

CryptoSharp

LACHAUD Samuel / PAZOLA Loïs – M1 Informatique





2/10

SOMMAIRE

Sur la partie de droite, nous retrouvons le décodage du message. L’utilisateur peut saisir ici le message à décoder. Le bouton « Decode » après avoir saisi l’ordre des cartes de la clé pour obtenir le message décodé qui sera visible dans la partie centrale.

Sur la partie centrale, nous retrouvons l’encodage du message. L’utilisateur peut saisir ici le message à encoder après avoir saisi l’ordre des cartes dans la première partie. Enfin il peut cliquer sur le bouton « Encode » afin d’encoder le message. Celui-ci sera alors visible dans la troisième partie.

Sur la partie de gauche, nous retrouvons une liste déroulante avec l’intégralité des 54 cartes du deck. L’utilisateur peut cliquer sur une carte et la déplacer avec une action de « glisser déposer ». Il y a également un bouton « Shuffle » permettant de mélanger aléatoirement le deck.

Il n’existe pour le moment aucune version compilée de CryptoSharp. Le projet a été réalisé sur Visual Studio 2022, il est donc nécessaire d’avoir cet IDE d’installé afin de lancer facilement notre projet. CryptoSharp est réalisé en C# (langage de programmation) + WPF (Framework graphique) et se présente comme une fenêtre de style Windows 10 où l’utilisateur pourra interagir avec le logiciel, et d’un terminal affichant des informations supplémentaires par rapport aux codages et décodages ainsi qu’a la génération de clé. Sur l’interface graphique, nous retrouvons 3 zones de contrôleurs :

Présentation de l’application

CryptoSharp est un projet de codage et cryptographie codé en C# portant sur l’encodage et le décodage d’un message à l’aide d’un paquet de cartes. La méthode utilisée est le solitaire de Bruce Schneier. Celle-ci permet un encodage facile à mettre en œuvre et très complexe à rompre. Le paquet de cartes utilisé est un jeu de 54 cartes (52 cartes + 2 Jokers). Il aurait également possible d’utiliser un jeu de Tarot (incluant les cavaliers) mais nous sortons du cadre conventionnel. Le projet est open source et trouvable à l’adresse suivante : <https://github.com/Mahtwo/CryptoSharp>

3/10

Qu'est-ce que CryptoSharp

3/10

I) Introduction



II) Fonctionnement de

l'application

Gestion des cartes et du deck

Une image contenant texte

Description générée automatiquement

Figure : Méthode MoveCard

Dans la modélisation du projet, le premier « Objet » qui est évident c’est le paquet de carte. C’est le centre de notre codage et il est donc primordial de le modéliser ainsi que l’ensemble des cartes qui le compose. Pour cela, nous devons créer une liste de cartes avec l’ensemble des valeurs qu’elles peuvent prendre. Nous avons choisi la notation anglaise pour leurs noms et nous avons utilisé une énumération pour les stocker (pas besoin d’une classe pour cela). Celle-ci se situe dans le fichier Card.cs.

Une fois cette énumération présente, nous pouvons coder les différentes méthodes permettant les manipulations sur le Deck (ensemble de toutes les valeurs possibles de l’énumération). Parmi celles-ci, nous retrouvons quelques fonctions basiques :

* Deck() // Constructeur qui initialise la LinkedList « cards » avec toutes les valeurs de l’énumération.
* Deck(LinkedList<Card> cards) // Constructeur qui initialise la LinkedList « cards » avec celle en paramètre.
* GetCard(int index) // Retourne la carte à l’index en paramètre dans la LinkedList.
* Shuffle() // Mélange aléatoirement la LinkedList
* FindCardPosition(Card c) // Retourne la position de la carte en paramètre dans la LinkedList
* ToString() // Retourne une string qui affiche l’ensemble des cartes et leur index.

Une fois les opérations basiques codées, nous pouvons nous attaquer aux fonctions permettant d’implémenter les différentes opérations nécessaires à l’encodage. Pour commencer, nous avons une fonction pour changer la position d’une carte : MoveCard(Card c, int direction).

4/10

6/10

6 / 9

Une image contenant texte

Description générée automatiquement

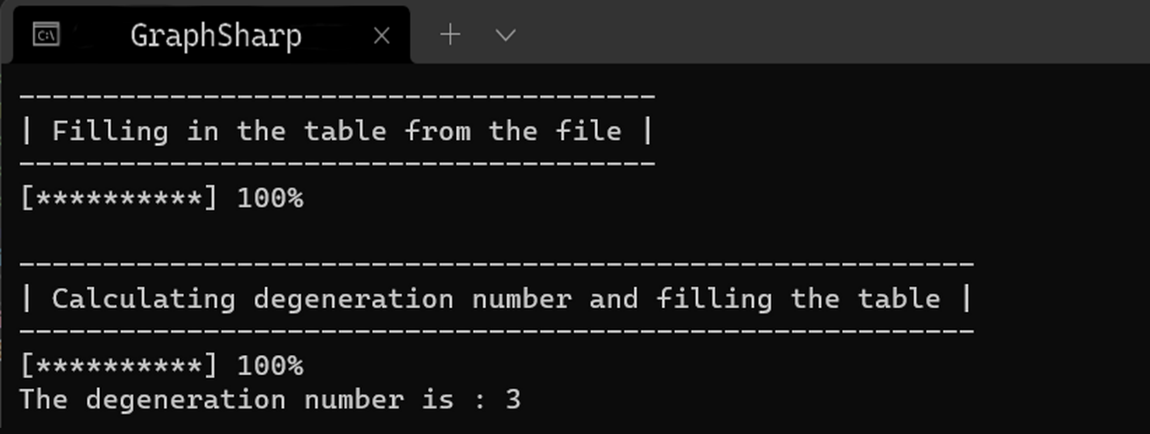
Cette fonction sert à déplacer une carte dans la liste de 1 ou plusieurs positions. Une position positive remontera la carte dans le paquet et une position négative la descendra dans le paquet. Le calcul de la nouvelle position utilise plusieurs modulos, c’est pour avoir des modulos fonctionnels avec des nombres négatifs (si on descend en dessous de la position 0 par exemple). On appelle cela un modulo canonique et ce n’est pas natif en C#. Le déplacement se fait de manière conventionnelle à part pour les bornes du deck.

En effet si on sort du dessus du deck, on ne se retrouve pas en dernière position, mais en avant dernière. De la même manière si on sort de la fin du deck, on ne retombe pas tout en haut du deck, mais en deuxième position. Cette fonction sert à réaliser les 2 premières opérations de la partie 3 du sujet :

*« 1 Recul du joker noir d’une position : Vous faites reculer le joker noir d’une place (vous le permutez avec la carte qui est juste derrière lui). Si le joker noir est en dernière position il passe derrière la carte du dessus (donc, en deuxième position).   
2 Recul du joker rouge de deux positions : Vous faites reculer le joker rouge de deux cartes. S’il était en dernière position, il passe en troisième position ; s’il était en avant dernière position il passe en deuxième. »*

La méthode suivante sert à réaliser la double coupe par rapport aux jokers : DoubleCutting()

3/11

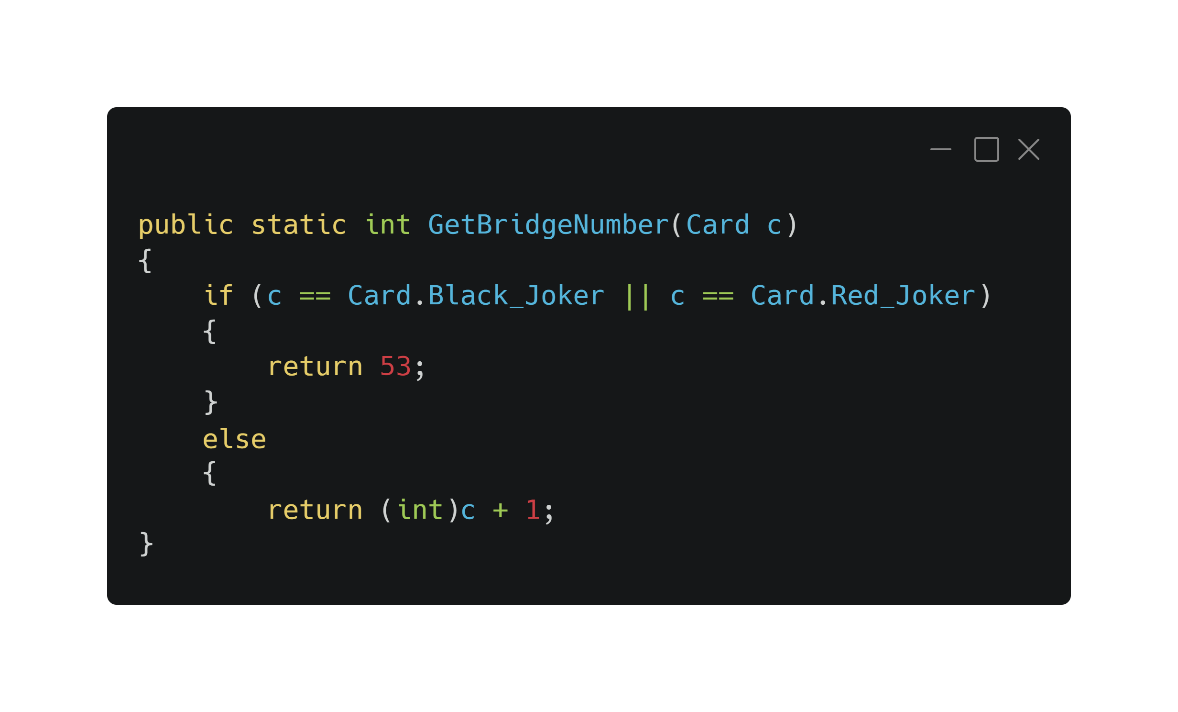
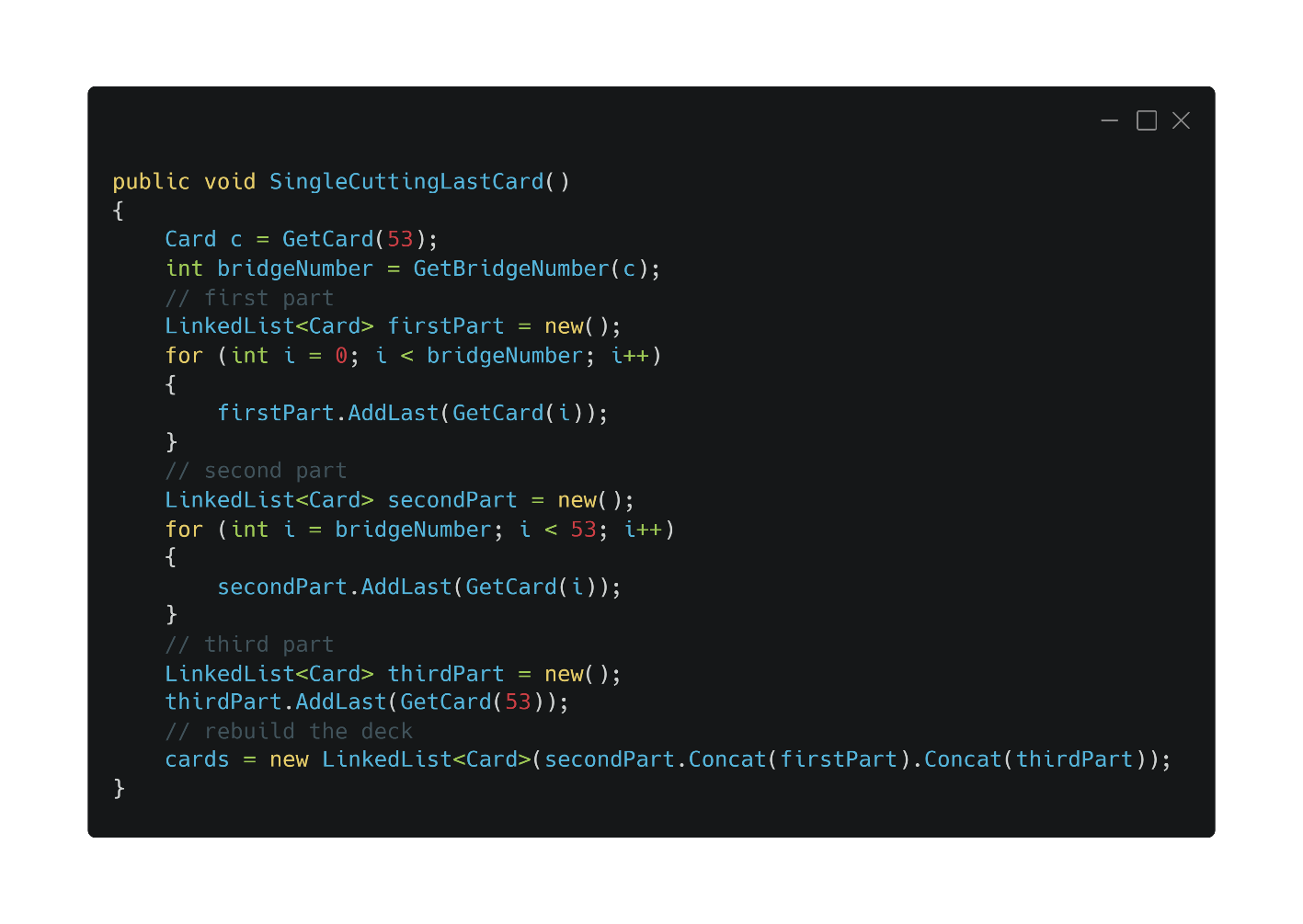


5/10

Figure : Méthode DoubleCutting

5/10

7/10



6/10

Figure 4 : Méthode SingleCuttingLastCard

Cette fonction toute bête nous sera très utile par la suite. Elle retourne simplement le numéro des cartes dans un deck dans l’ordre. L’ordre étant déjà le bon dans l’énumération, nous avons juste à récupérer le numéro de la carte dans celle-ci, sauf qu’elle commence à 0 et que les numéros du bridge commencent à 1. On rajoute juste 1 à cette valeur. Reste uniquement le cas des Joker, ceux-ci valent 53.

La fonction suivante nous sert à couper à partir de la dernière carte et de sa valeur de bridge : SingleCuttingLastCard()

Figure 3 : Méthode GetBridgeNumber

Comme dit précédemment, cette fonction sert à double couper le deck par rapport à la position des Jokers. On cherche donc d’abord les positions des deux Jokers avec les méthodes basiques épelées précédemment, puis on détermine celui qui a la position la plus haute, le Joker le plus proche du début du deck. On récupère le tas n°1 qui sont toutes les cartes entre les deux Jokers, puis le tas n°2 qui sont toutes les cartes entre le deuxième Joker et la fin du deck. Enfin, on intervertit ces deux tas. Notre double coupe est alors terminée. Cette fonction sert à réaliser la 3ème opération de la partie 3 du sujet :

*« 3* Double coupe par rapport aux jokers. Vous repérez les deux jokers et vous intervertissez le paquet des cartes situées au-dessus du joker qui est en premier avec le paquet de cartes qui est au-dessous du joker qui est en second. Dans cette opération la couleur des jokers est sans importance.*»*

La méthode suivante sert à récupérer le numéro du Bridge d’une carte : GetBridgeNumber(Card c)

Une image contenant texte

Description générée automatiquement

Cette fonction a pour but de récupérer la valeur de Bridge de la dernière carte du deck. Une fois cette valeur récupérée (que l’on appelle n par la suite), on prend donc les n première cartes puis on les met à l’avant dernière position du deck (en dessous du deck mais on garde la dernière carte à la fin). Cette fonction sert à réaliser la 4ème opération de la partie 3 du sujet :

*« 4 Coupe simple déterminée par la dernière carte : vous regardez la dernière carte et vous évaluez son numéro selon l’ordre du Bridge : trèfle-carreau-cœur-pique et dans chaque couleur as, 2, 3, 4, 5, 6, 7, 8, 9, 10, valet, dame et roi (l’as de trèfle a ainsi le numéro 1, le roi de pique a le numéro 52). Les jokers on par convention le numéro 53. Si le numéro de la dernière carte est n vous prenez les n premières cartes du dessus du paquet et les placez derrière les autres cartes à l’exception de la dernière carte qui reste la dernière »*

La prochaine méthode sert à vérifier le nombre de Bridge d’une carte à la position du nombre de Bridge d’une carte passée en paramètre et de vérifier si c’est un Joker : ReadingBridgeNumberNotJoker()

Une image contenant texte

Description générée automatiquement

Enfin, la dernière méthode sert à récupérer une lettre de l’alphabet à partir d’un nombre de Bridge : ReadingPseudoRandomLetter()

Figure 5 : Méthode ReadingBridgeNumberNotJoker



7/10

Pour cela on récupère le nombre de Bridge de la carte, puis le nombre de bridge de la carte à l’index du nombre de bridge précédent. Puis on force cette valeur entre 1 et 26 (Si c’est supérieur à 26, on retire 26). Le nombre ainsi obtenu sera une lettre de l’alphabet qui servira à créer la clé. Les deux fonctions précédentes servent à réaliser la 5ème et dernière opération de la partie 3 du sujet :

*« Lecture d’une lettre pseudo-aléatoire : Vous regardez le numéro de la première carte, soit n ce numéro. Vous comptez n cartes à partir du début et vous regardez la carte à laquelle vous êtes arrivé (la n + 1- i-ème), soit m son numéro. Si c’est un joker vous refaites une opération complète de mélange et de lecture (les points 1-2-3-4-5). Si m dépasse 26 vous soustrayez 26. Au nombre entre 1 et 26 ainsi obtenu est associée une lettre qui est la lettre suivante dans du flux de clefs. »*

Figure 6 : Méthode ReadingPseudoRandomLetter()



Encodage et Décodage

Une image contenant texte

Description générée automatiquement

Les fonctions que nous allons aborder dans cette partie sont présentes dans la classe encodeDecode.cs. Dans ce fichier, on retrouve tout d’abords deux petites fonctions de cast de valeurs l’une pour transformer un nombre entier en lettre de l’alphabet et une autre pour réaliser l’opération inverses : AlphabetToInt(char c) et IntToAlphabet(int i)

Figure 7 : Méthodes AlphabetToInt et IntToAlphabet

Une image contenant texte

Description générée automatiquement

Ces deux méthodes sont très utiles pour les deux autres méthodes de cette classe statique. Ce sont EncodeMessage(string message, int[] key) et DecodeMessage(string message, int[] key). Ces deux méthodes assez basiques effectuent les opérations d’encodage et de décodage à partir d’un message (codé ou non) et d’une clé sous la forme d’un tableau de nombre entiers (nombres récupérés grâce aux fonctions vu précédemment dans la classe Deck.cs. C’est ici qu’intervient les méthodes de cast, c’est pour passer d’une clé littérale a une clé numérique et inversement. Par simple soustraction pour l’encodage et addition pour décodage, le message est modifié.

8/10

Figure 8 : Méthodes EncodeMessage et DecodeMessage



Programme principal

Une image contenant texte

Description générée automatiquement

Une fois les deux parties précédentes réalisées, il est maintenant temps de les assembler. Pour cela nous avons donc un programme principal qui vas s’occuper d’exécuter les traitements dans l’ordres, de préparer les données pour le traitement suivant et de répéter les étapes tant que le résultat n’est pas obtenu. On retrouve donc cette partie centrale dans le programme MainWindow.xaml.cs.

Dans ce fichier, on retrouve l’ensembles des évènements associés à l’interface graphique (qui seront expliqués plus tard). Cependant 3 fonctions nous intéressent plus particulièrement dans cette partie :

* GetKeyFromCardsList(string message)
* ButtonEncodeMessage\_Click(object sender, RoutedEventArgs e)
* ButtonDecodeMessage\_Click(object sender, RoutedEventArgs e)

La première méthode permet de récupérer la liste des cartes dans l’ordre actuellement affiché, afin de l’utiliser comme clé d’encodage ou de décodage (tableau de nombre entier). Les deux méthodes suivantes permettent de récupérer les données des champs de décodage ou d’encodage (en fonction de la méthode), effectuer les opérations respectives, puis afficher le résultat (dans l’interface et dans la console pour des valeurs plus détaillées)

9/10

Figure 9 : Méthodes GetKeyFromCardsList, ButtonEncodeMessage\_Click et ButtonDecodeMessage\_Click



III) Application graphique

Interface et Framework

Une image contenant texte

Description générée automatiquement

Dans cette partie, nous allons revenir sur l’interface graphique, sa création, ainsi que les différents événements abordés précédemment. Nous allons tout d’abord nous pencher sur l’interface et son Framework.

L’interface de CryptoSharp utilise WPF comme interface graphique. Chacun des éléments de l’interface a été recréé afin d’obtenir une interface respectant la charte graphique utilisée par Visual Studio 2022 (Microsoft), pour obtenir un thème sombre harmonieux. Voici un exemple :

Figure 10 : Exemple utilisation style custom sur boutons en XAML (WPF)

Pour mettre en forme les différents contrôleurs maintenant remodelés, nous avons utilisé un Grid en WPF. Celle-ci nous permet de découper l’interface pour mieux placer ceux-ci. C’est un peu comme si on mettait Display : flex; sur la fenêtre parent. Nous avons donc une ligne seulement qui prend l’ensemble de la hauteur de la fenêtre (\*). Pour les colonnes, nous avons un découpage en 5 parties : La première colonne prend 1/5ème de la largeur totale, puis les deux colonnes suivantes prennent de la même manière 2/5ème de la largeur totale. Voici donc la description de cette Grid en XAML puis ce que cela donne graphiquement :

10/10

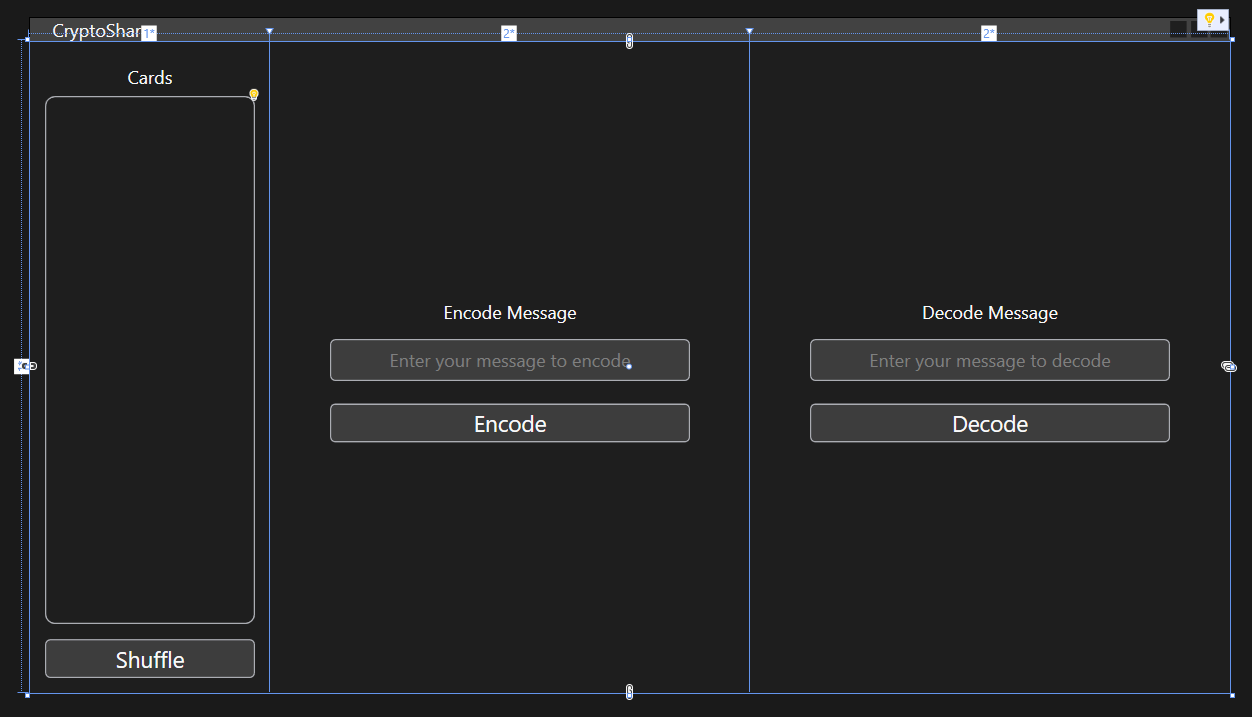




Figure 11 : Définition de la Grid et affichage de celle-ci

La dernière partie concernant l’interface est à propos des évènements. Voici donc une courte description de chacun d’entre eux.

* CardsList\_MouseDown(object sender, MouseButtonEventArgs e) // Evènement qui intervient quand l’utilisateur clique sur un élément de la ListBox. Celui-ci permet de commencer l’action de glisser-déposer.
* CardsList\_DragEnterOver(object sender, DragEventArgs e) //. Cet évènement permet le déroulement automatique de la liste si l’utilisateur veux poser la carte, plus bas ou plus haut que ce qui est affiché. Il permet également la prévisualisation d’où l’élément sera une fois relaché
* WaitAllowScroll() // Méthode permettant de baisser la vitesse de déroulement pour éviter que celui-ci soit trop rapide dans la liste
* CardsList\_Drop(object sender, DragEventArgs e) // Evènement qui intervient quand l’utilisateur relâche un élément de la ListBox dans celle-ci. Celui-ci permet de finir l’action de glisser-déposer
* TextBox\_LostFocus(object sender, RoutedEventArgs e) // Evènement qui intervient quand l’utilisateur ne cible plus une des deux zone de textes. Il sert à remettre le texte par défaut dans ce cas.
* TextBox\_GotFocus(object sender, RoutedEventArgs e) // Evènement qui intervient quand l’utilisateur cible une des deux zone de textes. Il sert à retirer le texte par défaut et à changer la couleur d’écriture.
* TextBox\_TextChanged(object sender, TextChangedEventArgs e) // Evènement qui intervient quand l’utilisateur change le texte d’une des deux zones de texte. Il permet de vérifier que le texte saisi est correct (texte comportant uniquement des lettres en majuscule)
* ButtonShuffle\_Click(object sender, RoutedEventArgs e) // Evènement lié au bouton de mélange de la liste de cartes. Celui-ci permet d’appeler la fonction shuffle() de la classe deck et d’ainsi mélanger les cartes en arrière-plan dans la classe et en premier plan dans la liste.

L’ensemble de ces méthodes d’évènements permettent à l’utilisateur d’utiliser le logiciel en toute transparence et d’assurer une expérience agréable à celui-ci.

11/10



Utilisation de l’application

Maintenant que nous avons expliqué l’intégralité du contenu de l’application, il est temps de lancer CryptoSharp. Pour cela deux manières sont possible :

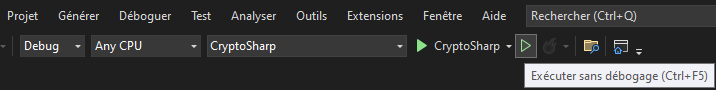
* Lancement de l’application non compilée (lancement par le biais du projet Visual Studio).
* Lancement de l’application compilée (lancement par le biais de l’exécutable compilé).

Une image contenant texte, écran, capture d’écran

Description générée automatiquement

Nous allons tout d’abord voir la procédure de lancement par Visual Studio : Tout d’abord, il nous faudra nous rendre à la racine du projet, puis lancer le fichier CryptoSharp.sln avec l’IDE :

Une fois dans l’IDE, il ne nous reste plus qu’a lancer le projet avec le bouton suivant :



Une image contenant texte, capture d’écran, écran, intérieur

Description générée automatiquement

Figure 12 : Logiciel lancé avec Visual Sudio

Une fois ceci fait, on se retrouve avec le logiciel lancé. On retrouve l’interface graphique où l’utilisateur peut effectuer les différentes opérations permises, ainsi qu’un terminal qui sert à récupérer des données supplémentaires.

Pour lancer la version compilée, il suffit juste de se rendre à la racine du projet, puis de lancer   
/publish build/CryptoSharp.exe. Et nous obtenons la même chose que sur la *Figure 12*.

12/10

Une image contenant texte, capture d’écran, écran

Description générée automatiquement

Figure 1 : Etape 1 - Saisir le texte à encoder

Il nous reste plus qu’à prendre un exemple pour montrer le fonctionnement de l’application. Pour cela, nous prenons la phrase d’exemple : ILOVERACLETTE. Nous lançons donc notre logiciel, puis dans la zone de texte « Encode Message » on saisit la phrase d’exemple :

Nous pouvons ensuite mélanger le paquet de cartes. Pour cela, ici on a modifié la position de 3 cartes dans le paquet (un mélange aléatoire aurait également pu être réalisé) :

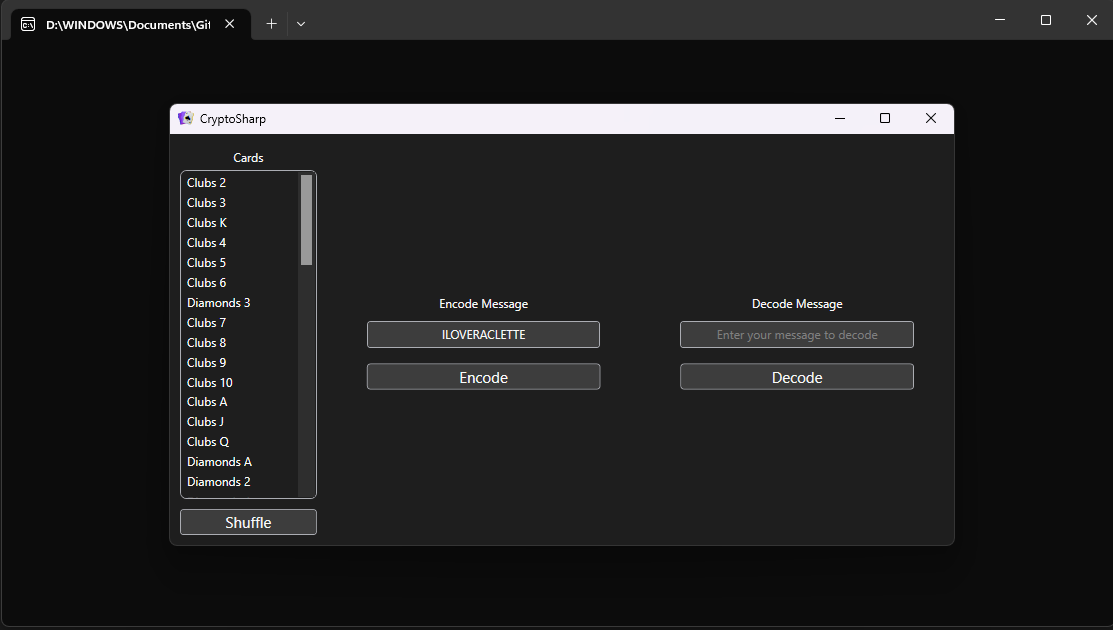


Figure 1 : Etape 2 - Mélanger le paquet de cartes en intervertissant celles-ci ou en appuyant sur "Shuffle"

13/10

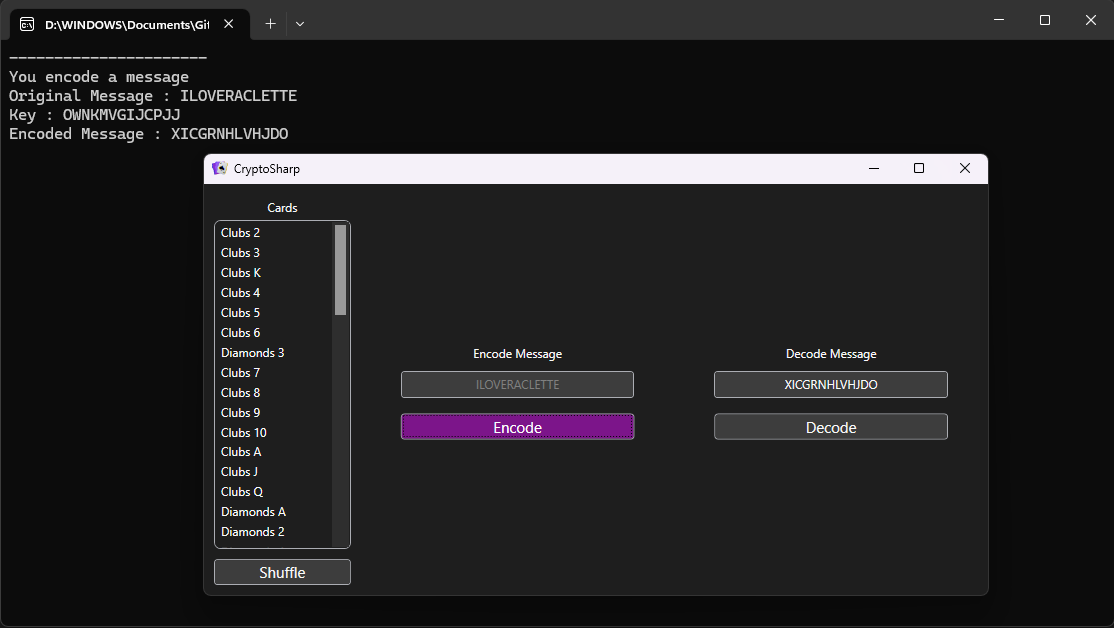


Figure 1 : Etape 3 - Lancer l'encodage et obtenir le résultat par appuis sur le bouton "Encode"

Nous pouvons maintenant cliquer sur le bouton « Encode ». Le message encodé apparait donc dans la zone de texte en dessous de « Decode Message ». Nous pouvons également voir des informations supplémentaires dans le terminal. En effet il est possible de voir le message original, le message encodé et la clé de ce chiffrement.

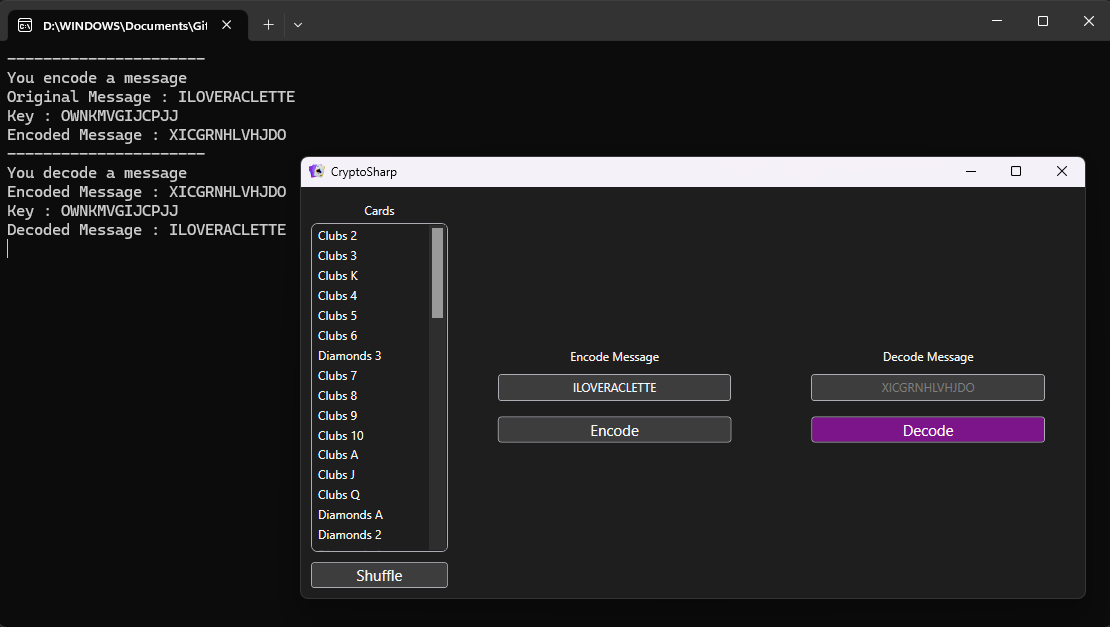


Figure 1 : Etape - Lancer le décodage et obtenir le résultat par appuis sur le bouton "Decode"

Nous pouvons maintenant vérifier le décodage. Partons du fait que l’on vient d’encoder le message et qu’on le donne à une autre personne ainsi que notre paquet de carte trié dans le même ordre. On reprend donc le logiciel, tout en supprimant les données. Puis on insère le message encodé (ici : XICGRNHLVHJDO) et on appuie sur « Decode ». On obtient donc le résultat dans le champ « Encode Message » et on peut voir les informations de décodage dans le terminal.

14/10



IV) Contenu supplémentaire

Tests Unitaires

La première amélioration réalisée sur notre application est les tests unitaires. En effet, la réalisation de ceux-ci nous est rapidement apparue comme une évidence. Nous avons commencé par créer l’énumération Card puis la classe Deck, cependant, le codage des différentes méthodes de manipulations d’ordre des cartes du deck fut difficile et sans tests nous n’aurions pas pu savoir ce qui n’allait pas dans ces différentes méthodes. Les tests unitaires de chaque méthode de la partie métier ont donc été réalisés. Cependant, nous voulions aller plus loin en vérifiant la couverture de tests de l’ensemble de ces méthodes. En effet, il se peux que certaines conditions des méthodes ne soient pas testées dans les test. Cette partie est d’autant plus importante pour nous, vu que l’on souhaite intégrer le Master 2 ISL (Ingénierie et Systèmes Logiciels) à Besançon.

Ayant Github Student (grâce à la facultée), nous pouvons bénéficier gratuitement de l’intégralité des services Jetbrains gratuitement. Cela inclut ReSharper, un lot d’extensions pour Visual Studio visant à aider le développement, mais surtout ce qui nous intéresse ici, les tests. Cet outil nous permet de vérifier rapidement nos tests déjà effectués, mais surtout de bénéficier d’un outil de couverture de tests (Tests Coverage).

Une image contenant texte

Description générée automatiquement

15/10

Figure 1 : Chemin de lancement de la couverture de tests dans Visual Studio avec l'extension ReSharper