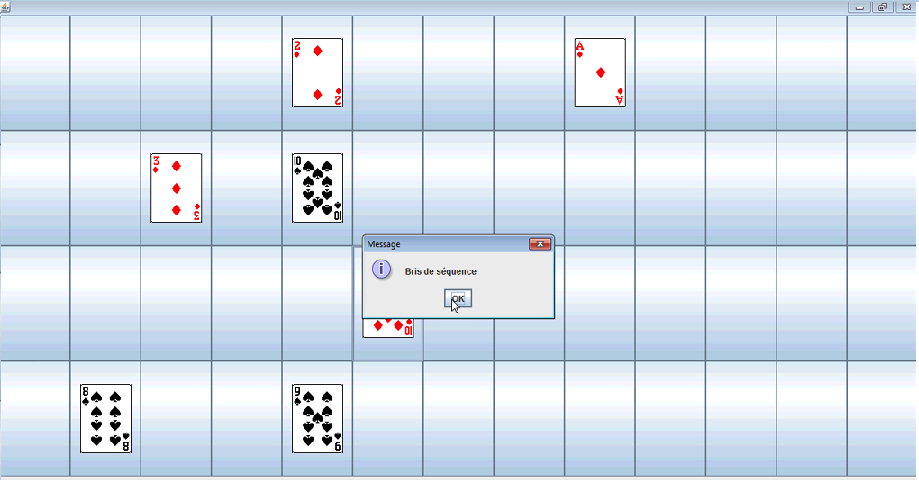
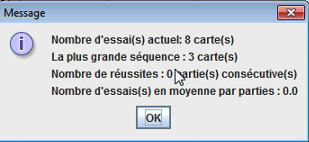
|  |  |
| --- | --- |
|  | **INF-111**  Travail pratique #1  **Groupe : 2 étudiants maximum:** (un seul rapport).  **Remise** : Selon votre groupe dans le plan de cours  **Auteur :** Pierre Bélisle (copyright A2022) |

# 1. Projet :

Nous voulons implémenter un jeu de cartes où l’utilisateur doit sélectionner des séries de cartes de même sorte d’un jeu de cartes dont on ne voit que l’endos. Au départ, les cartes sont montrées et ensuite retournées. L’utilisateur clique sur une carte et commence une séquence. Si la prochaine carte suit la séquence entamée, la carte est tournée. Autrement, il y a un bris dans la séquence. Dans l’image qui suit, l’utilisateur a tourné les 1,2 et 3 de carreau avant de se tromper. Ensuite, il a cliqué sur 8,9 et 10 de pique avant de cliquer sur le 10 de carreau. Ce qui fait un bris de la dernière séquence alors elle est terminée.

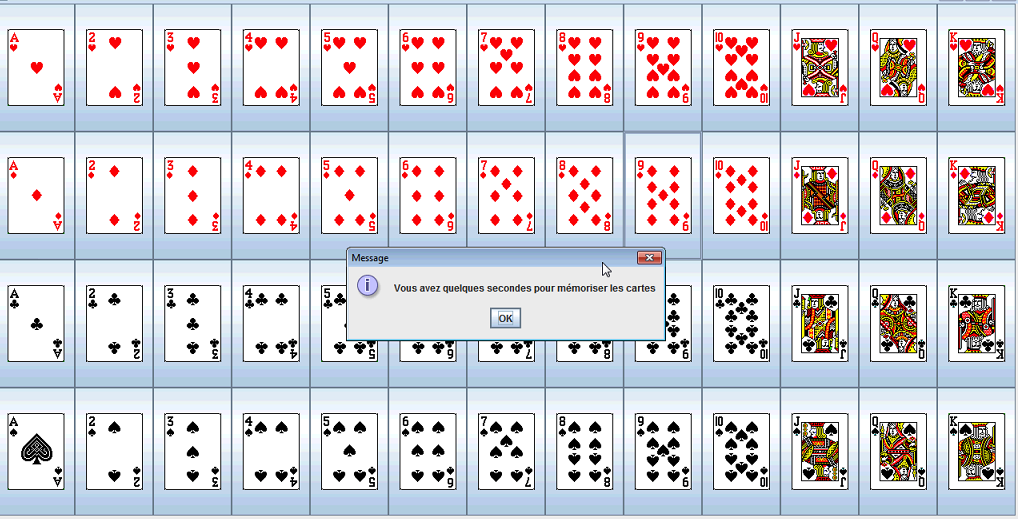


La prochaine carte tournée commencerait une nouvelle séquence. Voici les statistiques du jeu précédent, jusqu’à présent[[1]](#footnote-1) :

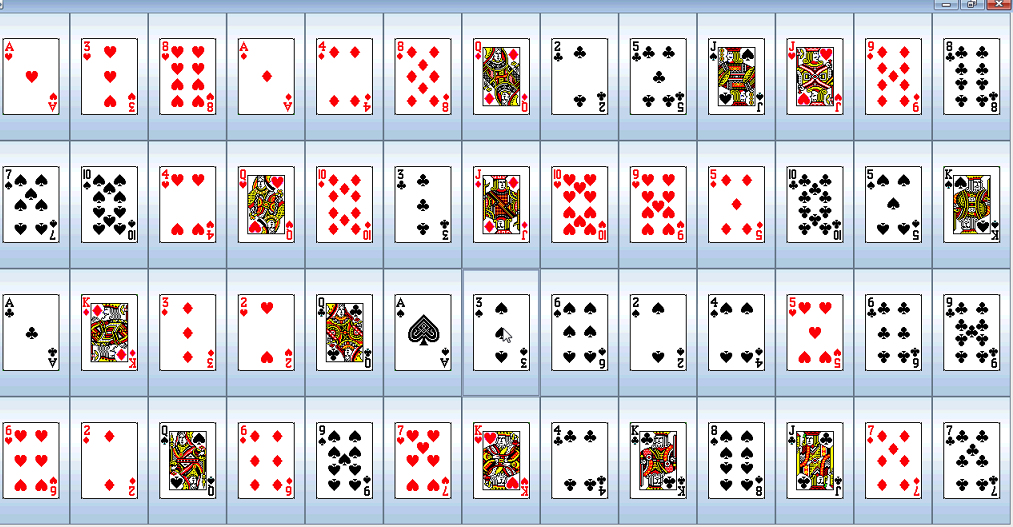


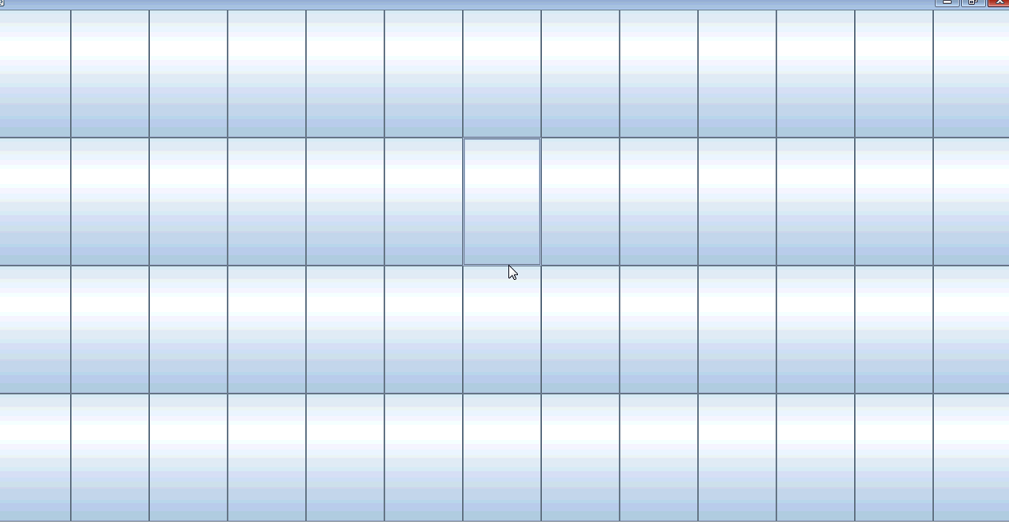
**2. Règles du jeu et déroulement :**

Au départ, l’utilisateur voit un jeu neuf et une boîte de dialogue.



Lorsqu’il clique ok, les cartes sont mélangées, montrées et tournées après quelques secondes.



****

**2.1 Début d’une séquence :**

Les deux premières cartes cliquées sont tournées. Si la deuxième ne suit pas la première, les deux cartes sont retournées. Si la deuxième est la suivante de la première, on passe en mode séquence.

**2.2 Mode séquence :**

Tant que l’utilisateur clique sur des cartes qui se suivent, elles sont tournées et comptabilisées dans le but de retenir quelle a été la plus longue séquence trouvée par le joueur.

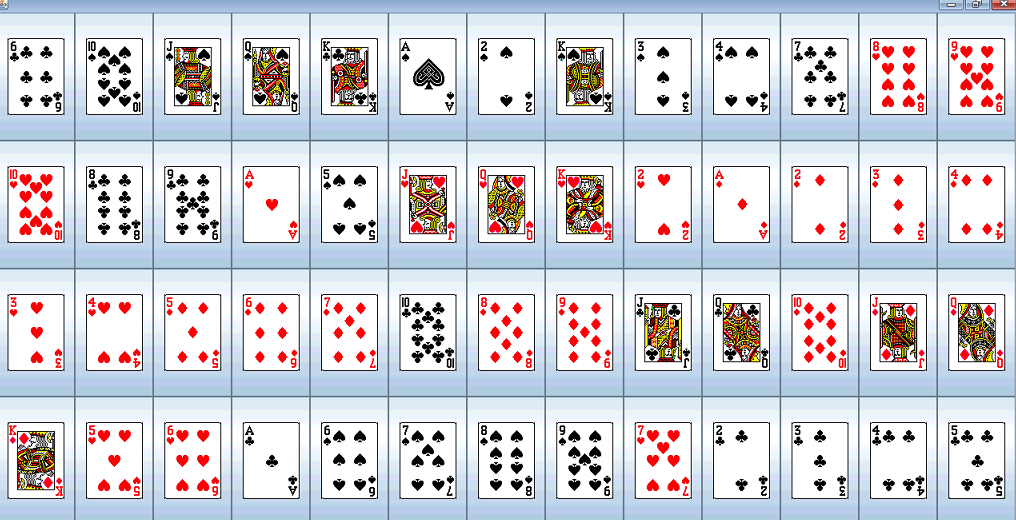
**2.3 Mode erreur :**

Si l’utilisateur brise la séquence, il est avisé par un message pertinent[[2]](#footnote-2). De même que s’il clique sur une carte déjà tournée, il reçoit un avis mais son essai compte quand même dans le nombre d’essais actuel (C’est un genre de pénalité).

La partie se termine lorsque toutes les cartes sont tournées. Il est possible qu’il reste des cartes non tournées mais qu’il ne reste plus de séquence. Vous permettez alors les séquences d’une carte. En d’autres mots, sauf pour ce cas, il faut au moins 2 cartes pour avoir une séquence.

# 3. Problème.

Un des problèmes est qu’il faut disposer les cartes. On veut éviter, autant que possible, d’avoir des cartes consécutives de même sorte qui se suivent sur une même ligne. Cela nous amène à se demander comment nous conservons les cartes et comment on les distribue. Voici un exemple de ce que nous **ne** voulons **pas** obtenir :



**4. Mandat :**

Il s’agit d’implémenter et de tester différentes façons de brasser des cartes dans un environnement numérique et d’utiliser la meilleure distribution. Nous vous en imposons 3 qui vous sont décrites plus loin. Il est possible qu’à la fin, des techniques soit équivalentes en distribution, si c’est le cas, choisissez celle qui prend le moins de temps.

En utilisant la technique retenue, il faut implémenter le jeu de mémoire de séquence de cartes selon les spécifications qui seront fournies plus tard.

**5. Conception imposée :**

Voici quelques informations sur les types que vous devez utiliser dans votre solution et qui vous sont déjà fournis. Consultez-les bien attentivement en Java, mais en voici une brève description :

**5.1 Coordonnee.java** :

Conserve une coordonnée ligne-colonne. Elle est utile pour l’interaction avec le GUI.

**5.2 Sorte** :

C’est un type énumératif (enum) définit dans un module de constantes (Constantes.java) qui vous est également fourni. Il énumère les quatre sortes (CŒUR, CARREAU, TREFLE et PIQUE). Pour voir une variable de ce type il suffit d’écrire : Constantes.Sorte sorte; Pour obtenir les constantes valeurs constantes, on utilise sort.values().

**5.3 Carte.java** :

Conserve un entier pour le numéro et une sorte (ex : AS PIQUE).

Il y a aussi une image de type javax.swing.Icon utile pour l’affichage. Vous devez obtenir ces images qui proviennent de fichiers (nous y reviendrons).

Finalement, un champ booléen pour l’état de la carte (visible ou non).

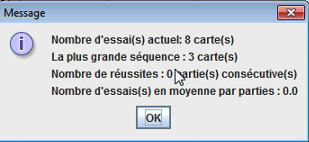
**5.4 EtatJeu.java :**

Nous voulons pouvoir passer plusieurs informations en paramètre concernant l’état du jeu en tout temps. Voici les champs retenus :

* + - Un champ booléen mis à vrai si la partie est terminée.
    - Un champ booléen qui permet de retenir s'il y a une séquence de cartes en cours par l’utilisateur (plus de 2 cartes).
    - Un tableau qui retient toutes les cartes de la séquence actuelle et sa longueur.
    - Nous offrirons la possibilité au joueur d’avoir des indices pendant le jeu. Nous retenons ici le nombre d’indices qu’il reste à l’utilisateur. Nous y reviendrons également.

**5.5 Stats.java :**

Pour les statistiques, nous ne conservons que celles dans la boîte présentées à la page **2** reproduites ici. Nous sommes conscients qu’on pourrait en ajouter mais pour ce travail, elles nous suffisent.



# 6. Stratégie :

**6.1** Vous n’avez pas vu les notions nécessaires pour écrire un programme dans un environnement graphique. C’est pourquoi nous vous fournissons un module de type **GrilleGui[[3]](#footnote-3)** qui le permet. Les seules utilisations sont pour afficher une image ou obtenir la position d’un clic de souris qui peut être sur une case ou sur un bouton du menu.

**6.2** Nous avons décidé d’utiliser unevariable-tableau à une dimension dans le programme principal qui compte 52 références du type **Carte**. Ces cartes sont transmises aux différents sous-programmes des différents modules. Vous avez donc :

// Le paquet de jeu neuf est créé dans un module spécifique.

Carte[] jeuNeuf = **UtilitaireSysteme**.*obtenirJeuCartesNeuf*();

Carte[] carteAffichee=**UtilitaireTableauCartes**.*copieDuJeu*(jeuNeuf)[[4]](#footnote-4);

GrilleGui gui – new GrilleGui(nbLignes, nbColonnes);

Le tableau de cartes est en 1D mais le GUI est en 2D.

# 7. Développement et description des modules à écrire :

L’ordre de présentation des modules et des sous-programmes tient compte de l’ordre de développement que nous vous suggérons. Même s’il est possible de développer différemment et que cela fonctionne (il n’y a pas qu’une façon), nous vous le déconseillons. Il est plus facile de vous aider si on sait où vous êtes rendu. Il est donc important de travailler assidûment à chaque semaine et de suivre l’ordre proposé.

Avec cette approche (descendante), vous n’avez pas besoin d’attendre de tout savoir ce que vous avez à produire pour commencer. Sachez que vous n’avez pas moins **25 sous-programmes** à écrire.

7.1 Développement :

Tous modules et les sous-programmes dont il est question sont décrits en détails plus loin.

* Créer un projet et y insérer les fichiers fournis. Prenez-en bien connaissance et corrigez les erreurs s’il y en a.
* Démontrer votre compréhension de la conception imposée (page 5). Vous devez montrer un diagramme des structures en séance de laboratoire avant de poursuivre.
* Consulter la fonction alea du module **UtilitaireFonction.**
* Écrire la procédure d’affichage des cartes dans le module **UtilitaireGrilleGui** jusqu’à ce que les cartes s’affichent.
* Écrire une première technique de brassage dans le module **UtilitaireTableauCartes** et l’appeler dans la procédure qui initialise le jeu. Vous devriez maintenant voir un jeu neuf suivi des cartes une fois brassée, en démarrant l’application.
* Compléter la méthode qui initialise le jeu.
* Écrire une deuxième technique pour brasser dans le même module et l’appeler dans la procédure qui initialise le jeu à la place de la première (mettre la première en commentaire). Testez votre méthode.
* Écrire la troisième méthode et la tester. À ce moment, nous vous fournirons une fonction pour permettre à l’utilisateur de sélectionner la méthode pour brasser les cartes et vous pourrez implémenter le jeu en écrivant la procédure efffectuerUnTour.

# 7.2 Modules à écrire :

# 7.2.1 UtilitaireGrilleGUI.java :

Il n’y a qu’un sous-programme[[5]](#footnote-5) à écrire pour afficher les cartes dans le GUI. Il n’y a pas qu’une façon mais ce que vous devez savoir c’est que vous devez utiliser gui.setImage(…) pour qu’une image apparaisse à l’écran.

Si l’image transmise est à **null**, on voit le dos de la carte.

**Test** : Aussitôt écrit, vous devez voir un jeu de cartes neuf en démarrant l’application.

7.2.2 UtilitaireStats.java :

Il n’y a qu’un sous-programme à écrire pour afficher les statistiques. Utilisez JoptionPane.showMessageDialog et la concaténation des chaînes de caractères (+) pour montrer les statistiques.

\*\*\*Test : Aussitôt votre procédure écrite, vous devez voir la boîte de message en cliquant sur le bouton de menu approprié.

# 7.2.3 UtilitaireTableauCartes.java:

Dans ce module, on retrouve tous les sous-programmes qui concernent les tableaux de cartes :

* Les 3 méthodes pour brasser et leurs sous-programmes privés.
* Les procédures utilitaires pour le jeu qui : copie un jeu, décale les cartes, …
* Les fonctions booléennes qui retournent : si deux cartes se suivent et si toutes les cartes sont tournées.
* Une procédure pour modifier la visibilité de toutes les cartes (true ou false).

**7.2.3.1 Méthode par brassage :**

Il nous est probablement tous arrivé de brasser un jeu de cartes neuf. La méthode la plus connue se divise en trois étapes qui sont répétées un certain nombre de fois. On commence par mélanger les cartes, ensuite on sépare le paquet puis on les fusionne pour redonner un seul paquet.

**Algorithme pour brasser**

**Début**

Générer un nombre aléatoire de fois que vous répéterez les trois étapes.

Boucler le nombre de fois qu’il faut

Mélanger le paquet de cartes à afficher.

Séparer le paquet de cartes à afficher dans deux autres tableaux de cartes à peu près égal (entre 20 et 30 cartes pour un paquet et le reste dans l’autre).

Fusionner les deux paquets dans le jeu de cartes à afficher.

Fin de la boucle

**Fin**

\*\*\* Il est préférable de définir les deux tableaux dynamiquement dans la boucle pour obtenir des tailles différentes à chaque tour.

7.2.3.2 Mélanger



**Algorithme pour mélanger**

Stratégie : On prend un peu de cartes de la fin et on les met au début.

**Début**

Générer un nombre aléatoire de fois que vous déplacer des cartes.

Pour chaque fois

Générer un nombre de cartes à déplacer (entre 3 et 10 par exemple).

Déplacer les cartes de la fin vers le début.

Fin de la boucle.

**Fin**

\*\*\*Nous sommes conscient que cet algorithme n’est pas parfait mais il convient aux objectifs que nous voulons atteindre. Ne perdez pas de temps à l’améliorer, à moins que vous ayez fini le reste et que cela vous tente.

# 7.2.3.3 Séparer



**Algorithme pour séparer**

**Début**

Utiliser un sous-programme utilitaire pour copier une partie du jeu de cartes à afficher dans un tableau et le reste dans un autre.

**Fin**

7.2.3.4 Fusionner



**Algorithme pour fusionner**

Habituellement, on prend un peu de cartes d’un paquet qu’on laisse tomber et on prend un peu de cartes de l’autre qu’on laisse tomber par-dessus et cela en alternance jusqu’à ce qu’il n’y ait qu’un paquet. Ici, on travaille avec des tableaux mais c’est le même principe. Il faut juste gérer l’alternance que l’on fera avec une variable qu’on incrémente % 2. Si c’est pair, on prend dans un paquet, sinon on prend dans l’autre.

**Début**

Il faut 3 itérateurs. Un par tableau. Tous à 0 au départ.

Initialiser une variable à 0 ou à 1 au hasard pour gérer l’alternance.

Tant qu’il reste des cartes à fusionner dans l’un ou l’autre des tableaux[[6]](#footnote-6)

Si l’alternance est paire et que l’itérateur du 1ier tableau est plus petit que taille du 1ier tableau

Générer le nombre de cartes du 1ier tableau à mettre dans le tableau de cartes à afficher. Entre 1 et la taille du 1ier tableau.

Copier les cartes du 1ier tableau vers le tableau de cartes à afficher en avançant les itérateur du nombre de cartes qui ont été déplacées.

Sinon (alternance impaire) si l’itérateur du 2ième tableau est plus petit que la taille du 2ième tableau

Générer le nombre de cartes du 2ier tableau à mettre dans le tableau de cartes à afficher. Entre 1 et la taille du 2ier tableau.

Copier les cartes du 2ier tableau vers le tableau de cartes à afficher en avançant les itérateur du nombre de cartes qui ont été déplacées.

Fin si

Incrémenter la variable pour gérer l’alternance.

Fin de la boucle

**Fin**

7.2.4 Méthode par paquets :

Il s’agit de subdiviser le jeu de cartes en plusieurs paquets (entre 6 et 8). Lorsque les paquets sont faits, il s’agit d’en prendre un et de distribuer ses cartes une à une sur les autres paquets. On recommence jusqu’à ce qu’il ne reste qu’un paquet.

Il faut un tableau de jeu de cartes. C’est donc un tableau 2D (un tableau de tableaux de C**arte)** où une case représente un paquet.

Voici une suggestion qui vous permettra de réutiliser le code. Déclarez le tableau avec une case de plus que le nombre de paquets. Mettez le jeu à distribuer dans la dernière case (ne compte pas dans le nombre de paquets). Au départ, c’est le jeu au complet.

Vous distribuer le paquet de la dernière case sur les autres. Lorsque le paquet est distribué au complet, on choisit au hasard le prochain paquet qu’on place à la dernière case et on recommence.

**Algorithme en pseudocode**

Début

Générer le nombre de paquets au hasard

Déclarer le tableau de jeu de cartes (tableau 2D).

Mettre le jeu de carte à distribuer dans la dernière case.

Tant qu’il reste plus d’un paquet,

Distribuer le dernier paquet sur les autres.

Choisir un paquet au hasard (case non nulle) et le mettre dans la dernière case (nbPaquet + 1).

Mettre la case du paquet choisi à **null**.

Fin de la boucle

Il reste à mettre à jour le jeu de cartes à afficher avec le dernier paquet (nbPaquet + 1).

7.2.5 Méthode par position aléatoire :

On simule de prendre une carte après l’autre et de les placer au hasard sur la table à sa position finale. Évidemment, on ne fera pas par exprès pour placer les cartes une à côté de l’autre.

Numériquement, cela revient à copier les cartes du jeu à afficher, une à une, dans un autre tableau de cartes à une position aléatoire qui n’est pas déjà occupée (non **null**). La fonction pseudo-aléatoire de Java distribue les valeurs normalement, vous avez peu de chance que les valeurs soient fortement consécutives. Il n’y a donc rien de spécial à écrire pour régler cela.

**Algorithme en pseudocode**

**Début**

Déclarer le tableau de cartes temporaire.

Pour chaque carte du jeu à afficher,

Obtenir une position de case qui n’est pas déjà utilisée dans le tableau temporaire.

Mettre la carte actuelle dans le tableau temporaire à la position.

Fin de la boucle.

Il reste à copier le tableau temporaire dans le tableau de cartes à afficher.

**Fin**

# 7.3 UtilitaireJeu.java :

Dans ce module, on retrouve tous les sous-programmes qui concernent les règles du jeu et la gestion de l’affichage des cartes lorsqu’elles sont cliquées. On ne vous décrit que le principal. À vous de découper en plus petit sous-programmes lorsque cela s’applique.

# 7.3.1 Initialiser le jeu

Il s’agit d’initialiser quelques champs de l’enregistrement qui retient l’état du jeu et d’appeler des sous-programmes.

**Debut**

Initialiser la longueur de séquence à 0 et le nombre d’indices à Constantes.*NB\_INDICE\_DEPART* de l’enregistrement qui retient l’état du jeu.

Brasser les cartes avec la méthode pour brasser que vous êtes rendu à tester.

Afficher les cartes du jeu neuf.

Afficher un message avec **JOptionPane** pour laisser l’utilisateur décider quand il est prêt (voir page 2).

Modifier la visibilité des cartes du jeu de cartes à afficher (pour les cacher). C’est important dans notre algorithme.

Copier le jeu de cartes à afficher dans un tableau de cartes temporaire et afficher ce jeu.

Faire une pause pour laisser le temps de voir (Constantes.***TEMPS\_VISIONNEMENT)***.

Modifier la visibilité du tableau temporaire à faux et l’afficher.

Réafficher le jeu de cartes à afficher original.

**Fin**

# 7.3.2 Effectuer un tour

Pour obtenir cette partie, vos trois techniques pour brasser les cartes doivent fonctionner, preuves à l’appui. Vous devez montrer votre code en séance de laboratoire. S’il est accepté, vous obtiendrez la partie qui décrit le jeu.

# 8. Modules utilitaires et main fournis :

Nous vous fournissons un autre module **UtilitaireSysteme** pour obtenir un jeu de cartes neuf qui contient les images en provenance des fichiers, que nous vous fournissons dans le répertoire « images/ ».

Nous vous fournissons aussi le programme principal et le module **UtilitaireMenu** qui gère les choix d’options de menu de l’utilisateur. Vous devez écrire les sous‑programmes qui y sont appelés. Évidemment, ces sous‑programmes sont à subdiviser en sous-programmes à leur tour tels que décrits. Vous en avez donc à ajouter pour optimiser le code et éviter la répétition inutile.

# 9. Normes de programmation et bonnes pratiques :

Votre programme doit respecter les contraintes suivantes les normes de programmation présentées au cours. D’ailleurs, la qualité du code vaut plus de points que le fonctionnement du programme.

Voici une liste non exhaustive des mauvaises pratiques pénalisantes :

1. Nom de fonction ou de procédure (nom + verbe) qui ne respectent pas la norme.
2. Identificateurs non significatifs (variables, constantes, sous-programmes, …).
3. Aération et/ou indentation et/ou impression laissent à désirer (80 colonnes max).
4. **Code qui devrait être dans un sous-programme. Découpage en sous programmes insuffisants.**
5. Code inutile; aucun impact.
6. Répétition inutile de code dû au manque de sous programmes.
7. Répétition inutile de code dû à l'incompréhension de l'utilité du paramétrage.
8. Constantes non définies.
9. Constantes non utilisées lorsque possible (même dans les commentaires).
10. Constantes en minuscules
11. Commentaires non judicieux ou inutile.
12. Commentaires des variables manquants.
13. Commentaires des constantes manquants.
14. **Commentaires de spécification des sous-programmes manquants (quoi).**
15. **Commentaire d'en-tête de programme manquant (explication, auteurs et version).**
16. **Commentaires manquants sur la stratégie employée dans chaque sous-programme dont l'algorithme n'est pas évident et nécessite réflexion (comment).**
17. Commentaires mal disposés.
18. Non utilisation d'un sous-programme lorsque c'est possible ou exigé.
19. Non utilisation d'une boucle lorsque c'est possible.
20. Sous-programme qui fait plus d'une tâche.
21. Affichage dans une fonction de calcul.
22. Qualité du français dans les commentaires.
23. Non-respect des droits d'auteurs.
24. Non utilisation de boucle lorsque possible.

**Pondération** :

**Qualité** : 60%

**Exécution** 40%

* Méthodes de brassage 25%
* Jeu 15%

**La remise se fera par Moodle.**  Il n’y aura aucune remise acceptée par courriel, donc **aucun retard ne sera accepté même d’une seconde.**

Bon travail!

1. L’affichage de la boîte a été effectué à l’aide de JOptionPane.showMessageDialog. [↑](#footnote-ref-1)
2. JOptionPane.showMessageDialog [↑](#footnote-ref-2)
3. Vous avez très peu à savoir sur ce module. Vous n’avez pas à lire tout le code. [↑](#footnote-ref-3)
4. Vous devez l’écrire et c’est une vraie copie des données, pas juste les références. [↑](#footnote-ref-4)
5. C’est un petit sous-programme (1 ou 2 boucles et un if-else). [↑](#footnote-ref-5)
6. Cela veut dire que l’itérateur est plus petit que la taille de son tableau. [↑](#footnote-ref-6)