Yann.Secq@univ-lille.fr
Fabien.Delecroix@univ-lille.fr

TP n°10

Objectifs: Comprendre la démarche à suivre pour résoudre un problème complexe:
- choisir comment représenter les données
- décomposer en sous-problèmes (en fonctions)
- résoudre et tester chaque sous-problème séparément

1 Le jeu de saute mouton

Nous allons nous intéresser aujourd'hui à un petit casse-tête, c'est-à-dire un jeu à un seul joueur. Celui-ci consiste à déplacer des moutons pour passer d'une situation initiale de troupeaux se faisant vis-à-vis (figure 1) à une situation finale où les troupeaux se tournent le dos (figure 2).

FIGURE 1 – Position initiale des deux troupeaux de moutons.

FIGURE 2 – Position finale à atteindre pour gagner le jeu.

Les règles du jeu sont très simples et ne reposent que sur les mouvements possibles des moutons:
- un mouton ne peut pas changer de direction,
- un mouton ne peut jamais reculer,
- un mouton peut avancer d'une case si il n'y a personne devant lui,
- un mouton peut sauter au dessus d'un autre mouton si la case suivant le mouton qu'il passe est libre.

La figure 3 illustre un déplacement simple de la situation initiale illustrée avec la figure 1 du troisième mouton. La figure 4 illustre le saut du quatrième mouton au dessus de celui qui venait de s'avancer (coup suivant de la figure 3).

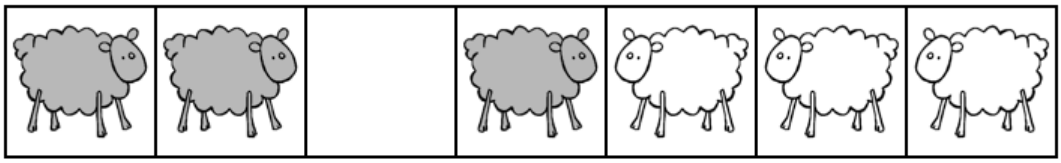

FIGURE 3 – Déplacement simple d'un mouton d'une case vers la droite.

Lorsque le joueur est efficace, il arrive à faire se croiser les deux troupeaux et atteint la position finale (figure 2) et résoud ainsi le casse-tête. Par contre, si il se trompe, il arrive rapidement dans une situation où il se trouve bloqué, comme l'illustre la figure 5. Dans cette situation, plus aucun coup n'étant possible, et on peut arrêter le jeu.

Vous allez écrire un programme qui permet à un joueur de jouer au saute-mouton.

2 Analyse et conception

Nous allons aborder les différentes étapes de l'analyse et la conception du problème, jusque l'obtention d'une solution. Pour ce problème nous allons vous proposer une décomposition, mais dans les TPs suivants vous serez de plus en plus amenés à proposer votre propre solution.

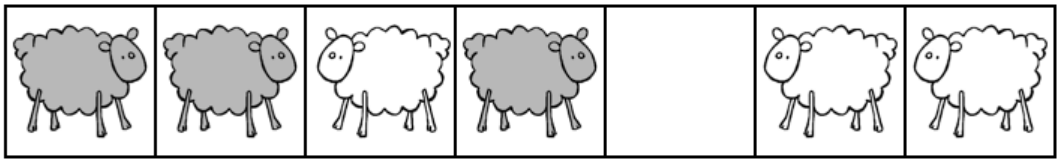

FIGURE 4 – Saut d'un mouton au dessus d'un autre mouton.

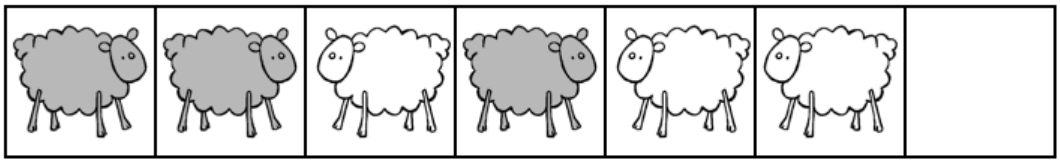

FIGURE 5 – Situation de blocage: il n'y a plus de coup valide possible.

L'étape zéro pour programmer la solution d'un problème (ici le saute mouton) est de bien comprendre le problème. Si nécessaire, faites une partie de saute mouton en utilisant par ex. des pièces de monnaie pour avoir une bonne compréhension du casse tête.

2.1 Les structures de données

La première étape consiste à identifier les objets qu'on manipule, et comment on va les représenter. Concrètement: Quel type utiliser pour représenter un mouton ? Quel type utiliser pour représenter la prairie, càd l'espace dans lequel évoluent les moutons ?

Dans la solution proposée, un mouton sera représenté par un caractère, et la prairie par un tableau de caractères. Un caractère sera réservé pour représenter la case vide de la prairie.

Commençons par initialiser la prairie telle qu'elle est au début du jeu, et l'afficher.

Q1. Créer un fichier `SauteMouton.java` qui contiendra le programme `SauteMouton`. Vous utiliserez ce fichier pour le reste du TP.

Q2. Déclarer ces constantes globales pour représenter les moutons:

```
final char G = '<';   // mouton qui va vers la gauche
final char D = '>';   // mouton qui va vers la droite
final char V = '_';   // l'espace vide de la prairie
```

Q3. Écrire l'algorithme principal qui construit et affiche la configuration initiale. Vous veillerez à utiliser les constantes définies précédemment. Écrire les deux fonctions utilisées dans `void algorithm()` ci-dessous:

```
void algorithm () {
   char[] prairie = new char[7];
   initialiser(prairie);          // Fonction à écrire
   println(toString(prairie));    // Fonction à écrire
}
```

Q4. Compilez et exécuter le programme pour vous assurer qu'il affiche bien la prairie initiale, à savoir

```
>>>_<<<
```

2.2 Identifier les sous-problèmes

Nous devons maintenant identifier quels sont les problèmes simples qu'on doit résoudre et qui nous permettront à construire une solution complète au saute-mouton. Commençons par faire une liste des sous-tâches que le programme devra pouvoir effectuer:

- initialiser la prairie (déjà fait)
- afficher la prairie (déjà fait)
- saisir le coup du joueur, càd le déplacement que le joueur veut faire
- faire le déplacement qui correspond au coup saisi
- détecter la fin de la partie, soit parce que la configuration finale est atteinte, soit parce qu'il y a blocage
- afficher le résultat du jeu: réussite ou non

Cette réflexion nous permet déjà de construire la forme générale du programme. Écrivons la en langage naturel:

```
- initialiser la prairie
- tant que le jeu n'est pas fini, càd la config. finale n'est pas atteinte et
                                                        il n'y a pas blocage
   - afficher l'état actuel de la prairie
   - saisir le coup du joueur
   - effectuer le déplacement
- afficher le résultat
```

Nous allons raffiner ce début de solution en déterminant comment chaque sous-tâche sera implémentée: par une fonction, ou par du code à écrire directement dans le programme. C'est la partie difficile, qu'on arrive à faire en accumulant de l'expérience. En règle générale, on essaye que:

— chaque tâche a sa fonction dédiée
— les noms des fonctions sont choisies de telle manière qu'en lisant le programme on comprend ce qu'il fait
— le code à écrire directement dans le programme se limite à des affectations de variables qui servent à transporter l'information entre les différentes fonctions, ou à des calculs très simples

Remplaçons chaque sous-tâche identifiée par une fonction qui lui sera dédiée. Pour l'instant nous donnons uniquement les noms des fonctions. Les signatures des fonctions (càd paramètres et valeurs retournées) seront précisées plus tard, quand on se posera la question des informations qui doivent circuler entre les différentes sous-tâches.

Le programme à écrire va ressembler à ceci:

```
void algorithm () {
   initialiserPrairie(...)
   while (! configFinaleAtteinte(...) && ! blocage(...)) {
      afficherPrairie(...)
      saisirCoup(...)
      effectuerDeplacement(...)
   }
   afficherResultatFinal(...)
}
```

FIGURE 6 – Ébauche du programme à écrire.

Il s'agit là d'une ébauche, qui nous servira de guide, et qu'on peut faire évoluer. Par exemple, nous avons vu que l'affichage se fait en utilisant une fonction `toString`, et que l'initialisation se fait par la fonction void `initialiser(char[] prairie)`.

2.3 Résoudre les sous-problèmes

2.3.1 Saisie d'un coup et déplacement

Pour saisir un coup, le joueur doit indiquer quel mouton il veut déplacer. Comment représenter cette information ?

Nous choisissons de représenter un coup par la position du mouton à déplacer, càd un entier entre 0 et 6. Donc, la fonction `saisirCoup` va retourner un entier qui sera ensuite passé à la fonction `effectuerDeplacement`. Effectuer un déplacement va mettre à jour la prairie, donc `effectuerDeplacement` prendra en paramètre le coup et la prairie.

Avant d'écrire la fonction `void effectuerDeplacement(int positionMouton, char[] prairie)`, il faudra s'assurer qu'elle se comporte comme attendu. Le comportement de notre programme final en dépend, et il est beaucoup plus facile de trouver une erreur dans une seule fonction que dans un programme entier. Donc, **on commence par écrire des tests pour cette fonction**.

Q5. Complétez ce test et l'écrire dans votre fichier.

```
void testEffectuerDeplacement () {
   char[] prairie;

   // de la prairie >>>_<<< en déplaçant le mouton 2, on doit passer à la prairie >>_><<<
   prairie = new char[]{D,D,D,V,G,G,G};
   effectuerDeplacement(2, prairie);
   assertArrayEquals(new char[]{D,D,V,D,G,G,G}, prairie);

   // de la prairie >>_><<< en déplaçant le mouton 4, on doit passer à la prairie >><>_<<
```

```
    prairie = new char[]{D,D,V,D,G,G,G};
    effectuerDeplacement(4, prairie);
    assertArrayEquals(new char[]{D,D,G,D,V,G,G}, prairie);

    // complétez ici par un autre exemple de déplacement
    ...
}
```

Puis, écrire une **fonction vide** `void effectuerDeplacement(int positionMouton, char[] prairie)`, juste de manière à pouvoir compiler le fichier. Renommer `void algorithm()` en `void _algorithm()`, compilez et exécuter le programme. Le test `void testEffectuerDeplacement()` est exécuté mais bien évidemment il est au rouge car pour l'instant la fonction testée ne fait rien.

Intéressons nous maintenant à la fonction `void effectuerDeplacement(int positionMouton, char[] prairie)` qui passe le test ci-dessus. Cela nous mène à la question: Que faire si `positionMouton` est un mouton qui ne peut pas être déplacé, par exemple la position 0 dans la prairie >>>_<<< ? Il y a deux manière de traiter ce problème:

- dans `effectuerDeplacement` on teste si `positionMouton` est valide, on doit alors décider ce que la fonction va faire si la position n'est pas valide,
- ou bien en suppose que `positionMouton` sera toujours valide lors des appels à `effectuerDeplacement`.

Nous optons pour la deuxième solution. Rappelons que le numéro du mouton à déplacer est donné par la fonction `saisirCoup`; ce sera donc `saisirCoup` qui va s'assurer de retourne une position de mouton valide.

Dans ce cas, le corps de `effectuerDeplacement` se limite à échanger le mouton de `positionMouton` avec la case vide.

Q6. Écrire la fonction `int indiceCaseVide(char[] prairie)` qui retourne l'indice de la case vide dans `prairie`. Puis, écrire un test pour la fonction `indiceCaseVide`, en complétant ceci:

```
void testIndiceCaseVide () {
    char[] prairie;

    prairie = new char[]{D,D,D,V,G,G,G};
    assertEquals(3, indiceCaseVide(prairie));

    prairie = new char[]{D,D,V,D,G,G,G};
    assertEquals(2, indiceCaseVide(prairie));

    // Complétez avec un autre test
    ...
}
```

Compilez et exécuter votre programme, pour vous assurer que `testIndiceCaseVide` passe.

Q7. Écrire la fonction `void effectuerDeplacement(int positionMouton, char[] prairie)` qui passe le test ci-dessus, et qui n'a pas besoin de tester que `positionMouton` est la position d'un mouton qui peut être déplacé. Vous devez utiliser la fonction `indiceCaseVide`. Compilez et exécutez votre programme pour vous assurer que les deux tests passent.

Vous allez maintenant écrire la fonction de saisie de coup. Cette fonction va redemander la saisie jusque ce que l'utilisateur propose la position d'un mouton qui peut être déplacé. Nous venons d'identifier une nouvelle sous-tâche: identifier si une position correspond à un mouton qui peut être déplacé dans une prairie donnée. Cette sous-tâche sera implémentée par la fonction `boolean estPositionValide(int positionMouton, char[] prairie)`. Décrivons le comportement désiré de cette fonction par un test.

Q8. Ajouter ce test à votre fichier en le complétant:

```
void testEstPositionValide () {
    char[] prairie;

    // un indice en dehors des indices du tableau n'est jamais valide, quel que
    // soit le contenu de la prairie
    prairie = new char[7];
    assertFalse(estPositionValide(-1, prairie));
    assertFalse(estPositionValide(7, prairie));

    prairie = new char[]{D,D,D,V,G,G,G};
    assertTrue(estPositionValide(2, prairie));
```

```
    assertTrue(estPositionValide(1, prairie));
    assertTrue(estPositionValide(4, prairie));
    assertTrue(estPositionValide(5, prairie));
    assertFalse(estPositionValide(0, prairie));
    assertFalse(estPositionValide(3, prairie));
    assertFalse(estPositionValide(6, prairie));

    prairie = new char[]{D,V,D,G,D,G,G};
    // Complétez les ... par True ou False
    assert...(estPositionValide(0, prairie));
    assert...(estPositionValide(1, prairie));
    assert...(estPositionValide(2, prairie));
    assert...(estPositionValide(3, prairie));
    assert...(estPositionValide(4, prairie));
    assert...(estPositionValide(5, prairie));
    assert...(estPositionValide(6, prairie));
}
```

Puis, implémenter la fonction `estPositionValide` qui passe ce test. La fonction `int indiceCaseVide(char[] prairie)` vous sera sans doute utile.

Q9. Écrire la fonction `int saisirCoup(char[] prairie)` qui demande à l'utilisateur de saisir un entier tant que la saisie n'est pas une position de mouton qui peut être déplacé dans la prairie donnée en paramètre. Vous utiliserez la fonction `boolean estPositionValide(int positionMouton, char[] prairie)`.

Tester la fonction que vous venez d'écrire en modifiant l'algorithme principal comme suit. Vous veillerez à saisir aussi bien des coups valides que des coups invalides, de manière à vous assurer du comportement correct de la fonction.

```
void algorithm() {
  char[] prairie = new char[]{D,D,D,V,G,G,G};
  // pour le test on effectue 3 saisies
  for (int i = 0; i < 3; i++) {
     println(toString(prairie));
     int positionMouton = saisirCoup(prairie);
  }
}
```

Vous pouvez déjà admirer le travail fait, en intégrant à l'algorithme principal les étapes déjà complétées. Si vous le souhaitez, testez votre algorithme partiel comme ceci:

```
void algorithm () {
  char[] prairie = new char[7];
  initialiser(prairie);

  // Pour l'instant on ne peut pas détecter la fin de partie, alors on fera 5 tours de jeu
  for (int i = 1; i <= 5; i++) {
    println("Tour de jeu " + i);
    println(toString(prairie));
    int positionMouton = saisirCoup(prairie);
    effectuerDeplacement(positionMouton, prairie);
    println(toString(prairie));     // pour voir le déplacement qui est fait
  }
}
```

2.3.2 Détection de fin de jeu

Par rapport à notre première conception de la figure 6, il nous faut faire les fonctions `configFinaleAtteinte` et `blocage`.

Q10. De quelles informations a-t-on besoin pour déterminer si la configuration finale de la prairie est atteinte ? En déduire le/les paramètres de la fonction `configFinaleAtteinte`.

Répondre à la même question pour la fonction `blocage`.

Q11. Quel doit être le type de retour des fonctions `configFinaleAtteinte` et `blocage`? Astuce: la réponse se trouve dans la manière d'utiliser ces fonctions dans l'algorithme de la figure 6.

Q12. Écrire les fonctions `configFinaleAtteinte` et `blocage` *sans écrire l'algorithme correspondant*. Pour l'instant, vous les ferez retourner une valeur quelconque arbitraire du bon type.

Attaquons nous maintenant à la fonction `configFinaleAtteinte`, qui est la plus facile des deux. Nous commençons par le test, qui permet de préciser ce que la fonction doit faire.

Q13. Écrire une fonction de test, en complétant ce squelette:

```
void testConfigFinaleAtteinte() {
    // Écrire une assertion qui teste que la configuration finale est atteinte
    // pour la prairie <<<_>>>
    ...

    // Écrire une assertion qui teste que la configuration finale n'est pas atteinte
    // pour la prairie <<_<>>>
    ...
}
```

Compiler pour vous assurer qu'il n'y a pas d'erreurs de compilation. Exécutez. Bien entendu, le test `testConfigFinaleAtte` ne passe pas pour l'instant.

Q14. Complétez le corps de la fonction `configFinaleAtteinte` pour qu'elle passe le test associé. Compilez et exécutez le programme pour vous assurer que le test passe.

Passons maintenant à la fonction `blocage`. On commence par les tests.

Q15. Écrire une fonction de test pour `blocage` en complétant ce squelette

```
void testBlocage () {
    // Écrire une assertion qui teste que cette prairie est un blocage: >><><<_
    ...

    // Écrire une assertion qui teste que cette prairie est un blocage: <<<_>>>
    ...

    // Écrire une assertion qui teste que cette prairie n'est pas un blocage: >>>_<<<
    ...

    // Écrire une assertion qui teste que cette prairie n'est pas un blocage: _<<<>>>
    ...
}
```

Compilez et exécuter (bien entendu, le test ne passe pas).

Remarquons qu'une prairie est dans une situation de blocage si aucun mouton ne peut se déplacer dans la case vide. Si l'indice de la case vide est `idxVide`, alors les candidats pour s'y déplacer sont les moutons aux positions `idxVide-2` (sauter à droite), `idxVide-1` (avancer d'une case vers la droite), `idxVide+1` (avancer d'une case vers la gauche), et `idxVide+2` (sauter vers la gauche). Donc, il y a blocage si aucun de ces quatre moutons (s'ils existent) n'est tourné dans la bonne direction.

Q16. Écrire le corps de la fonction `blocage`, puis exécutez les tests associés. Pensez à utiliser la fonction `indiceCaseVide` déjà écrite.

Revenons à notre ébauche d'algorithme de la figure 6. La seule chose qu'il nous reste à faire est afficher le résultat final. Plus précisément, si la prairie se trouve dans la configuration finale on affichera "Réussite", sinon on affichera "Blocage".

On se rend compte qu'écrire une fonction pour cela n'est pas aisé; il est plus facile d'écrire simplement l'alternative associée.

Q17. Écrire `void algorithm()` qui correspond à l'ébauche de la figure 6 en utilisant les fonctions déjà écrites. Ajouter l'affichage du résultat final.

Vous pouvez maintenant utiliser votre programme pour essayer de résoudre le casse tête saute-mouton!

Q18. Vérifions à présent que votre programme est bien réalisé avec un bon usage des constantes et une bonne paramétrabilité.

- Modifiez les valeurs de vos constantes G,D et V par `'{'`, `'}'` et `'.'`.
- Faîtes en sorte que le joueur puisse saisir la taille de sa prairie au début du jeu.
- Vérifiez que votre programme fonctionne toujours avec le nouvel affichage et une taille saisie de 3 ou de 9.

3 Conclusion

Nous avons déroulé le raisonnement qui nous a permis d'écrire un programme pour jouer à saute-mouton.

À partir de maintenant il vous sera demandé d'écrire des programmes complexes en les décomposant en des tâches plus simples. Ce sera notamment le cas pour votre projet.

Il est très important de tester les fonctions qui implémentent les tâches plus simples. C'est pourquoi vous serez également amenés à **écrire les fonctions de test** associées. En particulier, il peut sembler étrange d'écrire le test avant la fonction correspondante, comme ça a été fait à plusieurs reprises dans ce sujet. Nous conseillons cependant cette approche, qui a des avantages:

— écrire le test permet de préciser le comportement de la fonction. C'est souvent l'occasion de découvrir des difficultés potentielles. Le test confronte la programmeuse à sa compréhension du problème: si on n'arrive pas à définir les tests, il est fort probable qu'on n'arrivera pas à écrire la fonction correctement

— passer du test rouge (quand le corps de la fonction est encore vide) au test vert (quand la fonction est écrite et correcte) rythme l'avancement, donne confiance et augmente la motivation.

L'approche peut paraître complexe, mais comme pour toute chose, c'est en accumulant de l'expérience que vous serez capables de l'appliquer.