FANTASY FACTORY GAME

par LEDAUPHIN Eliott & ES-SADEL Mohamed Ghali

Sommaire :

Présentation du jeu :

Si vous êtes fan de jeux de stratégie, d'optimisation et de fantaisie. Alors fantastic factory est le jeu qui vous faut ! Développez vos talents de manager et investisseur tout en vous amusant, vous découvrirez des produits magiques tout au long de votre aventure. Seuls les meilleures et les meilleurs arriveront au niveau des Grands Industriels. Les graphismes rustiques du terminal plairont à nos amis geek, avec une petite touche de couleur et d'emojis pour rendre le peux plus intéressant.

Installation et Exécution :

Il faut disposer de github sur sa machine et pouvoir exécuter du code c++ dont un makefile, voici quelques commandes si possédez une machine sur linux :

$ sudo apt-get update
$ sudo apt-get install g++ make
$ sudo apt-get install git

Une fois que c'est fait, on peut passer à l'exécution !
Il va d'abord falloir cloner le répertoire sur son ordinateur en faisant la commande :
$ git clone https://github.com/Ghali062/Fantasy-Factory-Game.git
Puis il faut ouvrir le dossier créé : $ cdFantasy-Factory-Game
Pour exécuter le jeu, il suffit de taper dans le terminal la commande : $ make
Cela va générer le fichier exécutable du jeu.
Pour lancer le jeu : $ .fantasy-factory-game

Pour exécuter les tests unitaires, il suffit d'écrire la commande : $ make testcase

Règles du jeu :

- Vous êtes fournisseurs de supermarchés ayant une demande infinie et vous devez fabriquer des produits à partir de matières magiques.
- Gravissez les échelons jusqu'au niveau 5 pour rejoindre la Ligue des Grands Industriels 🤑
- L'avancement dans le niveau se fait en fonction des bénéfices à chaque tour
- A partir du niveau 3, vous accédez au méthodes de production 'Assembler' qui donnent accès aux objets Deluxe, à haute valeur ajoutée

- Le jeux se déroule en plusieurs tours, au début de chaque tour le joueur doit choisir de racheter ou non des matières premières et/ou de changer les méthodes de production des employés
- Vous **perdez** si vous vous endettez au-delà de -200 mana 💀
- A la fin de chaque tour vient le bilan financier, où le joueur doit payer des impôts. Le joueur est encouragé à faire de l'optimisation fiscale (en jouant à la jackpot machine) sinon il devra payer 100% d'impôt sur le revenu.
-

Consignes d'utilisation :

- Il faut mettre le terminal en mode plein écran et potentiellement dé-zoomer afin que les animations et zones de texte ne soient pas crop
- Les contrôles se font au clavier (pas de souris), principalement les chiffres, la SPACE BAR et ENTER

Comment gagner (la meta) :

START : premier tour

- Investir la mana dans la matière magique car aux tours suivants vous serez endetté et ne pourrez pas vous endetter pour de la matière.
- Après avoir consommé toute votre mana, il faut vous endetter un peu pour recruter des employés (max 1 par tour donc cela peut être un bottleneck)
- Assigner vos employés pour qu'ils travaillent tous

MID-GAME :

- Il ne faut pas avoir peur de trop vous endetter, il faut viser 4-5 employés avant d'atteindre le niveau 3, tant qu'ils travaillent ils sont rentables 🤑
- Attention à ne pas épuiser vos matières magiques !!!
- Dès que vous avez remboursé vos dettes, rachetez de la matière magique

END-GAME :

- Après avoir atteint le niveau 3, passez directement aux produits Deluxe
- Il faut faire attention à ne pas manquer d'un des 2 ingrédients avec la méthode assembler !
- Engranger un maximum de profit pour atteindre le niveau 5

Ce dont on est fier :

Nous sommes particulièrement fiers d'avoir développé Fantastic Factory de manière à le rendre plus attrayant pour les femmes que la plupart des jeux de gestion du marché. Nous avons opté pour un univers fantaisiste et coloré, mis en valeur par de petits ajouts visuels (comme les emojis), afin que les joueuses puissent évoluer dans un environnement agréable et accessible.

Sur le plan technique, quelques éléments clés méritent d'être soulignés :

1. Découpage modulaire du code :
   - #include "company.hh" : Gère la structure même de l'entreprise (employés, finances, matières premières).
   - #include "jeu.hh" : Englobe la logique principale du jeu (enchaînement des tours, transitions de niveau).
   - #include "mini_jeu_impots.hh" : Permet d'intégrer un mini-jeu d'optimisation fiscale pour réduire ses impôts.
   - #include "text_box.hh" : Simplifie l'affichage formaté des messages (grâce aux cadres et couleurs).

2. Effets visuels dans le terminal :
   - Le code g et r (ligne 12 et 13 du fichier jeu.cc) apportent de la couleur, rendant le style "console" plus vivant et ludique.
   - La fonction loading_screen() intègre un mini-effet de fumée ASCII (@) pour apporter un brin de dynamisme lors du chargement, tout en restant dans l'esprit « old school » apprécié par la communauté geek.

3. Facilité d'évolution :
   - Le fichier jeu.hh et la classe Jeu sont conçus pour accueillir facilement de nouvelles méthodes de production ou de nouveaux mini-jeux (comme mini_jeu_impots.hh).
   - L'organisation par fonctions (handleRawMaterials(), handleHiring(), handleProduction(), etc.) rend le code très lisible et facilite l'ajout ou la modification de fonctionnalités. Cela permet de garder des possibilités d'évolutions à venir et qui sait peut être qu'il finira sur steam ?

Nous avons donc réussi à concevoir un jeu proposant une réelle profondeur stratégique (achats, production, gestion), tout en conservant une présentation épurée et accueillante. Nous espérons ainsi que les joueuses et joueurs, notamment le public féminin, se sentiront à l'aise et prendront plaisir à grimper les échelons pour devenir des Grands Industriels !

Description du code :

Nous décrirons principalement le code de la classe Jeu car c'est elle qui organise le jeu et de la classe Company de manière brève qui joue un rôle important aussi.

1. Inclusion des Bibliothèques et Déclarations Globales

Le code commence par inclure les bibliothèques nécessaires et définir des variables globales pour les séquences d'échappement ANSI, utilisées pour colorer le texte dans la console.

2. Fonction de Chargement Animé

La fonction load affiche un message de chargement animé, améliorant l'expérience utilisateur pendant les temps d'attente.

3. Écran de Chargement Personnalisé

La fonction loading_screen affiche un écran de chargement animé avec des caractères ASCII, ajoutant une touche unique au jeu.

4. Messages de Bienvenue et de Fin de Jeu

Les fonctions Bienvenue et game_over affichent des messages bien formatés pour accueillir le joueur et signaler la fin du jeu.

5. Gestion des Choix de l'Utilisateur

Les fonctions choice et input gèrent les entrées de l'utilisateur de manière robuste, en vérifiant les erreurs et en fournissant des messages clairs.

6. Classe Jeu

La classe Jeu encapsule la logique principale du jeu, y compris la gestion des tours, des matières premières, des employés, et de la production.

Constructeur : Initialise le jeu avec un nombre initial d'employés et un montant d'argent.

```
Jeu::Jeu(int initial_nb_employees, float initial_money, bool dev_mode)
    : _company(initial_nb_employees, initial_money), _start(true), _tour(0),
      _dev_mode(dev_mode), level(1) {}
```

Tour : Gère la progression du jeu à chaque tour, y compris la mise à jour du niveau, l'affichage des progrès, et la gestion des finances.

```
void Jeu::Tour()
```

Gestion des Matières Premières : Permet aux joueurs d'acheter des matières premières nécessaires à la production.

```
void Jeu::ask_material(std::string material)
```

Gestion des Employés : Permet aux joueurs d'embaucher des employés et de les affecter à différents postes.

```
void Jeu::handleHiring()
```

Gestion de la Production : Permet aux joueurs de gérer la production de l'usine.

```
void Jeu::handleProduction()
```

Fin de Journée : Calcule les gains, les impôts, et les salaires à la fin de chaque journée.

```
void Jeu::endOfDay()
```

Exécution du Jeu : Boucle principale du jeu qui continue tant que le joueur souhaite jouer.

```
void Jeu::run()
```

7. Classe Company

En ce qui concerne la classe Company, ces méthodes permettent de gérer les niveaux de l'entreprise en fonction de l'argent accumulé. *getLevel* détermine le niveau actuel de l'entreprise, tandis que *getNextLevelGoal* fournit l'objectif d'argent nécessaire pour atteindre le niveau suivant. Les plages de valeurs dans les instructions *switch* facilitent la gestion des niveaux et des objectifs de manière claire et concise. La classe Jeu décrite ci-dessus organise le jeu est fait appelle au fonction de la classe Company comme :

```
void Company::produce()
```

```
void Company::buyRawMaterials(std::string material, int nb)
```

```
int Company::priceRawMaterials(std::string material, int nb)
```

```
void Company::hireWorker()
```

```
void Company::payImpots(float dollars)
```

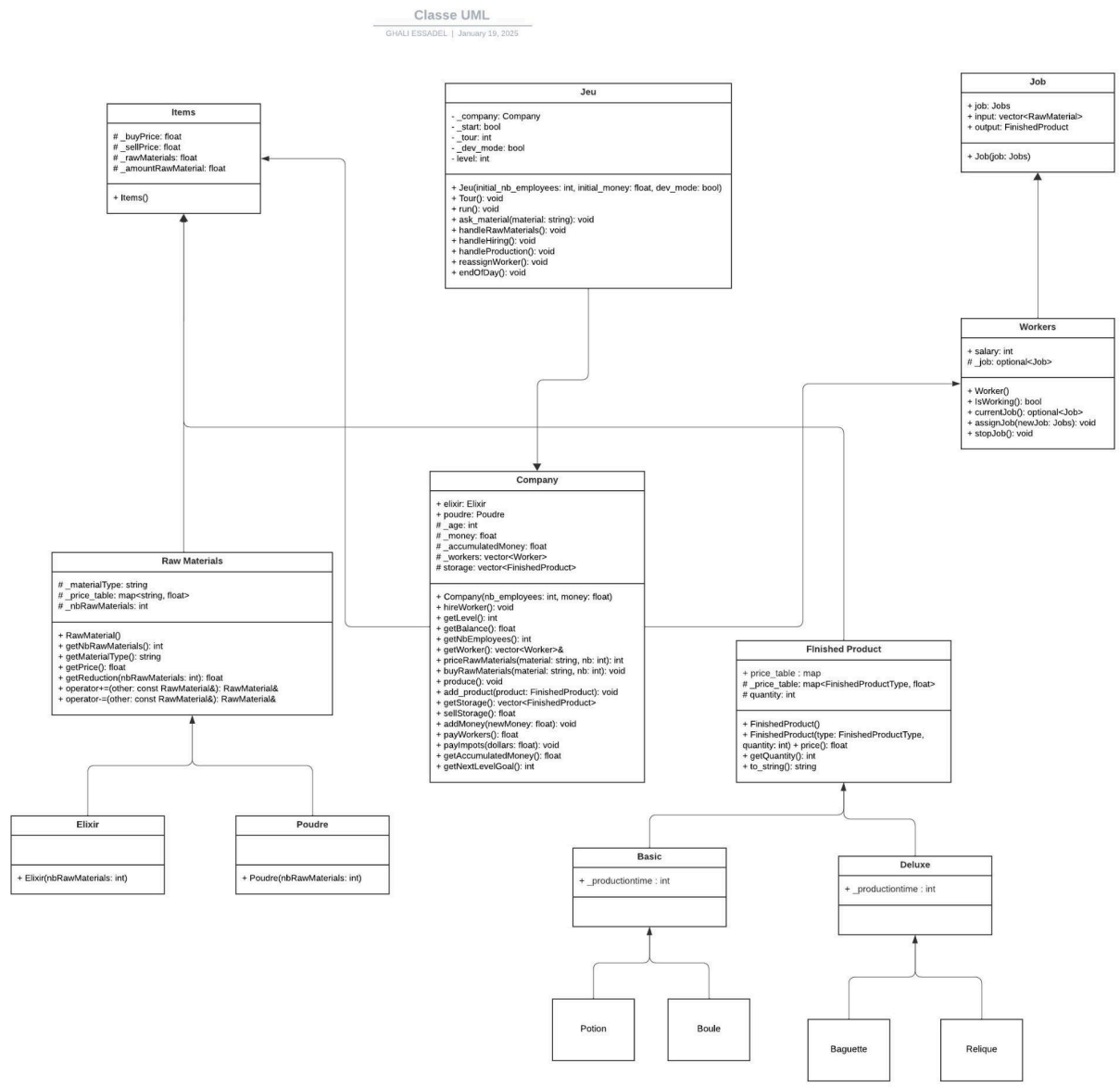
```
float Company::payWorkers()
```

```
int Company::getNextLevelGoal()
```

```
 int Company::getLevel()
```

La fonction Company s’occupe d’appeler des fonctions des autres classes.

Diagramme UML :



Utilisation de la programmation orientée objet (CONTAINERS) :

Nous avons principalement utilisés ces conteneurs de la stl : vector,map et optional

Voici un exemple d'utilisation de map :

```
std::map<std::string, float> RawMaterial::_price_table = {
    {"elixir", 1.5},
    {"poudre", 3.2},
};
```

L'utilisation du conteneur `std::map` dans rawMaterials.cc est justifiée car il permet de stocker des paires clé-valeur, associant les noms des matières premières à leurs prix respectifs. Cela offre un accès rapide et efficace aux prix en utilisant les noms comme clés. Enfin, `std::map` simplifie l'ajout de nouvelles matières premières ou la modification des prix existants.

Voici un exemple d'utilisation d'optional :

```
protected:
  /// None if workless
  std::optional<Job>  job;
```

L'utilisation de `std::optional` pour `_job` dans worker.hh permet de représenter de manière explicite l'état d'emploi d'un travailleur. `std::optional<Job>` peut contenir une valeur de type Job ou être vide indiquant que le travailleur est sans emploi. Cela évite l'utilisation de pointeurs nuls ou encore les risques d'erreurs liées à l'accès à des valeurs non initialisées.

Tests unitaires :

Les tests unitaires en C++ permettent de vérifier que chaque composant individuel d'un programme fonctionne correctement. Dans le fichier testcase.cc, nous utilisons la bibliothèque Doctest pour écrire et exécuter des tests. Chaque TEST_CASE représente un scénario de test distinct pour la fonction getReduction de la classe `RawMaterial`. Les macros CHECK et CHECK_THROWS_WITH_AS sont utilisées pour vérifier que les résultats de la fonction sont conformes aux attentes. Par exemple, le premier test vérifie que la fonction lance une exception avec un message spécifique lorsqu'un argument négatif est passé. Les autres tests vérifient que la fonction retourne les réductions correctes pour différentes quantités de matières premières. En exécutant ces tests, nous pouvons nous assurer que la fonction getReduction se comporte comme prévu dans divers cas.