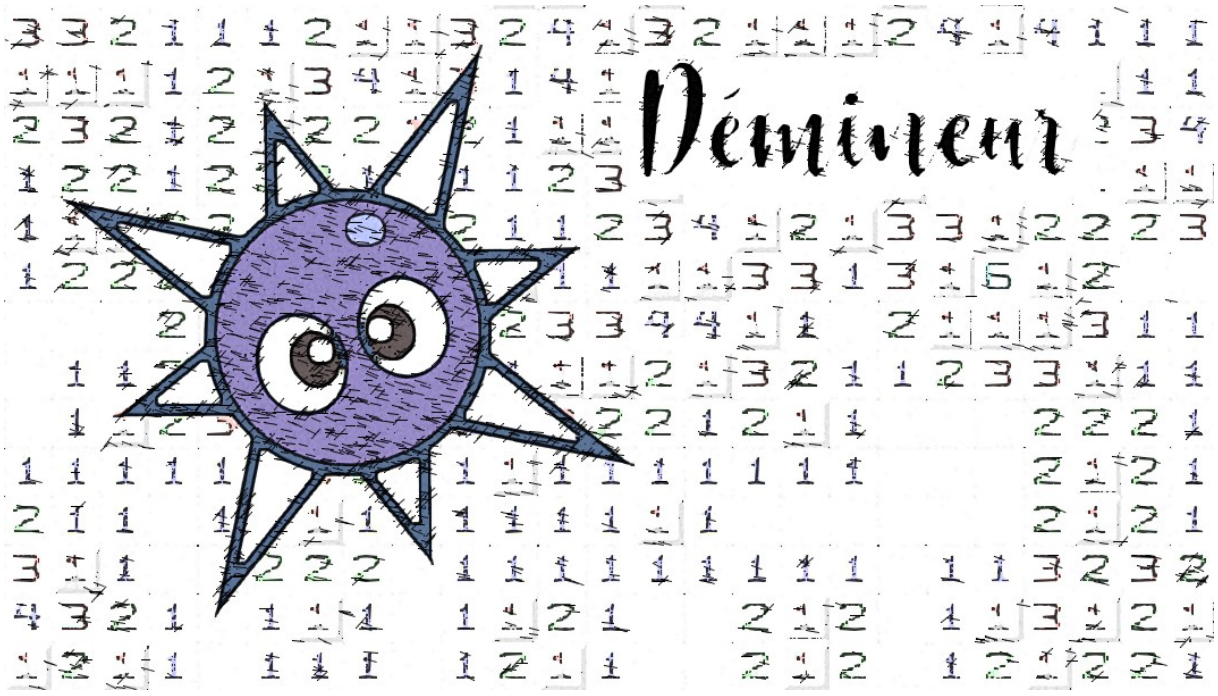
HE2B ESI

Attila Markus 42015 Chris Pipers 39864

Directeur : Michel Willemse

Professeur : Nicolas Vansteenkiste

DEV4 - C++II



Remise1

Février 2018

Executive summary

Le jeu Démineur est un jeu composé d'un tableau de cases. Les cases peuvent soit être vides, soit contenir une bombe.

Le but du jeu est de découvrir toutes les cases qui ne contiennent pas de bombes en un minimum de temps, et cela sans faire exploser une seule bombe. Si le joueur découvre une case contenant une bombe, la partie est terminée. Si le joueur soupçonne une case de contenir une bombe, il peut la marquer par un drapeau.

Dans le cadre de ce projet, le jeu possédera uniquement un mode solo.

Pour modéliser et gérer le jeu, le choix fut porté sur 8 classes : Board, Game, Observable, Observer, Player, Position, RulesEndGame, Time.

Table des matières

Introduction

Dans ce document, nous allons aborder le développement du jeu Démineur dans le langage de programmation C++. Actuellement, nous n'abordons que la modélisation métier du programme (le modèle). Cela signifie que nous allons développer les processus nécessaires à l'objectif du jeu.

Cette modélisation métier du programme reprend :

- Un diagramme de classe du projet ;
- Des diagrammes d'activités pour illustrer les actions possibles lors des différentes phases d'actions ;
- Une liste des headers (ou entêtes) documentée pour chaque classe.

Le diagramme de classe donne une vision du schéma d'ensemble des classes, ainsi que leurs interactions, qui vont être utilisées pour programmer le jeu démineur.

Pour chacune de ces classes, nous détaillerons leur utilité, leurs différentes méthodes et attributs. Nous donnerons également, si nécessaire, les algorithmes utilisés.

Les règles du jeu seront décrites brièvement afin de placer la modélisation métier dans son contexte.

Vous trouverez toute la documentation dans les annexes :

- **Annexe-A** : comporte les fichiers.h avec les headers et leurs documentations ;
- **Annexe-B** : comporte les PDF des diagrammes de classes et d'activités (schéma UML), ainsi que les images qui illustrent le rapport.

Les annexes sont les deux dossiers joints à ce rapport.

Le jeu Démineur

Le jeu Démineur est constitué d'un plateau de cases dans lequel se trouvent disséminées des bombes.

Par défaut, le plateau est de taille 9 * 9 cases. Selon le choix du joueur, la taille du plateau peut varier avec une certaine taille maximale pour que le jeu soit encore lisible à l'écran.

Le nombre de bombes présentes sur le plateau est déterminé :

- Soit par défaut par la formule : $0.0002 * (m\ n)^2 + 0.0938 * (m\ n) + 0.8937$;
- Soit selon le choix du joueur qui sélectionnera un certain nombre de mines ;
- Soit selon le choix du joueur qui sélectionnera un certain pourcentage de mines présentes.

La première action du joueur sera :

- Soit révéler (ou éclairer) une case ;
- Soit placer un drapeau sur une case.

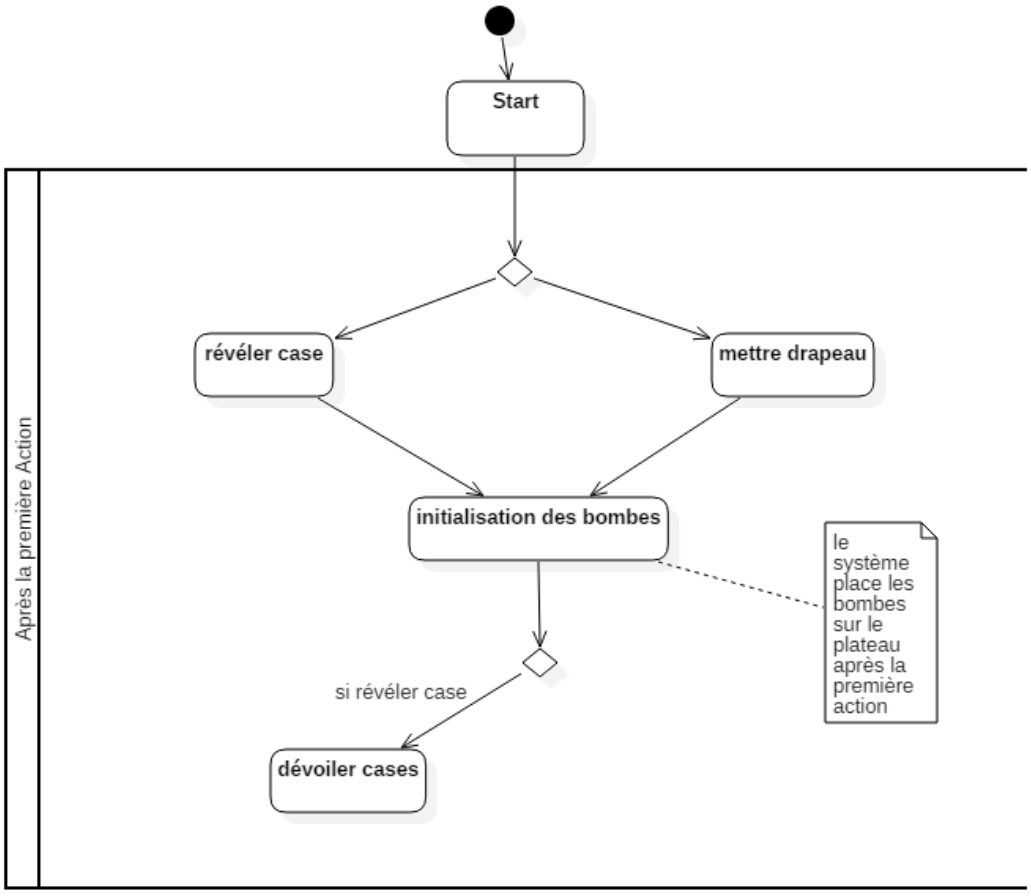


Figure 1 - représentation simplifiée des actions possibles pour la première action du jeu

Les actions suivantes du joueur seront de trois sortes :

- Soit révéler (ou éclairer) une case ;
- Soit placer un drapeau sur une case ;
- Soit supprimer un drapeau.

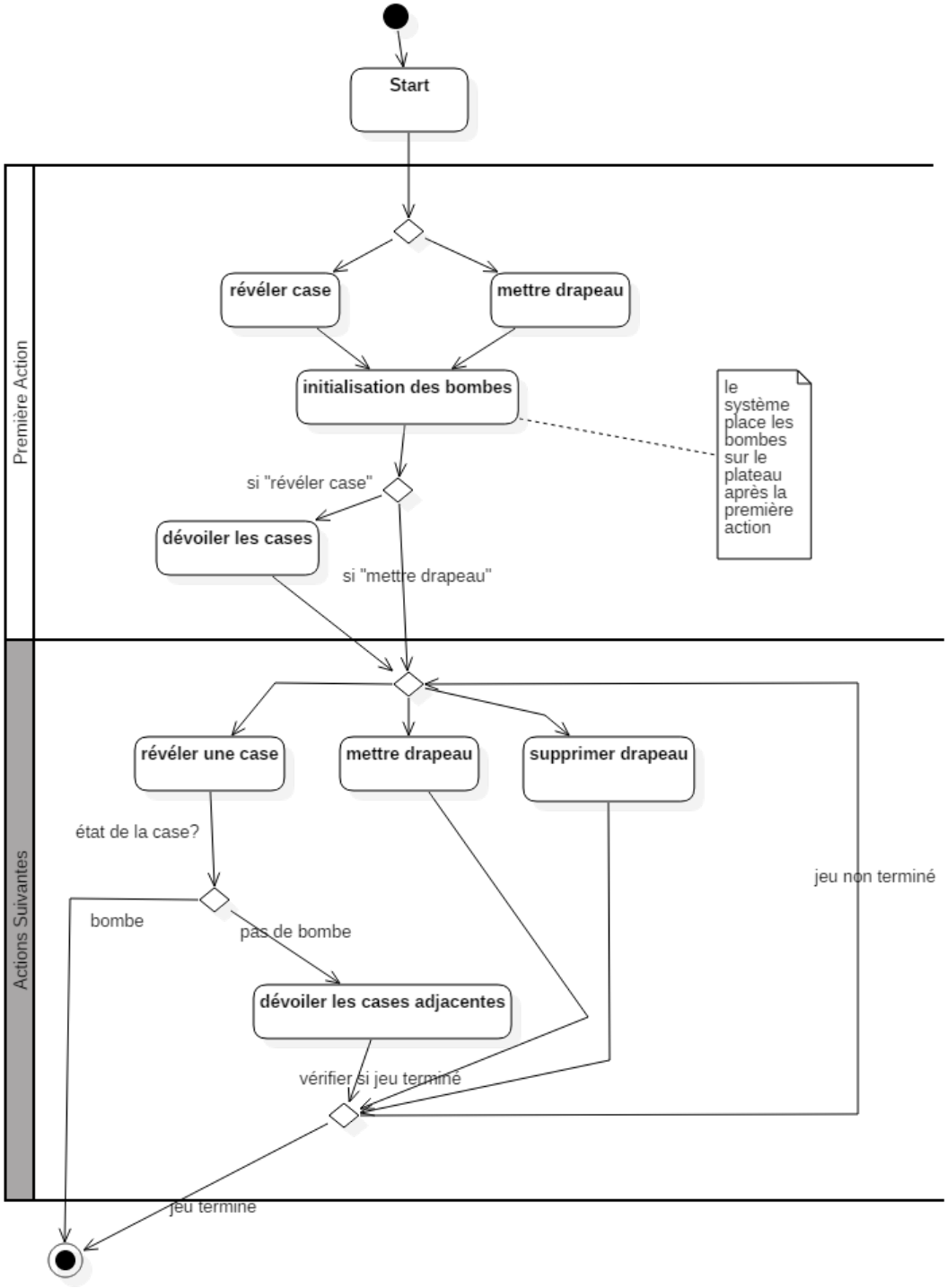


Figure 2 - représentation simplifiée des actions suivantes (après la première action)

Pour faciliter la vue, les cases non visitées resteront grisées tandis que les cases révélées seront bleutées. Pour aider le joueur à déminer le plateau, les cases ayant autour d'elles une ou plusieurs bombes indiquent combien de bombes sont présentes autour d'elles. Dans le cas où aucune bombe ne se trouve autour de la case, la case restera vide. Celles ayant une bombe dans le voisinage direct ont un indice respectivement de 1 à 3, de 1 à 5 ou de 1 à 8. Ce nombre maximum varie selon la position de la case :

- Les cases dans les 4 coins possèdent comme indice maximal 3, car autour d'elles ne se trouvent que 3 cases ;

- Les cases sur les bords possèdent comme indice maximal 5, car autour d'elles ne se trouvent que 5 cases ;

- Les autres cases, quant à elles, possèdent comme indice maximal 8, car autour d'elles ne se trouvent que 8 cases ;

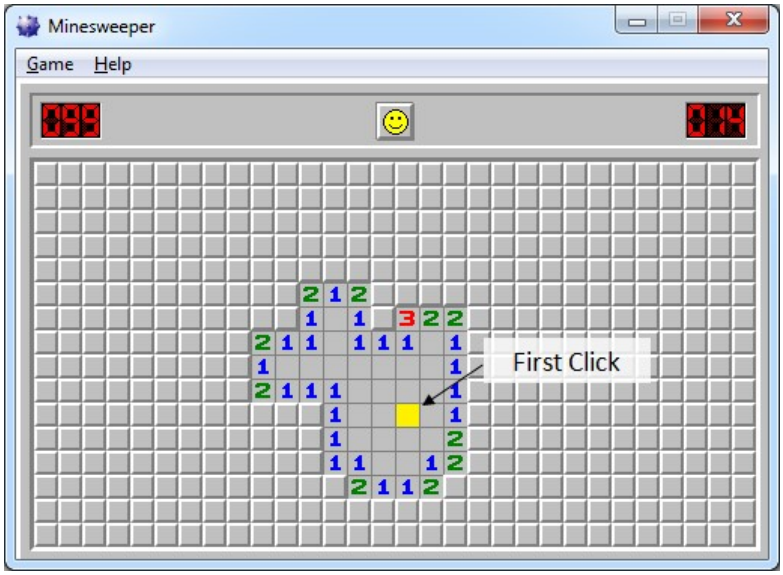
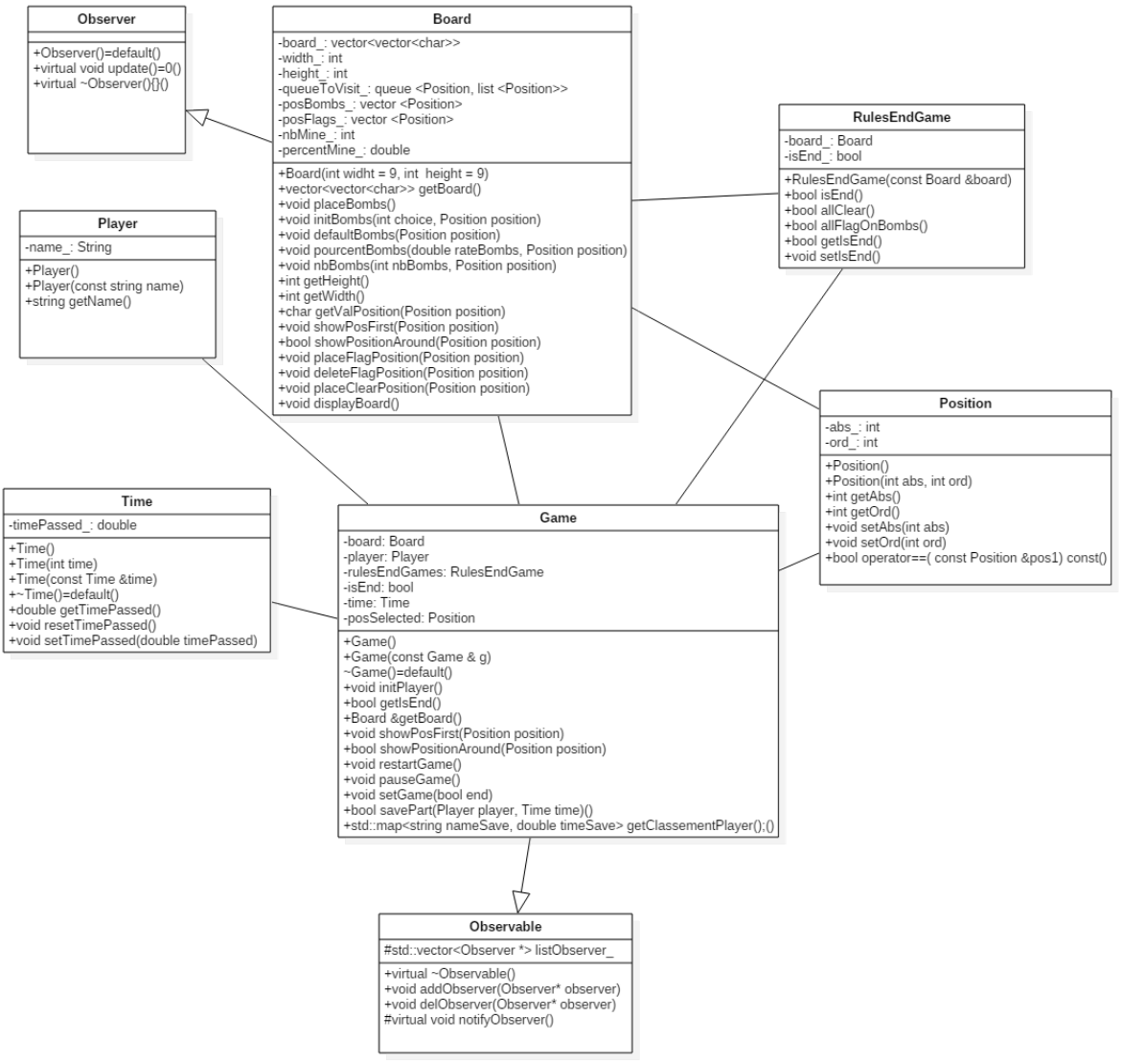


Figure 3 - révélation d'une case avec ses conséquences : on voit les cases vides et les cases numérotées qui indiquent ainsi la quantité de bombes dans les cases adjacentes pour chacune d'entre elles.

Source : http://datagenetics.com/blog/june12012/index.html

Le but du jeu est de révéler toutes les cases sans bombe ou de marquer toutes les bombes avec des drapeaux en un minimum de temps.

Classes et leurs compositions



La classe Board

Présentation de la classe Board

La classe Board contrôle le plateau du jeu. Elle hérite de la classe Observer ; on peut ainsi implémenter la méthode **update().** L'implémentation de **update()** permet de notifier les observateurs de tout changement.

De base, la taille du plateau aura 9 cases de hauteur et 9 cases de largeur. Le joueur peut sélectionner selon son envie la taille du plateau de jeu, une taille maximale sera déterminée lors de l'implémentation de cette classe, pour permettre d'avoir constamment un affichage idéal et lisible.

Pour la vue console, chaque case contient un caractère qui correspond à son état ou qui permet de retourner une information au joueur. Les cases peuvent contenir la valeur :

- 'c' pour *clear* (claire), la case a été révélée : il n'y aucune bombe et aucun indice (ce qui indique que les cases adjacentes[1] n'ont aucune bombe. Elle correspond à la case bleutée ;
- 'd' pour *dark* (sombre), la case est non révélée et non marquée. Elle correspond à une case grisée ou inconnue ;
- 'f' pour *flag* (drapeau), la case contient un *flag* (drapeau) qui permet au joueur de signaler que, selon son analyse, la case contient potentiellement une bombe. Elle correspond à la case avec un drapeau ;
- 'b' pour *bomb* (bombe), la case contient une bombe qui est invisible tant que le joueur n'a pas cliqué dessus, ce qui correspondrait à une fin de partie car le joueur aurait perdu ;
- '1' à '8', qui correspond à un indice destiné à aider le joueur ; ce chiffre indique le nombre de bombes présentes autour de la case contenant le chiffre.

Au commencement de la partie, le plateau contient uniquement des cases *dark* ('d'). Après la première action du joueur (soir révéler une case, soit placer un drapeau), les mines sont placées et la partie peut continuer (voir figure 2). Cela signifie que, pour la première action, il n'existe aucune bombe dans le plateau.

Si le joueur décide de placer un drapeau à la première action, il ne sait pas si des cases adjacentes contiennent des bombes.

Traitement des bombes

Pour le placement des bombes, la méthode **initBombs()** appelle, selon la valeur de l'entier *choice* , l'une des trois méthodes d'initialisation que sont :

- **defaultBombs()** pour le nombre de bombe par défaut ;
- **nbBombs()** pour le choix d'un certain nombre de bombes sélectionné par le joueur ;
- **pourcentBombs()** qui placera un pourcentage de bombes selon le nombre total de cases.

[1] 8 cases si c'est une case centrale, 5 cases si c'est une case sur les côtés, 3 cases si la case est dans un coin.

Traitement de la première case

La méthode **showPostFirst()** est la méthode appelée si le joueur décide de révéler une case. La position en paramètre ne contiendra jamais de bombe (car elles sont initialisées seulement par la suite). Ensuite, cette méthode appellera **showPositionAround().**

Traitement des adjacences

La méthode **showPositionAround()** utilise une file qui permettra d'explorer toutes les cases adjacentes à la case de départ. Le choix de la file facilite une vérification en chaîne. Cela permettra de placer, dans les cases visitées par la méthode, les indices (un entier) si une ou des bombes sont trouvées, ou un 'c' si aucune bombe n'est présente.

- Les cases contenant une bombe ne sont pas explorées par la méthode ;
- Les cases qui contiennent un indice ne sont pas ajoutées dans la file ;
- Une case contenant un drapeau ne sera pas visitée par la méthode, même si celle-ci ne contient aucune bombe.

Constitution de la classe Board

Les méthodes **public** de la classe Board :

```
Board (int width = 9, int height=9);
vector <vector<char>> getBoard();
void placeBombs();
void initBombs(int choice, Position position);
void defaultBombs(Position position);
void pourcentBombs(double rateBombs, Position position);
void nbBombs(int nbBombs, Position position);
int getHeight();
int getWidth();
char getValPosition(Position position);
void showPosFirst(Position position);
bool showPositionAround (Position position);
void placeFlagPosition(Position position);
void deleteFlagPosition(Position position);
void placeClearPosition(Position position);
void displayBoard();
```

Les attributs **private** de la classe Board:

```
vector<vector<char>> board_;
int height_;
int width_;
queue <Position, list <Position>> queueToVisit_;
vector <Position> posBombs_;
vector <Position> posFlags_;
```

```
int nbMine_;
double percentMine_;
```

Pour plus de détails sur les méthodes et attributs, veuillez consulter l'Annexe - A. Dans les headers, se trouve toute la documentation.

La classe Game

Présentation de la classe Game

La classe Game est une façade qui fera le lien entre les actions du joueur et toutes les méthodes contenues dans le **namespace model**.

En plus des méthodes de façade, cette classe permet de mettre le jeu en pause, de recommencer une partie ou d'afficher tous les scores sauvegardés des joueurs.

En fin de partie, si le score du joueur fait partie des meilleurs scores, le jeu préserve le score et le nom du joueur dans un fichier.

La classe hérite de la classe Observable ce qui permettra d'inscrire des observateurs et, en cas de changement d'état, de notifier tous les observateurs inscrits.

Constitution de la classe Game

Les méthodes **public** de la classe Game :

```
Game();
Game (const Game & g);
~Game()=default;
void initPlayer();
bool getIsEnd();
Board &getBoard();
void showPosFirst(Position position);
bool showPositionAround (Position position);
void restartGame();
void pauseGame();
void setGame(bool end);
bool savePart(Player player, Time time);
std::map<string nameSave, double timeSave> getClassementPlayer();
```

Les attributs **private** de la classe Game:

```
Board board_;
Player player_;
RulesEndGame rulesEndGame_;
bool isEnd_;
Time time_;
Position posSelected_;
```

Pour plus de détails sur les méthodes et attributs, veuillez consulter l'Annexe-A. Dans les headers se trouve toute la documentation.

La classe Observable

Présentation de la classe Observable

La classe Observer permet d'utiliser le pattern, Observateur – Observé (Design pattern Observer). Il est l'un des deux composants de ce pattern (patron). Il contient les méthodes permettant d'ajouter et supprimer un observateur, ainsi que de notifier (par la méthode **notify()** à tous les observateurs les changements survenus.

Constitution de la classe Observable

Les méthodes **public** de la classe Observable :

```
virtual ~Observable() {}
void addObserver(Observer* observer);
void delObserver(Observer* observer);
```

La méthode **protected** de la classe Observable :

```
virtual void notifyObserver() ;
```

L'attribut **protected** de la classe Observable :

```
std ::vector<Observer*> listObserver_ ;
```

Pour plus de détails sur les méthodes et attributs, veuillez consulter l'Annexe – A. Dans les headers se trouve toute la documentation.

La classe Observer

Présentation de la classe Observer

La classe Observer permet d'utiliser le pattern, Observateur – Observé (Design pattern Observer). Il est l'un des deux composants de ce pattern (patron). Il contient la méthode **update()** qu'il faut redéfinir et qui permet de se mettre à jour quand l'observateur est notifié.

Constitution de la classe Observer

Les méthodes **public** de la classe Observer :

```
Observer() = default ;
virtual void update()=0;
virtual ~Observer(){};
```

Il y a deux méthodes virtuelles. Elles sont destinées uniquement à être redéfinies dans les classes dérivant de la classe Observer ou à être déclarées à nouveau comme virtuelles pures dans les classes dérivées.

Pour plus de détails sur les méthodes et attributs, veuillez consulter l'Annexe – A. Dans les headers se trouve toute la documentation.

La classe Player

Présentation de la classe Player

La classe Player représente un joueur, qui possède un attribut *nom.* Cet attribut permet en fin de partie, si nécessaire, de sauvegarder le nom du joueur dans le classement du jeu.

Constitution de la classe Player

Les méthodes **public** de la classe Player :

```
Player();
Player(const string name);
string getName();
```

L'attribut **private** de la classe Player:

```
string name_;
```

Pour plus de détails sur les méthodes et attributs, veuillez consulter l'Annexe - A. Dans les headers se trouve toute la documentation.

La classe Position

Présentation de la classe Position

La classe Position permet de construire une position avec les coordonnées X (abscisse) et Y (ordonnée). Elle permet de simplifier l'écriture des coordonnées sur le plateau de jeu.

La classe Position permet de nous positionner sur une case du plateau.

Constitution de la classe Position

Les méthodes **public** de la classe Position :

```
Position ();
Position(int abs, int ord);
int getAbs();
int getOrd();
void setAbs(int abs);
```

```
void setOrd(int ord);
bool operator==( const Position &pos1) const;
```

Les attributs **private** de la classe Position :

```
int abs_;
int ord_;
```

Pour plus de détails sur les méthodes et attributs, veuillez consulter l'Annexe-A. Dans les headers se trouve toute la documentation.

La classe RulesEndGame

Présentation de la classe RulesEndGame

La classe RulesEndGame permet de vérifier si l'état de la partie est :

- **Gagné** (toutes les cases sans bombes ont été révélées) :

  Si un flag (drapeau) se trouve sur chaque case contenant une bombe, la partie est gagnée : la méthode **allFlagsOnBombs**() retourne un booléen à vrai ;

  Si toutes les cases claires sont explorées, la partie est gagnée : la méthode **allClear()** retourne un booléen à vrai ;

- **Perdu** (une case avec une bombe a été révélée) :

  Si un joueur demande à révéler une case contenant une bombe, la méthode **isEnd()** est appelée et elle retourne un booléen à vrai;

- **Rien** (ce qui correspond à en cours).

Constitution de la classe RulesEndGame

Les méthodes **public** de la classe RulesEndGame :

```
RulesEndGame(const Board &board);
bool isEnd();
bool allClear();
bool allFlagOnBombs();
bool getIsEnd();
void setIsEnd();
```

Les attributs **private** de la classe RulesEndGame :

```
Board board_;
```

```
bool isEnd_;
```

Pour plus de détails sur les méthodes et attributs, veuillez consulter l'Annexe – A. Dans les headers se trouve toute la documentation.

La classe Time

Présentation de la classe Time

La classe Time gère le chrono de la partie.

Cette classe initialise la valeur du chrono au début de la partie, le met en pause si la partie est mise en pause et le remet à zéro au cas où la partie est recommencée.

Constitution de la classe Time

Les méthodes **public** de la classe Time :

```
Time();
Time(int time);
Time(const Time &time);
~Time()=default;
double getTimePassed();
void setTimePassed(double timePassed);
void resetTimePassed();
```

L'attribut **private** de la classe Time :

```
double timePassed_;
```

Pour plus de détails sur les méthodes et attributs, veuillez consulter l'Annexe – A. Dans les headers se trouve toute la documentation.

Sauvegarde de la partie dans un fichier

La sauvegarde de la partie du joueur (si celui-ci répond au critère d'enregistrement) sera sauvegardée dans un fichier CSV, qui permettra une recherche plus rapide des données.

Un fichier .csv est un format de fichier dans lequel les données d'un tableau sont simplement séparées par une virgule

Cette recherche sera utile pour corriger le classement des joueurs.

Le fichier contiendra 7 informations :

1. La hauteur du plateau ;
2. La largeur du plateau ;
3. Le nombre par défaut ;
4. Le nombre de bombes (placées initialement) ;
5. Le pourcentage (taux de présence de bombes) ;
6. Le nom du joueur ;
7. Le temps (le résultat du chronomètre).

   Pour le nombre par défaut, le nombre de bombes ou le pourcentage de bombes, seule une des trois informations aura une valeur. Les autres sont à nul mais sont également présentes pour une simplification d'écriture et de recherche.

Trois cas possibles :

1. Le joueur bat le record : le nom du joueur et le nouveau temps du record remplacent le nom et le temps déjà présents dans le fichier CSV correspondant au même paramètre de la partie ;
2. Le joueur est le premier à effectuer un score pouvant entrer dans la liste des meilleurs scores : son nom et son temps sont sauvegardés dans le fichier CSV ;
3. Si le joueur bat son propre record et qu'il figure déjà dans le fichier : seulement son temps est modifié dans le fichier CSV.

Conclusion

Avec la première remise du projet nous pouvons tester l'intégration de plusieurs cours : analyse et C++. Dans nos travaux précédents, ces deux domaines étaient étudiés et pratiqués séparément (même Java et analyse ne sont abordés conjointement que dans ce quadrimestre-ci). Mais, ici, on ressent bien l'apport de l'analyse avant le codage, on a une vision plus claire du chemin qu'on va prendre.

On sent également que le projet va nous permettre d'apprendre à travailler en équipe (même si ce n'est qu'à deux), car il n'est pas toujours simple de voir comment diviser le travail.