Travail Pratique 1

6 février 2025

Préparé par
François Gagnon, Pierre Poulin et Karine Filiatreault

1 Résumé

Pour ce travail pratique, nous allons concevoir une petite application de Bingo simplifiée avec quelques règles particulières.

2 Conditions de réalisation

Valeur de la note finale	Type	Durée
15 %	Individuel	1 semaine ½

3 Précisions pour tous les programmes

1 Les programmes ne doivent jamais « planter »
2 On doit toujours redemander une valeur à l'utilisateur jusqu'à ce qu'elle soit correcte.
3 Un projet qui ne compile pas se verra attribuer la note zéro « 0 ».

4 Spécifications

Cette section vise à vous guider durant la réalisation de ce travail. Réalisez chaque étape dans l'ordre et passez à l'étape suivante uniquement lorsque vous avez terminé l'étape actuelle.

4.1 Création du projet

Utilisez le projet de départ fourni.

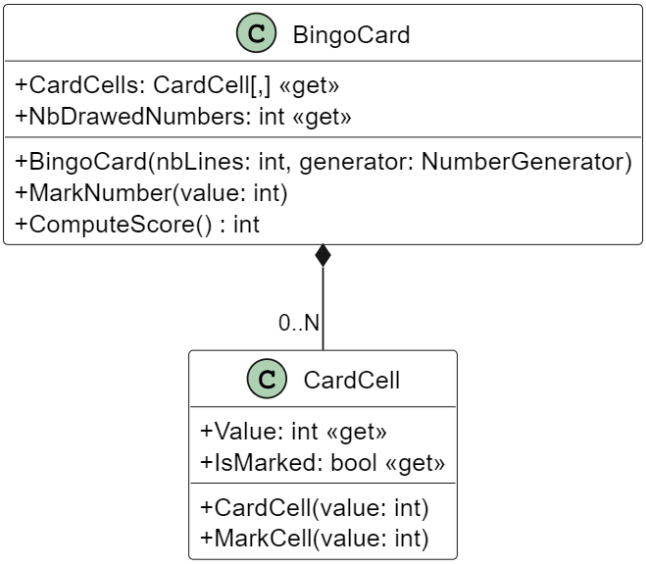
4.2 Intégration du générateur de nombres aléatoires dans le projet

Les fichiers `NumberGenerator.cs` et `NumberGeneratorTest.cs` sont fournis et ne doivent pas être modifiés.

4.3 Classes du projet

Vous aurez deux classes à produire (en plus du Main) pour ce travail en plus des classes de tests pertinentes.

Voir le diagramme de classes à la page suivante.



La classe `CardCell`

La classe `CardCell` est simple. Chaque case de Bingo contient un nombre (la valeur de la case) ainsi qu'un état pour savoir si la case est marquée ou pas (i.e. si son numéro a été pigé ou non).

Il est possible de marquer une case (méthode `MarkCell(int)`). Cela a pour effet de changer son état si le nombre associé à la case est le même que celui reçu en paramètre.

Il est également possible de demander l'état à la case (*Property* `IsMarked : bool`).

Finalement il est possible de demander la valeur de la case (*Property* `Value : int`).

Vous devez réaliser cette classe au complet. Contrairement au vrai Bingo, les valeurs tirées seront seulement des numéros (e.g., 12) et non une combinaison lettre numéro (e.g., B12). Il n'y a aucune restriction sur la position des numéros sur notre carte. Pour les besoins de ce travail, un numéro peut se trouver n'importe où sur la carte contrairement au Bingo usuel qui impose des restrictions par colonne.

La classe `BingoCard`

La classe `BingoCard` est plus compliquée. Une carte de Bingo aura une dimension variable mais toujours carrée.

La carte est composée d'un tableau à 2 dimensions contenant des `CardCell`. On voudra aussi que la carte conserve le nombre de numéros qui ont été pigés depuis le début (même si certains ne se trouvaient pas sur la carte courante).

Le constructeur de la classe `BingoCard` prend deux arguments (1. le nombre de lignes/colonnes ainsi que 2. une instance de la classe `NumberGenerator`). Le constructeur doit remplir la carte avec les nombres du générateur ligne par ligne. Par exemple, pour un générateur en mode séquentiel la carte construite serait :

```
 1  2  3  4  5  6
 7  8  9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
31 32 33 34 35 36
```

La classe **BingoCard** contient deux autres méthodes publiques importantes :

`void MarkNumber(int value)`

Cette méthode demande à la carte de marquer le numéro **value** sur la carte (s'il est présent). Ce numéro vient d'être « pigé », on veut donc le « cocher » sur notre carte.

`int ComputeScore()`

Cette méthode calcule le pointage de la carte dans son état actuel. Pour avoir un pointage différent de 0, une carte doit avoir au moins une rangée complète OU une colonne complète de cases marquées (cases dont le numéro a été pigé). On ne s'intéresse pas aux diagonales.

Vous devez donc produire le code nécessaire pour effectuer ce calcul selon les règles suivantes :

- Si aucune rangée ni colonne n'est complète, le pointage est 0.
- Si une (ou plusieurs) rangée(s) ou colonne(s) est (sont) complète(s), le pointage est la somme des cases sur toutes les lignes et colonnes qui sont complétées.

  La méthode **ComputeScore()** est définitivement à découper en sous-méthodes pour éviter d'avoir du code long difficile à lire et à comprendre.

Voici quelques exemples (3+ indique que le 3 est marqué sur la carte):

```
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
 16  17  18  19  20
 21  22  23  24  25

Draw: 3, 8, 13, 18, 23,

  1   2  3+   4   5
  6   7  8+   9  10
 11  12 13+  14  15
 16  17 18+  19  20
 21  22 23+  24  25

Score is: 65
```

```
  5   5   5   5   5
  5   5   5   5   5
  5   5   5   5   5
  5   5   5   5   5
  5   5   5   5   5

Draw: 3, 8, 13, 18, 5,

 5+  5+  5+  5+  5+
 5+  5+  5+  5+  5+
 5+  5+  5+  5+  5+
 5+  5+  5+  5+  5+
 5+  5+  5+  5+  5+

Score is: 250
```

```
 29 658 978 950 322 503
765 554 488 595 266 404
385 481  40 585 869 221
816 919 632 982 739 691
395 122 628 642 735 860
888 329 261 368 464 609

Draw: 29, 40, 13, 585, 869, 40

 29+658 978 950 322 503
765 554 488 595 266 404+
385 481  40+585+869+221
816 919 632 982 739 691
395 122 628 642 735 860
888+329 261 368 464 609

Score is: 0
```

Note :

Vous n'avez pas à reproduire cet affichage. Il est donné à titre indicatif seulement.

La méthode **Main**

La méthode **Main** doit instancier 5 cartes de Bingo qui seront initialisées à l'aide d'un générateur de nombres pseudo-aléatoires différent. Vous trouverez en annexe les informations pertinentes pour le générateur de nombres fourni.

Les 5 cartes doivent être placées dans une collection (tableau, liste, etc.) de **BingoCard** et doivent utiliser un générateur de nombre pseudo-aléatoire initialisé avec les valeurs 0 à 4 inclusivement. Cela permettra d'avoir des cartes différentes tout en nous assurant que tout le monde utilisera les 5 mêmes cartes dans leur programme.

Vous devez ensuite instancier un autre générateur de nombres qui servira comme « boulier » (pour le tirage). Utilisez la valeur 8473718269 pour l'initialisation.

Finalement, nous allons jouer au Bingo en tirant des nombres à l'aide de ce dernier générateur. Tirez un nombre à la fois et marquez-le dans chacune des 5 cartes dont vous disposez. Répétez cela jusqu'à ce que l'une des cartes ait un pointage supérieur à 0.

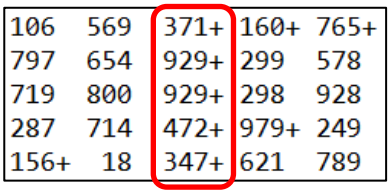
Affichez alors

- le numéro (de 0 à 4) de la carte gagnante
- le nombre de tirages qui a été nécessaire
- le pointage de la carte gagnante

Si deux cartes sont gagnantes au même tour, affichez les informations de celle dont le pointage est le plus grand.

Par exemple, si on prenait les 5 cartes initialisées tel que décrit plus haut, mais qu'on utilisait un générateur pseudo-aléatoire avec 1337 (au lieu de 8473718269) pour le tirage, le résultat serait :

- Carte gagnante : 1 (celle dont le générateur a été initialisé avec 1)
- Nombre de tirages : 394
- Pointage : 3048



Le **`Main`** est le seul endroit dans le code où vous pouvez faire de l'affichage à la console.

5 Tests unitaires

Vous devez produire les tests unitaires pour les classes **`BingoCard`** et **`CardCell`**.

Pour la correction des tests unitaires, vous serez évalués.ées de deux manières :

- Écriture de vos tests, notamment le niveau de couverture du code
- Exécution des tests du professeur.

Évidemment, pour que les tests du professeur puissent fonctionner, il faut que les signatures de méthodes soient similaires. Pour s'en assurer, deux fichiers vous sont fournis. Ils ne contiennent aucun test mais ont tous deux une méthode privée fournie qui fait des appels de méthodes. Vous devez vous assurer que cette méthode compile. Elle ne sert à rien d'autre.

Tests à effectuer

Pour la classe **`CardCell`**, vous devez tester les méthodes suivantes :

- Le constructeur
- **`MarkCell`**

Vous pouvez assumer que les *Properties* **`IsMarked`** et **`Value`** fonctionnent correctement.

Pour la classe **`BingoCard`**, vous devez tester les méthodes suivantes :

- **`ComputeScore`**

Assurez-vous de tester avec des cartes de grosseurs différentes ainsi qu'avec des générateurs aléatoires différents mais pour lesquels vous êtes en mesure de prédire le résultat attendu.

6 Évaluation

Vous trouverez dans le fichier ci-joint la grille d'évaluation utilisée pour la correction de ce travail pratique.

7 Modalités de remise

Remettez votre projet C# sur LÉA, dans la section travaux, à l'intérieur d'une archive *Zip*. Supprimez tous les dossiers temporaires

Annexe 1

Génération de nombres pseudo-aléatoires

La génération de nombres aléatoires ou plutôt pseudo-aléatoires sur un ordinateur vise à produire des nombres ayant l'apparence d'être le fruit du hasard. En fait, un générateur pseudo-aléatoire est un algorithme, donc une suite finie et non ambiguë d'opérations ; **à partir d'un nombre de départ donné, il générera toujours la même suite**. Utiliser des algorithmes pour créer des nombres aléatoires peut donc paraître paradoxal[1].

La génération de nombres pseudo-aléatoires est essentielle dans le cadre de ce travail et est réalisée par la classe `NumberGenerator`.

Cette classe est notamment utilisée pour créer (initialiser) une carte de Bingo, mais aussi pour simuler le tirage des numéros par la suite.

La classe `NumberGenerator` comporte trois modes de génération :

- Constant : génère toujours le même nombre. Pour utiliser ce mode, construire un objet avec le constructeur `NumberGenerator(int)` et passer en paramètre le nombre à générer.
- Séquentiel : génère une séquence incrémentale de nombres débutant à 1 (1, 2, 3, 4, 5, …). Pour utiliser ce mode, utilisez le constructeur `NumberGenerator()`.
- Pseudo-aléatoire : génère une séquence de nombres pseudo-aléatoires. Pour utiliser ce mode, construire un objet le constructeur `NumberGenerator(long)` et passer en paramètre le nombre de départ à utiliser pour générer la séquence. Pour deux nombres identiques, la même séquence sera générée. Par conséquent, même si nous sommes en pseudo-aléatoire, nous restons déterministes ce qui aide beaucoup pour les tests unitaires.

[1] Source : `https://fr.wikipedia.org/wiki/G%C3%A9n%C3%A9rateur_de_nombres_pseudo-al%C3%A9atoires`