Seance 3:

Développement par tests.

(durée max: 2h)

Etapes d'analyse de la ligne de commande:

Une fois le projet réalisé, l'échiquier doit être commandable à l'aide de *phrases* envoyées par l'utilisateur en **ligne de commande**.

Par exemple:
`> d 1 3 -> 3 2`
Devra permettre de déplacer le pion situé en (1,3) vers la case (3,2).

Les autres commandes sont les suivantes:
`> i :` Initialisera l'état de l'échiquier à l'état initial.
`> # ceci est un commentaire:` Toute phrase commençant par # est un commentaire non analysé.
`> lit <nom_du_fichier> :` permet de lire et de charger l'échiquier contenu dans "*nom_du_fichier*".
`> ecrit <nom_du_fichier> :` permet d'écrire l'état du fichier dans "*nom_du_fichie*r".

Chaque type de commande se traduit par une fonction de **l'API** du jeu.
Par exemple, la fonction `api_echec_deplacement_piece`, permet de demander le déplacement d'une pièce.
Ces fonctions sont considérées comme de "***haut niveau***". Leur paramètres ne doit pas faire apparaître **publiquement** des détails d'implémentation ou de contrat de code spécifiques.
Ce sont les fonctions qu'un **utilisateur externe** ne connaissant pas votre projet doit pouvoir **utiliser** facilement.

Le coeur du projet consiste à permettre le traitement correct du déplacement des pièces suite à la commande **`d`**.

Plusieurs étapes ont lieu successivement:

1. **Analyse de la ligne de commande** (étape dit de ***Parsing*** de texte) qui détecte si la ligne est valide au niveau textuelle, mais pas si le coup est valide au niveau du jeu.

ex.
```
# entrée textuelle considérée comme invalide:
> d 7-> 3 4 7
#entrée textuelle considérée comme valide:
> d 14 9 -> 5 12
```

L'analyse du texte sur une ligne de déplacement valide aboutit au stockage des coordonnées de départs (x0,y0) et de destination (x1,y1) dans des entiers.
Notez que cette étape est déjà implémentée.

2. **L'analyse des coordonnées** (x0,y0) et (x1,y1) pour savoir si le déplacement demandé est valide au niveau du jeu.

Cette étape peut se décomposer de la manière suivante:
Vérification que les coordonnées sont bien dans l'intervalle [0,7].

- Vérification qu'il **existe** une pièce aux coordonnées de départ.
- Vérification qu'un déplacement **non null** est bien défini.
- Vérification qu'aucune pièce du jeu du joueur n'est positionnée sur la case de **destination**.
- Vérification que le **déplacement** est bien valide pour cette pièce et qu'aucune pièce ne bloque son trajet.

3. Le **déplacement effectif** de la pièce.
Modification de ces coordonnées, et **suppression** potentielle d'une pièce du joueur adverse.

Méthodologie de tests:

Au fur et à mesure du développement de votre projet, il est important de réaliser des tests.

- Des **tests unitaires** après l'écriture de chaque fonctionnalité importante qui vont tester une ***fonctionnalité "unitaire"***.
- Des **tests d'intégrations** après un certain nombre d'étapes qui vont tester un ***comportement global***.

Le **développement par tests** consiste à écrire les tests des fonctionnalités à vérifier **avant** l'implémentation de la fonction elle même.
Il est en effet souvent plus facile de décrire les **appels** que l'on **souhaite** et la **sortie attendue**, plutôt que de décrire directement comment la fonction va être implémentée.

Par exemple, considérons le cas de la fonction `est_coordonnee_valide()`.
Avant l'écriture de la fonction, nous aurions pu définir un certain nombre de tests que doit passer cette fonction.

Par exemple:

```
int main()
{
    est_coordonnee_valide(0,0); //doit retourner 1
    est_coordonnee_valide(1,1); //doit retourner 1
    est_coordonnee_valide(2,6); //doit retourner 1
    est_coordonnee_valide(7,7); //doit retourner 1
    est_coordonnee_valide(-1,5); //doit retourner 0
    est_coordonnee_valide(1,-1); //doit retourner 0
    est_coordonnee_valide(1,8); //doit retourner 0
    est_coordonnee_valide(8,1); //doit retourner 0
}
```

Une fois les tests écrits, ils **guident** l'implémentation de la fonction.
Ici, l'écriture des tests guident sur le type d'arguments passés en paramètres (2 entiers).

Les tests dit **unitaires** doivent vérifier de manière méthodique chaque point particulier, autant les cas **valides** que les cas **non valides.**

Par exemple:
Tester uniquement que l'appel à `est_coordonnee_valide(1,5)` renvoie 1 ne permet **pas** de conclure que la fonctionnalité est correcte.

Par contre, le code suivant permet de tester que tous les cas d'entrées valides sont traités correctement:

```
int main()
{
    int kx=0,ky=0;
    for(kx=0;kx<=7;++kx)
    {
        for(ky=0;ky<=7;++ky)
        {
            if(est_coordonnee_valide(kx,ky)!=1)
            {printf("Test KO %d %d \n",kx,ky); exit(1);}
        }
    }
    printf("Test OK\n");
}
```

Ce code permet d'afficher **exhaustivement** toute erreur détectée sur l'intervalle [0,7]x[0,7].

Ce test permet de vérifier l'état valide, par contre il faut également vérifier que l'état **invalide** est correctement détecté.
Pour cela, on peut définir 6 cas possibles:

- `coordonnée x<0 && y OK`
- `coordonnée x>7 && y OK`
- `coordonnée y<0 && y OK`
- `coordonnée y>7 && y OK`
- `coordonnée x<0 && y<0`
- `coordonnée x<0 && y>7`
- `coordonnée x>7 && y<0`
- `coordonnée x>7 && y>7`

Il est important de tester les **cas limites** avec x/y valant strictement -1 et +7.
Par exemple, ne pas tester uniquement avec le nombre 78.

Mise en place des tests de l'application:

L'application doit permettre de déplacer les pièces de manière cohérente vis à vis d'une partie d'échec.

Par exemple, la combinaison
```
> i
> d 2 1 -> 2 2
> d 3 6 -> 3 5
> d 3 1 -> 3 2
> fin
```
Doit aboutir à un état de l'échiquier **valide** et connue (3 pions ont été déplacés d'une case).

Au contraire, la combinaison:
```
> i
> d 2 1 -> 2 5
> fin
```
Doit aboutir à la détection d'un coup **invalide** (un pion ne pouvant pas se déplacer de 4 cases vers l'avant).

Notez que ces combinaisons peuvent être écrites **dès maintenant**. Dans un code bien construit, ces tests de l'application ne dépendent pas du codage interne de la structure de données utilisée pour manipuler l'échiquier.
Ces tests doivent donc pouvoir être exécutés, peu importe le code interne.
Par exemple, les tests des binômes voisins peuvent être par la suite appliqués sur votre code.

Il vous est demandé d'établir la série de tests qui viendra **valider** que votre projet réalise bien la fonctionnalité d'un jeu d'échec.
Pour cela, vous allez écrire un ensemble de tests **unitaires** et **d'intégrations**.
Ils devront valider à la fois les coups **possibles**, et également confirmer que les coups **invalides** sont bien détectés comme tels.

Vos tests devront couvrir un **maximum de cas possibles**.
Réfléchissez à une **stratégie** adéquate.

Vous garderez ces tests tout au long de votre projet. Ils vous permettront de valider votre projet et vous aideront à ne pas oublier de cas particuliers lorsque vous coderez. Si ces tests sont insuffisants, ils ne pourront pas valider clairement votre travail.

Quelques conseils:

- Ne passez pas l'ensemble de vos tests à observer le déplacement valide d'un pion à différents endroits de l'échiquier. Mais **couvrez** l'ensemble des pièces possibles.

- Réaliser au minimum **1 test** de validité **par type** de déplacement
  (ex. Pion avançant en ligne droite, pion avançant en diagonale lorsqu'un adversaire est présent, notez que les pions noirs et blancs n'avancent pas dans la même direction. Tour avançant suivant x>0, cas d'une tour avançant suivant x<0, cas d'une tour avançant verticalement, présence d'une pièce adverse ou non, etc.)

- Pour les tests unitaires et pour les tests de fonctionnalité invalides, réalisez un **unique test** par fonctionnalité.
  (ex. Ne testez pas en même temps la tour et le fou, ne testez pas l'un après l'autre deux coups invalide puisque seul le premier passera à l'exécution, …)

- Réalisez quelques tests **d'intégrations** aboutissant à une configuration de plateau plus complexe.

- Pensez bien aux cas de coups **invalides**, aux **cas particuliers**.
  Etablir les tests des cas particuliers dès maintenant permet de ne pas les oublier plus tard au moment du développement de code.

- Utilisez les **outils** à disposition pour simplifier la mise en scène des tests: *lecture de fichiers de scènes, initialisation, commentaires*.

Format demandé:

Chaque test doit être écrit dans un **fichier séparé**.
On rangera ces fichiers dans une **arborescence spécifique**:

```
test/test_unitaire_ok/test_unitaire_ok_01.txt
                     (test_unitaire_ok_01_entree.txt)
                      test_unitaire_ok_02.txt
                     (test_unitaire_ok_02_entree.txt)
                      test_unitaire_ok_03.txt
                     (test_unitaire_ok_03_entree.txt)
                      ...
    /test_unitaire_ko/test_unitaire_ko_01.txt
                     (test_unitaire_ok_01_entree.txt)
                      test_unitaire_ko_02.txt
                     (test_unitaire_ok_02_entree.txt)
                      test_unitaire_ko_03.txt
                     (test_unitaire_ok_03_entree.txt)
                      ...
    /test_integration/test_integration_01.txt
                      test_integration_01_entree.txt
                      test_integration_01_sortie.txt
                      test_integration_02.txt
                      test_integration_02_entree.txt
                      test_integration_02_sortie.txt
                      test_integration_03.txt
                      test_integration_03_entree.txt
                      test_integration_03_sortie.txt
```

Tests OK:
Les fichiers de **`test_unitaires_ok`** sont des tests de **déplacement valides** qui doivent aboutir à un déplacement d'une pièce.

Ils possèdent la structure donnée en exemple.

```
#test_unitaire_ok_01
#
#NOM1: ZZZZ
#PRENOM1: ZZZZ
#NOM2: ZZZZ
#PRENOM2: ZZZZ
#GROUPE: ZZ
#
# Ce test permet de verifier qu'un pion puisse se deplacer en ligne droite d'une
case lorsqu'il n'y a pas d'obstacles.
#
i
d 1 1 -> 1 2
fin
#Sortie attendue:
# Pion initialement place en (1,1) est deplace en (1,2)
```

Le test précise en commentaire:

- le nom des auteurs,
- le but du test,
- les lignes de contrôles effectivement envoyées,
- la sortie attendue.

➔ Appelez votre programme avec la syntaxe suivante, testez avec le fichier exécutable solution:
`./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_01.txt`

➔ Quelle est la configuration de sortie après exécution de ce test?

Note: Le symbole “<” indique la **redirection d'entrée standard**. C'est à dire qu'à la place de venir lire les instructions sur la ligne de commande, le programme va les lire dans le fichier.

Pour directement s'aider d'une configuration particulière, on pourra écrire en dur des **fichiers** de configuration d'échiquier (voir documents annexes).

On pourra prendre exemple sur le cas *03* donné dans le jeu de tests fourni.
Notez que pour vous aider, vous pouvez visualiser l’échiquier correspondant à ce fichier en appelant le visualiseur de cette manière:
`./visualiseur ../test/test_unitaire_ok/test_unitaire_ok_03_entree.txt &`

Pour aller plus loin.
Lorsque beaucoup de fichiers de tests sont ecrits, il est utile de pouvoir les lancer automatiquement.
Ecrivez le fichier lanceur.sh suivant:

```
================
#!/bin/bash

./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_01.txt
./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_02.txt
./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_03.txt
================
```

Puis tappez en ligne de commande:
$ chmod +x lanceur.sh
$ ./lanceur.sh

Observez que les tests sont exécutés les uns derrière les autres.

Il est également possible de sauvegarder la sortie du projet dans un fichier plutôt que de l'avoir à l'écran. On se sert alors de la redirection de sortie. On pourra par exemple lancer:
$ ./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_01.txt > mon_fichier_de_sortie.txt

Observez alors le contenu de mon_fichier_de_sortie.txt

Tests KO:

Les tests du répertoire **`test_unitaire_ko`** doivent tester des déplacements ou des entrées textuelles invalides.
Testez bien chaque **cas particulier** que vous devrez gérer au niveau du code.

Lorsque vous coderez les fonctionnalités, vous devrez retrouver au moins un fichier de test correspondant à chaque cas particulier du code.
Vous pouvez vous inspirer des fichiers d'exemples fournis.

Pour aider à coder vos fonctionnalités, vous commenterez également le **type de message d'erreur attendu**, ainsi qu'à quel **niveau** cette erreur devra être détectée (parseur, API, déplacement propre à une pièce, etc.)

Tests d'intégrations:

Les tests d'intégrations vérifient des **combinaisons** de déplacements valides plus complexes afin de permettre de vérifier que le système global s'articule correctement.

La différence avec le `test_unitaire_ok` provient du fait que l'on ne vise pas à tester une unique fonctionnalité mais le **comportement global**.

Les tests aboutissent à des configurations plus complexes, et donc longues à vérifier. Il est avantageux de définir en supplément l'état formel de l'échiquier attendu après le déplacement des pièces (`fichier test_integration_XX_sortie.txt`).
De cette manière, il est possible de **comparer** les fichiers du résultat obtenu avec celui du résultat attendu afin de s'assurer que le résultat est bien celui escompté.

On pourra s'inspirer des exemples fournis dans le répertoire des tests d'intégration.

Lors le programme sera complété, vous pourrez exécutée les tests d'intégrations avec la syntaxe:
```
$ ./projet_3eti_solution < ../test/test_integration/test_integration_01.txt
```

➔ Comparez le fichier d'état attendu par rapport à l'état obtenu visuellement.

Pour aller plus loin:

Ecrivez le fichier le fichier d'état normalement obtenu dans echiquier_courant.txt
Comparez le fichier d'état attendu par rapport à l'état obtenu à l'aide de la commande diff:
$ diff -i -E -b -w -B echiquier_courant.txt ../test/test_integration/test_integration_01_sortie.txt

Modifiez le fichier test_integration_01_sortie.txt en modifiant les coordonnées d'une pièce.
Relancer la commande diff, et observez le résultat.
Concluez sur la mise en place d'une procédure de test automatique.

➔ Implémentez les différents tests.

Ecriture de l'état de l'échiquier.

(durée max: 2h)

L'échiquier est une structure contenant 2 jeux et un entier:

- le jeu des pièces blanches
- le jeu des pièces noires.
- le joueur courant (entier valant 0:blanc ou 1:noir).

L'état de l'échiquier peut être initialisé à l'aide de la fonction `echiquier_initialise`.

Pour récupérer le type et les coordonnées de la 3ème pièce du jeu noir, on pourra utiliser la syntaxe:

```
int main()
{
    echiquier echiquier_courant;
    echiquier_initialise(&echiquier_courant);
    jeu *jeu_courant=&(echiquier_courant.jeu_de_piece_noir);
    piece* piece_numero_3=&(jeu_courant->ensemble_de_piece[3]);
    type type_courant=piece_numero_3->type;
    int x_courant=piece_numero_3->coord_x;
    int y_courant=piece_numero_3->coord_y;

    //de la même maniere, on peut acceder à un champ précis sans étapes
intermédiaires:
    type type_courant2=echiquier_courant.jeu_de_piece_noir[3].type;
    int x_courant2 = echiquier_courant.jeu_de_piece_noir[3].coord_x;
    int y_courant2 = echiquier_courant.jeu_de_piece_noir[3].coord_y;

}
```

L'état complet de l'échiquier peut être **sauvegardé dans un fichier** comme il vous est proposé en exemple.

L'implémentation du corps de la fonction d'écriture vous est donnée en tant que **librairie** (Observez la syntaxe **-l** dans le programme de compilation).

Il s'agit d'un binaire obtenu après compilation, et donc **non lisible** par un humain. Seule l'en-tête vous est donnée.
Il s'agit d'un format d'échange standard de fonctionnalités.
Sous Linux, les librairies possèdent l'extension **.so** (librairie dynamique) ou **.a** (librairie statique).
Les librairies standards sont généralement situées dans le repertoire `/usr/lib/`
Sous *Windows*, les librairies possèdent l'extension **.dll** et sont généralement placées directement à côté de l'exécutable.

Vous devez re-coder la fonctionnalité de cette librairie.
Pour cela, réécrivez le corps de la fonction **`entree_sortie_ecrit_echiquier_fichier`** dans `entree_sortie.c`, et enlevez la ligne de "*link*" à la librairie fournie dans le Makefile (c'est à dire la partie associée à l'option -l).

Pour vous aider à vérifier le contrat, vous disposer d'une fonction vérifiant qu'un fichier est bien accessible.
Pour ouvrir un fichier et écrire en *ASCII*, on peut suivre ce type de syntaxe:

```
int main()
{
    FILE *fid=NULL; //struct contenant un descripteur de fichier
    fid=fopen(filename,"w"); //ouverture du fichier "filename" en mode ecriture

    if(fid==NULL)
    {
        printf("Erreur ouverture du fichier %s\n",filename); exit(1);
    }

    fprintf(fid,"J'ecris dans un fichier le nombre %d \n",3);
    fprintf(fid,"Et je sais que 2+2=%d",2+2);
    fprintf(fid,"Enfin si j'ai une piece de type tour, j'ecrirai le nombre
%d\n",tour);

    int c=fclose(fid);
    if(c!=0)
    {printf("Erreur fermeture fichier %s\n",filename);exit(1);}
}
```

Note: La syntaxe de `fprintf` est équivalente à celle de `printf` à l'exception que le descripteur du fichier est passé en paramètre.

Pour aller plus loin:
La syntaxe:
fprintf(stdout,"J'ecris dans le fichier 'ecran'\n");
est identique à l'utilisation de printf, car stdout pointe vers la sortie de la ligne de commande.

- ➔ **Implémentez** l'écriture de l'état de l'échiquier dans un fichier.
- ➔ **Testez** votre code sur différents états de l'échiquier.

Travail en autonomie

(durée estimé: 3h):

- Révision fichiers, chaîne de caractères (1h).
- Finalisez les scripts de tests et déposez votre résultat (2h)

➔ **Déposez** votre projet (code source+scripts de tests) tel que précisé dans les consignes.