UFR sciences & techniques

UNIVERSITÉ DE BOURGOGNE

CryptoSharp

LACHAUD Samuel / PAZOLA Loïs – M1 Informatique

SOMMAIRE

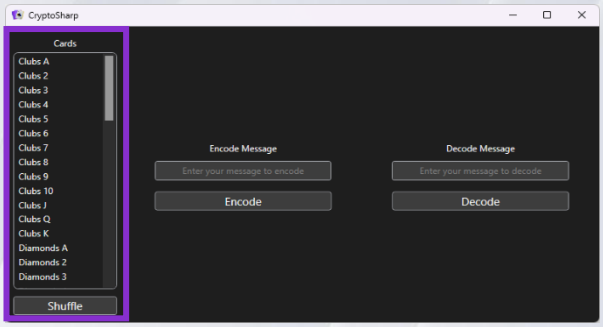
I) Introduction

Qu'est-ce que CryptoSharp

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

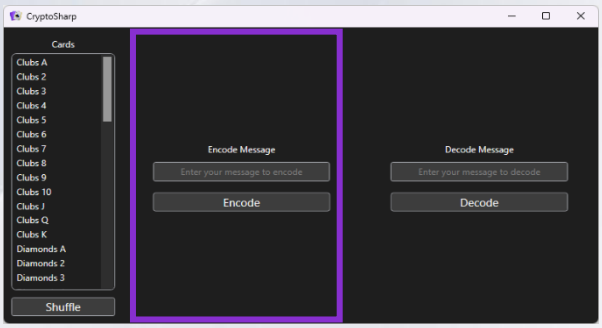
Présentation de l'application

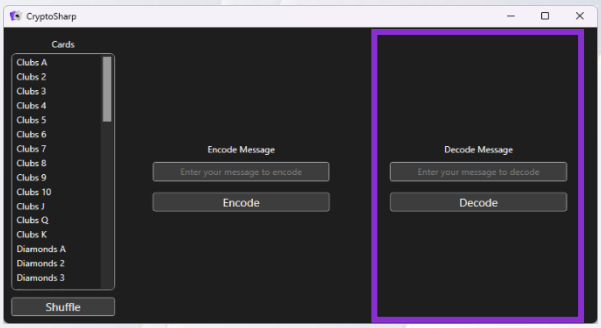
Il n'existe pour le moment aucune version compilée de CryptoSharp. Le projet a été réalisé sur Visual Studio 2022, il est donc nécessaire d'avoir cet IDE d'installé afin de lancer facilement notre projet. CryptoSharp est réalisé en C# (langage de programmation) + WPF (Framework graphique) et se présente 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 :



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.

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 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.

II) Fonctionnement de l'application

Gestion des cartes et du deck

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)`.

```
public void MoveCard(Card c, int direction) // direction -1 is down | direction +1 is up
{
    int pos = FindCardPosition(c);
    int newPos = (((pos + direction) % 54) + 54) % 54; // canonical modulus (work with negative numbers, not native in C#)
    LinkedListNode<Card> node = cards.Find(GetCard(newPos))!;
    cards.Remove(c);
    if (direction > 0)
    {
        cards.AddAfter(node, c);
    }
    else
    {
        cards.AddBefore(node, c);
    }
}
```

Figure 1 : Méthode MoveCard

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()`

```
public void DoubleCutting()
{
    // find jokers
    int posJoker1 = FindCardPosition(Card.Black_Joker);
    int posJoker2 = FindCardPosition(Card.Red_Joker);
    if (posJoker1 > posJoker2)
    {
        // swap jokers
        (posJoker2, posJoker1) = (posJoker1, posJoker2);
    }
    // get first part of the deck
    LinkedList<Card> firstPart = new();
    for (int i = 0; i < posJoker1; i++)
    {
        firstPart.AddLast(GetCard(i));
    }
    // get second part of the deck
    LinkedList<Card> secondPart = new();
    for (int i = posJoker1; i <= posJoker2; i++)
    {
        secondPart.AddLast(GetCard(i));
    }
    // get third part of the deck
    LinkedList<Card> thirdPart = new();
    for (int i = posJoker2 + 1; i < 54; i++)
    {
        thirdPart.AddLast(GetCard(i));
    }
    // rebuild the deck
    cards = new LinkedList<Card>(thirdPart.Concat(secondPart).Concat(firstPart));
}
```

Figure 2 : Méthode DoubleCutting

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)`

```
public static int GetBridgeNumber(Card c)
{
    if (c == Card.Black_Joker || c == Card.Red_Joker)
    {
        return 53;
    }
    else
    {
        return (int)c + 1;
}
```

Figure 3 : Méthode GetBridgeNumber

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()`

```
public void SingleCuttingLastCard()
{
    Card c = GetCard(53);
    int bridgeNumber = GetBridgeNumber(c);
    // first part
    LinkedList<Card> firstPart = new();
    for (int i = 0; i < bridgeNumber; i++)
    {
        firstPart.AddLast(GetCard(i));
    }
    // second part
    LinkedList<Card> secondPart = new();
    for (int i = bridgeNumber; i < 53; i++)
    {
        secondPart.AddLast(GetCard(i));
    }
    // third part
    LinkedList<Card> thirdPart = new();
    thirdPart.AddLast(GetCard(53));
    // rebuild the deck
    cards = new LinkedList<Card>(secondPart.Concat(firstPart).Concat(thirdPart));
}
```

Figure 4 : Méthode SingleCuttingLastCard

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()`

```
public bool ReadingBridgeNumberNotJoker()
{
    int bridgeNumberFirstCard = GetBridgeNumber(GetCard(0)); // = n
    // We want the card at "bridgeNumberFirstCard + 1",
    // however bridgeNumberFirstCard returns between 1 and 53 included AND our index starts at 0,
    // So we are basically doing "bridgeNumberFirstCard + 1 - 1"
    int bridgeNumberSecondCard = GetBridgeNumber(GetCard(bridgeNumberFirstCard)); // = m
    return bridgeNumberSecondCard != 53; // false if we have a Joker
}
```

Figure 5 : Méthode ReadingBridgeNumberNotJoker

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

```
public int ReadingPseudoRandomLetters()
{
    int bridgeNumberFirstCard = GetBridgeNumber(GetCard(0)); // = n
    int bridgeNumberSecondCard = GetBridgeNumber(GetCard(bridgeNumberFirstCard)); // = m
    if (bridgeNumberSecondCard > 26)
    {
        bridgeNumberSecondCard -= 26;
    }
    return bridgeNumberSecondCard;
}
```

Figure 6 : Méthode ReadingPseudoRandomLetter()

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. »

Encodage et Décodage

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)`

```
public static int AlphabetToInt(char c)
{
    return char.ToUpper(c) - 64;
}


public static char IntToAlphabet(int i)
{
    return (char)(i + 64);
}
```

Figure 7 : Méthodes AlphabetToInt et IntToAlphabet

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.

```
public static string EncodeMessage(string message, int[] key)
{
    int[] letters = new int[message.Length];
    for (int i = 0; i < message.Length; i++)
    {
        letters[i] = EncodeDecode.AlphabetToInt(message[i]);
    }
    string encodedMessage = "";
    for (int i = 0; i < letters.Length; i++)
    {
        int letterValue = letters[i] + key[i];
        if (letterValue > 26)
        {
            letterValue -= 26;
        }
        encodedMessage += IntToAlphabet(letterValue);
    }
    return encodedMessage;
}


public static string DecodeMessage(string message, int[] key)
{
    int[] letters = new int[message.Length];
    for (int i = 0; i < message.Length; i++)
    {
        letters[i] = EncodeDecode.AlphabetToInt(message[i]);
    }
    string decodedMessage = "";
    for (int i = 0; i < letters.Length; i++)
    {
        int letterValue = letters[i] - key[i];
        if (letterValue < 1)
        {
            letterValue += 26;
        }
        decodedMessage += IntToAlphabet(letterValue);
    }
    return decodedMessage;
}
```

Figure 8 : Méthodes EncodeMessage et DecodeMessage

Programme principal

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)

```
private int[] GetKeyFromCardsList(string message)
{
    // get all cards in ListBox
    LinkedList<Card> cardsListLinkedList = new();
    foreach (string card in cardsList.Items)
    {
        string cardUnderscore = card.Replace(' ', '_');
        cardsListLinkedList.AddLast(Enum.Parse<Card>(cardUnderscore));
    }
    deck = new Deck(cardsListLinkedList);
    // create the key
    int[] encodeKey = new int[message.Length];
    for (int i = 0; i < message.Length; i++)
    {
        bool readingBridgeNumberNotJokerOk = false;
        while (!readingBridgeNumberNotJokerOk)
        {
            deck.MoveCard(Card.Black_Joker, 1);
            deck.MoveCard(Card.Red_Joker, 2);
            deck.DoubleCutting();
            deck.SingleCuttingLastCard();
            readingBridgeNumberNotJokerOk = deck.ReadingBridgeNumberNotJoker();
        }
        int letterValue = deck.ReadingPseudoRandomLetters();
        encodeKey[i] = letterValue;
    }
    return encodeKey;
}



// Presque le même code pour ButtonDecodeMessage_Click
private void ButtonEncodeMessage_Click(object sender, RoutedEventArgs e)
{
    // Check if message exists
    string message = inputEncodeMessage.Text;
    if (message == "" || message == "Enter your message to encode")
    {
        return;
    }
    // Encode
    int[] encodeKey = GetKeyFromCardsList(message);
    string encodedMessage = EncodeDecode.EncodeMessage(message, encodeKey);
    inputDecodeMessage.Text = encodedMessage;
    inputDecodeMessage.Foreground = newSolidColorBrush((Color)ColorConverter.ConvertFromString("#FAFAFA"));
    inputEncodeMessage.Foreground = Brushes.Gray;
    // Display in terminal
    string key = "";
    for (int i = 0; i < message.Length; i++)
    {
        key += EncodeDecode.IntToAlphabet(encodeKey[i]);
    }
    Console.WriteLine("-----------------------");
    Console.WriteLine("You encode a message");
    Console.WriteLine("Original Message : " + message);
    Console.WriteLine("Key : " + key);
    Console.WriteLine("Encoded Message : " + encodedMessage);
}
```

Figure 9 : Méthodes GetKeyFromCardsList, ButtonEncodeMessage_Click et ButtonDecodeMessage_Click

III) Application graphique

Interface et Framework

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 :

```xaml
<!-- STYLE POUR LES BOUTONS -->
<Style TargetType="Button" x:Key="btn">
    <Setter Property="Background" Value="#3d3d3d"/>
    <Setter Property="Foreground" Value="#FAFAFA"/>
    <Setter Property="FontSize" Value="15"/>
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="Button">
                <Border Background="{TemplateBinding Background}"
                    CornerRadius="3"
                    BorderThickness="{TemplateBinding BorderThickness}"
                    Padding="2"
                    BorderBrush="#abadb3">
                    <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center" />
                </Border>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Background" Value="{Binding Source={x:Static SystemParameters.WindowGlassBrush}}"/>
            <Setter Property="Foreground" Value="#fff"/>
        </Trigger>
    </Style.Triggers>
</Style>


<!-- EXEMPLE APPLICATION DU STYLE -->
<Button x:Name="buttonDecodeMessage" Content="Decode" Click="ButtonDecodeMessage_Click" Style="{StaticResource btn}" />
```

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 :

```xaml
<Grid HorizontalAlignment="Right" Width="800">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
</Grid>
```

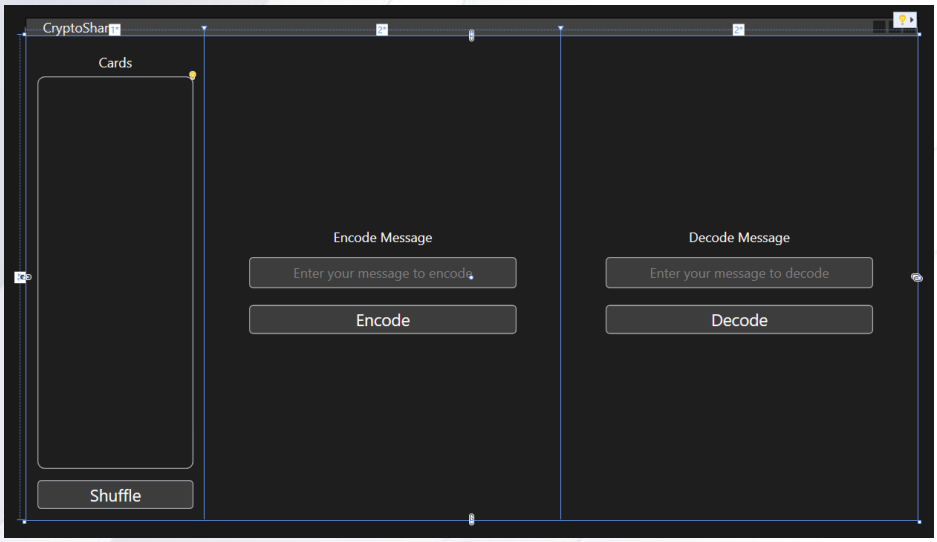


Figure 11 : Définition de la Grid et affichage de celle-ci

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.

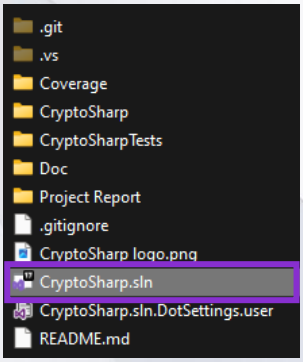
L'ensemble de ces méthodes d'évènements permettent à l'utilisateur d'utiliser le logiciel en toute transparence et d'assurer une expérience agréable à celui-ci.

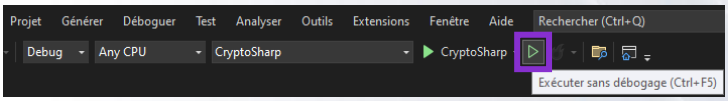
Utilisation de l'application

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).

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 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.

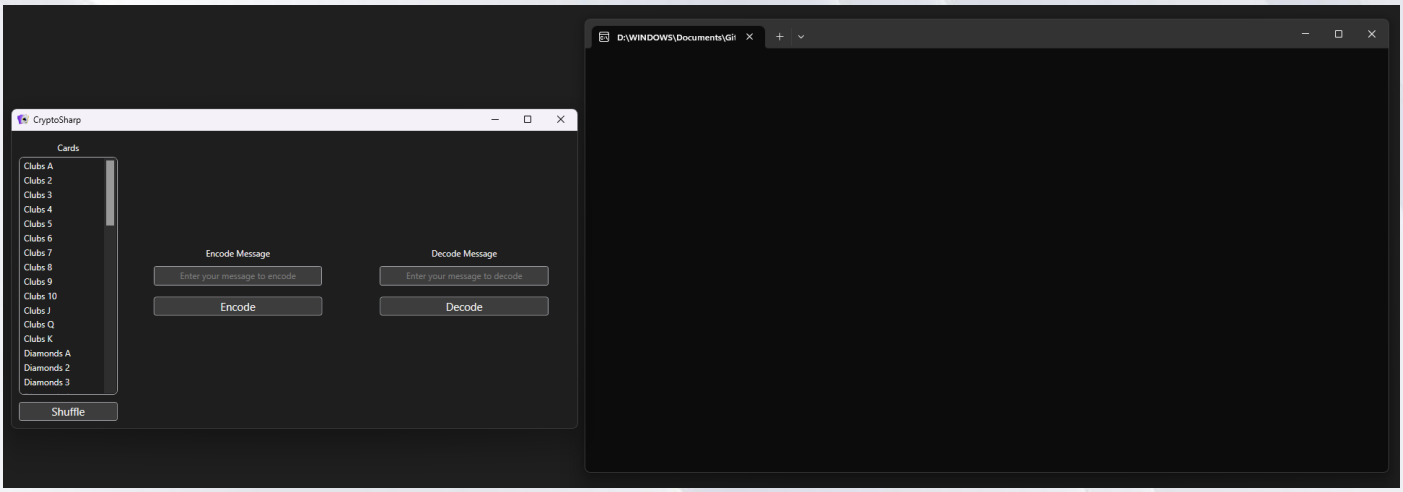


Figure 12 : Logiciel lancé avec Visual Sudio

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*.

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 :

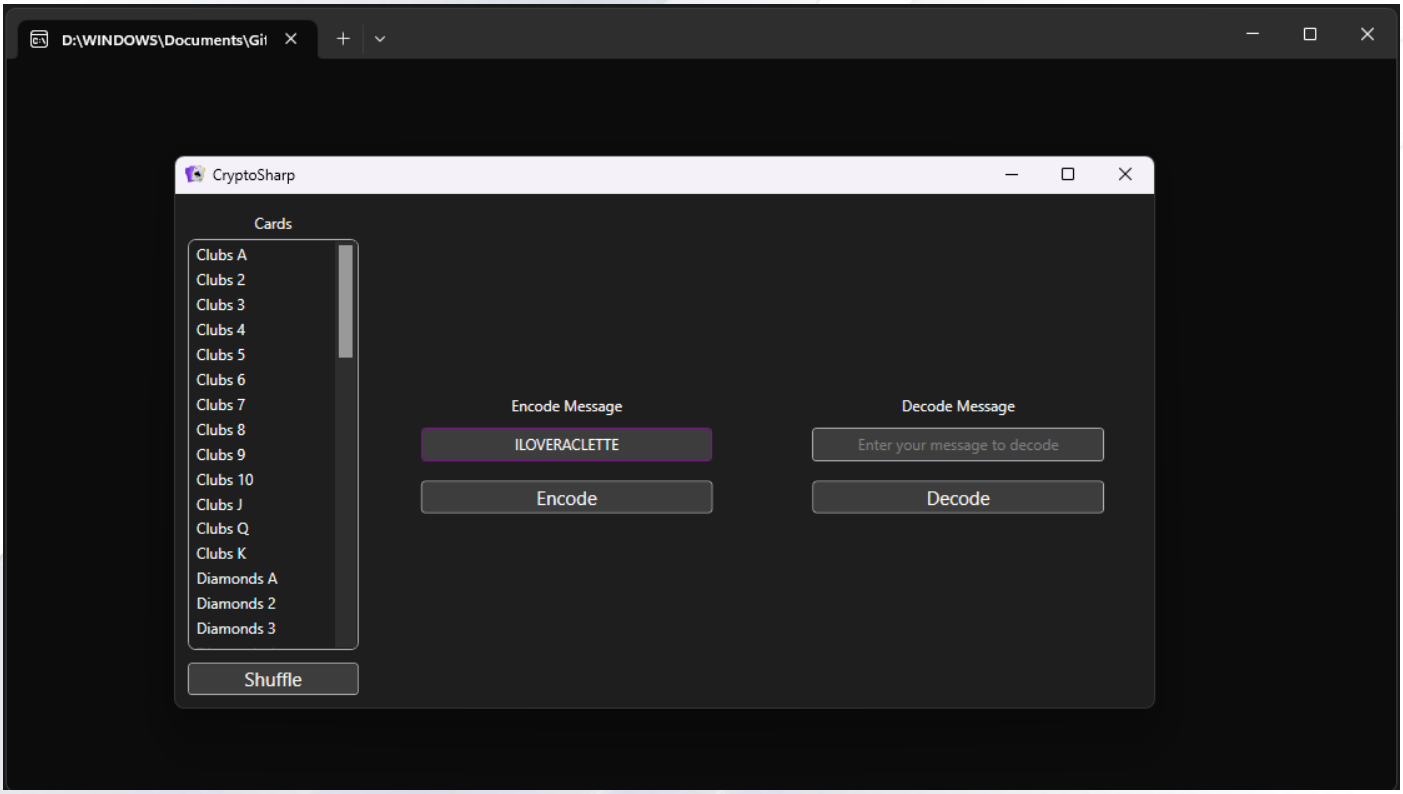


Figure 13 : Etape 1 - Saisir le texte à encoder

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) :

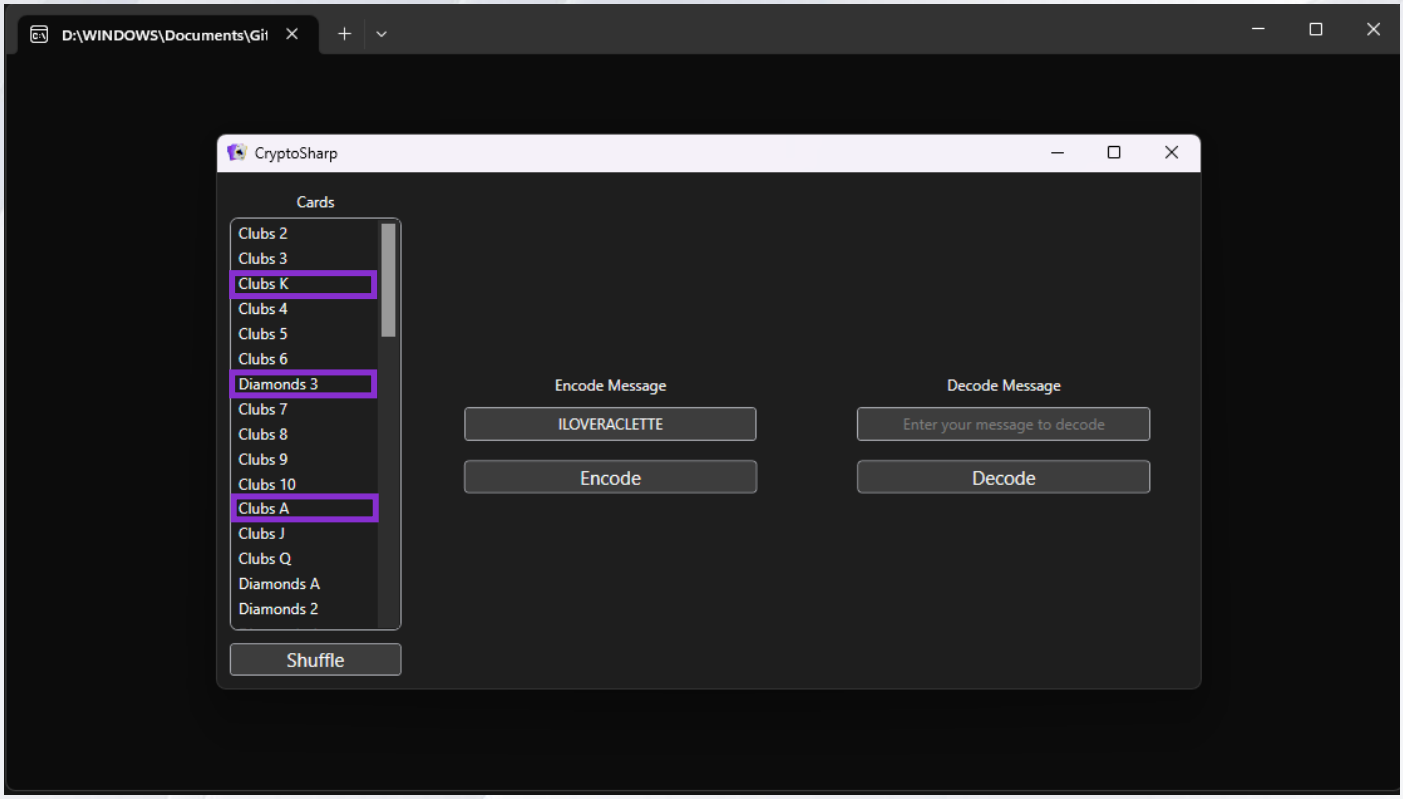


Figure 14 : Etape 2 - Mélanger le paquet de cartes en intervertissant celles-ci ou en appuyant sur "Shuffle"

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.

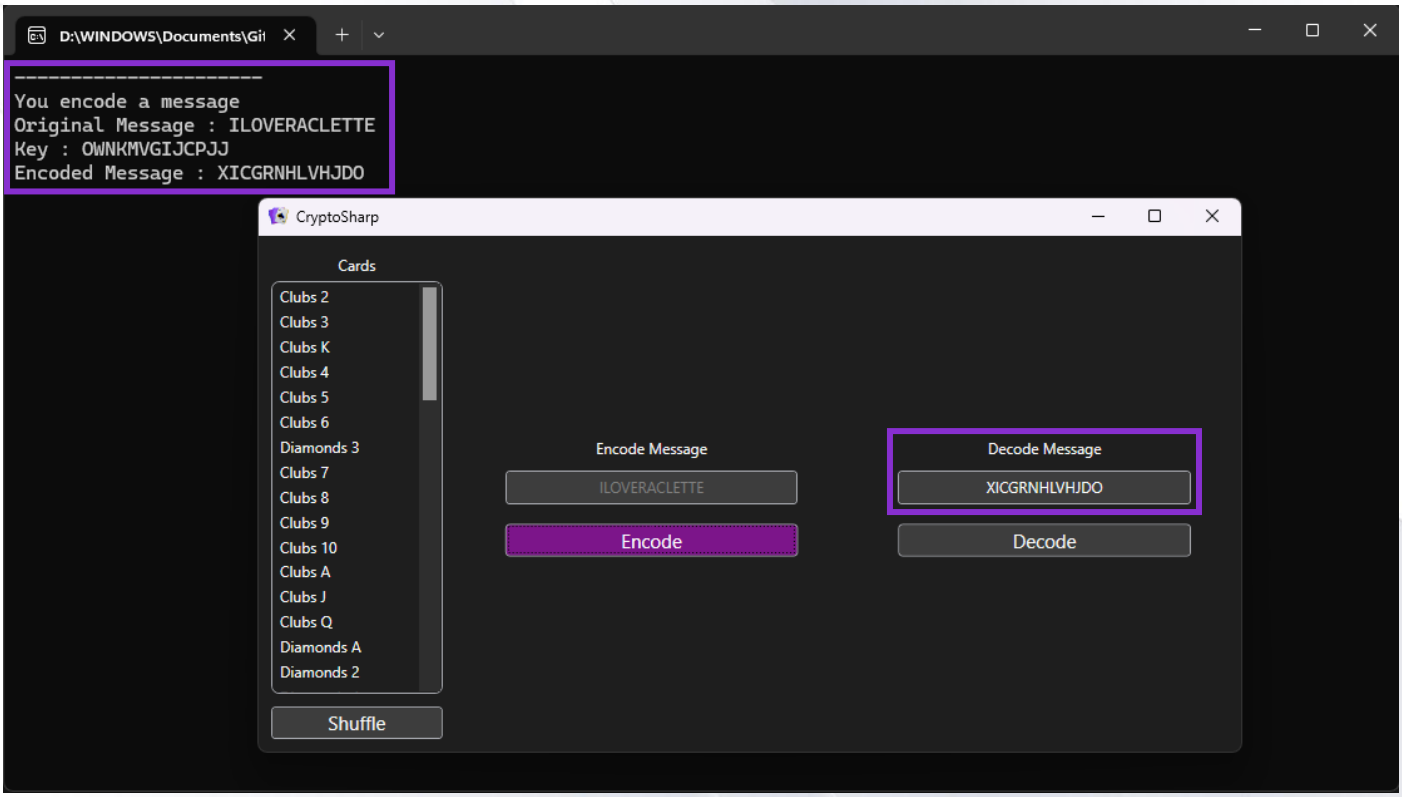


Figure 15 : Etape 3 - Lancer l'encodage et obtenir le résultat par appuis sur le bouton "Encode"

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.

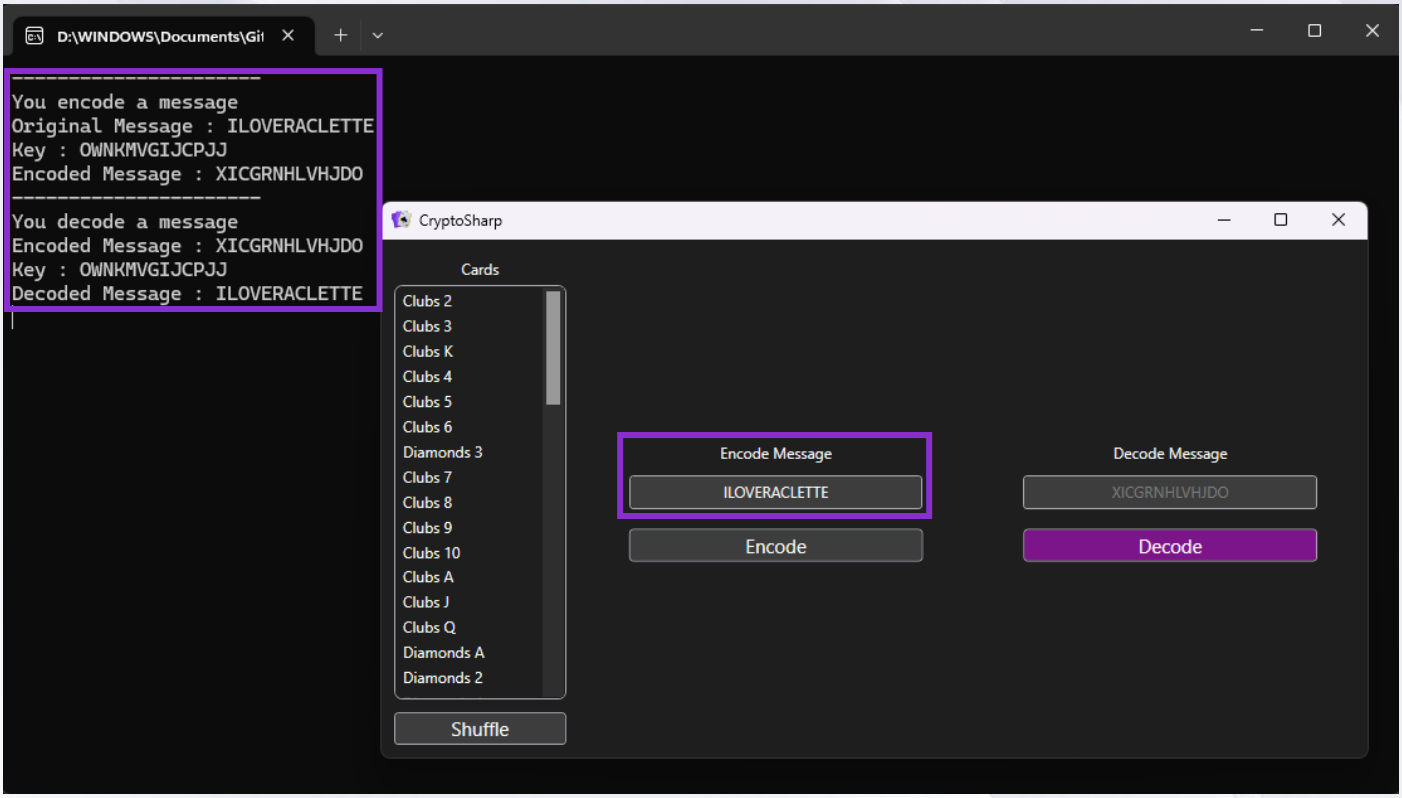


Figure 16 : Etape - Lancer le décodage et obtenir le résultat par appuis sur le bouton "Decode"

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 l'université), 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).

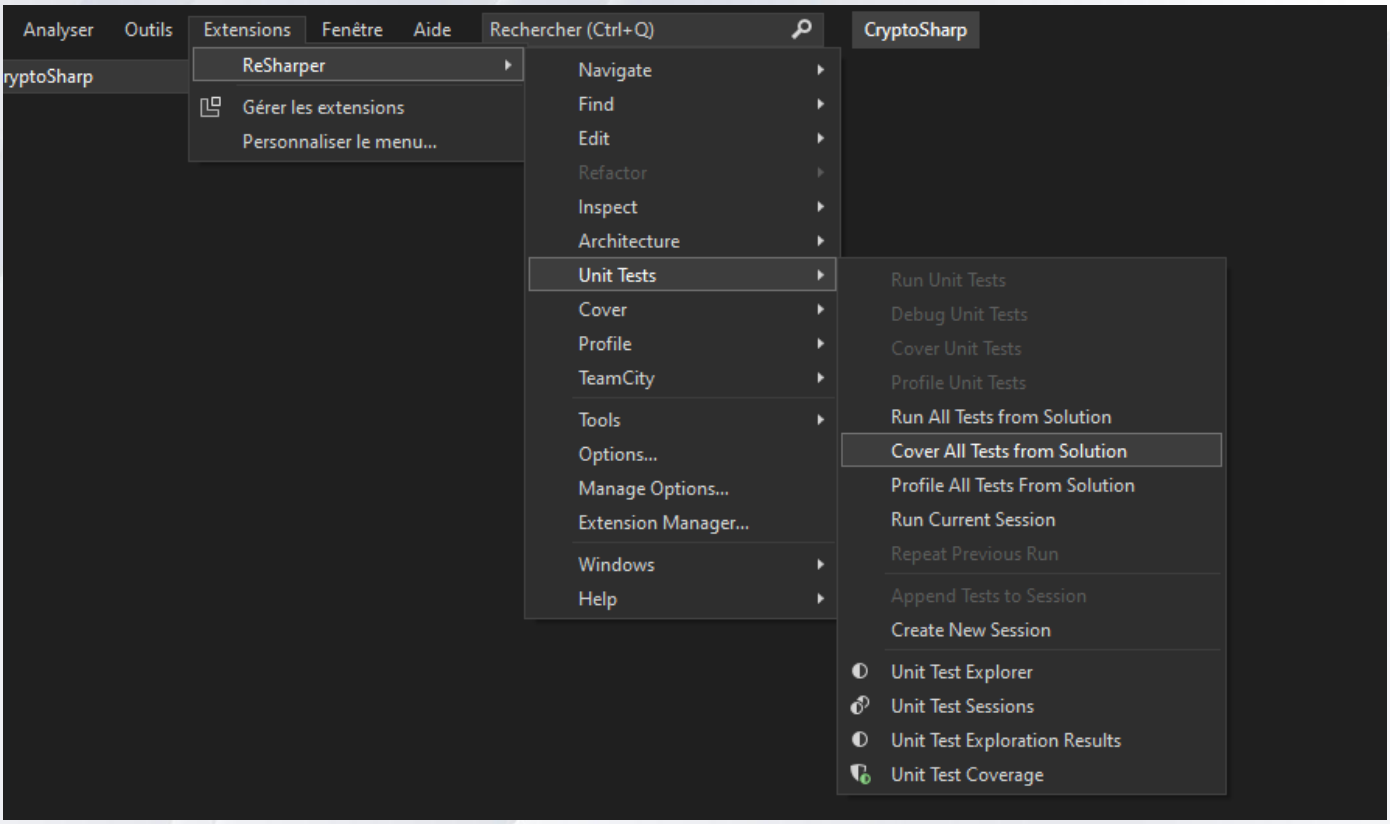


Figure 17 : Chemin de lancement de la couverture de tests dans Visual Studio avec l'extension ReSharper

Nous obtenons donc deux documents d'informations. Le premier présente l'exécution des tests et si les tests passent. Nous pouvons voir ici que nous avons 10 méthodes de tests différents dans la classe `Deck` et 4 dans `EncodeDecode`. Nous pouvons voir ici que tous les tests passent :

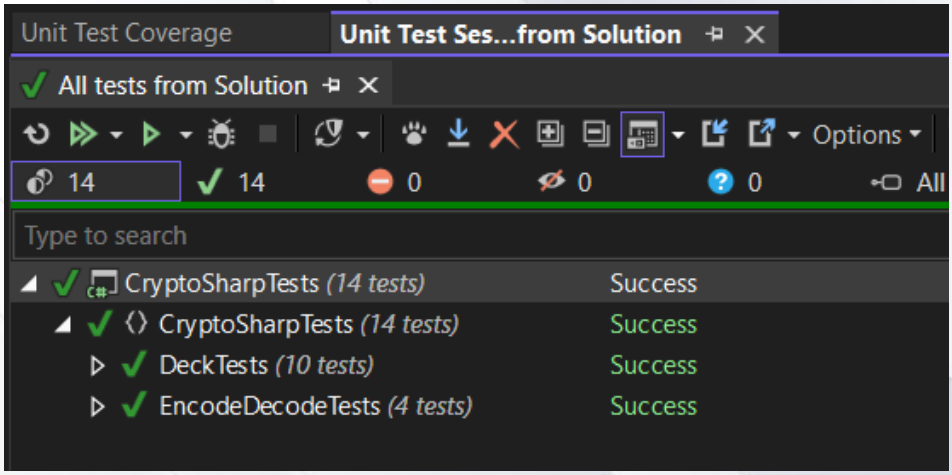


Figure 18 : Document qui montre si les différents tests passent

Cependant, c'est le deuxième document qui est vraiment important pour nous : l'`Unit Test Coverage`. Il présente une arborescence comme suit : Les projet → Les classes → Les méthodes. Ainsi pour chaque projets, classes et méthodes, on récupère l'informations de si tout est testé dedans. Prenons l'exemple d'une méthode avec un if puis else, il se pourrait qu'on ait testé que le if, ou bien que le else. Le code coverage nous dira donc le pourcentage de couverture de tests de chaque méthode.

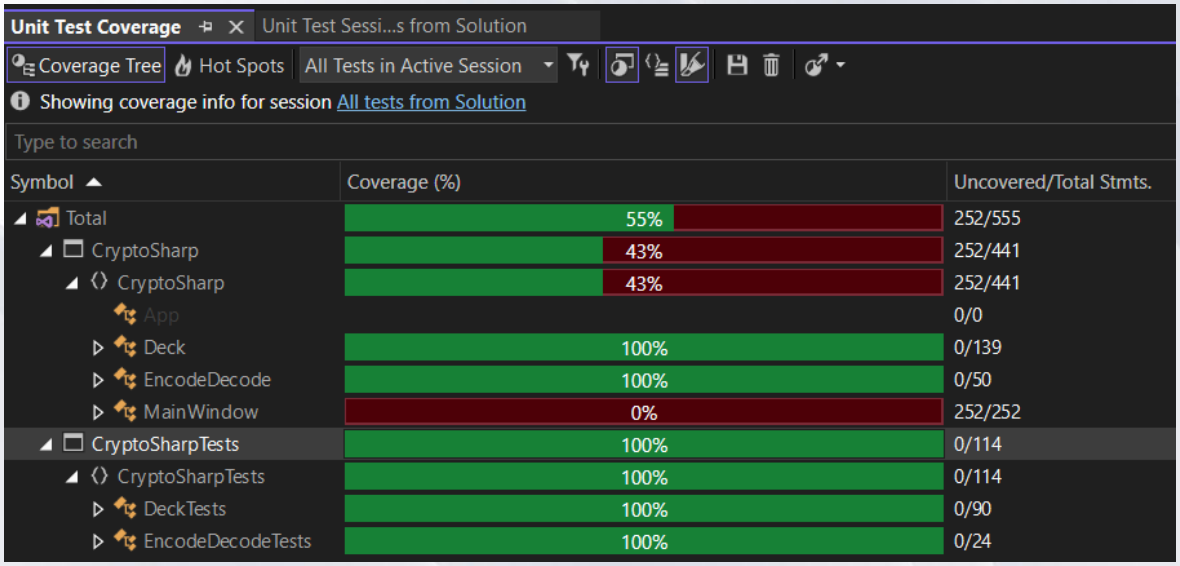


Figure 19 : Code Coverage de CryptoSharp

Nous voyons donc sur le document ci-dessus que le code coverage total est de 55%, ce qui est plutôt mauvais. Cependant, il correspond à des choix personnels. En effet, nous avons uniquement testé la partie « Métier » du projet et non pas la partie IHM. Ainsi, nous pouvons laisser le code coverage de `CryptoSharpTests` de côté car c'est notre projet de tests, il sera donc obligatoirement à 100%, et nous pouvons laisser de côté `CryptoSharp/MainWindow` qui est l'IHM (non testée). Il nous reste donc `CryptoSharp/Deck` et `CryptoSharp/EncodeDecode`, nos deux classes testées. Nous voyons que ces deux classes sont à 100%, ce qui veut dire que toutes leurs méthodes (précédemment vues dans ce rapport) sont totalement testées.

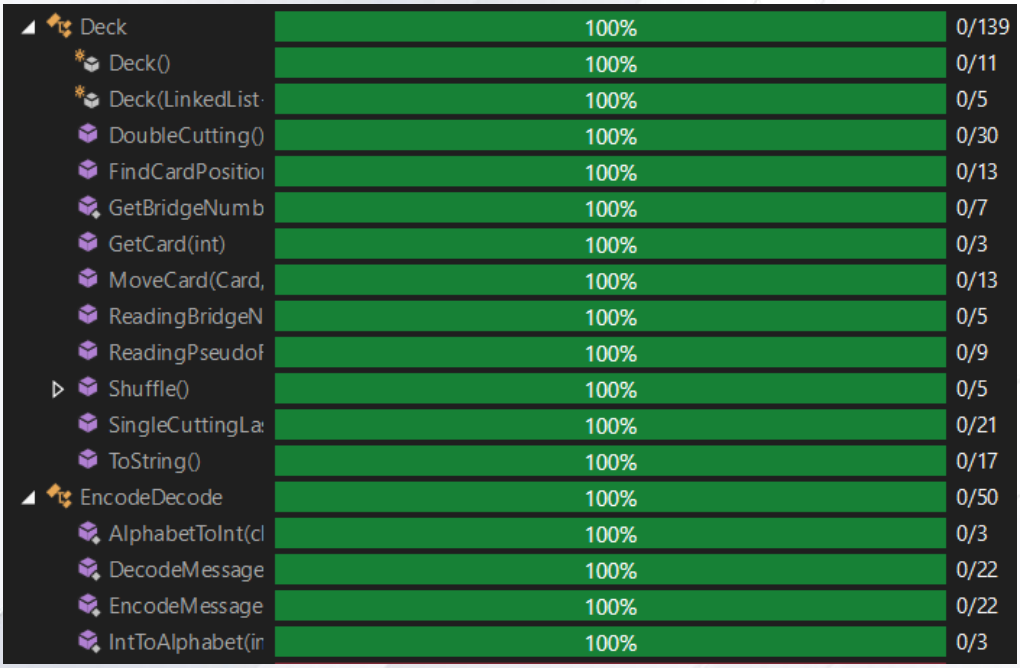


Figure 10 : Code Coverage de Deck et EncodeDecode

Comme vu précédemment, ici nous avons un code coverage de 100% sur chaque méthode, mais que ce soit pour montrer que tous les tests fonctionnent ou pour montrer quels tests ne sont pas complètements fonctionnels (s'il y a des lignes non testées), il est intéressant de créer un document de couverture de code. Vous le retrouverez à la racine puis dans `/Coverage/CryptoSharp.html`. En voici un court aperçu, il nous montre ainsi quelles lignes sont testées efficacement :

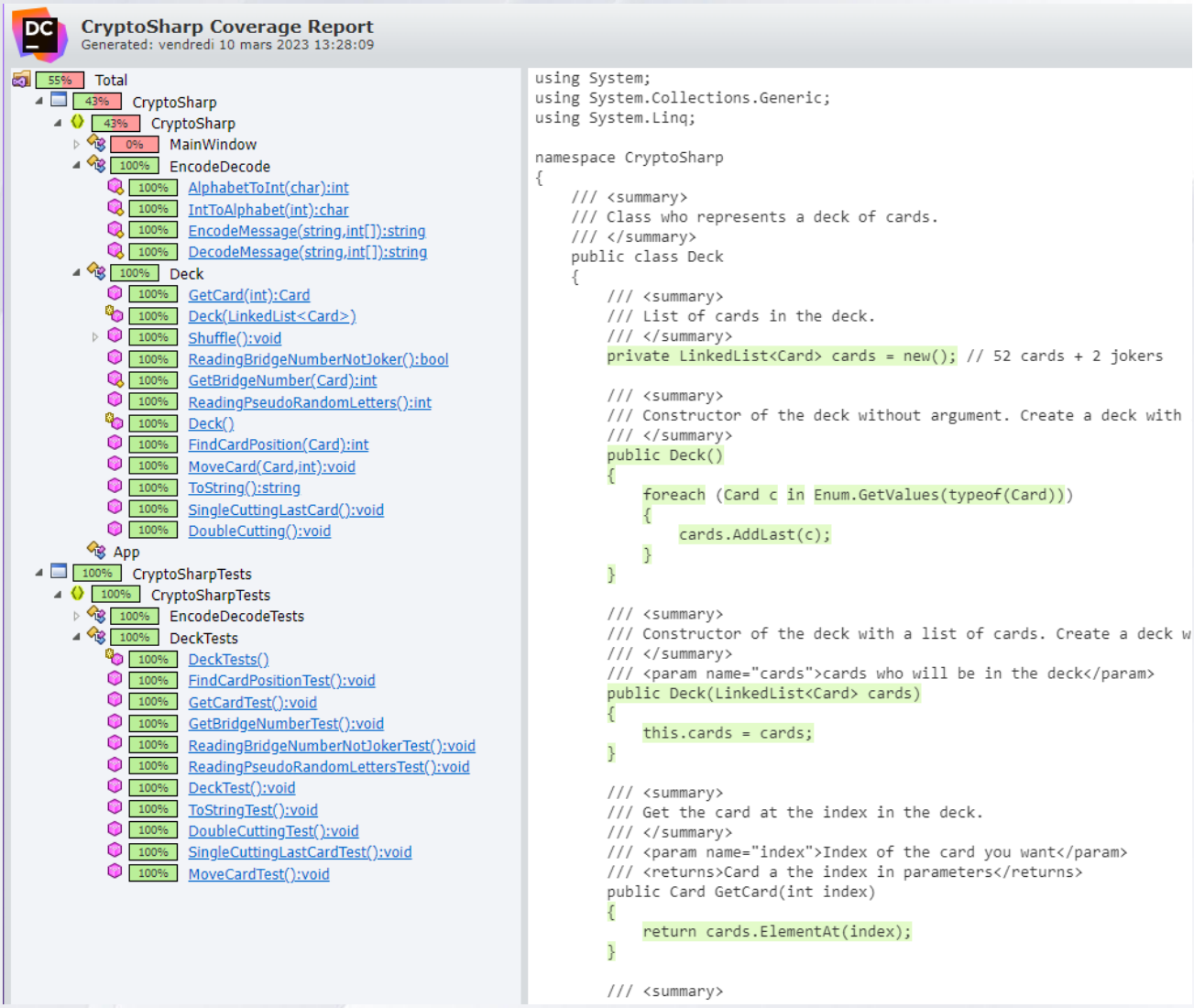


Figure 11 : Document HTML de code coverage de l'application CryptoSharp

Documentation Doxygen

L'autre amélioration du projet, c'est la documentation. En effet celle-ci est super importante. Elle permet tout d'abord de nous souvenir de ce que l'on a fait, d'à quoi sert une méthode ou une fonction, mais elle permet également à un développeur tiers de comprendre ce que l'on a fait. Notre projet étant Open Source et disponible à tous les développeurs et surtout à tout le monde, il est important que celui-ci soit complètement compréhensible. Tout est donc rédigé en Anglais (commits, documentations, variables, méthodes etc.), et une documentation Doxygen a été réalisée. Voici un exemple de commentaire permettant la documentation de notre projet :

```
/// <summary>
/// Methode to encode a message with a key generated by solitaire of Bruce Schneier.
/// </summary>
/// <param name="message">string message to encode composed only by letters (a-z)</param>
/// <param name="key">table of int key who permise to encode the message</param>
/// <returns>string encoded message</returns>
public static string EncodeMessage(string message, int[] key)
{
  ...
}
```

Figure 12 : Exemple de bloc de documentation C#

Une fois la documentation de chaque méthode de chaque classe de chaque projet réalisé, nous pouvons maintenant à l'aide de Doxygen et de notre fichier DoxyFile (fichier de configuration de la documentation Doxygen) générer un site HTML pour notre documentation. De plus après la génération de la documentation, le fichier « DoxygenWarnings.txt » nous donne les emplacements qui ne sont pas documentés. Ici le ce fichier est vide, tout est bien documenté.

Il nous reste plus qu'à lancer la documentation, celle-ci est disponible à la racine, puis `/Doc/html/index.html`. En voici un court extrait :

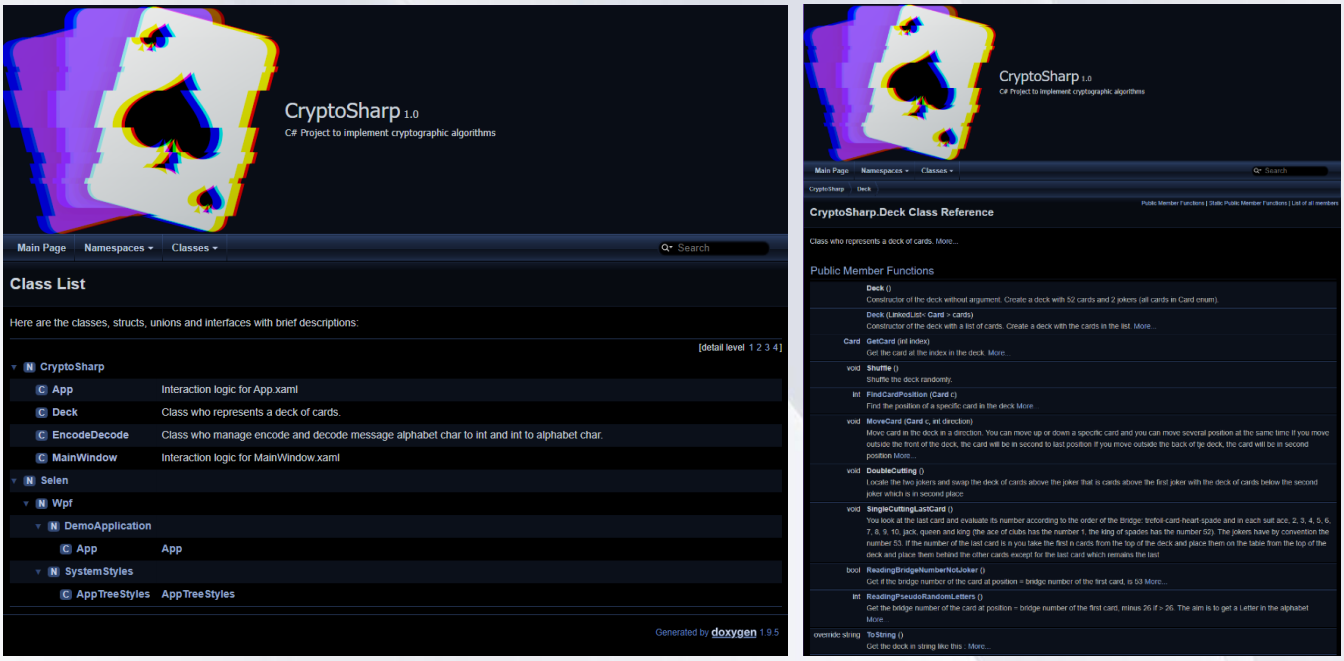


Figure 13 : Extrait de la documentation Doxygen de CryptoSharp

Conclusion

Pour démarrer cette conclusion, vous trouverez à coté de ce document, le projet Visual Studio, le document de code coverage, la documentation Doxygen, ainsi que le code compilé dans un exécutable tout prêt.

Tout d'abord, nous allons parler de notre ressenti sur ce projet. En effet, il est important pour nous de passer par la pratique pour comprendre un principe, des algorithmes, et plus encore, il est important pour nous de savoir à quoi sert réellement ce qu'on apprend. Le fait de passer par un projet, qui plus est, en langage libre nous a permis de réaliser l'algorithme et les principes vus en cours, de pouvoir, dans une démarche scientifique, expérimenter celui-ci.

Ce projet a été très enrichissant sur la partie développement logiciel. En effet, comme dit précédemment, nous nous destinons à un Master 2 ISL (Ingénierie et Systèmes Logiciels) à Besançon et c'est la base de ce Master. Nous avons donc pu prendre grand soin de faire un logiciel optimisé, bien codé, bien documenté et surtout bien testé. Toutes ces compétences nous seront utiles l'année prochaine et nous nous sommes surpris par exemple à réaliser des tests unitaires en en ayant réellement besoin, chose qui jusqu'ici nous paraissait superflu dans les cas d'études vus en cours.

Pour fini, le projet est open source et disponible à l'adresse suivante : https://github.com/Mahtwo/CryptoSharp. Merci d'avoir lu ce long rapport et nous espérons qu'il aura répondu à vos attentes et que tout y a bien été expliqué. Si des détails ne sont pas clairs ou si vous voulez des informations supplémentaires, nous sommes bien sûr disponibles via nos adresses électroniques étudiantes.