# Rapport de stage

Design d'un nouveau type de code-barres 2D

Richard Dumoulin

Mai 2012

## Table des matières



## Introduction

De nos jours, la plupart des gens sont en possession d'un smartphone et le monde devient de plus en plus mobile. L'accès à internet se fait depuis n'importe quel lieu et les gens ont le besoin de tout savoir, tout de suite. On commence à constater l'apparition, dans nos pays occidentaux, de *codes-barres à deux dimensions*. Ces matrices de pixels contiennent une certaine quantité d'informations. Une fois scannés avec un smartphone, ils sont décodés et dévoilent alors l'information qu'ils contiennent : que ce soit un texte, une URL, une carte de visite, etc.

Ce rapport de stage décrit un travail dont le but était de proposer un nouveau type de code-barres à deux dimensions. La tâche consistait à proposer une spécification précise et modulable d'un nouveau format de code-barres 2D. Seuls les aspects purement théoriques ont été pris en compte, et aucune implémentation n'a été réalisée.

La première section reprend le cahier des charges de ce projet, tel que commandé par le maitre de stage. La seconde section développe le format du code-barres 2D proposé dans le cadre de ce travail. Enfin, la dernière section conclut ce stage et propose des pistes pour des éventuelles futures extensions du format de code-barres 2D proposé.

# 1 Cahier des charges

La demande formulée par le maitre de stage consistait à développer un nouveau format de code-barres 2D. Seuls les aspects théoriques sont considérés dans ce travail. Cette section présente le cahier des charges sous ses trois aspects que sont les fonctionnalités de la solution attendue, les performances espérées ainsi que les contraintes.

## 1.1 Fonctionnalités

- Le format doit proposer différentes tailles de code-barres.
- Il doit être possible d'encoder du texte anglais, français et des URL.
- Le code-barres 2D doit proposer un mécanisme de correction d'erreurs en cas d'altérations. Autrement dit, il doit être robuste.
- Le format proposé doit être configurable et extensible.

## 1.2 Performances

- Les opérations d'encodage et de décodage doivent prendre moins de deux secondes pour des tailles raisonnables.
- La taille nécessaire pour encoder un message donné doit être aussi faible que possible.

## 1.3 Contraintes

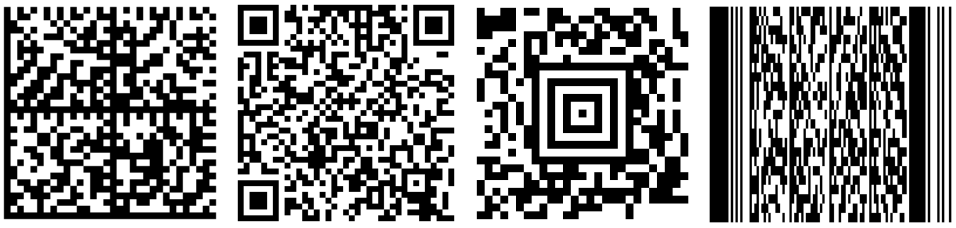
- Le code-barres imprimé doit avoir une surface raisonnable, tout en permettant sa lecture avec des lecteurs optiques standards.

# 2 Codes-barres 2D

Il existe actuellement plusieurs formats de code-barres 2D qui ont été développés. La figure 1 montre quatre d'entre eux, parmi les plus courants. Un code-barres 2D est une manière compacte de représenter de l'information. Cette information est encodée sous forme d'une matrice de pixels. De plus, ces codes-barres étant destinés à être placardés à divers endroits, pouvant être sensibles comme sur un colis postal par exemple, il est important qu'ils soient *robustes*. Cela signifie que si le code-barres est altéré, de manière raisonnable, il doit quand même être possible de lire l'information s'y trouvant ou au moins de signaler qu'il n'est pas valide.

## 2.1 Opérations

Essentiellement, comme l'illustre la figure 2, deux opérations sont possibles à partir de code-barres 2D. Tout d'abord, en partant d'un code-barres 2D, on obtient une matrice



(a) DataMatrix (b) QR Code (c) Aztec (d) PDF 417

FIGURE 1 – Exemples de codes-barres 2D courants.

de bits par la phase de *lecture* du code-barres. Ensuite, cette matrice est *décodée* pour retrouver l'information qui se trouve dans le code-barres (les flèches descendant sur la droite). Ensuite, en partant d'une information, qui va tout d'abord être *codée* en une matrice de bits, on peut obtenir un code-barres 2D par la phase d'*écriture* (les flèches remontant sur la gauche).

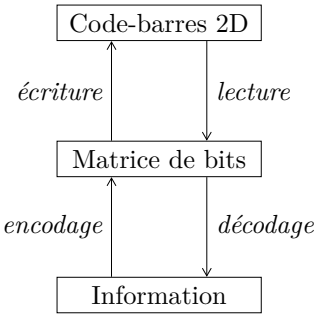



FIGURE 2 – Codage et décodage d'un code-barres.

Pour en revenir aux différents aspects concernant l'information stockée dans un code-barres, il y en a plusieurs qu'il convient d'identifier et définir lorsqu'on développe un nouveau format de code-barres 2D :

1. **Informations de configuration :** certains codes-barres 2D supportent plusieurs options et les valeurs de ces options sont alors encodées dans le code-barres 2D ;
2. **Informations de données :** il s'agit ici à proprement parler des données qui doivent être encodées dans le code-barres 2D ;
3. **Information de contrôle :** ces informations permettent de se rendre compte si le code-barres 2D est bien valide et n'a pas été altéré ;
4. **Information redondante :** une partie de l'information du code-barres 2D est dupliquée, et cela afin d'être robuste aux éventuelles altérations.

## 3 Spécification du format CB2D

Cette section décrit le format « CB2D » qui est celui proposé dans le cadre de ce travail. La description du format va se baser sur les quatre types d'informations présentés dans la section précédente.

### 3.1 Taille

De manière générale, deux choix peuvent être fait pour spécifier la taille des codes-barres 2D. Par taille d'un code-barres, on entend la taille de la matrice de bits résultant de la lecture d'un code-barres 2D. Les code-barres peuvent être de taille fixe ou de taille variable. Dans le premier cas, la taille est fixée une fois pour toute, et dans le second cas, la taille s'adapte à la quantité d'information que l'on souhaite stocker.

Nous proposons d'utiliser des tailles fixes, mais d'en autoriser plusieurs. Ce choix permet une certaine souplesse au niveau des tailles, tout en simplifiant les opérations de codage et de décodage de part le fait que la taille est connue. Les codes-barres seront toujours carrés. Quatre tailles différentes seront supportées : $32 \times 32$, $64 \times 64$, $128 \times 128$ et $256 \times 256$. Le format proposé stocke la taille sur 3 bits, cela afin de supporter d'éventuelles tailles additionnelles dans le futur. Le tableau 1 reprend les configurations possibles de taille.

| **Paramètre** | **Taille** |
|---|---|
| 0 | $32 \times 32$ |
| 1 | $64 \times 64$ |
| 2 | $128 \times 128$ |
| 3 | $256 \times 256$ |

Table 1 – Tailles des codes-barres.

### 3.2 Données

Plusieurs types de données peuvent être encodés dans le code-barres 2D. Si on se tourne vers les autres formats existants, on pense à des URL, des vCards, du simple texte, des kanjis japonais, etc. Afin de ne fermer aucune porte et rester aussi ouvert et extensible, nous proposons de réserver quatre bits d'information pour stocker le type de données stockées dans le code-barres. Le tableau 2 reprend différents types de données.

| **Paramètre** | **Type de données** | **Norme** |
|---|---|---|
| 0 | Texte ASCII 7-bits | ISO 646 |
| 1 | Texte ASCII étendu | ISO 8859-1 |
| 2 | URL | RFC 3986 |
| 3 | Japanese Kanji | Shift JIS |

Table 2 – Types de données.

Dans le cadre de ce travail, seul le premier type de données sera développé. Les trois autres types proposés dans le tableau 2 sont réservés pour une future extension du format. Notez qu'une URL pourrait très bien être encodée comme un texte ASCII. L'avantage de définir un type de données explicite est que les applications peuvent se baser sur ce type, pour par exemple proposer à l'utilisateur de directement charger la ressource identifiée par l'URL.

### 3.3 Code correcteur d'erreurs

Afin de pouvoir détecter et éventuellement corriger des erreurs dues à des altérations du code-barres, il faut utiliser ce qu'on appelle un *code correcteur d'erreur*. L'idée est d'ajouter

de l'information qui permettra de savoir si le code-barres a été altéré ou non et également d'ajouter de l'information redondante qui permettra de corriger les éventuelles erreurs. La solution choisie dans le cadre de ce travail est plutôt basique, par rapport à d'autres solutions existantes. Ce choix a été fait afin de rendre le plus efficace possible l'encodage et le décodage de code-barres.

La figure 3 montre la forme générale du code-barres. On peut distinguer deux zones : la zone blanche contient les informations de configuration, ainsi que les données (*zone contenu*) tandis que la zone grise contient une partie des informations redondantes permettant de détecter et corriger des éventuelles erreurs (*zone code correcteur d'erreurs*).

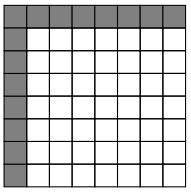


FIGURE 3 – Format du code-barres 2D.

Le code correcteur d'erreurs qui va être utilisé sera un *code de parité*. Les bits de la zone code correcteur d'erreurs vont être choisis de telle sorte que pour chaque ligne et pour chaque colonne du code-barres, le nombre de bits à 1 doit être pair. La figure 4 montre un exemple de petit code-barres, avec la zone du code correcteur d'erreurs qui a été remplie. Notez que le bit tout en haut à gauche joue un rôle tout particulier.

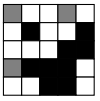


FIGURE 4 – Exemple de code correcteur d'erreur.

### 3.3.1 Détection et correction d'erreurs

Le code de parité proposé permet de détecter au maximum deux erreurs. De plus, il permet de corriger une erreur. La figure 5 montre un exemple de code-barres possédant une erreur. On remarque en effet qu'il y a une ligne et une colonne pour lesquelles la parité n'est pas respectée. En réalité, cela signifie que le pixel se trouvant à l'intersection de cette ligne et de cette colonne n'est pas correct et qu'il faut dès lors inverser sa valeur pour corriger l'erreur.

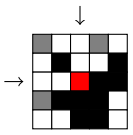


FIGURE 5 – Exemple de détection et correction d'une erreur dans le code-barres.

Il faut bien préciser les propriétés de ce pouvoir de correction. En partant d'un code-barres correct, si un seul de ses bits a été altéré, il sera possible de le détecter et de le corriger. Si deux de ses bits ont été modifiés, il sera possible de dire avec certitude que deux erreurs ont été faites. On peut donc détecter qu'il y a eu deux erreurs, mais on ne sait pas les corriger. Maintenant, des fausses alarmes sont possibles avec ce système. En effet,

plusieurs erreurs bien placées peuvent donner l'illusion qu'il n'y a eu qu'une seule erreur. Le code-barres sera alors corrigé, car il est impossible de détecter de telles situations. Par exemple, si trois erreurs sont faites, étant les trois coins d'un triangle rectangle dont les côtés sont parallèles aux zones du code correcteur d'erreurs.

### 3.4 Compression

Une des performances visées est la compacité des données. En effet, si on parvient à compacter les données, on pourra stocker plus d'information pour un espace fixé. Afin de rester le plus flexible possible, le format proposé prévoit la possibilité de *compresser* les données stockées dans le code-barres. Une des options de configuration indiquera donc le type de compression qui sera appliqué aux données, avant d'être écrite dans le code-barres. Le tableau 3 reprend les types de compression supportés.

| **Paramètre** | **Compression** |
|---|---|
| 0 | Aucune compression |
| 1 | Compression de Huffman<br>(table de fréquences pour le français) |

TABLE 3 – Types de compression des code-barres.

Si on reprend la figure 2 décrivant le codage et décodage d'un code-barres, les phases de compression de de décompression viennent se positionner entre l'information et la matrice de bits. Pour la création du code-barres, les informations sont d'abord compressées et ensuite encodées. Pour la phase de lecture du code-barres, ce n'est qu'une fois la matrice de bits décodée que la décompression se fait afin de récupérer l'information stockée. La figure 6 résume cela.

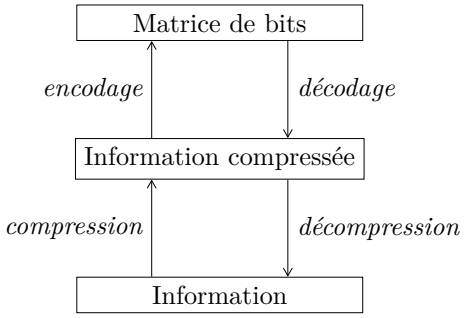



FIGURE 6 – Compression et décompression.

### 3.5 Format global

Enfin, venons-en au format global. Cette section décrit comment les différents éléments présentés jusqu'ici s'agencent dans la matrice de bits représentant le code-barres 2D. La figure 7 montre les différentes zones présentes dans un code-barres 2D. On peut clairement y identifier cinq zones parmi lesquelles on a déjà vu en détail la « *zone code correcteur d'erreurs* ».

Ensuite, la « *zone contenu* » se découpe en quatre zones successives :

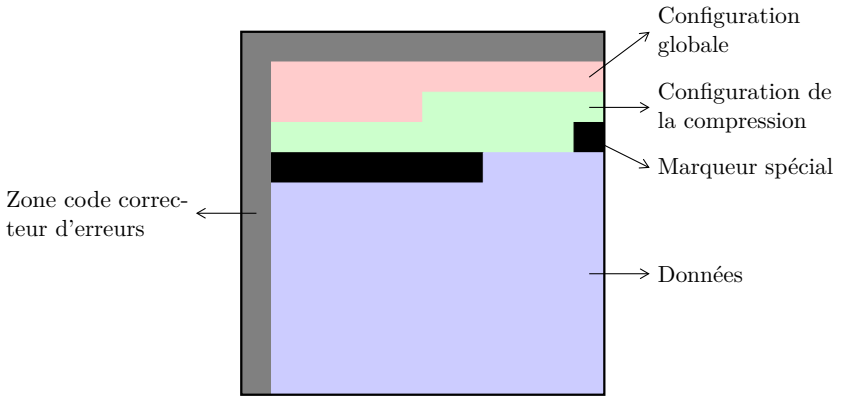



FIGURE 7 – Organisation des zones du code-barres 2D.

1. La configuration globale contient les paramètres qui sont globaux au code-barres. Cette zone se compose de trois éléments :
   (a) 3 bits pour la taille ;
   (b) suivis de 4 bits pour le type de données ;
   (c) et enfin 3 bits représentent le type de compression appliqué.

   Les 6 bits suivants sont fixés à zéro, de telle sorte que les éléments de configuration occupent au total 2 octets. Ils sont dits *réservés pour une éventuelle utilisation future.*
2. Vient ensuite une zone réservée pour des éventuelles informations de configuration liées au mode de compression. La longueur de cette zone doit être un multiple d'octets et dépend du type de compression choisi. La fin de cette zone est signalée par un octet spécial `11111111`. Ce marqueur ne peut donc pas apparaitre dans les informations de configurations du mode de compression. S'il n'y a pas de compression, alors cette zone sera complètement absente.
3. Enfin, le reste de l'espace disponible est dédié aux données. La figure 7 montre les différentes zones du code-barres 2D.

### 3.6 Le type de données texte ASCII 7-bits

La section précédente décrit le format global du code-barres. Dans ce format, une zone est réservée aux données à proprement parler. L'organisation interne de cette zone dépend du type de données qui est choisi. Cela permet une grande souplesse et flexibilité.

L'encodage de caractères ISO 646 permet de représenter du texte ASCII dont les caractères sont encodés sur 7 bits (il y a donc $2^7 = 128$ caractères représentables). La figure 8 montre la table des caractères ASCII.

La première colonne du tableau représente les quatre premiers bits et la première ligne les quatre derniers. Prenons par exemple le caractère `V` : sa valeur est $(56)_{16} = (86)_{10} = (1010110)_2$ et donc, le caractère `V` sera représenté par les 7 bits suivants : `1010110`. De plus, les seuls caractères intéressants pour représenter du texte sont ceux compris entre 32 (inclus) et 127 (exclu).

Chaque lettre du texte à encoder sera représentée sur 8 bits. Les 7 bits de poids faible correspondent à l'encodage du caractère selon la norme ISO 646 et le bit de poids fort est utilisé comme *bit de contrôle* qui sera simplement un bit de parité. La figure 9 montre un octet illustrant ce principe de contrôle d'erreur.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | *A* | *B* | *C* | *D* | *E* | *F* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STH | ETH | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | CD2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | spc | ! | " | # | $ | % | & | ’ | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ‘ | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ∼ | DEL |

FIGURE 8 – La table de caractères ASCII 7-bits (iso-646).

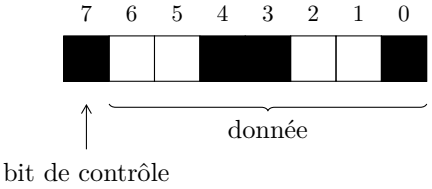



FIGURE 9 – Exemple d’encodage d’un caractère, avec son bit de contrôle, pour le type de données « Texte ASCII 7-bits ».

Pour l’encodage d’une phrase complète, les octets encodant les caractères sont simplement placés consécutivement dans la zone de données du code-barres. Lorsque la quantité de données n’est pas suffisante pour couvrir tout l’espace du code-barres, la fin de la zone de données est signalées par le marqueur spécial `11111111`. Ce marqueur ne doit être évidemment utilisé que si la place restante non utilisée est au moins plus grande que 8 bits.

Notez que l’ajout d’un bit de parité à chaque octet de la zone de données permet d’augmenter le nombre d’erreurs qu’il est possible de détecter. En effet, pour chaque octet, si un de ses bits a été altéré, cela pourra être détecté. Plusieurs réactions à cela sont possibles : soit afficher un message d’erreur lors du décodage, soit décoder chaque octet contenant une erreur par un caractère particulier (par exemple, le caractère « `?` »).

## Conclusion

Pour conclure ce rapport, la solution proposée est bien en accord avec le cahier des charges, tout en étant souple et flexible. Ainsi, le format proposé est modulable et très ouvert. Il est ainsi possible de facilement l’étendre et d’ajouter des fonctionnalités supplémentaires. Le format proposé peut en effet être vu comme un format de type conteneur pouvant inclure plusieurs formats de données possibles.

Ce format de code-barres 2D reste néanmoins à valider par une implémentation qui pourra démontrer sa simplicité d’utilisation. À cette fin, quelques pistes sont proposées en annexe de ce rapport.

# A Interfaces pour l'implémentation

Cet annexe reprend une proposition d'interface en Java pour l'encodeur et le décodeur des codes-barres 2D. Le type de données utilisé pour la matrice de bits pourrait également être de type `boolean` au lieu du type `int` comme utilisé dans la proposition présentée dans cet annexe. Il s'agit d'interfaces, cela permettant à tout un chacun d'avoir ses propres implémentations, notamment pour implémenter ou non différentes extensions.

## A.1 Configuration

```
public interface Configuration
{
   /**
    * @pre -
    * @post La valeur renvoyée contient la taille du code-barres
    */
   public int getSize();

   /**
    * @pre -
    * @post La valeur renvoyée contient le type de données du code-barres
    */
   public int getDataType();

   /**
    * @pre -
    * @post La valeur renvoyée contient le type de compression du code-barres
    */
   public int getCompressionMode();
}
```

## A.2 Encodeur

```
public interface Encoder
{
   /**
    * @pre msg != null
    * @post La valeur renvoyée contient une matrice de bits correspondant
    *       au message msg, encodé en respectant la configuration de cet encodeur
    * @throw EncodingException au cas où le message msg ne peut pas être encodé
    */
   public int[][] encode (String msg);

   /**
    * @pre -
    * @post La valeur renvoyée contient la configuration de cet encodeur
    */
   public Configuration getConfiguration();
}
```

## A.3 Decodeur

```
public interface Decoder
{
   /**
    * @pre data != null
    *      data est une matrice carrée, de taille 32, 64, 128 ou 256
    *      data ne contient que des 0 et des 1
    * @post La valeur renvoyée contient le décodage de la matrice de bits data
    *       décodée en respectant la configuration de ce décodeur
    *       Si la matrice contient une erreur, celle-ci est corrigée
    * @throw DecodingException au cas où la matrice data ne peut pas être décodée
    */
   public String decode (int[][] data);

   /**
    * @pre data != null
    *      data est une matrice carrée, de taille 32, 64, 128 ou 256
    *      data ne contient que des 0 et des 1
    * @post La valeur renvoyée contient true si data ne contient pas
    *       d'erreurs (la parité de la matrice de bits est valide),
    *       et false sinon
    */
   public boolean check (int[][] data);

   /**
    * @pre -
    * @post La valeur renvoyée contient la configuration de ce décodeur
    */
   public Configuration getConfiguration();
}
```