

# Documentation : *Projet Shogi*

---

## Ou autrement dit : Rapport de Projet

---

**Rédaction : NoobZik**

Le rapport il à été rédigé en markdown, flemme d'ouvrir Office qui met 5 ans à démarrer pour rédiger un rapport projet. Accrochez vous bien, car le rapport que vous allez lire n'est pas un rapport comme les autres.

---

## Introduction

---

Le sujet projet à été présenté durant le 20 mars 2017 en cours d'amphi de programmation impérative. Le sujet est donc le **Shogi**

J'étais à la base dans un trinôme composé de 2 Informatique et 1 Maths (Oui je sais je me suis inclus, ne vous demandez pas pourquoi). Mais suite au restriction nouvelle d'être exclusivement entre la même promotion introduite cette année. J'ai du être tout seul. Je vous remercie Christophe Tollu pour cette superbe annonce :( .

Le projet du shogi à été immédiatement commencé le jour même de la présentation. C'est pas marrant de passer les vacances à programmer alors que les maths c'est bien pour réviser.

Tout le projet a été développé à l'aide d'un GIT dans lequel nous avons développé de manière à pouvoir continuer à développer l'application à distance et permettre de retracer entièrement l'historique du projet (rapport de bug, état de développement des branches du programme et des différentes manipulations dans le code). L'avantage principale est de supprimer totalement la fragmentation des versions du projet. Vous voyez, j'aime pas avoir plusieurs versions.

La supervision du répertoire git est exclusivement faite par NoobZik, en même temps c'est le seul qui a des connaissance dessus et qui n'aime pas attendre les modification par clé USB ou transfert par G-Drive ou mail.

La méthode de développement est la « méthode agile ». C'est-à-dire de par adaptation systématique de l'application aux changements du besoin détecté lors de la conception-réalisation du projet et par remaniement régulier du code déjà produit (re-factoring). Avec un planning de développement à base de secteurs dans le programme sans réelle précision

En gros, on prends une fonction au hasard et on fait avec jusqu'à qu'on rencontre un problème.

Le projet est fait en sorte qu'il écrit sous forme de MVC (ou presque ...). Un petit rappel qui ne fait pas de mal : Modele, Vue, Controleur. La priorité absolue du projet est de modéliser la vue. Une fois que c'est fait, on peut commencer à coder en free-style.

## Répartition des tâches

J'ai tout fait :) L'avantage d'être tout seul, c'est qu'on finit le projet plus vite.

Prenez exemple chez Free. Lors du lancement de Free Mobile, le site officiel était tellement surchargé qu'il a planté en fin de journée avec des conséquence de grosse base de données rendu inexploitable. Il y a eu seulement 7 500 requete enregistreé, tout le reste ... à la corbeille.

Rani Assaf, directeur technique de Free, a re-coder tout l'infrastructure en 4 jours seulement, tout le travail de 6 Ingénieur faite en 2 ans. Vous voyez, avec de la motivation de le faire tout seule, on fini vite.

Pendant ce temps, l'ENT de l'Université Paris 13 tombe souvent en rade depuis un très long moment. A qui la faute

d'ailleurs ?

Bref, en faite, je suis déjà dans un trinôme, le truc, c'est qu'on fait le projet dans notre coin et on rassemble à la fin... J'attends juste qu'ils finissent leur projet.

---

## Descriptions des fichiers

---

### La liste des fichiers

Il y a au total 8 fichiers (à l'exclusion du main).

1. piece.c
  2. pile.c
  3. file.c
  4. mouvement.c
  5. restriction.c
  6. game.c
  7. sauvegardes.c
  8. debug.c
- 

### Description détaillé des fichiers

Cette partie détaille les fonctions qui ont été modifié du sujet ou rajouté.

#### Un MAKEFILE pas comme les autres

La description de l'utilisation du makefile est indiqué dans le [README.md](#)

Cependant, le projet avait l'objectif très strict d'avoir 0 erreurs et 0 warnings à la compilation.

De ce fait, le makefile contient tout les affichage warnings possible. C'est pour ça que l'affichage du terminal est très surchargé à la compilation du programme.

Mais c'est pas du tout marrant de compiler avec tout les warnings dès le début du projet. C'est comme tricher. Je trouve que c'est vraiment marrant de le faire à la fin, quand -Wall n'affiche plus rien. Entre nous, compiler avec seulement le warning -Wall, ce n'est pas assez jouissif pour régler tout les warnings. -Wall c'est pour les débutants. Nous ce qu'on veut c'est du zéro tolérance, c'est amusant d'avoir environ 600 Warnings d'un coups.

---

#### piece.c

Dans ce fichier, contient tout les fonctions concernant à la gestion des pièces

Une structure pièce est caractérisé par trois champs :

- Un entier représentant la couleur, il y en a trois {Blanc, Noir et VIDE}.
- Un entier énuméré type représentant l'ensemble des pièces en plus de VIDE, SELECT\* (Plus de détails plus bas).
- Un statut énuméré pour la représentation d'une piece pièce ou non promu.

**promote\_grant, demote\_grant, demote\_grant\_reserve**

```
piece_couleur(); Initialement piece_joueur();
```

Prend en argument une pièce `piece_t` et renvoie sa couleur associé.

```
promote_grant(); Permet de promouvoir la piece.
```

```
demote_grant_reserve(); Permet de dépromouvoir la piece en changeant de couleur.
```

```
demote_grant(); Exactement la même chose qu'au dessus mais sans changer de couleur.
```

```
switch_color(); Elle permet de changer de couleur avant de placer la pièce dans la réserve.
```

---

## pile.c / file.c

La *file* représente l'historique des coups jouée et ses événement associé (Capture de piece et promotion).

La *pile* représente l'historique brute des pièces capturés.

Pour la pile et la file, elle marche comme des liste qui sont doublement chaînée vu en TD. Cependant, deux fonctions on été introduite pour plus de clarté

```
file.c
```

```
void file_thread();
```

Permet d'ajouter un mouvement, un état de promotion et un état de capture dans l'historique.

```
void file_unthread();
```

Fait le contraire, mais retire le dernier mouvement ajouté.

```
pile.c
```

```
void pile_stacking(); Ajoute une pièce dans la pile.
```

```
void pile_unstacking(); Retire la dernière pièce ajouté dans la pile.
```

---

## mouvement.c

Les mouvement (autrement dit les déplacement dans le jargon du sujet) sont répertorié dans ce fichier.

Il s'agit de vérifier si les déplacements sont possible ou pas, de manière très naïve.

Ce fichier est décomposé en 5 blocs :

### 1. Validation de coordonnées d'entrée et de sortie :

- Pour les coordonnées d'entrée, on vérifie qu'il sont bien dans l'échiquier 11x11
- En revanche pour les coordonnées d'arrivée, on doit vérifier qu'il sont dans l'échiquier 9x9 (c'est-à-dire sans la réserve). Une valeur de coordonnées est créer pour désélectionner la case, qui est (42,42), si aucun mouvement n'est possible (Notament des parachute de pions).

### 2. déplacement\_valide et valide\_win

- déplacement\_valide est composé de switch case pour appeler la fonction de déplacement valide adéquate. Ici il y a aussi un truc pour désélectionner la pièce, il suffit de mettre les mêmes coordonnées de départ et d'arrivée.
- valide\_win Permet juste de savoir que si les coordonnées d'arrivée pointe un roi, bah la partie se termine. (**Remarque:** apparemment d'après plusieurs application shogi, il n'y a aucune annonce d'échec ou échec et mat dans le monde de compétition, en conséquence, ces deux fonction ne sont pas implémenté).

### 3. déplacement\_valide(Inserer type de piece)

- Ce bloc contient tout les déplacement des pièces non promu et des pièces non promu

#### 4. `movement_valid_helper`

- Ce bloc sert essentiellement pour la restriction qui sera abordée plus en bas.

#### 5. `deplacement_apply`

La fonction `deplacement_apply` est un peu complexe à comprendre.

A la base, `deplacement_apply` permet justement d'appliquer de manière naïve les déplacements dès qu'elles sont vérifiées. D'inscrire ce déplacement dans la file d'historique et s'il y a capture, retirer la pièce capturée de l'échiquier et le rajouter dans la pile de capture de pièce.

Mais dès l'introduction de la réserve de parachute, il a fallu le refactoriser. Puisqu'il faut afficher cette pièce capturée dans la réserve. Et en plus de cela, il faut récupérer les données de promotion, s'il y a eu ou pas.

Avant de commencer à commenter la réserve, **il faut avoir en tête que:**

- La réserve du haut et de la gauche sans la case 0,9 constitue la réserve noir
- La réserve du bas et de la droite sans la case 10,0 constitue la réserve blanc

On commence donc à initialiser la valeur de promotion par

```
int promoted_v = is_promoted();
```

La condition est de vérifier si les coordonnées d'arrivée contiennent une pièce. Une fois qu'elle est vérifiée, on la place naturellement dans la capture des pièces en retirant de l'échiquier, et de boucher la pièce.

Vient ensuite le placement dans la réserve.

Avant de commencer les boucles while tant attendu, il se peut que la pièce capturée est une pièce promu, dans ce cas, les conditions de promotions sont vérifiées. Si c'est vrai, la pièce est dépromu en changeant de couleur. Dans le cas contraire, on change juste la couleur.

Comme il y a deux bloc de réserve pour chaque joueur, il y aura donc au totale 4 boucles while

**Cas universel:** (Noir et vous devinerez pour les blanc qui est presque la même chose).

La première boucle va parcourir la réserve du haut, de la droite vers la gauche. Pendant ce temps, on teste si la case est vide :

- Si elle est vide : On place la pièce capturée\*(cf `note_1`) à cette case, et on change la valeur de boucle pour sortir
- Sinon on passe à la case suivante.

Dans le cas où toutes les cases sont occupées, on change la variable de la deuxième boucle pour qu'on puisse entrer dans la deuxième boucle.

Donc on parcourt maintenant la réserve de gauche, du haut vers le bas. C'est le même algorithme que ci-dessus. Mais on parcourt jusqu'à l'avant dernière case (Car la dernière case c'est pour l'autre joueur).

Une fois qu'on a fait tout ça, il ne reste plus qu'à ajouter les données dans la file et de changer de joueur.

S'il n'y a pas de pièce à l'arrivée, on applique tout simplement les déplacements.

\* cf `note_1` : Dans le cas où la pièce est promu, on fait appel à la fonction

```
demote_grant_reserve();
```

#### 6. `annuler_deplacement()`

Cette fonction permet de faire le chemin inverse de `deplacement_apply`.

On fait une extraction du dernier élément inséré de la file. Puis en fonction des données de cet élément on peut soit :

- Dépromouvoir une pièce
- Restaurer une pièce capturée en prenant soin de traiter la réserve. (Il est possible qu'il est toujours buggé, Utilisez la fonction Signalement de bug sur le repo du bitbucket)
- Et dans tous les cas, restaurer la position initiale de la pièce en question avant le déplacement.

On prend aussi le soin de changer de joueur. Sinon c'est pas marrant de jouer deux fois d'affiler...

---

## restriction.c

Cette partie du code est un peu délicate. Lorsqu'on a développé les déplacements, on s'est rendu compte que les pièces peuvent se balader sur tout l'échiquier même s'il y a des obstacles sur leur parcours.

Pour contrer ce comportement anormal, il a fallu introduire des restrictions de mouvement. Il y a cependant un nouveau problème. Les pièces pouvaient quand se déplacer partout.

Donc pour contrer encore une fois ce comportement anormal, la fonction

```
mouvement.c  
  
deplacement_valide();
```

est modifiée pour que les pièces peuvent se déplacer exclusivement sur des cases de type '\*' (SELECT).

C'est ici que se fait le prototype de l'aide visuelle, avec les cases '\*' (SELECT) L'aide visuelle permet d'afficher les déplacements possibles sur l'échiquier.

## Détails du fonctionnement des restrictions

Nous avons atteint les limites du code, en effet, il n'est pas possible d'avoir une fonction qui possède un argument optionnel.

Nous avons donc une coordonnée par défaut qui est donc comme d'habitude :

```
coordinate_t COORDINATE_NULL = {42, 42};
```

Pendant l'affichage de l'échiquier, il y a une condition qui permet d'entrer dans la condition qui vérifie si les coordonnées sont différentes de 42,42 justement.

Lorsque l'utilisateur entre une coordonnée d'entrée, la condition est vérifiée et c'est ici que entre en jeu les restrictions.

La fonction

```
movement_restriction();
```

est appelée durant `deplacement_valide`. Cette fonction appelle la restriction adaptée en fonction de la pièce.

Si la restriction n'exige pas de conditions particulières, on fait appel à

```
Situé à mouvement.c :  
movement_valid_helper();
```

par une boucle `for` pour tester les cases.

Entrons dans les détails de ***movement\_valid\_helper***.

Cette fonction permet de savoir si la case testé par la boucle for est valide pour déplacement ou pas. Ils font appel naturellement aux déplacements valide des pièces respectifs, **A l'exception de tour\_promu et fou\_promu qui font appel à roi.**

Si vous voulez comprendre pourquoi cette exception : Tout simplement car si on ne met pas ces exceptions, alors l'aide visuel va tout simplement permettre de sauter d'autre pieces pendant leur déplacement. Et justement c'est le but des restrictions on le rappelle.

Pour en revenir à ***movement\_restriction***. Pour les quatre cas *tour, fou, lance et parachute* et leurs promotions respectifs ont leur propres restrictions spécifique. qui sont défini dans le même fichier. Je vous invite donc à regarder les commentaires pour leur fonctionnement.

- **Note** : Les parachutes, sauf pions, tout les cases sont changé en '\*'.

Après avoir effectué les restrictions, le joueur saisie les coordonnées d'arrivé. La condition dans

```
deplacement_valide(); et  
restriction_detector();
```

doivent être tout les deux vérifier pour pouvoir appliquer les déplacements.

```
restriction_detector();  
permet de tester si la case d'arrivée est vide ou occupé par une pièce de couleur  
différente.
```

### Un commentaire pour pion parachute

Donc les pions ont une restriction particulière, ils ne peuvent pas se placer sur la colonne où il y a déjà son compatriote du même type.

Pour ça, il y a 3 Boucles for. Il faut aussi noter que pour accéder à la troisième boucle il faut que la condition de test soit vérifié (Elle est changé par la deuxième boucle si besoin).

- La première permet de parcourir les colonnes
- La deuxième permet de parcourir chaque ligne de la colonne actuelle, si il y a un compatriote, on change la variable conditionnel pour ne pas rentrer dans la troisième boucle
- La troisième boucle permet de changer tout les case de la colonne en case SELECT pour l'aide visuel.

---

### game.c

Pas grand chose à commenter ici, vu que game.c se focalise exclusivement de l'interaction utilisateur et machine par une interface graphique, appel les fonctions en fonction de l'interaction de l'utilisateur.

Il suffit de lire les commentaires dans le dossier associé. Je ne vois pourquoi je détaillerai cette partie dans ce document...

---

### sauvegardes.c

Donc déjà pour commencer

```
struct stat st = {0};
```

Va nous servir à check si le dossier existe ou pas par la fonction

```
stat();
```

---

La fonction

```
partie_sauvegarder();
```

est divisé en deux sous fonctions

```
game_save_board(); Qui permet de sauvegarder l'échiquier  
game_save_meta(); Qui permet de sauvegarder le contenu de la pile et file.
```

Aussi, cwd permet de récupérer le chemin du projet actuelle pour pouvoir créer un dossier qui n'existe pas et de sauvegarder dans ce dossier.

### Un commentaire pour partie charger :

J'avais eu un problème de chargement de partie. Les pièces été affecté a la mauvaise position et à entraîné pour la première fois les erreurs de segmentations et les erreurs qui sont directement pointé les fichiers includes du système.

Il s'avère qu'en changeant la boucle de 0 à 12 (Initialement de 0 à 11) cela règle le problème mais je n'ai pas compris pourquoi...

---

C'est la fin de cette documentation ou rapport de projet.