IA. Ses classes respectent la patron Stratégie et Fabrique Simple :

- Patron Stratégie : Il existe une classe par difficulté qui héritent toutes de la classe IA et qui implémentent d'une manière différente, le choix d'une case dans le jeu.
- Patron Fabrique Simple : Permet de centraliser le code lié à la création de l'IA. A partir d'un cas de l'enum **Difficulty**, **IAFactory** retourne l'instance de l'IA adaptée.

1.3.1.1. EasyIA

- Hérite de IA. Implémente **chooseCase**.
- Stratégie : choisit une case libre aléatoirement.

1.3.1.2. HardIA

- Hérite de IA. Implémente **chooseCase**.
- Stratégie : choisit une case contenant un bateau 1 fois sur 2.

1.3.1.3. IA

- Classe abstraite.
- **public abstract Point chooseCase(GrillMdl grille) ;**

1.3.1.4. IAFactory

public static IA createIA(Difficulty d) :

- Retourne une instance d'une des classes filles de l'IA en fonction de la difficulté.

1.3.1.5. MediumIA

- Hérite de IA. Implémente **chooseCase**.
- Stratégie : tire au hasard jusqu'à trouver un bateau, puis choisit les cases aux alentours.

1.3.2. Observer

Dossier contenant les classes impliquées dans le patron Observeur. Nous utilisons ce patron pour permettre aux Grille de la vue de se mettre à jour, lorsque la partie évolue.

1.3.2.1. IObserver

Interface que les observeurs doivent implémenter (Grille, BatailleController).

void update() :

- Appelée lors d'une notification.

1.3.2.2. Partie

- Hérite de « Subject ».
- La classe qui contient toutes les informations liées à la partie en cours, que l'on passe d'un contrôleur à l'autre.
- C'est elle :
    - Qui fait référence aux 2 Grille (Player et IA), en les initialisant et en les mettant à jour à chaque coup.
    - Qui crée l'IA grâce à la Factory.
    - Qui gère le déroulement de la partie (qui doit jouer, nombre de coup, durée, vainqueur).
    - Qui notifie la vue pour qu'elle se mette à jour.
    - Qui crée et met à jour les listes de messages.

private List<List<Point>> generateRandomPos() :

- « boatsSize » est un tableau d 'entiers qui définit le nombre de bateaux et leurs tailles..
- Pour chaque bateau de « boatsSize » :
    - On choisit une case aléatoire non utilisée.
    - On choisit aléatoirement l'orientation du bateau.
    - On place de part et d'autre de la case choisie, autant de case que la taille du bateau (on veille à ce qu'elle ne soit pas déjà utilisée).

    - Si le bateau n'est pas positionnable sur la Grille, on recommence avec une nouvelle case.
- Cette méthode retourne une liste contenant les positions de chaque bateaux.

public void updatePartie(int x, int y) :

- Méthode appelée lorsque l'un des joueurs à choisit une case.
- Elle permet :
    - De vérifier que la partie n'est pas finie.
    - En fonction du joueur qui vient de jouer, elle met à jour le bon « GrilleMdl », en appelant **discover**, qui retourne l'état de la case cliquée. En fonction de cet état, on ajoute le message adéquat dans la liste de messages.
    - On vérifie si la partie est gagnée.
    - On notifie les observateurs du changement.
    - Si c'est au tour de l'IA, on lui fait choisir une case et on rappelle **updatePartie**.

1.3.2.3. Subject

- Classe abstraite qui définit le comportement d'un notifieur.
- Elle contient les informations qui ont besoin d'être envoyés aux observateurs.
- Elle permet d'ajouter / de supprimer des observateurs et de les notifier.

1.3.3. Util

Contient les classes indépendantes du projet.

1.3.3.1 ScoreReader

Classe permettant, la lecture d'un fichier texte de scores.

public List<Score> getScores() :

- Parcours toutes les lignes d'un fichier.
- Pour chaque ligne, appelle la méthode **format**, qui retourne un objet « Score ».
- Retourne la liste des scores.

private Score format(String line) :

- Découpe la ligne lue grâce au séparateur « // ».
- Récupère chacun des éléments d'un Score, et le crée.

1.3.3.2. ScoreWriter

Classe permettant, l'ajout d'un score dans un fichier texte.

public void insertScore(Score score) :

- Lit le fichier de scores.
- Trouve la position du score.
- Met à jour la liste des scores.
- Réécrit tout dans le fichier texte.

Private void writeToFile(List<Score> scores) :

- Parcours les scores.
- Ecrit chacun d'eux au bon format dans le fichier.

1.3.3.3. SoundBox

- Gère les sons de l'application (musiques et bruitages).
- Fournit une méthode statique par sons.
  Les bruitages sont joués 1 seule fois, les musiques sont répétées à l'infinie.

1.3.4. Difficulty

Enumération désignant les difficultés d'une IA.

1.3.5. GrilleMdl

Classe métier d'une Grille, qui est mise à jour par la Partie et synchronisée avec une vue Grille.

Elle contient les différentes positions :

- Des bateaux.
- Des bateaux touchés.
- Des bateaux détruits.
- Des cases découvertes.

public int discover(int x, int y) :

- Pour une case donnée, cela met à jour les listes et retourne l'état de la case (manqué, touché, coulé).
- On part du plus général au plus spécifique.
- On vérifie que :

- La case touchée contient un bateau, sinon retourne 0 (manqué).
- Toutes les cases du bateau sont touchées. Si oui, retourne 2 (coulé) et l'ajoute dans la liste des bateaux détruits, sinon retourne 1 (touché).

public boolean isWin() :

- Retourne vrai si la partie est finie, c'est-à-dire s'il y a autant de bateaux détruits que bateaux.

1.3.6. Point

Classe métier qui encapsule 2 coordonnées pour en faire un Point.

- On redéfinit la méthode **equals()**.

1.3.7. Score

Classe métier qui encapsule toutes les informations liées à un Score.

- Redéfinit le **toString()** pour permettre l'affichage dans la ListView.
- Redéfinit le **compareTo()** pour permettre de comparer 2 Score (utilisé pour le ScoreWriter).

IV/ Patrons de conception

Nom du patron	Classes impliquées	Pourquoi ?
Décorateur	Package « DecoratorButton »	Un bouton (case d'une Grille) pouvant être à la fois découvert, touché et coulé, nous pouvons le décorer successivement, ce qui aura pour effet d'ajouter des icônes dans son StackPane. Cela rend le modèle plus extensible car il suffit de rajouter une classe pour ajouter un type de bouton, sans modifier le code existant. Evite la prolifération des classes.
Stratégie	Package « IAStrategie »	Toutes les classes filles IA, utilisent des algorithmes différents pour le choix d'une case en fonction de la difficulté.
Fabrique Simple	Classe **IAFactory**	Permet de centraliser le code lié à la création de l'IA. A partir d'un cas de l'enum **Difficulty**, **IAFactory** retourne l'instance de l'IA adaptée.
Patron de méthode	Classes **Grille, IAGrid, PlayerGrid**	La création d'un bouton standard est effectuée par la méthode abstraite **configureStandardButton()**. C'est le seul élément de la méthode qui varie d'une classe fille à l'autre. Un bouton type dépendra donc de la classe fille instanciée (Player ou IA). Permet d'éviter la duplication de code.

Observateur	Classes **Partie, Subject, BatailleController, Grille, PlayerGrid, IAGrid** Interface **IObserver**	Permet la connexion entre le modèle (Partie) et la vue (Grille).