Projet Jeu de Cartes – R2.02 Développement d'applications avec IHM

Préambule légal : ce projet est basé sur un jeu de cartes commercialisé en France sous le nom de 6 qui prend !, *adapté de* 6 nimmt ! *par Wolfgang Kramer, chez Amigo Spiel + Freizeit GmbH, adapté et distribué pour la France par Gigamic. Tous droits réservés. La protection intellectuelle et le droit d'auteur en Allemagne, France, Europe et à l'international doivent être respectés. Ce projet est à but éducatif et pédagogique uniquement, il est en particulier interdit de le distribuer ou de distribuer le code produit hors contrat avec l'auteur et les éditeurs. Voyez* `https://fr.wikipedia.org/wiki/6_qui_prend_!`.

Résumé du projet R2.02

Le but est de développer une application permettant de jouer au jeu en question.

Le projet se fait en **binôme** ou **solo** (pas d'équipe de plus de deux), avec une base de code fournie. Il se déroule sur **cinq séances** au total (il ne devrait pas être nécessaire de travailler en-dehors des séances). L'organisation est la suivante :

1. Un *tutoriel* de **au moins une séance et au plus deux** qui permettent à chacun d'entre vous de vous approprier le projet et le code fourni à l'aide d'instructions précises et de code (généralement donné) à insérer.
   Livrables du tutoriel : un fichier contenant les réponses aux questions posées **et** l'archive du projet au niveau du tutoriel. **Instructions** : travaillez en parallèle individuellement sur le tutoriel. L'aide entre coéquipiers d'un même binôme est encouragée, mais nous exigeons un rendu par personne.
2. Une phase de *conception* (*design*, environ 30 minutes), dans laquelle vous préparerez ce que vous pensez faire dans votre interface utilisateur **avant** de coder. Ce sera fait par discussion dans chaque binôme.
   Livrable de conception : une maquette (croquis) de l'interface de votre fenêtre principale, au choix sur papier ou PDF. Un par binôme.
3. Une phase de *réalisation du projet* (trois à quatre sessions), dans laquelle chaque binôme code **leur propre** conception, l'essaient, l'améliorent autant que souhaité.
   Livrable de réalisation du projet : l'archive du projet final. Une par binôme (avec les noms des auteurs).

La date limite de rendu pour tout le projet est 9h du soir (21h) le vendredi 15 avril 2022.

Le sujet

Version courte

Nous avons développé une version **minimaliste** de *6 nimmt !* qui forme la base du projet. Le jeu est un classique des jeux de cartes d'ambiance avec des règles simples et quelques tactiques ; notre version est à peu près fonctionnelle (quelques règles ont été légèrement modifiées), mais elle n'est fournie qu'en tant que **jeu interactif en mode texte** (et en anglais), ce qui est rédhibitoire. Votre but est de donner une interface graphique utilisateur fonctionnelle, agréable, si possible ergonomique et esthétique pour ce code, **pas** de coder le jeu (cette partie est faite ; vous n'êtes pas responsables des erreurs ou problèmes de performance).

L'organisation du travail

Il y a donc trois parties au projet :

1. Le tutoriel : en une ou deux séances, nous vous donnons des instructions détaillées, des questions à répondre dans un fichier texte (obligatoire) ; cette partie vous apprendra à vous repérer dans le code donné.
2. Le maquettage.

3. Le projet : dans les trois dernières séances (au moins), vous concevrez (donnez la maquette d'abord) puis fournirez les fonctionnalités de l'interface graphique suivant **votre** maquette. Chaque rendu final sera différent, suivant les choix de chaque binôme.

À la fin du *tutoriel*, vous rendrez votre code et les réponses aux questions ; cela sera noté (6 sur 20 dans la note finale).

Ensuite, vous rendrez votre maquette, qui sera utilisée pour noter le projet final.

À la fin du projet, vous rendrez le code final, qui sera donc aussi noté avec la maquette (14 sur 20 dans la note finale).

Attention ! Ce projet est une épreuve évaluée ; tout plagiat de code est un délit qui sera sévèrement réprimé.

Ce projet est donc prévu pour des **binômes** (des étudiants peuvent le réaliser **seuls** après validation par l'enseignant de TD).

Le jeu

6 nimmt ! ou *6 qui prend !* est, normalement, un jeu de cartes d'ambiances pour deux à dix joueurs.

Il consiste en un seul jeu de 104 cartes, toutes avec un nombre (de 1 à 104, pas de cartes identiques). Chaque carte à aussi une **valeur de malus** (pénalité, de 1 à 7). Le but du jeu est d'avoir le **plus petit malus** (le moins de points possible) à la fin.

Au début d'une partie, chaque joueur reçoit dix cartes tirées au sort. Quatre cartes sont placées sur la table (face visible), créant le début de quatre lignes.

À chaque tour, chacun des joueurs choisit une carte, qu'ils placent devant eux face cachée.

À la fin du tour, toutes ces cartes (une par joueur) sont retournées et ordonnées par ordre numérique croissant (plus petite carte en premier, plus grande en dernier) ; chacune des cartes dans cet ordre est placée à la fin d'une des quatre lignes de la table suivant ces trois règles :

— la nouvelle carte est toujours **plus haute** numériquement (1 à 104) que la précédente dans la ligne (chaque ligne est donc toujours triée par ordre numérique croissant),
— la nouvelle carte est obligatoirement placée **après la carte la plus proche** (la différence de valeurs est la plus petite),
— chaque ligne ne contient que **cinq cartes au plus**.

Quand une carte est jouée (placée), il peut se passer l'une de ces trois choses :

— il y a un espace libre correspondant aux règles, la carte y est placée,
— la ligne dans laquelle la carte devrait se placer contient déjà cinq cartes, auquel cas la sixième carte *prend tout* : le joueur qui a préparé cette carte récupère les cinq cartes de la ligne et marque le malus total (pénalité) de ces cinq cartes, ce que les joueurs essayent d'éviter (quand cela arrive, la carte jouée devient la première dans la ligne qui a été vidée en prenant toutes les cartes pour calculer le score de ce joueur),
— la carte est plus basse que toutes les (dernières cartes de toutes les) lignes, et ne peut se placer nulle part, auquel cas *le joueur choisit une ligne à prendre* et récupère le malus total (pénalité) de toutes les cartes de la ligne ; l'avantage de faire cette action est que le joueur peut avoir une pénalité bien moindre qu'en prenant une ligne pleine.

Une manche (le jeu complet d'une donne) fait donc dix tours, avec chaque joueur qui a une carte de moins en main à chaque tour ; à la fin, les joueurs n'ont plus de cartes, le score de chacun est calculé par l'addition de tous les malus reçus.

Normalement, d'autres manches sont jouées avec le score de chacune ajouté à celui de chaque joueur jusqu'à atteindre une valeur fixe (66 dans le jeu de base) ou plus pour au moins un joueur ; le jeu est terminé à ce moment, le rang des joueurs est l'ordre croissant de leurs scores.

Pour plus de détails, voyez :

— La vidéo officielle de présentation de deux minutes : `https://youtu.be/6u5rp96PAqI`

— Une copie du livret de règles : `https://www.philibertnet.com/fr/index.php?controller=attachment&id_attachment=2979`

Le code fourni

Différences

Il y a trois différences (principales) entre le code fourni et le vrai jeu de cartes :

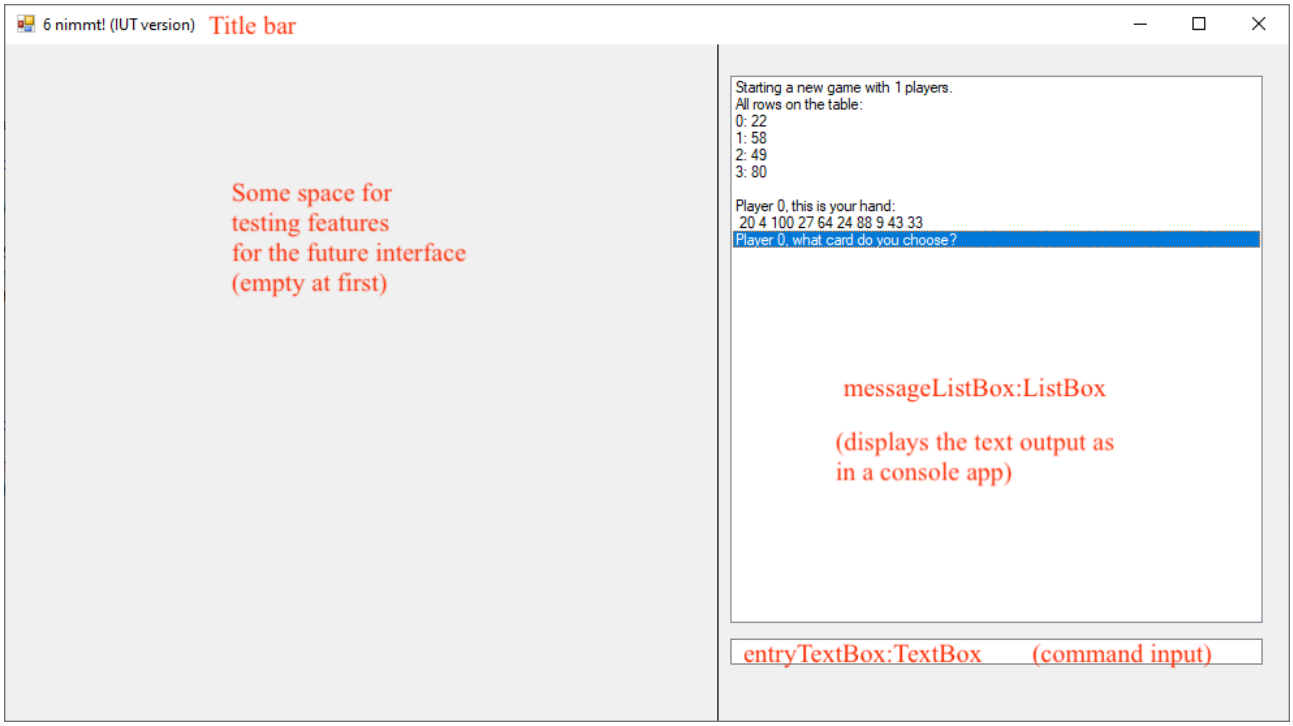
— Dans ce code, une seule manche est jouée (la possibilité de rejouer sera à ajouter dans votre projet).

— Ce code a été produit afin que le jeu en solo soit possible, et d'ailleurs lancé par défaut : en solo, le joueur place simplement ses cartes pour que vous puissiez tester les règles (le seul joueur gagne et perd toutes les parties).

— Ce code n'utilise pas les valeurs *officielles* de pénalité pour les vraies cartes du vrai jeu, mais compte juste un point de pénalité par carte (vous changerez la méthode de calcul dans le projet).

Comment se présente l'application fournie

Voici la fenêtre principale :



Au début, il n'y a que deux éléments avec lesquels l'utilisateur peut interagir :

— Une `TextBox` pour entrer des commandes (ici, des entiers),

— Une `ListBox` pour afficher des messages.

(Il n'y a donc qu'un événement externe et un événement résultat.)

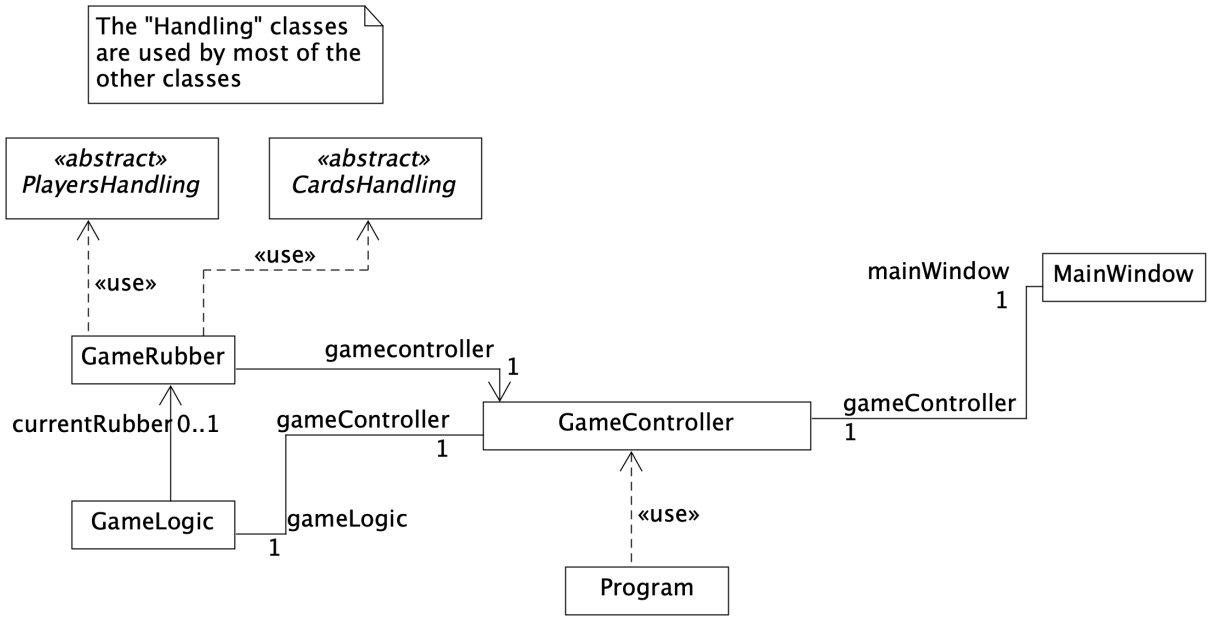
Sans rien changer, le jeu est jouable depuis le message de démarrage jusqu'au message de *game over*, simplement en tapant des entiers pertinents dans la `TextBox`. Pour un nouveau jeu, relancez l'application.

Structures de données, classes et responsabilités

Le jeu est très facile à modéliser, puisque les cartes sont identifiées par une valeur de 1 à 104 : résultat, les cartes sont stockées dans des `int`. Les joueurs sont aussi identifiés par des `int` (de 0 au nombre de joueurs moins 1). Donc, la main (liste de cartes) d'un joueur sera une `List<int>`; les quatre lignes de cartes sur la table seront une `List<List<int>>`.

Nous séparons la **logique du jeu** (classes métier, le modèle) de la **vue du jeu** (la fenêtre principale et tous autres formulaires ou fenêtres), et ces deux parties communiquent en utilisant un objet `controller` (*contrôleur*). Le but est de séparer ce qui est fourni : les classes `GameRubber`, `GameLogic`, `CardsHandling` et `PlayersHandling` sont terminées et ne nécessitent pas d'intervention (au moins au début, à la fin, seule `GameRubber` sera identique à sa première version), alors que vous changerez la classe `MainWindow` et ajouterez d'autres classes de type fenêtre (`Form`) pour l'interface utilisateur. Le `GameController` sera aussi changé quand ce sera nécessaire afin de maintenir opérationnelle la communication entre toutes les parties de l'application.

Ce qui suit est un diagramme de classes UML en spécification générale reflétant la conception de départ de l'application :



1 Tutoriel

Une ou deux sessions, rendus individuels, aide autorisée au sein de chaque binôme. Rendez les réponses et le code à la fin.

1.1 Exécution du code

Exercice 1. Ouvrez la solution, lancez-la : c'est un jeu en solo. Essayez une ou deux parties, lisez le code.

Question 1. Donnez la séquence d'appels qui se termine par l'affichage du **message de démarrage** suivant : `Starting a new game with 1 players..`

Question 2. Quelle est la responsabilité de (que fait, en une phrase) chacun des objets placés dans les régions `Association attributes` des classes `MainWindow`, `GameController`, `GameLogic` ?

Question 3. Selon la méthode `MalusValue` de la classe `CardsHandling` class, quelle est la valeur de pénalité associée à une carte arbitraire n ?

Question 4. Lisez la méthode `PickThisRow` de la classe `CardsHandling`. Quand est-elle appelée ? Comment fonctionne-t-elle ? Donnez un exemple.

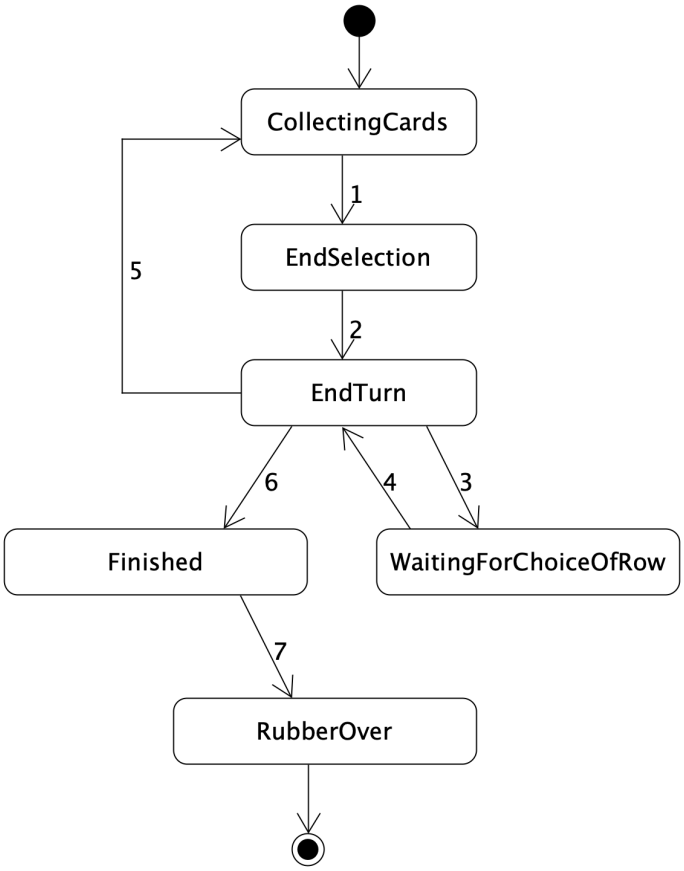
1.2 Changement : la manche à deux joueurs

Exercice 2. Dans `Program.cs`, changez le 1 en 2. Essayez.

Question 5. Quand il y a deux joueurs, chacun choisit la carte qu'il joue. Lisez en diagonale la méthode `OneAction` de la classe `GameRubber`. Comment la phase de sélection de cartes passe-t-elle du premier joueur (0) au second (1) ?

Regardez de plus près la méthode `OneAction`. **Vous n'avez pas besoin de comprendre le détail complet**, mais elle contient le cœur de la logique du jeu, principalement qui consiste à attendre une entrée et savoir quoi faire quand cette entrée arrive. La phase de la manche actuelle est stockée en permanence dans l'attribut `state`, et des données supplémentaires dépendant de cet état sont stockées dans des attributs tels que `cardsSelectedByPlayers`, `playerForSelection` et `playerChoosingARow`.

Ce qui suit est un diagramme d'états incomplet (ce n'est pas vraiment au programme de première année, ce n'est qu'un schéma explicatif) :



Explication : la manche est créée au niveau du rond plein et détruite (remplacée par `null` dans la classe `GameLogic`) dans le rond entouré d'un cercle. Les numéros signifient :

1. Chaque joueur a fini de sélectionner une carte de sa main.
2. Les cartes sont montrées, vont être placées.
3. Un joueur a une carte plus petite que toutes les lignes.
4. Ce joueur a pris une ligne, on reprend le placement des cartes.
5. Toutes les cartes sont placées, il reste des cartes en main, un nouveau tour commence.
6. Toutes les cartes sont placées, plus personne n'a de cartes en main, la manche est finie.
7. Les scores seront renvoyés, la manche supprimée.

1.3 Une interface plus utilisable

Exercice 3. Dans le constructeur de la classe `MainWindow`, dé-commentez les lignes commentées. Essayez.

Question 6. Comment sont affichées les lignes de cartes dans la classe `MainWindow` ?

Question 7. Combien de joueurs au maximum peuvent jouer avec uniquement les contrôles posés dans cette fenêtre ?

1.4 Première modification : une meilleure interface

Les `Label` permettent de suivre le flux d'une partie.

Exercice 3. Vérifiez que vous comprenez ce à quoi sert chaque `Label`. Vérifiez chacune en mode Design dans `MainWindow` à l'aide de la touche TAB, assignez un espace suffisant dans la propriété `Text` de chaque étiquette (quatre caractères par cinq nombres font environ 20 espaces, par dix, environ 40 espaces). Personnalisez les propriétés `BackColor`, `ForeColor` et `Font` de chacun des groupes (ligne, main, score) pour les distinguer ; re-positionnez les un peu si vous le souhaitez, et ajoutez des **étiquettes explicatives** (avec une police différente) pour que tout soit plus utilisable.

Ne prenez pas trop de temps pour cet exercice, vous aurez bientôt de meilleures idées pour cette interface.

C'est la première étape vers une interface graphique du jeu dans laquelle les `ListBox` et `TextBox` ne seront plus nécessaires (mais nous n'en sommes pas là).

1.5 Seconde modification : choix du nombre de joueurs

Le but est d'ajouter un dialogue affiché lors du lancement de l'application, permettant de choisir entre un et deux joueurs.

Exercice 4. Dé-commentez la ligne commentée dans `Program.cs`. Vérifiez comment et quand le dialogue apparaît.

Le code suivant :

```
MessageBoxButtons buttons = MessageBoxButtons.YesNo;
bool two = MessageBox.Show("Do you want to play a two-players game?",
                           "", buttons) == DialogResult.Yes;
```

affiche un dialogue avec des boutons oui et non, et vérifie si l'utilisateur clique sur oui.

Exercice 5. Remplacez le dialogue donné en exemple avec ce code, utilisez la valeur de retour pour créer un jeu à un ou deux joueurs.

Plus tard dans le projet, vous remplacerez ce dialogue par un plus joli.

1.6 Troisième modification : bouton nouvelle partie

Pour l'instant, quand une manche est terminée, l'application ne fait rien.

Exercice 6. Ajoutez un bouton `new game` à l'interface qui permet de démarrer une nouvelle manche (par exemple, quand la première est terminée). Pour ce faire, vous devrez :

1. Créer une méthode qui gère l'événement `Click` de ce bouton [1].
2. Cette méthode appelle une méthode (action sans paramètres) dans le `GameController` que vous devez créer.
3. Cette dernière méthode appelle une méthode (action sans paramètres) dans la `GameLogic`, que vous devez créer aussi.
4. Enfin, la méthode crée dans `GameLogic` appelle `UpdateRubbed` (pour créer une nouvelle manche et l'assigner à l'attribut `currentRubber`) puis (important) `OneAction` pour lancer la manche.

1. Le focus sera sur le bouton ; pour le repositionner dans la `TextBox`, vous pouvez faire appel à `entryTextBox.Focus();`.

La classe `GameRubber` ne devrait pas être modifiée lors de cet exercice.

1.7 Quatrième modification : scores

À la fin de chaque manche, les scores s'affichent dans la `ListBox`.

Exercice 7. Affichez-les plutôt dans un dialogue dédié (par exemple une `MessageBox`).

Exercice 8. Stockez les scores en fin de manche et affichez les scores cumulatifs (manche un plus manche deux plus...). Pour ce faire, vérifiez la méthode `UpdatePlayerScores` de `GameLogic`.

La classe `GameRubber` ne devrait pas être modifiée lors de ces exercices.

2 Conception d'une maquette

Lisez les spécifications du projet ci-après, puis prenez du temps en tant que binôme pour discuter et produire **votre vision pour le design de la fenêtre principale**. À la fin, il n'y aura plus ni `ListBox` ni `TextBox`, tout sera fait à la souris. Dans une maquette de votre design, donnée soit sous forme d'image (fichier PDF) soit sur papier, détaillez comment la fenêtre est organisée, quels contrôles sont placés ou, et ainsi de suite.

Exercixe 9. Rendez la maquette à votre enseignant **avant** de commencer l'implémentation.

3 Réalisation du projet

Le but est de produire une version graphique complète du jeu.

3.1 Éléments obligatoires

3.1.1 CardView

Nous utilisons des listes d'entiers convertis en chaînes de caractères posées dans des étiquettes pour afficher les cartes (mains des joueurs et lignes sur la table). Ce n'est pas satisfaisant : on désire avoir des **objets graphiques** qui représentent les cartes, affichant la valeur numérique mais aussi la valeur de pénalité (malus), et ces objets seront **cliquables** pour que les joueurs ne doivent plus taper le numéro d'une carte pour la jouer, mais juste cliquer dessus.

Tout ceci nécessite une classe `CardView` qui :

— Contienne une propriété (ou un attribut)`Point Position` pour l'affichage,
— Contienne un attribut (ou une propriété) `int card` qui représente la carte (logique),
— Fait appel à la méthode `CardsHandling::MalusValue` pour récupérer (peut-être dans une propriété calculée) la valeur de pénalité (malus),
— A une méthode `Draw` qui lui permet de se dessiner, par exemple en tant que chaîne de caractères dans un rectangle pour commencer,
— A une méthode `Contains` qui permet à la vue de savoir si un point est dedans.

L'objet contenant la fenêtre principale aura une collection (ou plusieurs collections, ou tableaux associatifs) de `CardView` pour les afficher, qui devra être utilisée dans la réponse à l'événement `Paint` (la méthode dédiée peut être changée, elle ne fait qu'afficher un séparateur vertical pour l'instant).

Pour commencer, les quatre lignes sur la table devraient être affichées avec des `CardView`.

Si une mise à jour est effectuée, il faudra appeler la méthode `Refresh` de la fenêtre.

3.1.2 La main des joueurs : une à la fois

Dans la version textuelle, même si les joueurs ne regardent pas quand l'un d'entre eux choisit sa carte, tout l'historique est visible et tout le monde connaît les cartes de tout le monde. Cela devra changer dans la version finale afin que seule la main du joueur courant est affichée dans la fenêtre à tout instant. Un message (par exemple `MessageBox.Show("Player "+p+", this is your turn.");`) devrait s'afficher avant la main, pour que la partie puisse être sincère.

3.1.3 Sélection des cartes et lignes

Quand le joueur choisit une carte, le jeu devrait attendre que le joueur clique sur une carte de sa main au lieu d'attendre que son numéro soit entré.

Quand toutes les cartes sont choisies, la vue devrait les afficher toutes par ordre croissant de valeur numérique et attendre une confirmation avant de les assigner.

Enfin, quand un joueur a une carte plus basse que toutes les dernières cartes des lignes, il devrait pouvoir choisir la ligne à prendre par un mécanisme de votre choix à la souris (cliquer n'importe où sur la ligne, par exemple) au lieu d'entrer du texte.

3.1.4 Les scores de tous les joueurs : tout le temps

Les scores de chaque joueur devraient toujours être affichés pour tous dans la fenêtre principale. Comme il peut y avoir de un à dix joueurs, ils seront positionnés automatiquement (par du code).

Vous devrez aussi changer le dialogue d'ouverture afin que le nombre de joueurs puisse être choisi entre un et dix, avec un dialogue personnalisé.

Après toutes ces modifications, les `ListBox` et `TextBox` ne devraient plus être nécessaires.

3.2 Éléments recommandés

3.2.1 Fin du jeu et ordre du score des joueurs

Normalement, de nouvelles manches sont jouées jusqu'à ce qu'au moins un des joueurs (n'importe lequel) atteigne ou dépasse un certain score (66 dans les règles). Changez `GameLogic` pour le permettre. Améliorez aussi l'affichage de chaque manche et des scores définitifs d'une partie en triant l'ordre d'affichage pour que le meilleur joueur (avec le plus petit score) soit en haut.

Le bouton (ou item de menu) lançant une nouvelle manche devrait être indisponible avant la fin de chaque manche (ou bien : les nouvelles manches sont lancées automatiquement). Un bouton (ou item de menu) lançant une nouvelle partie peut être ajouté, qui remplace complètement la partie en cours (en oubliant les scores et permettant de choisir un nombre de joueurs différent).

3.2.2 Vraies règles

Changez la méthode `MalusValue` de la classe `CardsHandling` pour que les vraies pénalités pour les cartes soient utilisées au lieu de 1. Vérifiez que votre affichage est à jour.

3.2.3 Compteurs de tour et de manche

Affichez (avec des `Label` ou des instructions `DrawString`) le numéro du tour (un à dix – le même que le nombre de cartes jouées par chacun), et le numéro de la manche courante.

3.3 Éléments facultatifs

3.3.1 Préférences

Ajoutez un dialogue **Settings** (préférences, paramètres) qui peut être ouvert pour changer des valeurs. Par exemple :

- Mode de jeu : nombre déterminé de manches (avec possibilité de choisir ce nombre) ou bien score cible (avec possibilité de choisir ce score, 66 par défaut) – ce changement nécessite une nouvelle partie.
- Taille d'affichage : permettez au moins de choisir entre une *petite* et *grande* taille d'affichage pour les cartes.

D'autres paramètres peuvent être pertinents suivant **votre** conception : couleurs, animations, alignement vertical ou horizontal, ordre commençant à 0 ou 1, etc.

Suivant votre conception, ce dialogue sera affiché au début d'une partie ou disponible à l'aide d'un bouton ou item de menu.

3.3.2 Nom des joueurs

Demandez son nom à chaque joueur au lieu de `Player 0` ou *First player*.

3.3.3 Jeu automatique

S'il reste du temps et que vous le souhaitez, demandez au début si chaque joueur est humain (et joue en cliquant sur les cartes) ou une IA (et joue automatiquement), et implémentez une IA pour ce jeu. (Ceci n'est pas un cours d'algorithmique, jouer au hasard est suffisant.)

3.4 Travail à faire

Trois à quatre séances.

Exercice 10. Réalisez le projet, autant que possible avec les éléments obligatoires en très forte priorité (éléments recommandés en priorité bien plus basse, facultatifs encore plus basse, et l'IA en dernier) en vous efforçant de maintenir une certaine qualité de code (surtout le nommage des variables et méthodes, et la documentation des méthodes non-privées si possible).

Gardez un œil sur le calendrier et l'horaire, ce projet ne devrait pas nécessiter de temps de travail personnel hors séances.