Projet info – Rapport Final

I. Structure des données

Le projet se compose de plusieurs classes, dont la principale est la classe MonImage. Chacune a son importance dans ce projet, et elles ont été conçues de telle façon à ce que chaque fonctionnalité soit le plus facile à manier, et que tout le code soit clair. Je vais donc dans cette partie m'appuyer sur la structure de chacune de ces classes, et l'expliquer. Je commencerai d'abord par expliquer 2 classes simples qui seront utilisées par la suite dans MyImage pour que tout reste clair. Les méthodes n'ayant pas servi en dehors d'une classe, ou non testées, n'ont pas été mises en « public ». Par habitude, je n'ai pas précisé « private » pour tout ce qui n'est pas public, car par défaut tout est en « private ».

1. Pixel

Cette classe est la plus simple, la plus courte, et la plus rapide à expliquer de tout le programme. Pour faire simple, sur une image, chaque pixel possède *3 intensités* de couleurs, pouvant chacune prendre une valeur de **0 à 255**. Il y a une intensité de rouge, une de vert, et une de bleu. Les champs et attributs de cette classe sont donc des variables codantes chacune de ces couleurs. Le type **byte** a été privilégié pour de nombreuses raisons, notamment le fait qu'une intensité de couleurs ne dépassera jamais 255 (tout comme un byte).
Au niveau des méthode, on n'en a que trois, une pour calculer la somme des intensités, une pour vérifier l'égalité de 2 pixels, utile pour certains tests, et une méthode ToString.
J'aurais pu ajouter quelques attributs comme Noir qui serait un pixel (0,0,0), Blanc (255,255,255), etc. mais j'ai préféré faire, lorsque j'en avais besoin, Pixel noir = new Pixel(0,0,0), plutôt que = Pixel.Noir par exemple.

2. Complexe

Cette classe sert à manipuler des nombres de l'espace complexe. A première vue, on peut se demander ce que vient faire cette classe ici, mais nous aurons l'occasion d'en reparler en évoquant les fractales. Tout ce qu'il faut savoir de cette classe, c'est qu'un **complexe** est désigné par une **partie réelle**, et une **partie imaginaire**, et que chaque opérateur de cette classe est appliqué pour faire un calcul de 2 complexes, classiques. Un attribut donne le module du complexe.

3. MonImage

Maintenant nous pouvons évoquer cette classe. Une instance de la classe MonImage possède beaucoup de champs qui lui sont utiles.

Son type, **typeImg**, précise s'il s'agit d'une image *bmp* ou autre. Finalement, le projet ne portera que sur des images *bmp*.
La taille de l'offset, **tailleOffset**, qui précise à partir de quel octet démarre le premier pixel dans l'encodage du pixel.
La **largeur** et la **hauteur**, de l'image.
Le nombre de bits par couleurs, **nbBits**, précisant sur combien de bits sont codés chaque pixel. Ici, le projet ne porte que sur des images *bmp* codées sur 24 bits, c'est-à-dire que chaque pixel est codé sur 24 bits. En effet, comme je l'ai évoqué plus tôt, chaque pixel est composé de 3 intensités allant de 0 à 255. Il s'agit de 3 octets. Un octet étant codé sur 8 bits, on a chaque pixel codé sur 24 bits. Grace à cela, on peut obtenir jusqu'à 16 millions de couleurs différentes par pixels. Si on le souhaite, on peut coder une image sur 8 bits, cela veut dire que chaque pixel a une couleur allant de 0 à 255, donc 256 couleurs différentes, mais cela a l'avantage de réduire considérablement la taille du fichier. Il faut alors préciser la palette de couleur utilisé, dans une partie du header du fichier. Ou mieux encore, coder un pixel sur 1 bit reviendrait à coder l'image sur 2 pixels, par exemple 0 = noir et 1 = blanc, et ainsi, chaque octet coderait 8 pixels. Malheureusement, je n'ai pas réussi à traiter autre chose que des images sur 24 bits.
La taille du fichier, **tailleFichier**, correspondant au nombre total d'octets que comprend le fichier. Elle se calcule de cette façon : tailleFichier = tailleOffset + nbBits * hauteur * largeur / 8, soit, en particulier pour les images 24 bits : tailleFichier = tailleOffset + 3 * largeur * hauteur.
La taille du header informatif, **headerInfo**, qui est importante à indiquer pour enregistrer l'image correctement.
La matrice de pixels, **image**, qui est une matrice d'éléments de la classe Pixel, dans laquelle sont conservés les données de chaque pixel. Elle a pour nombre de lignes la hauteur de l'image, et pour nombre de colonnes la largeur, c'est pourquoi, c'est pourquoi la matrice sera toujours de la forme **Pixel[hauteur, largeur]** et non **Pixel[largeur, hauteur]**, par analogie avec les matrices que l'on a l'habitude d'étudier. J'aurais pu évidemment faire une matrice de tableau de byte, mais le code aurait vite été lourd à lire, et c'est pourquoi il m'a semblé beaucoup plus pratique de créer la classe Pixel citée précédemment. Dans cette classe, je n'inverse pas la matrice. Les images se lisent du bas vers le haut, et les matrices se parcourent du haut vers le bas. Une fois enregistrée, la matrice se positionne automatiquement dans le sens inverse.
Le tableau des données de l'image, **imageByte**, est le tableau dans lequel sont stockées toutes les données de l'image sous forme d'octets. Il se récupère lorsque l'on crée une nouvelle instance de la classe à partir d'un fichier *bmp*, et est utile pour enregistrer les images.
Enfin, le **nom** est purement esthétique et n'apparaît même pas dans les constructeurs, c'est pour simplifier le nommage du fichier dans le programme principal lorsqu'on enregistre une nouvelle image.

Au niveau des constructeurs, il en existe 3. Le premier prend en paramètre le nom d'un fichier, et récupère toutes les données à partir du tableau d'octets codant le fichier, grâce à certaines méthodes de conversion notamment. Le 2^{e} est un constructeur par copie, il recopie toutes les données d'une autre instance de MyImage entrée en paramètre. Enfin, le 3^{e} est un constructeur d'une image « vide », entièrement blanche, dont on définit la largeur et la hauteur en paramètres. On le définit par défaut avec les attributs les plus couramment utilisés, soit **tailleOffset = 54**, **nbBits = 24**, **headerInfo = 40**. Enfin, on remplit la matrice de pixels par des pixels blancs (255,255,255).

Concernant les attributs rendus publics, ils ne sont tous quasiment utilisés que pour effectuer des tests unitaires (sauf le nom pour le programme principal, et la matrice de pixels pour les QR Code).

Le but de ce rapport n'étant pas d'expliquer chacune des méthodes, les commentaires et le balisage XML étant là pour ça, je ne vais pas m'attarder sur chacune des méthodes. La première permet d'**enregistrer** une image en faisant le schéma inverse du 1er constructeur qui récupère un fichier image. Les 4 méthodes suivantes sont des méthodes de **conversions** dans les 2 sens, binaire, en little endian. Pour les conversions en binaire, on peut

noter que les séquences binaires sont des tableaux de booléens. C'est un choix que j'ai fait car, un bit peut prendre 2 états, tout comme les booléens, alors pourquoi pas. De plus, ces méthodes seront utiles pour cacher une image et récupérer une image (**sténographie**), donc la manipulation des tableaux va s'avérer plus évidente que celle d'une chaîne de caractères.
Pour l'**histogramme**, j'ai voulu que le ratio largeur/hauteur soit égal à 16/9 pour que ça reste joli, d'où le petit calcul au début de la méthode. Pour ne pas avoir une image trop petite, j'ai pris comme **largeur** 3*256 qui me semblait bien. Le but est que, pour chaque niveau d'intensité existant, entre 0 et 255, on compte le nombre d'occurrences, et qu'on les affiche sous la forme d'un histogramme, avec le plus haut pic ayant une influence sur la **hauteur** de l'image. J'avais d'abord fait un algorithme qui parcourait 255 fois tous les pixels pour compter les occurrences, mais j'ai changé ça par la suite pour baisser la complexité temporelle. Les histogrammes des images 1080p pouvaient prendre plus d'une minute à s'afficher… J'ai donc inversé l'ordre des boucles pour ne parcourir qu'une seule fois la matrice de pixels. Ça reste long pour les grosses images (~10s), mais beaucoup plus rapide ! Une méthode ne sert donc plus à rien (mais j'aime laisser des traces de mon ancien code).
Pour les autres méthodes, elles sont très bien expliquées en commentaire.

4. QRCode

Concernant les **champs** de cette classe, il y en a beaucoup et ils sont tous expliqués en commentaires // sur le code.

Il y a 2 constructeurs, un pour **créer** le QR Code et un pour le **décoder**. J'ai choisi de faire le générateur et le lecteur de QR Code dans la même classe car on réutilise tous les mêmes champs dans les 2 cas. Les méthodes sont séparées en 2 parties, la 1ere pour le générateur et la 2e pour le lecteur. Pour générer un QR Code, on a un constructeur qui fait appel à plusieurs fonctions pour encoder le message entré en paramètre, et une autre méthode que l'on appel après pour tout mettre sur une matrice et convertir par la suite instance de la classe MonImage.

Pour le lecteur, tout se fait dans le 2e constructeur. Je ne voyais pas de réel intérêt à faire autrement, appeler les méthodes une à une dans le programme principal n'aurait pas été fortement plus utile. Le constructeur prend en paramètre le nom de fichier d'une image *bmp* dans laquelle est codé un QR Code, j'aurais pu mettre directement comme paramètre une instance de la classe MonImage, mais ça revient au même de la créer directement dans ce constructeur, et la méthode **ToString** renvoie toutes les informations le concernant, dont la phrase codée évidemment. Le petit problème est que je n'ai pas trop géré les cas où les QR Code seraient encadrées par exemple, donc à moins de rogner, il est difficile de décoder un QR Code téléchargé sur internet (ça reste possible), mais cela marche pour tous les QR Code que vous pourrez créer dans ce programme. De plus, il est même possible de dessiner (et cacher jusqu'à 30% des pixels du QR Code selon le niveau de correction) sur les images et de **décoder** tout de même, avec le lecteur comme avec n'importe que scanner sur mobile.

Dans cette classe, on travaille beaucoup en binaire, et contrairement à la classe MonImage, j'ai décidé d'utiliser des chaines de caractères ici, pour écrire les bits, car les méthodes s'y prêtaient plus. J'ai préféré créer moi-même ces méthodes de conversion plutôt que les méthodes de conversion **Convert.ToInt32** ou **Convert.ToString**, pour m'exercer à faire des **méthodes récursives**, et aussi pour pouvoir entrer le nombre de bits sur lequel je souhaite coder chaque entier.
Beaucoup de fonctions sont en void, très peu retournent des valeurs. C'est un choix que j'ai fait de modifier les champs directement dans les méthodes plutôt que de faire des méthodes qui retournent quelque chose à chaque fois. Ça a ses avantages et ses inconvénients, notamment au niveau des tests, c'est vrai.
Je reviendrai sur la classe QRCode dans la partie innovation car le boulot que j'y ai fourni est plutôt grand.

II. Innovations

1. Fractales

Comme je le disait précédemment, je pense avoir fourni un peu plus de travail que nécessaire concernant les fractales. Lorsque je me suis intéressé au sujet, j'ai vite trouvé ça intéressant, et j'ai voulu essayer. Cependant, les deux algorithmes étaient assez simples à faire, j'aurais même pu me passer de la classe Complexe, et le résultat n'était pas aussi satisfaisant que ce que je voulais. Une fractale en noir sur fond blanc, avec impossibilité de se déplacer comme c'est possible de le faire sur certains logiciels, j'ai voulu un peu améliorer ça. Ce n'est pas grand-chose, mais je voulais quand même l'ajouter.

Pour créer une fractale, l'utilisateur entre de nombreux paramètres que je vais détailler.
- **Complexe c**. Il s'agit du complexe à entrer pour tester la convergence de la suite complexe suivante : $Z_{n+1} = Z^2_n + c$ (je reviens sur cette histoire de convergence plus tard). Celui-ci ne doit être entré par l'utilisateur que pour créer une fractale de Julia. Il existe plein de fractales de Julia, et celles-ci dépendront de ce paramètre. Initialement, j'avais prévu de créer 2 méthodes différentes pour les fractales de Julia, et celle de Mandelbrot, mais leurs algorithmes n'étant pas si différents, j'ai décidé de tout mettre dans une seule méthode. Donc, pour créer une fractale de Mandelbrot, il suffit de définir **c** comme *null*. Si **c** est *null*, il sera défini plus tard selon chaque point du plan complexe. Là où **c** est fixé et Z_0 variable pour créer une fractale de Julia, **c** sera variable et Z_0 fixé pour une fractale de Mandelbrot.
-**float zoom.** C'est le **zoom**, en *pourcentage*, de la fractale. Il permet à l'utilisateur de zoomer ou dézoomer sur la fractale. Le zoom optimal est établi à zoom = 50. D'ailleurs, dans la Console, si l'utilisateur n'entre rien, juste *ENTREE*, le **zoom** par défaut sera établi à **50** par le programme principal. C'est un réel et non un entier car on le divise après par 100.
-**float centreRe** et **float centreIm.** Ce sont nos deux paramètres qui vont nous permettre de nous déplacer sur la fractale. **centreRe** est l'emplacement de l'axe des réels par rapport au centre de l'image, et **centreIm** est l'emplacement de l'axe des imaginaires par rapport au centre de l'image. Si par exemple on donne **centreRe = 10** et **centreIm = -50**, le centre de l'image sera à **[10, -50]** sur le plan complexe, ou encore, le centre de la fractale (ou du plan complexe) aura décalé de *10 vers la gauche* et de *50 vers le haut*, ou bien encore, le cadrage de l'image se sera déplacé de *10 vers la droite* et de *50 vers le bas*. Si ce n'est pas clair, il suffit de tester sur la console (il est possible de continuer de se déplacer, et de zoomer sur la même fractale autant de fois que vous voulez sans avoir à devoir rentrer tous les paramètres à chaque fois. Le seul inconvénient est qu'à chaque fois il faut attendre que les changements soient enregistrés sur l'image et que l'image s'ouvre, on perd 5 secondes à chaque fois). Par défaut dans le programme, ces 2 paramètres seront initialisés à **0**, si l'utilisateur n'entre rien et met *ENTREE* directement. C'était vraiment cette idée de déplacement que je voulais ajouter aux fractales, mais il manquait encore quelque chose : de la couleur.

-**int central** et **int nuance**. Il s'agit là de deux entiers codant pour 7 ou 8 couleurs différentes. Ces couleurs seront proposées dans l'interface de la Console, et seront appliquées à l'aide de la méthode **ColorerFractales**, qui applique tout simplement un switch, rien de bien compliqué, et purement esthétique. Pour **central**, il s'agit de la couleur des points centraux (idéalement noir) pour les quels la suite complexe est convergente, et **nuance**, il s'agit de la couleur entourant la fractale, qui est plus ou moins intense selon le temps qu'a mis la suite à diverger. Par défaut, si l'utilisateur met *ENTREE* dans la console, les points convergents seront noirs, et les points divergents seront blancs, soit pas de nuance.
-**int max**. Il s'agit du nombre d'itérations maximum pour déterminer si la suite complexe converge. Plus ce nombre est grand, plus la fractale sera précise. On peut le voir notamment avec la fractale de Julia pour *c = 0,285 + 0,01i*, elle n'est pas vraiment la même si **max = 50** ou si **max = 100** par exemple. Cependant, plus ce nombre d'itérations est grand, plus le calcul est long, et ça fait déjà beaucoup de paramètres à entrer, donc finalement, il reste **par défaut** à **50** et n'est même plus demandé d'entrer par l'utilisateur.

L'algorithme n'est pas compliqué mais je vais tout de même l'expliquer dans cette partie pour que ça soit clair, on place le centre du plan complexe au centre de l'image, avec un décalage : **largeur**/2 – **centreRe**/2, pareil pour **hauteur**. On divise alors le **zoom** par 100 puis on définit un booléen pour voir si la fractale est une fractale de mandelbrot ou non, si **c** est *null*.
Ensuite on parcourt la matrice de pixels, et pour chaque Pixel, on définit une **partie réelle** et une **partie imaginaire** selon sa place dans le plan complexe, et on divise ça par **zoom*hauteur**, que ce soit pour la partie réelle ou imaginaire. En effet, le fait de choisir un même coefficient pour diviser avec le zoom (ici hauteur) permet de ne pas aplatir la fractale. On pour le voir avec la fractale de Julia pour *c = 0 + 0i*, il s'agit d'un cercle parfait. Si j'avais divisé la partie réelle par la hauteur, et la partir imaginaire par la largeur, et qu'on affiche la fractale sur une image non carrée, on voit alors une ellipse aplatie. Comme dit précédemment, pour une fractale de Mandelbrot, **c** varie selon chaque point du plan complexe, il aura donc pour parties réelles et imaginaires les 2 nombres précédemment créés, et la première itération de Z est définie à 0. Au contraire, pour une fractale de Julia, la première itération, Z_0, aura pour partie réelle et imaginaire les 2 nombres créés, et **c** vaudra le complexe entré en paramètres.
Ensuite, on applique **max** fois le calcul : z = z*z + c, à l'aide des opérateurs créés dans la classe complexe. Si z diverge avant le nombre max d'itérations, on quitte la boucle avant. Un calcul mathématique permet rapidement de démontrer que la suite diverge si le **module de z** est supérieur ou égal à 2, donc on considèrera Z_n comme divergent dans le cas où son module sera supérieur à 2. Ensuite, si la suite a convergé, c'est-à-dire si le compteur est égal à **max**, on colorie le pixel de la couleur **central**. Sinon, on crée une intensité proportionnelle au compteur par rapport au **max**, soit le nombre d'itérations qu'a mis z a diverger, et on remplit de cette intensité le pixel de la couleur **nuance**.
C'est dans le fait de pouvoir choisir *l'emplacement précis*, le *zoom* et la *couleur* que je considère cet algorithme comme une innovation.

2. Sténographie

Il n'y a pas grand-chose à dire sur ça, et je n'avais même pas envie de placer ça dans innovation tellement c'est subtil, mais j'ai mis du temps à avoir cette idée pour cacher une image dans une autre de **taille différente**.
Pour faire simple, on convertit en *séquence de 8 bits* chaque valeur des pixels, et on met les 4 bits de poids fort de la 2e image à la place des 4 bits de poids faible de la 1ere image (en jouant avec les divisions entières pour les images de tailles différentes), et on fait l'action inverse pour décoder l'image, en mettant tous les bits de poids faible de a 2e image à 0. Evidemment, on a une baisse de qualité, mais celle-ci n'est pas énorme !
Je voulais cacher la **hauteur** et la **largeur** de l'image cachée quelque part afin de la récupérer, j'avais pensé au *header* éventuellement, puis j'ai eu une petite idée : convertir la **hauteur** et la **largeur** dans un tableau de 3 bytes en little endian, puis poser ces 2 fois 3 valeurs dans les **2 premiers pixels** de l'image, ce qui ne se voit à peine à l'œil nu. Le résultat satisfaisant.
J'avoue cependant avoir partagé cette astuce à 2 ou 3 connaissances, sans être vraiment précis sur l'astuce, en disant juste que l'on peut placer des informations dans les pixels par exemple. Je n'ai jamais fourni le moindre code ou écrit sur l'ordinateur de personne pour aider les autres.

3. QR Code

Là encore, il s'agit de quelque chose du sujet sur lequel je suis allé plus loin que prévu, mais je pense que ça mérite aussi sa place dans innovation. Je ne vais pas réexpliquer comment est encodé un QR Code en alphanumérique pour un niveau de correction L, car tout est très bien expliqué dans le sujet, mais je vais m'attarder sur certains points.

Tout d'abord, les innovations sur le QR Code par rapport au sujet sont les suivantes, mon QR Code peut **générer** et **lire** :
-Un texte en *alphanumériques* et en *UTF-8* (reconnaissance automatique), alors que le sujet ne demandait que alphanumérique.
-De la *version 1* à la *version 4*, alors que le sujet demandais uniquement les versions 1 et 2.
-Tous les *niveaux de correction (L,M,Q,H : 7% à 30%)*, le sujet ne demandait que 7% (L).
-Tous les *masques parmi les 32 existant* (8 pour chaque niveau de correction), avec *détection* du meilleur masque à appliquer si l'utilisateur le souhaite. Dans le sujet, seul le masque 0 pour le niveau de correction L était demandé.
-Enfin, j'ai pu arriver jusqu'au bout malgré quelques coquilles dans le sujet (pas d'explications sur le masque, sur l'algorithme de Reed-Solomon, le QR Code généré peut être lu par n'importe quel scanner alors que ce n'était pas forcément demandé). Ce sont toutes ces recherches qui m'ont poussé à aller aussi loin dans l'élaboration du QR Code.

Pour générer un QR Code, l'utilisateur entre son texte et son niveau de correction. La taille des données à encoder, ou de correction, la version, vont dépendre de la taille du texte, des caractères (*UTF-8* ou *alphanumérique*), et du **niveau de correction**. Pour déterminer ça, j'ai fait une méthode avec un grand switch et des conditions, ça fait lourd au niveau de l'écriture, mais je ne voyais pas comment faire autrement, j'ai fait au mieux !
En *alphanumérique*, l'indicateur de mode est **0010**, le nombre de caractères est codé sur **9 bits**, et on prend les lettres **2 par 2**, qu'on code sur **11 bits**, avec un **poids** allant jusqu'à **44** pour chaque lettre, sauf si le nombre de lettres est **impair**, dans ce cas, on code la dernière lettre sur **6 bits**.
En *UTF-8*, l'indicateur de mode est **0100**, on code la taille du texte sur **8 bits**, et chaque lettre est codée sur un **octet** (**8 bits**). Les phrases sont alors plus lourdes qu'en *alphanumériques*, c'est évident. Mais l'encodage est plus facile à faire.
Après avoir ajouté les **terminaisons** manquantes (cf le sujet), il faut encoder les données avec l'algorithme de *Reed-Solomon*. Pour cela, on doit convertir tout le code binaire en une séquence d'octets (1 octet tous les 8 bits), et encoder le message, avec la méthode :
ReedSolomonAlgorithm.Encode(byte[] message, int ecc, ErrorCorrectionCodeType eccType),
avec eccType pouvant prendre la valeur 0 (= ErrorCorrectionCodeType.QRCode) ou 1 (= ErrorCorrectionCodeType.DataMatrix). Comme nous travaillons avec des **QR Codes**, ici le **3e paramètre** prend la valeur **0**.
Pour utiliser cet algorithme qui était fourni, j'ai dû enlever une ligne dans *ReedSolomonEncoder*, la condition ligne 94, qui posait un problème pour

certains niveaux de correction, lorsque le nombre de données à coder est supérieur au nombre de données pour la correction (ça arrive). Cependant, pour certaines versions, le message ainsi que le code d'erreur sont divisés en blocs. Par exemple, pour la **version 3-Q** :

Version and EC Level	Total Number of Data Codewords for this Version and EC Level	EC Codewords Per Block	Number of Blocks in Group 1	Number of Data Codewords in Each of Group 1's Blocks
3-Q	34	18	2	17

On voit que le nombre d'octets à encoder est de **34**, et **18** pour la correction d'erreurs. Cependant, si on regarde bien les 2 dernières cases, on évoque **2 blocs** et **17 octets** dans chaque blocs. Cela veut dire que nous n'allons pas corriger **34 données**, mais **2 * 17 données**.

Faisons un exemple avec des numéros. Imaginons que je doive appliquer la correction d'erreurs à la séquence d'octets *0123456789*, où chaque chiffre représente grossièrement un numéro, en **2 blocs**, où chaque bloc contient **5 octets**, avec **3 octets** de EC (error Code). On va d'abord appliquer l'algorithme sur la séquence *01234*, admettons qu'elle retourne *ABC*, où chaque lettre est également un octet, et ensuite on fait pareil avec *56789*, admettons que cela retourne *DEF*. On a alors naïvement envie de mettre le code final en binaire dans l'ordre suivant : *0123456789ABCDEF*. Mais non, c'est trop facile comme ça ! On va alterner selon le nombre de blocs que l'on a ! Ici, c'est 2 blocs, donc on va faire *0516273849*, et pareil pour la correction d'erreur, *ADBECF*, soit au final *0516273849ADBECF*.

Pour coder *123456789* en **3 blocs** de **3**, avec un EC de **2 octets**, on fera *123* => *AB*, *456* => *CD* et *789* => *EF*, l'ordre sera *147258369ACEBDF* (alternance une fois sur 3). Que ce soient pour le générateur ou le lecteur, les fonctions faisant intervenir les blocs m'ont pris beaucoup de temps ! Mais j'ai fini par avoir un algorithme fonctionnel !

Pour placer les éléments dans le QR Code, j'ai initialisé une matrice remplie de 100 (en affichant l'image, je pouvais vérifier quels pixels avaient été placés, 100 affiche une case grise), j'ai placé les motifs et le masque petit à petit, pour qu'il ne reste plus que le nombre exact de cases égale à 100 que le nombre de bits dans le code en binaire. Pour placer chaque bit, il faut appliquer un **masque**. J'ai créé 4 algorithmes à partir de ce que j'ai pu trouver sur internet qui comptent un certain nombre de *pénalités* en fonction des couleurs, et qui teste sur tous les masques appliqués pour voir lequel est le meilleur (celui qui donne le moins de pénalités). Pour les versions **2 à 6**, il faut ajouter **7 « 0 »** à la fin du code pour remplir entièrement la matrice.

Pour le **décodage**, le constructeur récupère une image, et vérifie la **taille** du 1er motif de synchronisation. En effet, je peux avoir enregistré des QR Code de même version avec des tailles différentes, il faut s'adapter et trouver le **coefficient** d'agrandissement. La plupart des méthodes sont exactement les mêmes, redéfinies pour être appliquées à l'inverse. On commence avec une image, on termine avec le texte décodé. Pour obtenir le numéro de la **version**, on regarde la **dimension** de la matrice. Pour obtenir le **niveau de correction**, on récupère le **masque** (en plusieurs fois au cas où il serait erroné à certains endroits), puis on regarde les 2 premiers chiffres du masque pour obtenir le niveau de correction. On vérifie le masque à partir d'un algorithme en brut. Ensuite je remplace tout ce qui n'est pas le code binaire par 100, pour récupérer le code après, de la même façon que pour le poser, en appliquant le **masque** de la même façon. De même, il faut faire attention aux **blocs** et réadapter le code en fonction du nombre de **blocs** si celui-ci est supérieur à 1

III. Problèmes rencontrés

Pas tant de problèmes rencontrés que ça, il a juste fallu m'armer de patience et approfondir mes recherches pour en arriver là. Je peux tout de même noter le problème du temps qui m'a empêcher d'avancer sur l'interface WPF. Je la laisse quand même dans le dossier même si elle est inutilisable, tous les projets de fin d'année sont arrivés en même temps, et je n'ai pas de semaine de révision, je suis complètement déçu de ne pas avoir pu faire cette interface, mais le temps m'a vraiment manqué !

Petits problèmes pendant l'avancement du projet à cause d'inversions de hauteur et largeur par exemple, ou d'inversion de couleurs… Parfois je me rendais compte d'erreurs beaucoup plus tard, mais j'ai toujours fait au mieux pour corriger ça.

On m'a beaucoup demandé de l'aide autour de moi, et il est parfois difficile de dire non, mais l'aide que j'ai apporté était uniquement des explications supplémentaires, des schémas, ou des liens vers d'autres pages non mentionnées dans le sujet qui ont pu m'aider. Pas de code, pas de capture d'écran, et pas d'écriture sur l'ordi des autres, peut-être une ou deux astuces simples données à une ou deux personnes de temps en temps (par exemple le fait que la largeur doit être multiple de 4).

IV. Autocritique

Même si je suis allé très loin dans ce que j'ai fait, je suis déçu sur certains points. Le premier étant que je n'ai pas pu faire l'interface WPF en XAML. Je savais exactement ce que je voulais faire, mais le temps m'a malheureusement manqué (j'étais persuadé que j'irai jusqu'au bout). J'aurais voulu également pouvoir traiter d'autres images que des 24 bits, par exemple pour la méthode qui transforme en nuance de gris, ou pour les fractales, j'aurai pu enregistrer ces images sur 8 bits, car dans l'encodage des données, les pixels se répètent 3 fois à chaque fois, donc beaucoup d'informations inutiles. Pareil pour les fractales, très peu de couleurs utilisées j'aurais pu enregistrer les images sur 8 voire 4 bits. Pour les conversions d'images en noir et blanc, cela aurait pu être codé sur 1 bit, pour conserver énormément de place dans l'encodage de l'image, notamment pour les QR Codes. Mais pareil il manquait de temps je n'ai pas pu m'en occuper. Enfin, j'aurais voulu m'intéresser davantage aux méthodes de compression d'images.

Au niveau du QR Code, c'est pareil j'aurais pu aller plus loin que la version 4, et surtout, je trouve certaines méthodes un peu lourdes, notamment celle qui donne les informations sur la version, le nombre de données à encoder, etc., qui est un énorme switch mêlé à plein de conditions, c'est un peu dommage, ou bien d'autres endroits où je trouve le code un peu lourd.

A part ça, je suis très content d'avoir travaillé sur ce sujet, je pense avoir beaucoup progressé en informatique notamment grâce à ça, et j'ai aimé aller aussi loin.

Bien que j'aie toujours une certaine préférence pour les tests en brut sur le programme principal, à mettre des Console.WriteLine un peu partout ou à lire l'explorateur de variables en mode Debug, je dois avouer que les tests unitaires ont pu m'aider à découvrir des erreurs de codes que je n'aurai sûrement pas découvert autrement, sur des vieilles méthodes dont j'ai pu modifier un paramètre à un moment donné.

Je pense que beaucoup se plaindront de comment sont organisées les séances, qu'on n'a pas forcément l'impression d'être aidés en cours, mais je comprends qu'on ait voulu nous laisser faire tout en autonomie, c'est comme ça qu'on comprend le mieux. Certains disaient « on reçoit plus d'aide des autres gens de la classe que du prof ce n'est pas normal », mais si c'est normal, c'est avec l'entraide que l'on progresse le mieux, on n'a pas un « professeur » mais un encadrant c'est normal, mais je comprends que certains aient pu se sentir perdus à un moment ou à un autre.