Universidad Autónoma Metropolitana Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería

Licenciatura en Ingeniería en Computación

Reporte de Proyecto Terminal:

“Algoritmo de generación de código QR implementado en C# como biblioteca de código abierto”

Aurelia Lissette López Huerta

207332442

Risto Fermín Rangel Kuoppa

Profesor Titular

Departamento de Sistemas

ÍNDICE.

INTRODUCCIÓN.

El presente reporte describe el Proyecto Terminal titulado: "Algoritmo de generación de código QR implementado en C# como biblioteca de código abierto", en el cual se llevó a cabo la construcción de una biblioteca de código abierto donde se implementaron la primera y segunda parte del algoritmo de generación de código QR [1][2], dicha biblioteca se construyó en lenguaje C#.

Para el proyecto, en un principio, se propusieron los objetivos que a continuación se enlistan:

OBJETIVO GENERAL:

- Sintetizar automáticamente códigos QR con una biblioteca en lenguaje C#.

OBJETIVOS ESPECÍFICOS:

- Delimitar el tipo de datos y capacidad de caracteres a codificar en el código QR resultante de la biblioteca.
- Determinar el nivel de corrección de errores que contendrá el código QR.
- Establecer el tamaño del código QR que se generará.
- Diseñar biblioteca de código abierto basada en el algoritmo de código QR.
- Implementar la biblioteca antes mencionada.
- Realizar pruebas codificando información de entrada.
- Generar código QR en una imagen en formato PNG.
- Diseñar una aplicación para comprobar la funcionalidad de la biblioteca.
- Implementar la aplicación ya mencionada.

Es importante mencionar que algunos de los objetivos específicos propuestos, no se alcanzaron puesto que no se dimensiono correctamente la robustez del algoritmo de generación de códigos QR, ya que implementar todo el algoritmo hasta llegar a la última parte, que es la generación de la imagen resultante de la codificación de los datos de entrada a código QR, va más allá de un solo proyecto terminal, es decir, para poder tener todo el algoritmo en una biblioteca de código abierto y poder realizar el análisis, diseño e implementación del mismo se debe dividir en dos proyectos terminales. La primera parte se elaboró en el presente proyecto y la correspondiente segunda parte se explica en la sección de Trabajo a Futuro.

En la primer etapa del proyecto se realizó el análisis, diseño e implementación de la primera parte del algoritmo en una biblioteca orientada a obtener los mismos resultados que los mostrados como ejemplo en el tutorial [1], donde se demuestra cómo codificar la cadena **Hello World** en la versión 1 para un código QR, con la corrección de error de nivel Q, como se muestra en la Tabla 1 **[3]**.

Version	Modules	ECC Level	Data bits	Numérico	Alfanumérico	Binario	Kanji
1	**21x21**	L	152	41	25	17	10
		M	128	34	20	14	8
		Q	**104**	**27**	**16**	**11**	**7**
		H	72	17	10	7	4

Tabla 1

En la segunda etapa del proyecto se llevó a cabo el análisis, diseño e implementación de la segunda parte del algoritmo en una biblioteca orientada a obtener los mismos resultados que los mostrados como ejemplo en el tutorial [2], En él se demuestra cómo generar las palabras de corrección de errores, donde se llegó hasta el segundo paso, que *Crear el mensaje del polinomio*, esto por las razones de robustez anteriormente expuestas.

Así mismo, una vez obtenidos los mismos resultados en la implementación de la biblioteca (primera y segunda parte del algoritmo), se hicieron los ajustes necesarios para implementar la Versión 8 con un nivel de corrección L (Tabla 2 **[3**]), ya que este tipo de versión no se encuentra disponible si cualquier usuario desea codificar un número máximo de 279 datos alfanuméricos.

Versión	Modules	ECC Level	Data bits	Numérico	Alfanumérico	Binario	Kanji
8	**49x49**	**L**	**1,552**	**461**	**279**	**192**	**118**
		M	1,232	365	221	152	93
		Q	880	259	157	108	66
		H	688	202	122	84	52

Tabla 2

Mencionado lo anterior y con la biblioteca creada se tiene la capacidad de crear una aplicación[1], que recibirá como entrada, por ejemplo, los siguientes datos:

Examen tipo: Parcial 3
Subtipo: 2
Grupo: CSI06 (trim. 12P)
Fecha: 16/Jul/2012 11:31
UEA: ESTRUCTURA DE DATOS CON ORIENTACION A OBJETOS (1151008)
Estudiante: HERNANDEZ PEREZ JOSE ALBERTO
Matrícula: 210332442

Los codificadores disponibles en la red **[5**] solo aceptan un máximo de 180 caracteres como serían los siguientes:

Examen tipo: Parcial 3
Subtipo: 2
Grupo: CSI06 (trim. 12P)
Fecha: 16/Jul/2012 11:31
UEA: ESTRUCTURA DE DATOS CON ORIENTACIÓN A OBJETOS (1151008)
Estudiante: HERNANDEZ PEREZ JOSE AL

Donde el nombre del alumno está incompleto además de que no se incluye la matrícula.

Obteniendo como resultado la Imagen 1.

[1] Esta aplicación fue necesaria para comprobar el buen funcionamiento de la biblioteca con casos reales.

Imagen 1 - Código QR
(codificación de infomración de entrada)

Como se observa, la información de entrada que va más allá de 180 caracteres, no puede ser codificada, esto es un ejemplo del porque es importante poder tener acceso al código fuente, esto para poder modificar la capacidad de entrada y codificación del algoritmo de generación de código QR.

DESARROLLO DEL PROYECTO.

PRIMERA ETAPA.

En esta sección de describe la primera etapa del proyecto donde se realizó el análisis, diseño e implementación, en lenguaje C#, de la primera parte del algoritmo de generación de código QR. Esta primera parte del algoritmo contiene 6 pasos que a continuación se describen, donde una vez analizado cada uno, se efectuó el diseño e implementación de una clase por paso. Una vez obtenidas las 6 clases, se analizó, diseñó e implementó una aplicación con la que se comprobó el correcto funcionamiento de las clases, esto al comparar los resultados obtenidos con los expuestos en el material de consulta [1].

DESCRIPCIÓN, ANÁLISIS E IMPLEMENTACIÓN DE LA PRIMERA PARTE DEL ALGORITMO.

El primer paso para crear un código QR es generar una cadena binaria que incluya sus datos, información sobre su modo de codificación y la longitud de los datos.

Paso 1: Codificar el Indicador de modo.

El indicador de modo es una cadena de cuatro bits que representa el modo de datos que está utilizando, es decir, numérico, alfanumérico, binario o japonés [1], como se muestra en la Tabla 3.

Cadena de bits	Modo de datos
0001_2	Modo numérico
0010_2	**Modo alfanumérico**
0100_2	Modo binario
1000_2	Modo japonés

Tabla 3

El modo de entrada que se utilizó para este proyecto fue el de cadenas alfanuméricas, por lo tanto, el indicador de modo o cadena binaria correspondiente es la 0010_2, indicando "Modo alfanumérico".

Así se obtiene la primera cadena binaria:
0010_2

Análisis y diseño de la clase referente al Paso 1.

La clase contiene un solo atributo llamado "strPaso1", el cual representa la cadena resultante del Paso 1. Así mismo, la clase contiene dos métodos como son el constructor "Paso1" y el método llamado "hazPaso1", el cual, como su nombre lo indica, está encargado de realizar el *Paso 1* ya descrito y retornar el atributo "strPaso1" con el valor de la cadena resultante del paso. Lo anterior se representa en el Diagrama 1.

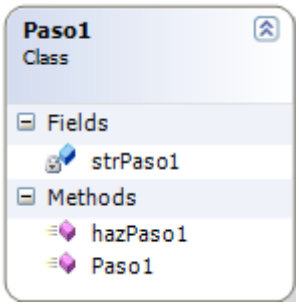

Diagrama 1 – Estructura de la Clase Paso1

Implementación de la clase referente al Paso 1.

```
public class Paso1
    {
        private string strPaso1; //Atributo de clase.
        public Paso1() //Método constructor de clase.
        { }
        public string hazPaso1()
        {
            strPaso1 = "0010"; //Asignación de modo alfanumérico.
            return strPaso1; //Retorno de cadena resultante.
        }
    }
```

Código 1 – Clase referente al Paso 1.

Paso 2: Codificar la longitud de los datos.

En este paso, se determina el número de caracteres que hay el mensaje de entrada, el cual será convertido a un número binario.

En el ejemplo propuesto en el tutorial [1] se codifica la frase **Hello World**, en la cual hay cinco caracteres por palabra, además de un espacio entre palabras, sumando un total de 11 caracteres. Entonces el número 11_{10} se convierte al número binario 1011_2.

Cuando se codifica la longitud de los datos es necesario codificar utilizando un número determinado de bits. Entonces, como ejemplo se va a crear una Versión 1 de código QR; como se puede ver en la Lista 1, cuando se utiliza el modo alfanumérico y la versión de la 1 a la 9, tenemos que usar 9 bits para codificar la longitud de los datos. Por lo tanto, si tenemos 1011_2 se pondrán 0´s a la izquierda con el fin de obtener una palabra con longitud de 9-bits, así: 000001011_2.

Lista 1

Versiones 1 al 9

- Modo Numérico: 10 bits
- Modo Alfanumérico: 9 bits
- Modo binario: 8 bits
- Modo Japonés: 8 bits

Para consultar demás versiones *véase* [1]

Una vez obtenida la cadena binaria del paso 2, esta se une a la cadena binaria obtenida en el paso 1, como se muestra a continuación:
0010 **000001011**$_2$

Análisis y diseño de la clase referente al Paso 2.

La clase *Paso 2* contiene un solo atributo llamado "strPaso2", éste representa la cadena resultante del Paso 2. Así mismo, la clase contiene dos métodos, como son: el constructor "Paso2" y el método llamado "hazPaso2", el cual, como su nombre lo indica, está encargado de realizar el *Paso 2* y retornar el atributo "strPaso2" con el valor de la cadena resultante del paso 2. Lo anterior se representa en el Diagrama 2.

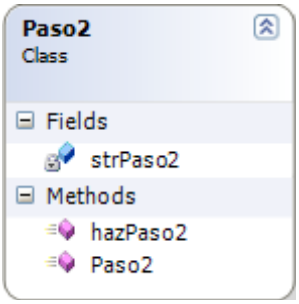

Diagrama 2 – Estructura de la Clase Paso2

Implementación de la clase referente al Paso 2.

```
public class Paso2
    {
        string strPaso2; //Atributo de clase.

        public Paso2() //Constructor de clase.
        { }
        public string hazPaso2(string datos) //Recibe la cadena de entrada.
        {
           convertidorBin longDConv = new convertidorBin(); //Declaración de un Objeto de
           la Clase "convertidorBin" encargada de convertir números decimales a binarios,
           descrita en el Anexo 1.

           strPaso2 = longDConv.convierteABin(datos.Length);//Se manda la longitud de la
           cadena de entrada al método "convierteABin" de la clase convertidorBin, donde
           la variable strPaso2 recupera el valor en binario del número de caracteres de
           entrada.

           while (strPaso2.Length < 9)//Ciclo while que verifica la longitud de la cadena,
           si es menor a 9 realiza ciclos con el fin de que la cadena sea de longitud 9.
            {
                strPaso2 = "0" + strPaso2; //Concatena 0s a la izquierda de la cadena.
            }

            return strPaso2; //Regresa la longitud de los datos de entrada codificados.
        }
    }
```

Código 2 – Clase referente al Paso 2.

Paso 3: Codificar los datos.

Para codificar los datos alfanuméricos, primeramente, la cadena de entrada, por ejemplo *HELLO WORLD*, se debe romper en pares de caracteres: HE, LL, O , WO, RL, D.

Para cada par de caracteres, se toma el valor alfanumérico del primer carácter y se multiplica por 45, obtenido dicho valor éste se suma al valor alfanumérico del segundo carácter. Este valor alfanumérico se encuentra en la Tabla 4, la cual proporciona una referencia para los caracteres que se admiten en el modo alfanumérico para códigos QR **[4]**.

Carácter	Valor	Carácter	Valor	Carácter	Valor
0	0	**F**	15	**U**	30
1	1	**G**	16	**V**	31
2	2	**H**	17	**W**	32
3	3	**I**	18	**X**	33
4	4	**J**	19	**Y**	34
5	5	**K**	20	**Z**	35
6	6	**L**	21	**espacio**	36
7	7	**M**	22	**$**	37
8	8	**N**	23	**%**	38
9	9	**O**	24	*****	39
A	10	**P**	25	+	40
B	11	**Q**	26	-	41
C	12	**R**	27	**.**	42
D	13	**S**	28	/	43
E	14	**T**	29	**:**	44

Tabla 4

Luego, el resultado obtenido se convierte a una cadena binaria de 11-bits. Si se desea codificar un número impar de caracteres, se tomar valor alfanumérico del carácter final y se convierte a una cadena binaria de 6-bits. Las cadenas resultantes se concatenarán con las obtenidas en los pasos anteriores.
A continuación se muestra lo expuesto anteriormente:

	HE	**LL**	**O (espacio)**	**WO**	**RL**	**D**
	(45 * 17) +14	(45 * 21) +21	(45 * 24) +36	(45 * 32) +24	(45 * 27) +21	13
	779	966	1116	1464	1236	13
0010 000001011	01100001011	01111000110	10001011100	10110111000	10011010100	001101
Cadenas binarias de los pasos 1 y 2	Cadenas binarias del Paso 3					

Análisis y diseño de la clase referente al Paso 3.

La clase Paso3 contiene un solo atributo llamado "strPaso3", el cual representa las cadenas resultantes del *Paso 3*. Así mismo, la clase contiene dos métodos como son: el constructor "Paso3" y el método llamado "hazPaso3", el cual, como su nombre lo indica, está encargado de realizar el Paso 3 ya descrito y retornar el atributo "strPaso3" con el valor de las cadenas resultantes del paso. Lo anterior se representa en el Diagrama 3.

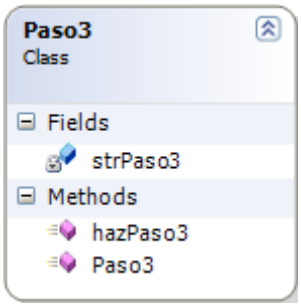

Diagrama 3 - Estructura de la Clase Paso3

Implementación de la clase referente al Paso 3.

```
public class Paso3
    {
        string strPaso3 = null; //Inicializando el atributo.
        public Paso3() //Método constructor de clase.
        { }
        public string hazPaso3(string datos)
        {
            convertidorAlfanum convAlfanum = new convertidorAlfanum(); //Declaración de
               //un objeto de la Clase "convertidorAlfanum" encargada de recibir la
               //cadena de entrada y retornar el valor alfanumérico de cada carácter,
               //descrita en el Anexo 2.
            convertidorBin convABin = new convertidorBin(); //Declaración de un Objeto de
               //la Clase "convertidorBin" encargada de convertir números decimales a
               //binarios, descrita en el Anexo 1.

            int[] datosEnAlfanum = new int[datos.Length]; //Arreglo donde se guarda el
               //valor alfanumérico de los caracteres.
            int[] datosSumPares = new int[(datos.Length / 2) + 1]; //Arreglo donde se
               //guarda el resultado de la suma del valor alfanumérico del par de
               //caracteres.
            string[] datosABin = new string[(datos.Length / 2) + 1]; //Arreglo donde se
               //guarda el resultado de la conversión de suma del valor alfanumérico de
               //cada par de caracteres.
            datosEnAlfanum = convAlfanum.convierteAAlfanum(datos); //Hace la primera fase
               //del paso 3.

            if (datos.Length % 2 == 0)//Número de datos par.
            {
                for (int j = 0; j < (datos.Length / 2); j++)
                {
                    datosSumPares[j]=(datosEnAlfanum[j*2]*45)+datosEnAlfanum[(j*2)  +  1];
```

```
                    //Se toma el valor alfanumérico del primer carácter y se
                    //multiplica por 45 y se le suma el valor alfanumérico del segundo
                    //carácter.
                }
                for (int i = 0; i < (datos.Length / 2); i++)
                {
                    datosABin[i] = convABin.convierteABin(datosSumPares[i]);
                    //Convierte el resultado de cada suma a binario.
                }

                int p = 0;// Índice.
                while (p < datosABin.Length && datosABin[p] != null)
                {//Ciclo que completa palabras de 11-bits.
                    if (datosABin[p].Length < 11)
                    {
                        datosABin[p] = "0" + datosABin[p];
                    }
                    else p++;
                }
            }
            else //Para número de caracteres impar.
            {
                int k;
                for (k = 0; k < (datos.Length / 2); k++)
                {
                    datosSumPares[k]=(datosEnAlfanum[k*2]*45)+datosEnAlfanum[(k*2)+1];
                    //Ciclo en el cual se toma el valor alfanumérico del primer
                    //carácter y se multiplica por 45 y se le suma el valor
                    //alfanumérico del segundo carácter.
                }
                datosSumPares[k] = datosEnAlfanum[k * 2]; //Al último dato impar no se le
                   //hace el mismo proceso.

                for (int i = 0; i <= (datos.Length / 2); i++)
                {
                    datosABin[i] = convABin.convierteABin(datosSumPares[i]);
                    //Convierte el resultado de cada suma a binario.
                }

                int p = 0;
                while (p < (datosABin.Length-1) && datosABin[p] != null)
                { //Ciclo para completar palabras de 11 bits.
                    if (datosABin[p].Length < 11)
                    {
                        datosABin[p] = "0" + datosABin[p];
                    }
                    else p++;
                }
                while (p < datosABin.Length && datosABin[p] != null)
                { //Ciclo para completar palabras de 6 bits.
                    if (datosABin[p].Length < 6)
                    {
                        datosABin[p] = "0" + datosABin[p];
                    }
                    else p++;
                }
            }
```

```
            strPaso3 = datosABin[0]; // A strPaso3 se le asigna la primera cadena binaria
               //obtenida.

            for (int i = 1; i < (datos.Length / 2) + 1; i++)
            { //Ciclo para concatenarle a strPaso3 las demás cadenas obtenidas y retornar
              //una sola supercadena.
                strPaso3 = strPaso3 + " " + datosABin[i];
            }

            return strPaso3;
        }
    }
```

Código 3 – Clase referente al Paso 3.

Al término de la creación de las clases referentes a los pasos 1, 2 y 3 se llegó a la conclusión de que se requería de una clase extra encargada de unir las palabras generadas en dichos pasos, esta clase fue llamada "Paso123", la cual solo consta de un atributo y dos métodos, uno constructor y el otro encargado de la creación de objetos de tipo Paso 1, Paso 2 y Paso 3 para que las cadenas resultantes de cada paso, mediante concatenación, formaran una sola.
El diseño de la clase antes mencionada se muestra en el Diagrama 4.

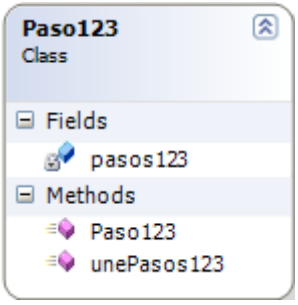

Diagrama 4 - Estructura de la Clase Paso4

Implementación de la clase Paso123.

```
    public class Paso123
     {
         string pasos123 = null; //Atributo principal de la clase.
         public Paso123() //Constructor.
         { }
         public string unePasos123(string datos)
         {   //***Creación de los objetos de cada paso.***//
             Paso1 strPaso1 = new Paso1();
             Paso2 strPaso2 = new Paso2();
             Paso3 strPaso3 = new Paso3();
             pasos123 = pasos123 + strPaso1.hazPaso1(); //Primero toma el valor de la
                //cadena del paso  1.
             pasos123=pasos123+" "+strPaso2.hazPaso2(datos)+" "+strPaso3.hazPaso3(datos);
                //Se concatenan las cadenas de los pasos 2 y 3.
             return pasos123; // Retorna el conjunto de cadenas en una sola variable.
         }
     }
```

Código 4 – Clase referente al Paso123.

Paso 4: Termine los bits.

Una vez que se tiene la cadena de bits de los pasos anteriores hay que asegurarse de que es de la longitud correcta. Esto depende del número de bits de datos que son necesarios para generar la versión y la corrección de errores que se esté usando.

En el ejemplo propuesto [1] se ha optado por utilizar una versión de código QR con la corrección de nivel de error Q. Para ello, se debe generar 104 bits de datos como se muestra en la Tabla 5 **[3]**.

Versión	Modules	ECC Level	Data bits	Numérico	Alfanumérico	Binario	Kanji
1	21x21	L	152	41	25	17	10
		M	128	34	20	14	8
		Q	**104**	**27**	**16**	**11**	**7**
		H	72	17	10	7	4

Tabla 5

Si la cadena de bits es menor que 104, entonces se agregan hasta cuatro 0´s al final. Si dicha adición hace que la cadena sea mayor de 104, entonces sólo se agrega el número de ceros necesarios para que la cadena sea de 104 bits de longitud.

En el ejemplo de *HELLO WORLD,* la cadena es de 59 bits de largo, por lo tanto se añaden cuatro 0s en el final. Si dicha cadena hubiera terminado siendo de 102 bits de longitud (sólo como ejemplo), únicamente se le añadirían dos 0s hasta el final, para un total de 104 bits.
Por lo tanto, para este paso la cadena binaria del ejemplo seria:

0010_2 000001011_2 01100001011_2 01111000110_2 10001011100_2 10110111000_2 10011010100_2 0011012_2 $\mathbf{0000}_2$

Análisis y diseño de la clase referente al Paso 4.

La clase Paso4 contiene un solo atributo llamado “strPaso4”, el cual representa la cadena resultante del Paso 4. Así mismo, la clase contiene dos métodos como son el constructor “Paso4” y el método llamado “hazPaso4”, el cual, como su nombre lo indica, está encargado de realizar el Paso 4 y retornar el atributo “strPaso4” con el valor de la cadena resultante del paso. Lo anterior se representa en el Diagrama 5.

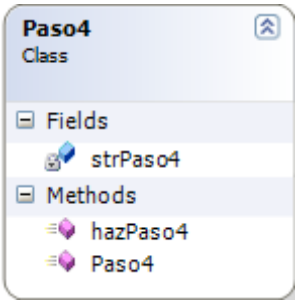

Diagrama 5 - Estructura de la Clase Paso5

Implementación de la clase referente al Paso 4.

```
public class Paso4
    {
        string strPaso4; //Atributo de clase.

        public Paso4() //Método constructor de clase.
        { }

        public string hazPaso4(string datos)
        {
            Paso123 strPaso123 = new Paso123(); //Declaración de un Objeto de la Clase
               //Paso123 para tener las cadenas generadas de los pasos 1, 2 y 3.
            strPaso4  =  strPaso123.unePasos123(datos).Replace("  ",  ""); //Instrucción
               //encargada de quitar los espacios en blanco.
            int i=0; // Índice para el ciclo while.

            while(strPaso4!=null && (104 - strPaso4.Length) < 4 && strPaso4.Length < 104)
            { //Ciclo while que agrega menos de 4 ceros, si es que así se completa la
              //cadena de 104 bits de longitud.
                strPaso4 = strPaso4 + "0";
            }
            while (strPaso4 != null && (104 - strPaso4.Length) > 4 && i < 4)
            { //Ciclo while que agrega un máximo de 4 ceros, si es que la cadena es menor
              //de 104 bits de longitud.
                strPaso4 = strPaso4 + "0";
                i++;
            }

            return strPaso4; //Retorna la cadena con el numero de 0´s necesarios según
sea //el caso.
        }
    }
```

Código 5 – Clase referente al Paso 4.

Paso 5: Delimitar la cadena en palabras de 8 bits.

En éste paso la cadena se rompe en grupos de 8-bits. Si el último grupo no es de 8-bits de largo, se rellena con 0s a la derecha.

En el ejemplo, el bloque final de la cadena es de sólo seis bits de largo, como se puede ver, así que la almohadilla de la derecha será con dos ceros.

Quedando, hasta el momento, la siguiente cadena binaria:

00100000_2 01011011_2 00001011_2 01111000_2 11010001_2 01110010_2 11011100_2 01001101_2 01000011_2 $01000\mathbf{00}_2$

Análisis y diseño de la clase referente al Paso 5.

La clase Paso5 contiene un solo atributo llamado "strPaso5", el cual representa las cadenas resultantes del Paso 5. Así mismo, la clase contendrá dos métodos como son: el constructor "Paso5" y el método llamado "hazPaso5", el cual, como su nombre lo indica, está encargado de realizar el Paso 5 ya descrito y retornar el atributo "strPaso5" con el valor de las cadenas resultantes del paso. Lo anterior se representa en el Diagrama 6.

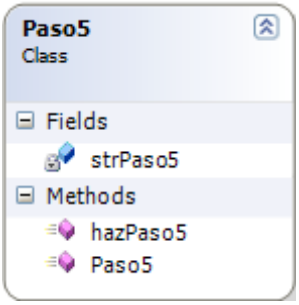

Diagrama 6 - Estructura de la Clase Paso5

Implementación de la clase referente al Paso 5

```
public class Paso5
    {
        string strPaso5 = null; //Inicializando el atributo de clase.

        public Paso5() //Método constructor de clase.
        { }

        public string hazPaso5(string datos)
        {
            Paso4 strPaso4 = new Paso4(); //Declaración de un Objeto de la Clase Paso4.
            string aux = strPaso4.hazPaso4(datos); //La variable "aux" toma el valor de la
               //cadena generada en el paso 4.
            int j,k=0; //Variables utilizadas como índices.
            string[] paso5Array = new string[(aux.Length / 8) + 1]; //Arreglo en el cual
               //se guardarán los segmentos de 8 bits de la cadena generada en el paso 4.

            for (int i = 0; i < paso5Array.Length; i++)
            { //Ciclo que secciona en palabras de 8 dentro de un arreglo.
                j=0; //Se inicializa la variable usada como índice del siguiente while.
                while (j < 8 && k<aux.Length)
                {
                    paso5Array[i] = paso5Array[i]+aux[k]; //Concatena bits hasta formar
                       //palabras de 8 (bits), cada bit lo toma de la cadena proveniente
                       //del Paso 4 (aux).
                    k++;//En cada ciclo aumenta en uno la variable.
                    j++;//En cada ciclo aumenta en uno la variable.
                }
            }
            while(paso5Array[paso5Array.Length-1]!=null&&(paso5Array[paso5Array.Length-
            1].Length<8))
            { //Ciclo que agrega 0s, de ser necesario, para a completar la última
```

```
                //partición en 8 bits.
                paso5Array[paso5Array.Length - 1] = paso5Array[paso5Array.Length - 1]+"0";
                //Instrucción que concatena 0s por la derecha.
            }
            strPaso5 = paso5Array[0];//Se asigna directamente el primer valor del arreglo.
            for (int i = 1; i < paso5Array.Length; i++)
            { //Ciclo utilizado para retornar una sola cadena y no un arreglo.
                strPaso5 = strPaso5 + " " + paso5Array[i];
            }
            return strPaso5; // Regresa la cadena resultante del paso 5.
        }
    }
```

Código 6 – Clase referente al Paso 5.

Paso 6: Agregar palabras al final si la cadena es demasiado corta.

Si aún la cadena de bits no es lo suficientemente larga, hay dos cadenas de bits especiales que son: 11101100_2 y 00010001_2, donde la especificación de códigos QR nos obliga a poner al final de la cadena, alternando entre las dos hasta que tengamos el número necesario de palabras de 8 bits (también referido como bloques de datos).

Como se ha mencionado anteriormente, se deben generar 104 bits de datos, para un total de trece palabras de 8 bits (104/8 = 13). La cadena hasta el momento sólo cuenta con 10 bloques de datos, así que se tienen que añadir tres cadenas especiales más. Comenzando con 11101100_2, luego añadir 00010001_2, a continuación agregar 11101100_2. Si se necesitan más bloques de datos, continuaríamos con 00010001_2 y se mantendría alternando entre esas dos palabras hasta que se tuviera el número necesario de bloques.

Finalmente cadena binaria queda de la siguiente manera:

00100000_2 01011011_2 00001011_2 01111000_2 11010001_2 01110010_2 11011100_2 01001101_2 01000011_2 01000000_2 $\mathbf{11101100}_2$ $\mathbf{00010001}_2$ $\mathbf{11101100}_2$.

Análisis y diseño de la clase referente al Paso 6.

La clase Paso6 contiene un solo atributo llamado “strPaso6”, el cual representa las cadenas resultantes del Paso 6. Así mismo, la clase contendrá dos métodos como son el constructor “Paso6” y el método llamado “hazPaso6”, el cual, está encargado de realizar el Paso 6 ya descrito, y retornar el atributo “strPaso6” con el valor de las cadenas resultantes del paso. Lo anterior se representa en el Diagrama 3.

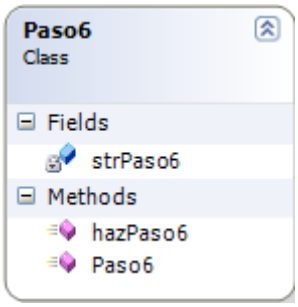

Diagrama 7 -- Estructura de la Clase Paso 6.

Implementación de la clase referente al Paso 6.

```
public class Paso6
    {
        string strPaso6 = null; //Inicializando el atributo de la clase.
        public Paso6() //Método constructor de clase.
        { }
        public string hazPaso6(string datos)
        {
            Paso5 strPaso5 = new Paso5(); //Declaración de un Objeto de la Clase Paso5.
            string aux = strPaso5.hazPaso5(datos).Replace(" ", ""); //Obtiene la cadena
               //del paso 5 y quita los espacios.
            string cadena = "1110110000010001"; //Cadenas especiales: 11101100 y 00010001.
            int x = 0, j = 1, k = 0, p = 0; //Variables utilizadas como índices.

            while (aux.Length < 104)
            {//Ciclo condicional que evalúa sí la longitud de la cadena obtenida en el
              //Paso 5 es menor a 104 a completa con los bits de las cadenas especiales.
                if (x < cadena.Length) //Estructura condicional que permite el recorrido
                  //de bit por bit en la cadena.
                {
                    aux = aux + cadena[x]; //Instrucción que concatena cada bit de las
                      //cadenas especiales a la obtenida del Paso 5.
                    x++;//Índice que aumenta en uno cada vez que se cumple la sentencia
                      //que se evalúa en la instrucción IF.
                }
                else x = 0;//El índice x se vuelve a inicializar en 0, para volver a
                  //recorrer desde 0 la variable nombrada cadena.
            }

            k = aux.Length / 8 - 1; //Al índice k se le asigna el valor de la longitud de
                //la cadena aux dividido entre (8-1), esto permitirá en el ciclo for
                //separar la cadena de 104 en 13 subcadenas de 8 bits.

            for (int i = 0; i < aux.Length+k; i++)
            {//Ciclo para separar cadenas de 8 bits, para que la lectura y comprensión por
              //parte del usuario sea mejor.
                if (j <= 8)
                {
                    strPaso6 = strPaso6 + aux[p];
                    p++;
                    j++;
                }
                else
```

```
                {
                    strPaso6 = strPaso6 + " ";
                    j = 1;
                }
            }
            return strPaso6; //Retorna la cadena con una longitud de 104 bits.
        }
    }
```

Código 7 – Clase referente al Paso 6.

ANÁLISIS, DISEÑO E IMPLEMENTACIÓN DE UNA APLICACIÓN.

En esta sección se describe el análisis, diseño e implementación de una aplicación creada con el fin de comprobar el correcto funcionamiento de las clases referentes a los pasos de la primera parte del algoritmo de generación de código QR. Esto con el fin de hacer comparaciones y que se obtengan los mismos resultados a los expuestos en el ejemplo del material de consulta [1].

La aplicación se desarrolló utilizando como IDE Visual Studio Forms, donde primero se diseñó la interfaz gráfica mostrada en la **¡Error! No se encuentra el origen de la referencia.**.

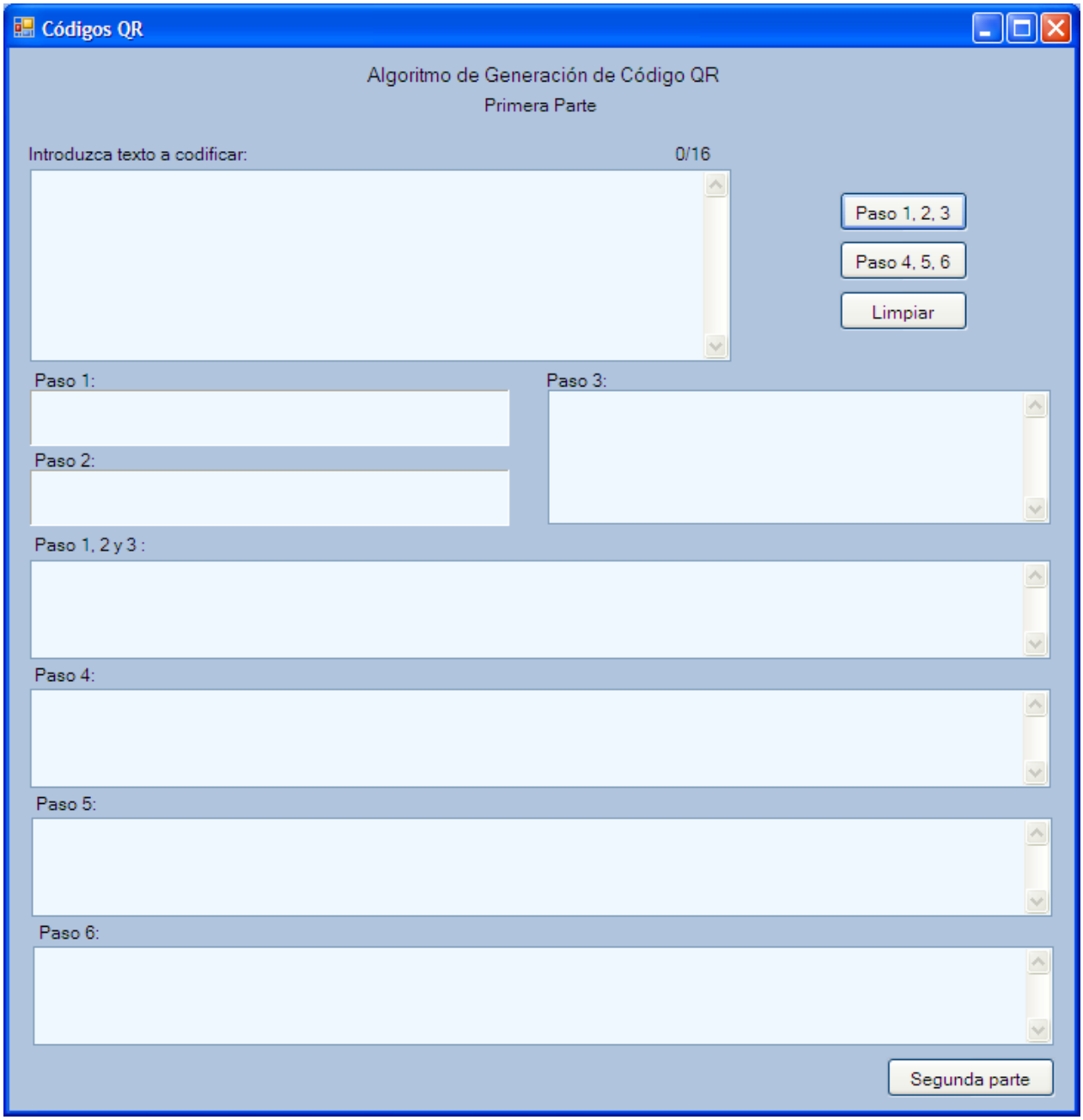

Figura 1 - Interfaz gráfica de la aplicación implementada.

Una vez concluida la interfaz gráfica se realizó la programación de cada componente de la misma, obteniendo como resultado el siguiente código:

```
public partial class Form1raPart : Form
    {
        String datos; //Variable que recupera los datos capturados por el usuario.
        bool p123Hechos = false, p456Hechos = false;//Variables booleanas para comprobar
           //que los pasos 1,2,3,4,5,6 se han realizado.
        Paso1 strPaso1 = new Paso1(); //Declaración de un Objeto de la Clase Paso1.
        Paso2 strPaso2 = new Paso2(); //Declaración de un Objeto de la Clase Paso2.
        Paso3 strPaso3 = new Paso3(); //Declaración de un Objeto de la Clase Paso3.
        Paso123 strPaso123 = new Paso123(); //Declaración de un Objeto de la Clase
           //Paso123.
        Paso4 strPaso4 = new Paso4(); //Declaración de un Objeto de la Clase Paso4.
        Paso5 strPaso5 = new Paso5(); //Declaración de un Objeto de la Clase Paso5.
        Paso6 strPaso6 = new Paso6(); //Declaración de un Objeto de la Clase Paso6.

        public Form1raPart() //Método constructor.
        {
            InitializeComponent(); //Método que se crea automáticamente y es administrado
               //por el Diseñador de Windows Forms y define todo lo que es visto en el
               //formulario.
        }

        private void btnPaso123_Click(object sender, EventArgs e) //Método referente a la
           //acción que realiza el botón titulado "Paso 1,2,3" al momento que se le da
           //click.
        {
            if (String.IsNullOrWhiteSpace(txBxentradaDatos.Text))// Verifica que haya
               //datos de entrada diferentes de null o de espacio.
            {
                MessageBox.Show("***INGRESE DATOS***"); //Mensaje que advierte que no hay
                   //datos validos a codificar.
            }
            else
            {
                datos = txBxentradaDatos.Text; //Recupera la información capturada por el
                   //usuario en el TextBox titulado "Introduzca texto a codificar:".

                txBxPaso1.Text=strPaso1.hazPaso1()+"Modo Alfanumérico"; //Instrucción que
                   //muestra el resultado de la ejecución del Paso 1 en el TextBox
                   //titulado "Paso 1:".

                txBxPaso2.Text=txBxentradaDatos.TextLength.ToString()+"
                "+strPaso2.hazPaso2(datos); // Instrucción que muestra el resultado de la
                   //ejecución del Paso 2 en el TextBox titulado "Paso 2:".

                txBxPaso3.Text = strPaso3.hazPaso3(datos); // Instrucción que muestra el
                   //resultado de la ejecución del Paso 3 en el TextBox titulado "Paso 3:"

                txBxPaso123.Text = strPaso123.unePasos123(datos); // Instrucción que
                   //muestra el resultado de la ejecución de la clase Paso123 en el
                   //TextBox titulado "Pasos 1, 2 y 3:".

                p123Hechos = true; //A la variable se le asigna el valor true ya que se
                   //realizaron los pasos 1,2,3.
```

```
            //***Asigna un valor nulo a estos TextBox en caso de que contengan
            información previa.***//
            txBxPaso4.Text = null; //Referencia al TextBox titulado "Paso 4:".
            txBxPaso5.Text = null; //Referencia al TextBox titulado "Paso 5:".
            txBxPaso6.Text = null; //Referencia al TextBox titulado "Paso 6:".
        }
    }

    private void btnLimpiar_Click(object sender, EventArgs e) //Método referente a la
       //acción que realiza el botón titulado "Limpiar" al momento que se le da click.
    {//***Asigna un valor nulo a estos TextBox en caso de que contengan información
     previa.***//
        txBxentradaDatos.Text = null; //Referencia al TextBox titulado "Introduzca
           //texto a codificar:".
        txBxPaso1.Text = null; //Referencia al TextBox titulado "Paso 1:".
        txBxPaso2.Text = null; //Referencia al TextBox titulado "Paso 2:".
        txBxPaso3.Text = null; //Referencia al TextBox titulado "Paso 3:".
        txBxPaso123.Text = null; //Referencia al TextBox titulado "Pasos 1, 2 y 3:".
        p123Hechos = false; //A la variable se le asigna el valor false ya no se han
           //realizado los pasos 1,2,3.
        txBxPaso4.Text = null; //Referencia al TextBox titulado "Paso 4:".
        txBxPaso5.Text = null; //Referencia al TextBox titulado "Paso 5:".
        txBxPaso6.Text = null; //Referencia al TextBox titulado "Paso 6:".
    }
    private void entradaDatos_TextChanged(object sender, EventArgs e) //Método
       //encargado de acciones referentes al TextBox titulado "Introduzca texto a
       //codificar:".
    {
        txBxentradaDatos.MaxLength = 16; //Delimita la longitud de las cadenas de
           //entrada, se restringe a 16 caracteres.
        lblContador.Text = Convert.ToString(txBxentradaDatos.TextLength) + "/16";
           //Contador mostrado en la parte superior derecha del TextBox titulado
           //"Introduzca texto a codificar:".
    }
    private void btnPaso456_Click(object sender, EventArgs e) //Método referente a la
       //acción que realiza el botón titulado "Paso 4,5,6" al momento que se le da
       //click.
    {
        if(p123Hechos==true) //Comprueba que ya se hayan realizado los pasos 1, 2 y 3.
        {
            txBxPaso4.Text = strPaso4.hazPaso4(datos); //Muestra el resultado de la
               //ejecución del Paso 4 en el TextBox titulado "Paso 4:".
            txBxPaso5.Text = strPaso5.hazPaso5(datos); //Muestra el resultado de la
               //ejecución del Paso 5 en el TextBox titulado "Paso 5:".
            txBxPaso6.Text = strPaso6.hazPaso6(datos); //Muestra el resultado de la
               //ejecución del Paso 6 en el TextBox titulado "Paso 6:".
            p456Hechos = true;  //A la variable se le asigna el valor true ya que se
               //realizaron los pasos 4, 5, 6.
        }
        else
        {
            MessageBox.Show("Primero realice los pasos 1, 2 y 3"); //Si no se han
               //realizado los pasos 1, 2 y 3 se le informa al usuario.
        }
    }
```

Código 8 – Programación de componentes de `Form1raPart : Form`

La estructura de este código se observa en el Diagrama 8.

Diagrama 8 -
Estructura del código de la programación de componentes de `Form1raPart : Form`

A continuación, se realizaron las siguientes pruebas codificando información de entrada y para cada paso se obtuvieron los mismos resultados a los propuestos en el tutorial [1], como se muestra en la Figura 2.

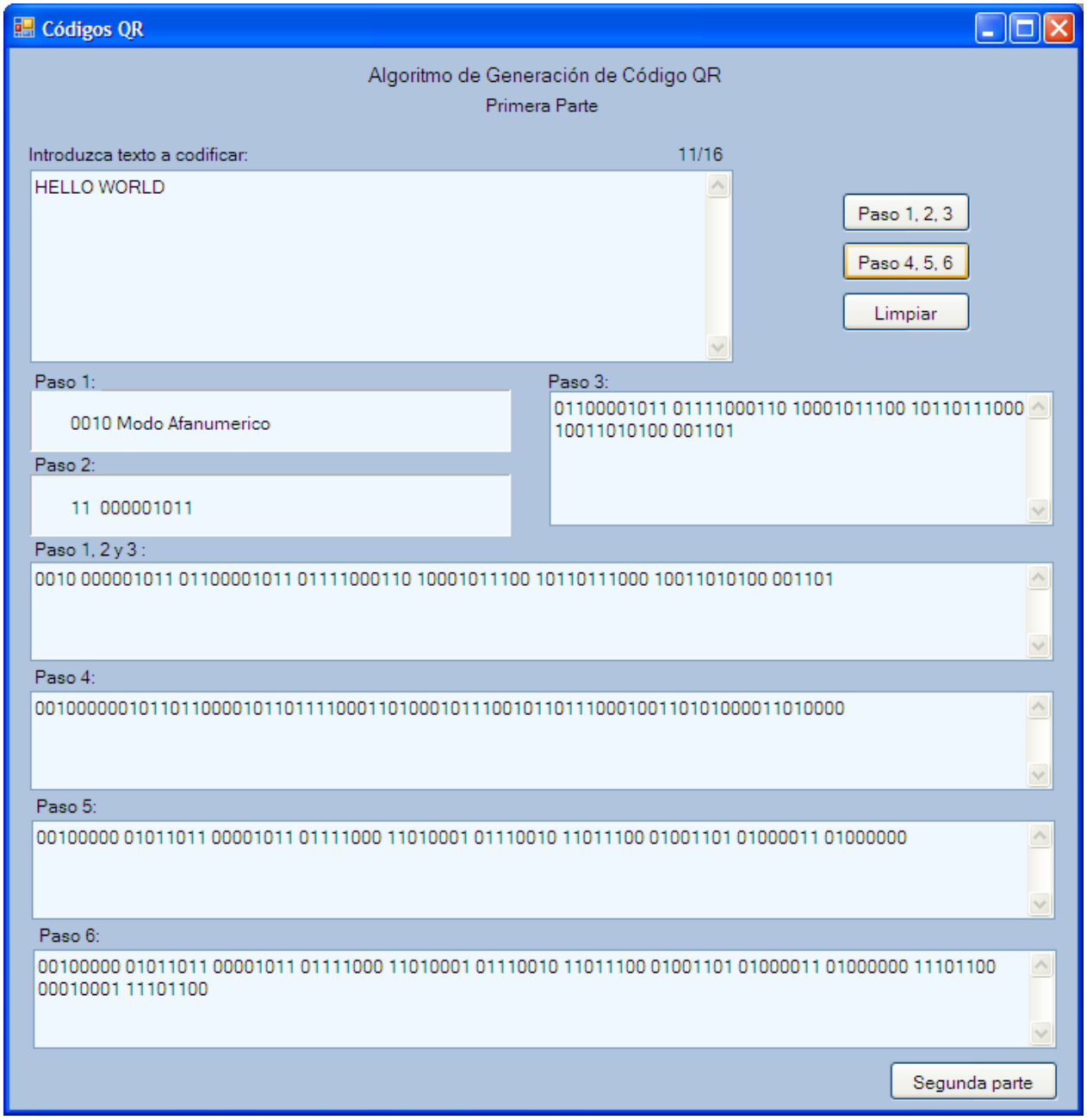

Figura 2 -
Captura de la aplicación donde se aprecia la funcionalidad de los Pasos de la primera parte del algoritmo de generación de Códigos QR.

Comprobados los resultados anteriores se procedió a agregar un botón a *Form1raPart*, llamado "Segunda parte", al que se le programó con el siguiente código:

```
private void btn2daPart_Click(object sender, EventArgs e) //Método que se ejecuta al dar
   //clic en el botón con la etiqueta "Segunda parte".
{
   if (p456Hechos == true) //Instrucción que comprueba que los pasos 4, 5 y 6 ya se
     //hayan //ejecutado.
    {
```

```
            Form2daPart f2P = new Form2daPart(txBxPaso6.Text); //Construcción de un
                //objeto de tipo Form2daPart, donde se le envía, como dato de entrada, la
              //cadena resultante del Paso 6.
            f2P.Show();//Instrucción que muestra el formulario Form2daPart.
        }
        else
        {
             MessageBox.Show("Primero realice los pasos 4, 5 y 6"); //Método que muestra
              //un cuadro con el mensaje especificado.
         }
      }
   }
```

Código 9 – Programación del botón llamado "Segunda parte",

La segunda parte del algoritmo y su análisis, diseño e implementación; así como de que trata el siguiente formulario que se ejecutará (`Form2daPart`), se explica a continuación, en la sección titulada **SEGUNDA ETAPA**.

SEGUNDA ETAPA.

En esta sección de describe la segunda etapa del proyecto donde, primeramente, se realizo el análisis, diseño e implementación, en lenguaje C#, de la segunda parte del algoritmo de generación de código QR. Esta segunda parte del algoritmo, llamada "Generación de palabras de código de corrección de errores", está compuesta por 3 pasos de los cuales 2 se describen a continuación[1]; una vez analizado cada uno, se realizó el diseño e implementación de una clase por paso. Una vez obtenidas las 2 clases, se analizó, diseñó e implementó una aplicación con la que se comprobó el correcto funcionamiento de las clases, esto al comparar los resultados obtenidos con los expuestos en el material de consulta [2].

DESCRIPCIÓN, ANÁLISIS E IMPLEMENTACIÓN DE LA SEGUNDA PARTE DEL ALGORITMO

Los códigos QR incluyen palabras de código de corrección de errores. Estos bloques de datos redundantes aseguran que el código QR aún puede leerse incluso si una parte de él no se puede leer. Los códigos QR en sus palabras de código de corrección de errores, están basados en la corrección de errores de Reed-Solomon [7].

Paso 1: Averigüe cuántas palabras de código de corrección de error es necesario generar.

El primer paso para la generación de las palabras de código de corrección de errores es averiguar cuántas palabras tiene que generar para la versión elegida QR y el nivel de corrección de error. Esto se puede consultar en la Tabla 6[2].

Versión y Nivel de Corrección de Errores	Número Total de datos de palabras en clave para esta Versión y Nivel de Corrección de Errores	Palabras en clave de corrección de errores por bloque	Número de bloques en el Grupo 1	Número de palabras en clave de datos en cada uno de los bloques del Grupo 1	Número de bloques en el Grupo 2	Número de palabras en clave de datos en cada uno de los bloques del grupo 2	Total de datos de palabras en clave
1-L	19	7	1	19			(19*1) = 19
1-M	16	10	1	16			(16*1) = 16
1-Q	**13**	**13**	1	13			**(13*1) = 13**
1-H	9	17	1	9			(9*1) = 9

Tabla 6

Como ejemplo se usa la versión 1 con el nivel de corrección de errores Q. Esta combinación requiere 13 bloques de datos, que se generan en el paso anterior y 13 palabras de código de corrección de errores, los cuales generarán en el paso 2.

[1] El 3er paso será mencionado en la sección de Trabajo a Futuro.
[2] Para informarse acerca demás versiones consulte [6].

Análisis y diseño de la clase referente al Paso 1.
(Segunda Parte del Algoritmo)

La clase contiene un solo atributo llamado "strPaso1_2", el cual representa las cadenas resultantes del Paso 1 del a segunda parte del algoritmo. Así mismo, la clase contendrá dos métodos como son el constructor "Paso1_2" y el método llamado "hazPaso1_2", el cual, como su nombre lo indica, está encargado de realizar el Paso 1 ya descrito, y retornar el atributo "strPaso1_2" con el valor de las cadenas resultantes del paso. Lo anterior se representa en el Diagrama 9.

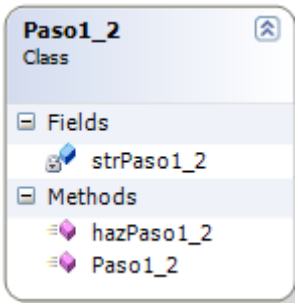

Diagrama 9 - Estructura de la Clase Paso1_2.

Implementación de la clase referente al Paso 1.

```
class Paso1_2
    {
        string strPaso1_2; //Atributo de clase, el "_2" indica que es referente al paso 1
           //de la 2da parte del algoritmo.
        public Paso1_2() // Método constructor.
        { }
        public string hazPaso1_2()
        {
            strPaso1_2 = "Versión: 1-Q \r\nCE palabras codificadas por bloque: 13";
               //Cadena que indica la versión del código QR implementado y cuántas
               palabras tiene que generar.

            return strPaso1_2; //Retorna los datos solicitados.
        }
    }
```

Código 10 – Clase referente al Paso 1 de la segunda parte del algoritmo de generación de Códigos QR.

Antes de continuar con el análisis, diseño e implementación del siguiente paso es importante realizar la siguiente explicación:

REVISIÓN DE POLINOMIOS.

Reed-Solomon de corrección de errores utiliza polinomios, como el siguiente:

$$2x^{8} + 3x^{61} + 5x + 7$$

Los números a la izquierda de las X (2, 3, 5, 7) son **coeficientes**.
Cada cosa que no es un signo más ($2x^{8}$, $3x^{61}$, 5x, 7) es un **término**.
Los números por encima de las X (8, 61) son **exponentes**.

Paso 2: Crear un polinomio del mensaje.

El primer paso es el de construir un polinomio llamado el -**polinomio del mensaje**- que utiliza los bloques de datos que se generan en la primera parte del algoritmo (primera etapa del proyecto). Para una mejor comprensión de este paso se explicara por medio del ejemplo propuesto que se ha venido mencionando en pasos anteriores.

Se han creado 13 bloques de datos en el paso anterior (Paso 6 de la 1ra parte del algoritmo), por lo que el polinomio del mensaje tendrá 13 términos en el mismo. Cada bloque de datos será el coeficiente de cada término.

Aquí están los 13 bloques de datos que se generan en la etapa anterior.

00100000_2 01011011_2 00001011_2 01111000_2 11010001_2 01110010_2 11011100_2 01001101_2 01000011_2 01000000_2 11101100_2 00010001_2 11101100_2

Si convierte cada palabra de 8 bits de binario a decimal, obtenemos:

32, 91, 11, 120, 209, 114, 220, 77, 67, 64, 236, 17, 236.

El uso de éstos resultan ser los coeficientes de un polinomio, obteniendo el siguiente **polinomio del mensaje:**

$$32x^{25} + 91x^{24} + 11x^{23} + 120x^{22} + 209x^{21} + 114x^{20} + 220x^{19} + 77x^{18} + 67x^{17} + 64x^{16} + 236x^{15} + 17x^{14} + 236x^{13}$$

El exponente del primer término es: $(1)^1$.

(Número de bloques de datos) + (Número de palabras de código de error de corrección) – 1

En este caso, esto es 13 + 13 - 1 = 25. Por lo tanto, el primer término del polinomio es $32x^{25}$.

Análisis y diseño de la clase referente al Paso 1.
(Segunda Parte del Algoritmo)

La clase contiene un solo atributo llamado "strPaso2_2", el cual representa los resultados obtenidos del Paso 2 del a segunda parte del algoritmo. Así mismo, la clase contendrá dos métodos como son el constructor "Paso2_2" y el método llamado "hazPaso2_2", el cual, como su nombre lo indica, está encargado de realizar el Paso 2 ya descrito, y retornar el atributo "strPaso2_2" con el valor de las cadenas resultantes del paso. Lo anterior se representa en el Diagrama 10.

[1] Numero para tomar referencia en la implementación.

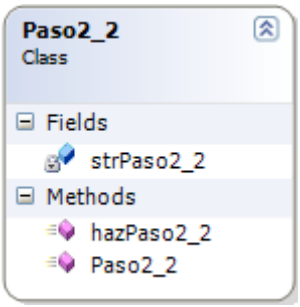

Diagrama 10 - Estructura de la Clase Paso2_2.

Implementación de la clase referente al Paso 2.

```
class Paso2_2
    {
        string strPaso2_2 = ""; //Atributo de clase, el "_2" indica que es referente al
          //paso 2 de la 2da parte del algoritmo.

        public Paso2_2() // Método constructor.
        { }

        public string hazPaso2_2(string strPaso6)
        {
            string aux = strPaso6.Replace(" ", ""); //Se quitan espacios de la cadena del
              //paso 6 de la parte 1 del algoritmo.
            string[] paso6Array = new string[(aux.Length / 8)]; //Arreglo que contendrá
              //cadenas de 8 obtenidas de la partición del resultado del paso 6.
            convertidorDec num = new convertidorDec(); // "num" Objeto de la clase
              //convertidorDec la cual convierte números binarios a decimales para
              //obtener los coeficientes del polinomio.
            int[] numDec = new int[(aux.Length / 8)]; //Arreglo que guarda el valor de
              //los coeficientes del polinomio.
            int j, k = 0; //Índices.

            for (int i = 0; i < paso6Array.Length; i++)
            {//Ciclo encargado de llenar el arreglo "paso6Array".
                j = 0;
                while (j < 8 && k < aux.Length)
                { //Ciclo encargado de particional la cadena resultante del paso 6 en
                  //palabras de 8.
                    paso6Array[i] = paso6Array[i] + aux[k];
                    k++;
                    j++;
                }
            }

            for (int i = 0; i < paso6Array.Length; i++)
            {//Ciclo que convierte cada palabra de 8 en su número decimal.
                numDec[i] = num.convierteADec(paso6Array[i]);
            }

            for (int i = 0; i < numDec.Length; i++)
            {//Ciclo encargado de generar el polinomio del mensaje donde los coeficientes
             //de obtienen de "numDec" y los exponentes de la operación
             //"(13+(13 - i)-1)" (1).
```

```
                strPaso2_2 = strPaso2_2+numDec[i] + "x^" + (13+(13 - i)-1);
                if (i < numDec.Length - 1) strPaso2_2 = strPaso2_2 + " + "; //If que
                  //coloca el símbolo de suma "+" entre cada término del polinomio.
            }
            return strPaso2_2; //Retorna el polinomio del mansaje.
        }
    }
```

Código 11 – Clase referente al Paso 2 de la segunda parte del algoritmo de generación de Códigos QR.

ANÁLISIS, DISEÑO E IMPLEMENTACIÓN DE UNA APLICACIÓN.

En esta sección se describe el análisis, diseño e implementación de una aplicación creada con el fin de comprobar el correcto funcionamiento de las clases referentes a los pasos de la segunda parte del algoritmo de generación de código QR, esto al complementarse con las generadas en la primera etapa del proyecto. Esto con el fin de hacer comparaciones y que se obtengan los mismos resultados a los expuestos en el ejemplo del material de consulta [2].

La aplicación se desarrollo utilizando como IDE Visual Studio Forms, donde primero se diseño la interfaz gráfica mostrada en la **¡Error! No se encuentra el origen de la referencia.**.

Figura 3 - Interfaz gráfica de la aplicación implementada.

Una vez concluida la interfaz gráfica se realizó la programación de cada componente de la misma, obteniendo como resultado el siguiente código:

```
public partial class Form2daPart : Form
    {
        string strPaso6; //Variable que recupera los datos de entrada, provenientes de la
           //1ra fase.
        bool pa1_2 = false; //Variable booleana para comprobar que el paso 1 de la 2da
           //fase del algoritmo, se ha realizado.
```

```
        public Form2daPart(string strPaso6)
        {
            InitializeComponent();//Método que es automáticamente creado y gestionado por
               //Windows Forms Designer y define todo lo que se ve en el formulario.
            this.strPaso6 = strPaso6; //Asignación de valor enviado del primer Form.
        }

        private void btnPaso1_2_Click(object sender, EventArgs e)
        {
            Paso1_2 strPaso1_2 = new Paso1_2();//Declaración de un Objeto de la Clase
              //Paso1_2.
            txBxPaso1.Text = strPaso1_2.hazPaso1_2();//Muestra el resultado de la
              //ejecución del Paso 1 en el TextBox titulado "Paso 1:".
            pa1_2 = true; //Paso 1 realizado.
        }

        private void btnPaso2_2_Click(object sender, EventArgs e)
        {
            if (pa1_2 == true) //Sentencia que verifica que el Paso 1 se ha realizado.
            {
                Paso2_2 strPaso2_2 = new Paso2_2(); //Declaración de un Objeto de la Clase
                  //Paso2_2.
                txBxPaso2.Text = strPaso2_2.hazPaso2_2(strPaso6);//Muestra el resultado de
                  //la ejecución del Paso 2 en el TextBox titulado "Paso 2:".
            }
            else MessageBox.Show("Primero realice el paso 1"); //Si no se ha realizado el
              //paso 1 se le informa al usuario.
        }
    }
```

Código 12 - Programación de componentes de `Form2daPart : Form`

La estructura de este código se observa en el Diagrama 11.

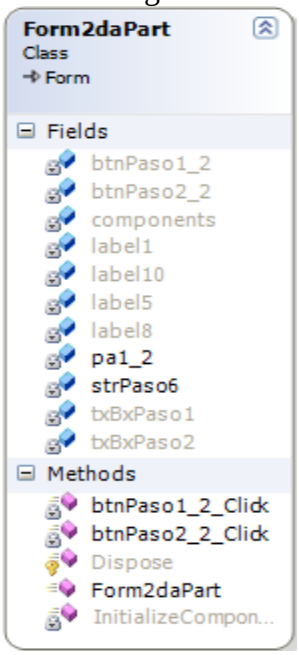

Diagrama 11 -
Estructura del código de la programación de componentes de `Form2daPart : Form`

A continuación, se realizaron las siguientes pruebas codificando información de entrada y para cada paso se obtuvieron los mismos resultados a los propuestos en el tutorial [2], como se muestra en la Figura 2Figura 4.

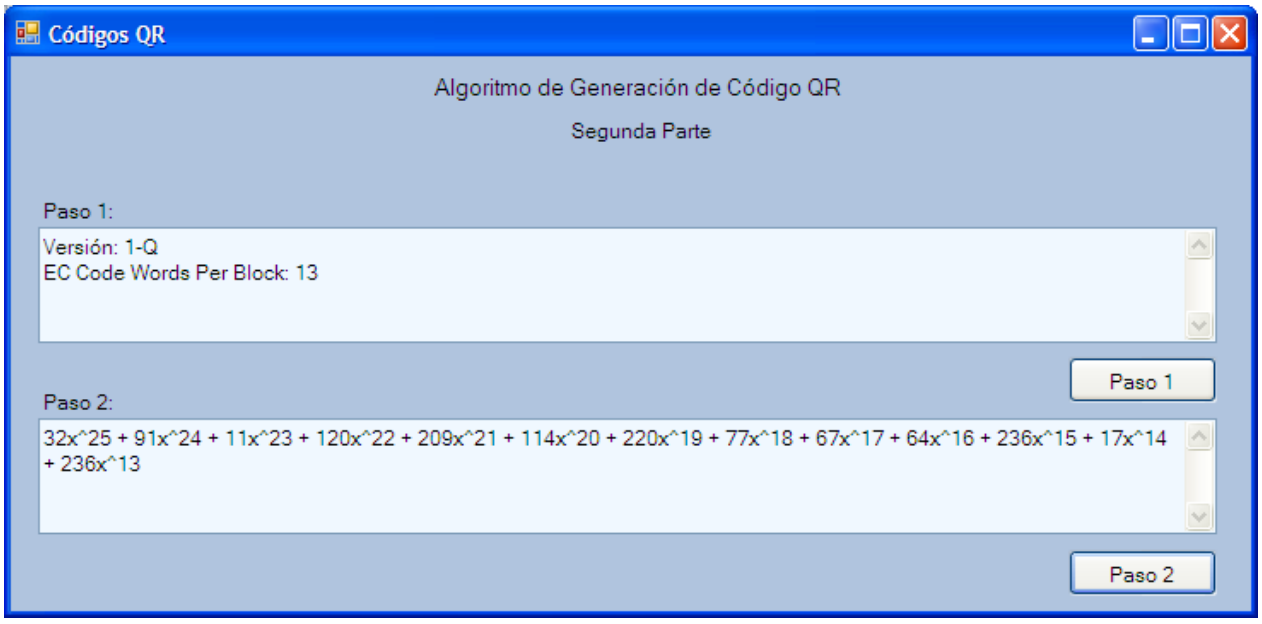

Figura 4
Captura de la aplicación donde se aprecia la funcionalidad de los Pasos 1 y 2, de la segunda parte del algoritmo de generación de Códigos QR.

- **MODIFICACIONES A LAS CLASES Y A LAS APLICACIONES, RESULTANTES DE LA PRIMERA Y SEGUNDA ETAPA.**

En esta sección del presente trabajo, se va a presentar el cómo es posible la modificación de las clases resultantes de la primera y principio de la segunda etapa del proyecto, con el fin de implementar la Versión 8 de generación de código QR; con el propósito de demostrar la utilidad de tener implementado el algoritmo de generación de código QR como biblioteca de código abierto, pues esta puede ser modificada con el fin de poder codificar mayor información de entrada.

PRIMERA PARTE DEL ALGORITMO

Paso 1: Codificar el Indicador de modo.

Para esté paso el indicador de modo será el mismo al ya explicado anteriormente, pues el modo de datos será el mismo, alfanumérico (Tabla 7):

Cadena de bits	Modo de datos
0001_2	Modo numérico
0010_2	**Modo alfanumérico**
0100_2	Modo binario
1000_2	Modo japonés

Tabla 7

Por lo tanto a la clase *Paso1*, en la cual se implementó este paso, no se le hará modificación alguna.

Paso 2: Codificar la longitud de los datos.

En este paso, como ya se explicó, se determina el número de caracteres que hay en nuestro mensaje de entrada, el cual será convertido a un número binario. Ahora, para la implementación de la Versión 8 el mensaje de entrada va a ser de un mayor número de caracteres, como el mencionado en la introducción:

Examen tipo: Parcial 3
Subtipo: 2
Grupo: CSI06 (trim. 12P)
Fecha: 16/Jul/2012 11:31
UEA: ESTRUCTURA DE DATOS CON ORIENTACION A OBJETOS (1151008)
Estudiante: HERNANDEZ PEREZ JOSE ALBERTO
Matrícula: 210332442

Haciendo un recuento de caracteres es un total de 212 caracteres. Entonces el número 212 se convierte al número binario 11010100_2.

Para codificar la longitud de los datos se codifica utilizando un número determinado de bits. Entonces, para la Versión 8, como se muestra en la Lista 1, cuando se utiliza el modo alfanumérico y la versión de la 1 a la 9, tenemos que usar 9 bits para codificar la longitud de los

datos. Por lo tanto, si tenemos 11010100_2 se pondrán 0's a la izquierda con el fin de obtener una palabra con longitud de 9-bits, así: 011010100_2.

Lista 2

Versiones 1 al 9

- Modo Numérico: 10 bits
- Modo Alfanumérico: 9 bits
- Modo binario: 8 bits
- Modo Japonés: 8 bits

Para consultar demás versiones *véase* [1]

Uniendo esta ultima cadena binaria a la cadena binaria obtenida en el Paso 1, como se muestra a continuación:

0010_2 **011010100_2**

El análisis, diseño e implementación de esta clase (*Paso 2)* se conserva estructuralmente igual a la descrita en la primera etapa; la diferencia va a radicar en el código donde se va a implementar, pues los parámetros, como el número máximo de caracteres que se pueden ingresar en los datos de entrada, va a aumentar de 16 a 279 caracteres (Tabla 8).

Versión	Módulos	ECC Nivel	Data bits	Numérico	Alfanumérico	Binario	Kanji
8	**49x49**	**L**	**1,552**	**461**	**279**	**192**	**118**
		M	1,232	365	221	152	93
		Q	880	259	157	108	66
		H	688	202	122	84	52

Tabla 8

Paso 3: Codificar los datos.

Para implementar la Versión 8, este paso no tiene modificaciones, pero a continuación se hace un recordatorio, que para codificar los datos alfanuméricos, primeramente, las cadenas de entrada, como son:

Examen tipo: Parcial 3
Subtipo: 2
Grupo: CSI06 (trim. 12P)
Fecha: 16/Jul/2012 11:31
UEA: ESTRUCTURA DE DATOS CON ORIENTACION A OBJETOS (1151008)
Estudiante: HERNANDEZ PEREZ JOSE ALBERTO
Matrícula: 210332442

Se debe romper en pares de caracteres obteniendo 106 pares, donde para cada par de caracteres, se toma el valor alfanumérico del primer carácter y se multiplica por 45, obtenido dicho valor esté se suma al valor alfanumérico del segundo carácter.

Luego, el resultado obtenido se convierte a una cadena binaria de 11-bits. Si se desea codificar un número impar de caracteres, se tomar valor alfanumérico del carácter final y se convierte a una

cadena binaria de 6-bits. Las cadenas binarias resultantes (106) se concatenarán con las obtenidas en los pasos anteriores, como se muestra en la Tabla 9.

Cadenas binarias de los pasos 1 y 2	Cadenas binarias del Paso 3
0010_2 011010100_2	010100101111_2 001110110001_2 010100011101_2 110011100011_2 011010000111_2 100011001001_2 110011011011_2 001110111011_2 010001011101_2 001110101111_2 110010101111_2 000000000001_2 101000010101_2 010000011001_2 011010000111_2 100011001001_2 110010101101_2 000000000001_2 010111010111_2 101010111111_2 100011001001_2 110011000001_2 100111111101_2 000000001101_2 110010101001_2 101001101001_2 011010000001_2 111100001101_2 000001011111_2 100011001011_2 000000000001_2 010101100011_2 010001011011_2 001111011101_2 110010101011_2 001001110011_2 011011101011_2 011110111001_2 000010110101_2 000001011111_2 110010101011_2 000010110011_2 000100010001_2 000000000001_2 101010101001_2 001111011101_2 110011000101_2 101000010011_2 100110111011_2 010001011011_2 011100011111_2 100011001011_2 101011000011_2 001111001101_2 011000010111_2 100011001111_2 110010111101_2 110011011001_2 010100010111_2 100010101001_2 110011000001_2 100010011111_2 110011011001_2 100110001011_2 010100010111_2 101001001111_2 110010111101_2 110011011001_2 010000000101_2 010100100111_2 100010101001_2 110010101001_2 000001011101_2 000111000101_2 000000000001_2 001011010001_2 000000000001_2 010100100101_2 101001101111_2 010010110111_2 001110110011_2 101001000111_2 111111000001_2 011000010111_2 100110101101_2 001110110011_2 010010101111_2 110010010111_2 100011100111_2 100110011011_2 110010010111_2 011011011111_2 100111110101_2 110010111101_2 011101111001_2 010100100011_2 101001100011_2 000000011011_2 011111010001_2 101001101001_2 001011101001_2 101010110111_2 001111011101_2 110010101101_2 000001011011_2 000100010101_2 000010111101_2 000101101101_2

Tabla 9

Cabe mencionar que la clase llamada “Paso123”, no se le hicieron modificaciones, pues solo es encargada de la unión de las cadenas resultantes de los pasos 1,2 y 3.

Paso 4: Termine los bits.

Una vez que se tiene la cadena de bits de los pasos anteriores hay que asegurarse de que es de la longitud correcta. Esto depende del número de bits de datos que son necesarios para generar la versión y la corrección de errores que se esté usando.

Para la implementación de la Versión 8 se ha optado por utilizar una versión de código QR con la corrección de nivel de error L. Para ello, se deben generar 1,552 bits de datos como se muestra en la Tabla 10 **[3]**.

Versión	Módulos	ECC Nivel	Data bits	Numérico	Alfanumérico	Binario	Kanji
8	49x49	**L**	**1,552**	**461**	**279**	**192**	**118**
		M	1,232	365	221	152	93
		Q	880	259	157	108	66
		H	688	202	122	84	52

Tabla 10

Si la cadena de bits es menor que 1,552, entonces se agregan hasta cuatro 0´s al final. Si dicha adición hace que la cadena sea mayor de 1,552, entonces sólo se agrega el número de ceros necesarios para que la cadena sea de 1,552 bits de longitud.

Para el ejemplo mencionado, la cadena es de 1,179 bits de largo, por lo tanto se añaden cuatro 0s en el final, obteniendo un total de 1,183 bits de longitud. Si dicha cadena hubiera terminado siendo de 1,150 bits de longitud (sólo como ejemplo), únicamente se le añadirían dos 0s hasta el final, para un total de 1,552 bits.

Por lo tanto para este paso la cadena binaria del ejemplo seria:

0010_2 011010100_2 010100101111_2 001110110002_2 010100011012_2 110011100012_2 011010000112_2
100011001002_2 110011011012_2 001110111012_2 010001011102_2 001110101112_2 110010101112_2
000000000002_2 101000010102_2 010000011002_2 011010000112_2 100011001002_2 110010101102_2
000000000002_2 010111010112_2 101010111112_2 100011001002_2 110011000002_2 100111111102_2
000000001102_2 110010101002_2 101001101002_2 011010000002_2 111100001102_2 000001011112_2
100011001012_2 000000000002_2 010101100012_2 010001011012_2 001111011102_2 110010101012_2
001001110012_2 011011101012_2 011110111002_2 000010110102_2 000001011112_2 110010101012_2
000010110012_2 000100010002_2 000000000002_2 101010101002_2 001111011102_2 110011000102_2
101000010012_2 100110111012_2 010001110012_2 101011000012_2 001111001102_2 010010101112_2
110011000012_2 001110111112_2 100010101002_2 110011000002_2 100010011112_2 110011011002_2
100110100012_2 010100011012_2 101001000112_2 010001011102_2 100010011112_2 110010111102_2
110011011002_2 010000000102_2 010100100112_2 100010101002_2 110010101002_2 000001011102_2
000111000102_2 000000000002_2 001011010002_2 000000000002_2 010100100102_2 101001101112_2
010010110112_2 001110110012_2 101001001112_2 111111000002_2 011000010112_2 100110101102_2
001110110012_2 010010101112_2 110010010112_2 100011100112_2 100110011012_2 110010010112_2
011011011112_2 100111110102_2 110010111102_2 011101111002_2 010100100012_2 101001100012_2
000000000002_2 011111010002_2 101001101002_2 001011101002_2 101010110112_2 001111011102_2
110010101102_2 000001011012_2 000100010102_2 000010111102_2 000101101102_2 $\mathbf{0000}_2$

Referente a éste paso los únicos cambios realizados se hicieron en el código fuente.

La estructura de este código es la misma (ya explicado anteriormente), por lo tanto, solo se verán reflejados los comentarios de las líneas de código modificadas.

```
public class Paso4
    {
        string strPaso4;

        public Paso4()
        { }

        public string hazPaso4(string datos)
        {
            Paso123 strPaso123 = new Paso123();
            strPaso4 = strPaso123.unePasos123(datos).Replace(" ", "");
            int i=0;

            while(strPaso4!=null && (1552-strPaso4.Length) < 4 && strPaso4.Length < 1552)
            { //Ciclo while que agrega menos de 4 ceros, si es que así se completa la
              //cadena de 1,152 bits de longitud.
                strPaso4 = strPaso4 + "0";
            }
            while (strPaso4 != null && (1552 - strPaso4.Length) > 4 && i < 4)
```

```
            { //Ciclo while que agrega un máximo de 4 ceros, si es que la cadena es menor
              //de 1,152 bits de longitud.
                strPaso4 = strPaso4 + "0";
                i++;
            }

            return strPaso4;
        }
    }
```

Código 13 – Modificaciones a la Clase referente al Paso 4.

Paso 5: Delimitar la cadena en palabras de 8 bits.

En éste paso la cadena se rompe en grupos de 8-bits. Si el último grupo no es de 8-bits de largo, se rellena con 0´s por la derecha.

En el ejemplo, el bloque final de la cadena es de sólo siete bits de largo, como se puede ver, así que la almohadilla de la derecha será con solo un cero.

Quedando, hasta el momento, la siguiente cadena binaria:

00100110_2 10100010_2 10010111_2 00111011_2 00001010_2 00110111_2 00111000_2 10110100_2
00111000_2 11001001_2 10011011_2 01001110_2 11101010_2 00101110_2 00111010_2 11111001_2
01011100_2 00000000_2 01010000_2 10100100_2 00011000_2 11010000_2 11100011_2 00100110_2
01010110_2 00000000_2 00001011_2 10101110_2 10101111_2 11000110_2 01001100_2 11000001_2
00111111_2 10000000_2 00110110_2 01010100_2 10100110_2 10001101_2 00000011_2 11000011_2
00000010_2 11111000_2 11001010_2 00000000_2 00010101_2 10001010_2 00101101_2 00111101_2
11011001_2 01010100_2 10011100_2 10110111_2 01010111_2 10111000_2 00010110_2 10000001_2
01111110_2 01010101_2 00001011_2 00100010_2 00100000_2 00000000_2 01010101_2 01000011_2
11011101_2 10011000_2 10101000_2 01001100_2 11011101_2 01000111_2 00110101_2 10000100_2
11110011_2 00100101_2 01111100_2 11000010_2 01110111_2 11100010_2 10100110_2 01100000_2
10001001_2 11111001_2 10110010_2 01101000_2 10101000_2 11011010_2 01000110_2 10001011_2
10100010_2 01111110_2 01011110_2 11001101_2 10001000_2 00001001_2 01001001_2 11000101_2
01001100_2 10101000_2 00001011_2 10000111_2 00010000_2 00000000_2 00101101_2 00000000_2
00000001_2 01001001_2 01010011_2 01110100_2 10110110_2 01110110_2 01101001_2 00111111_2
11100000_2 01100001_2 01110011_2 01011000_2 11101100_2 10100101_2 01111100_2 10010111_2
00011100_2 11100110_2 01101110_2 01001011_2 01101101_2 11110011_2 11101011_2 00101111_2
00111011_2 11000101_2 00100011_2 01001100_2 01000000_2 00000011_2 11101000_2 10100110_2
10000101_2 11010010_2 10101101_2 10011110_2 11101100_2 10101100_2 00001011_2 01000100_2
01010000_2 01011110_2 00010110_2 11000000_2

Paso 6: Agregar palabras al final si la cadena es demasiado corta.

Si aún la cadena de bits no es lo suficientemente larga, hay dos cadenas de bits especiales que son: 11101100_2 y 00010001_2, donde la especificación de códigos QR nos obliga a poner al final de la cadena, alternando entre las dos hasta que tengamos el número necesario de palabras de 8 bits.

Como se ha mencionado anteriormente, se deben generar 1552 bits de datos, para un total de ciento noventa y cuatro palabras de 8 bits (1552/8 =194). La cadena hasta el momento sólo cuenta con 148 bloques de datos, así que se tienen que añadir cuarenta y seis cadenas especiales más.

Finalmente la cadena binaria queda de la siguiente manera:

00100110_2 10100010_2 10010111_2 00111011_2 00001010_2 00110111_2 00111000_2 10110100_2
00111000_2 11001001_2 10011011_2 01001110_2 11101010_2 00101110_2 00111010_2 11111001_2
01011100_2 00000000_2 01010000_2 10100100_2 00011000_2 11010000_2 11100011_2 00100110_2
01010110_2 00000000_2 00001011_2 10101110_2 10101111_2 11000110_2 01001100_2 11000001_2
00111111_2 10000000_2 00110110_2 01010100_2 10100110_2 10001101_2 00000011_2 11000011_2
00000010_2 11111000_2 11001010_2 00000000_2 00010101_2 10001010_2 00101101_2 00111101_2
11011001_2 01010100_2 10011100_2 10110111_2 01010111_2 10111000_2 00010110_2 10000001_2
01111110_2 01010101_2 00001011_2 00100010_2 00100000_2 00000000_2 01010101_2 01000011_2
11011101_2 10011000_2 10101000_2 01001100_2 11011101_2 01000111_2 00110101_2 10000100_2
11110011_2 00100101_2 01111100_2 11000010_2 01110111_2 11100010_2 10100110_2 01100000_2
10001001_2 11111001_2 10110010_2 01101000_2 10101000_2 11011010_2 01000110_2 10001011_2
10100010_2 01111110_2 01011110_2 11001101_2 10001000_2 00001001_2 01001001_2 11000101_2
01001100_2 10101000_2 00001011_2 10000111_2 00010000_2 00000000_2 00101101_2 00000000_2
00000001_2 01001001_2 01010011_2 01110100_2 10110110_2 01110110_2 01101001_2 00111111_2
11100000_2 01100001_2 01110011_2 01011000_2 11101100_2 10100101_2 01111100_2 10010111_2
00011100_2 11100110_2 01101110_2 01001011_2 01101101_2 11110011_2 11101011_2 00101111_2
00111011_2 11000101_2 00100011_2 01001100_2 01000000_2 00000011_2 11101000_2 10100110_2
10000101_2 11010010_2 10101101_2 10011110_2 11101100_2 10101100_2 00001011_2 01000100_2
01010000_2 01011110_2 00010110_2 11000000_2 **11101100_2 00010001_2 11101100_2 00010001_2**
11101100_2 00010001_2 11101100_2 00010001_2 11101100_2 00010001_2 11101100_2 00010001_2
11101100_2 00010001_2 11101100_2 00010001_2 11101100_2 00010001_2 11101100_2 00010001_2
11101100_2 00010001_2 11101100_2 00010001_2 11101100_2 00010001_2 11101100_2 00010001_2
11101100_2 00010001_2 11101100_2 00010001_2 11101100_2 00010001_2 11101100_2 00010001_2
11101100_2 00010001_2

Referente a éste paso los únicos cambios realizados se hicieron en el código fuente.

La estructura de este código es la misma al ya explicado anteriormente, por lo tanto, solo se verán reflejados los comentarios de las líneas de código modificadas.

```
public class Paso6
    {
        string strPaso6 = null;
        public Paso6()
        { }
        public string hazPaso6(string datos)
        {
            Paso5 strPaso5 = new Paso5();
            string aux = strPaso5.hazPaso5(datos).Replace(" ", "");
```

```
            string cadena = "1110110000010001";
            int x = 0, j = 1, k = 0, p = 0;
            while (aux.Length < 1552)
            {//Ciclo condicional que evalúa sí la longitud de la cadena obtenida en el
//Paso 5 es menor a 1,552, acompleta con los bits de las cadenas especiales.
                if (x < cadena.Length)
                {
                    aux = aux + cadena[x];
                    x++;
                }
                else x = 0;
            }

            k = aux.Length / 8 - 1; //Al índice k se le asigna el valor de la longitud de
//la cadena aux dividido entre (8-1), esto permitirá en el ciclo for separar //la cadena
de 1,552 en 194 subcadenas de 8 bits.

            for (int i = 0; i < aux.Length+k; i++)
            {
                if (j <= 8)
                {
                    strPaso6 = strPaso6 + aux[p];
                    p++;
                    j++;
                }
                else
                {
                    strPaso6 = strPaso6 + " ";
                    j = 1;
                }
            }
            return strPaso6;
        }
    }
```

Código 14 – Modificaciones a la Clase referente al Paso 6.

Hechas las mencionadas modificaciones en el código fuente, el Forms ya diseñado en la primera etapa (`Form1raPart : Form`) nos muestra el siguiente resultado (Figura 5):

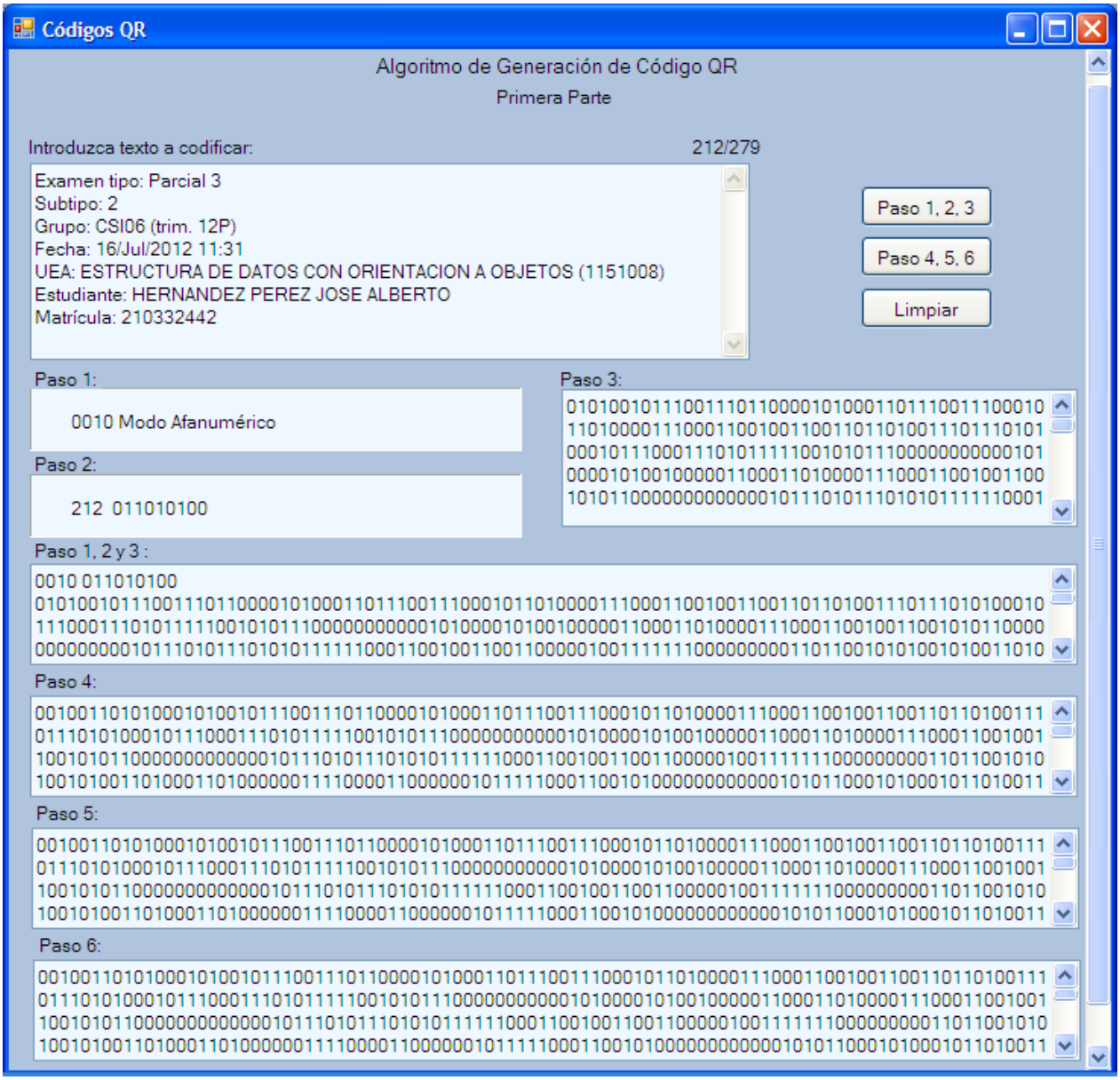

Figura 5 - Captura de la aplicación donde se aprecia la funcionalidad de los Pasos de la primera parte del algoritmo de generación de Códigos QR con las modificaciones hechas.

SEGUNDA PARTE DEL ALGORITMO

Paso 1: Averigüe cuántas palabras de código de corrección de error es necesario generar.

Como ya se ha mencionado, el primer paso para la generación de las palabras de código de corrección de errores es averiguar cuántas palabras tiene que generar para la versión elegida QR y el nivel de corrección de error. Esto se puede consultar en la Tabla 11[1].

Versión y Nivel de Corrección de Errores	Número Total de datos de palabras en clave para esta Versión y Nivel de Corrección de Errores	Palabras en clave de corrección de errores por bloque	Número de bloques en el Grupo 1	Número de palabras en clave de datos en cada uno de los bloques del Grupo 1	Número de bloques en el Grupo 2	Número de palabras en clave de datos en cada uno de los bloques del grupo 2	Total de datos de palabras en clave
8-L	**194**	**24**	**2**	**97**			**(97*2) = 194**
8-M	154	22	2	38	2	39	(38*2) + (39*2) = 154
8-Q	110	22	4	18	2	19	(18*4) + (19*2) = 110
8-H	86	26	4	14	2	15	(14*4) + (15*2) = 86

Tabla 11

Para la implementación de la Versión 8, con el nivel de corrección de errores L, se requiere que la combinación sea de 194 bloques de datos, que se generan en el paso anterior (Paso 6, primera parte del algoritmo) y 24 palabras de código de corrección de errores, los cuales generarán en el Paso 2 (Segunda Parte del Algoritmo).

Para la implementación de la Versión 8, referente a éste paso los únicos cambios realizados se hicieron en el código fuente. La estructura de este código es la misma al ya explicado anteriormente, por lo tanto, solo se verán reflejados los comentarios de las líneas de código modificadas.

```
class Paso1_2
    {
        string strPaso1_2;
        public Paso1_2()
        { }
        public string hazPaso1_2()
        {
            strPaso1_2 = "Versión: 8-L \r\nCE palabras codificadas por bloque: 24";
              //Cadena donde se está indicando la implementación de la Versión 8 del
              //Código QR y que se tienen que generar 24 palabras.

            return strPaso1_2;
        }
    }
```

Código 15 – Modificaciones a la Clase referente al Paso 1 de la segunda parte del algoritmo de generación de Códigos QR.

[1] Para informarse acerca demás versiones consulte [6].

Paso 2: Crear un polinomio del mensaje.

El primer paso es el de construir el polinomio del mensaje que utiliza los bloques de datos que se generan en la primera parte del algoritmo (primera etapa del proyecto). Ya se han creado 194 bloques de datos en el paso anterior (Paso 6 de la 1ra parte del algoritmo), por lo que, el polinomio del mensaje tendrá 194 términos en el mismo. Cada bloque de datos será el coeficiente de cada término, por lo tanto, si se convierte cada palabra de 8 bits de binario a decimal, obtenemos:

38, 162, 151, 59, 10, 55, 56, 180, 56, 201, 155, 78, 234, 46, 58, 249, 92, 0, 80, 164, 24, 208, 227, 38, 86, 0, 11, 174, 175, 198, 76, 193, 63, 128, 54, 84, 166, 141, 3, 195, 2, 248, 202, 0, 21, 138, 45, 61, 217, 84, 156, 183, 87, 184, 22, 129, 126, 85, 11, 34, 32, 0, 85, 67, 221, 152, 168, 76, 221, 71, 53, 132, 243, 37, 124, 194, 119, 226, 166, 96, 137, 249, 178, 104, 168, 218, 70, 139, 162, 126, 94, 205, 136, 9, 73, 197, 76, 168, 11, 135, 16, 0, 45, 0, 1, 73, 83, 116, 182, 118, 105, 63, 224, 97, 115, 88, 236, 165, 124, 151, 28, 230, 110, 75, 109, 243, 235, 47, 59, 197, 35, 76, 64, 3, 232, 166, 133, 210, 173, 158, 236, 172, 11, 68, 80, 94, 22, 192, 236, 17

Por lo tanto, se obtiene el siguiente polinomio del mensaje:

$$38x^{217} + 162x^{216} + 151x^{215} + 59x^{214} + 10x^{213} + 55x^{212} + 56x^{211} + 180x^{210} + 56x^{209} + 201x^{208} + 155x^{207} + 78x^{206} + 234x^{205} + 46x^{204} + 58x^{203} + 249x^{202} + 92x^{201} + 0x^{200} + 80x^{199} + 164x^{198} + 24x^{197} + 208x^{196} + 227x^{195} + 38x^{194} + 86x^{193} + 0x^{192} + 11x^{191} + 174x^{190} + 175x^{189} + 198x^{188} + 76x^{187} + 193x^{186} + 63x^{185} + 128x^{184} + 54x^{183} + 84x^{182} + 166x^{181} + 141x^{180} + 3x^{179} + 195x^{178} + 2x^{177} + 248x^{176} + 202x^{175} + 0x^{174} + 21x^{173} + 138x^{172} + 45x^{171} + 61x^{170} + 217x^{169} + 84x^{168} + 156x^{167} + 183x^{166} + 87x^{165} + 184x^{164} + 22x^{163} + 129x^{162} + 126x^{161} + 85x^{160} + 11x^{159} + 34x^{158} + 32x^{157} + 0x^{156} + 85x^{155} + 67x^{154} + 221x^{153} + 152x^{152} + 168x^{151} + 76x^{150} + 221x^{149} + 71x^{148} + 53x^{147} + 132x^{146} + 243x^{145} + 37x^{144} + 124x^{143} + 194x^{142} + 119x^{141} + 226x^{140} + 166x^{139} + 96x^{138} + 137x^{137} + 249x^{136} + 178x^{135} + 104x^{134} + 168x^{133} + 218x^{132} + 70x^{131} + 139x^{130} + 162x^{129} + 126x^{128} + 94x^{127} + 205x^{126} + 136x^{125} + 9x^{124} + 73x^{123} + 197x^{122} + 76x^{121} + 168x^{120} + 11x^{119} + 135x^{118} + 16x^{117} + 0x^{116} + 45x^{115} + 0x^{114} + 1x^{113} + 73x^{112} + 83x^{111} + 116x^{110} + 182x^{109} + 118x^{108} + 105x^{107} + 63x^{106} + 224x^{105} + 97x^{104} + 115x^{103} + 88x^{102} + 236x^{101} + 165x^{100} + 124x^{99} + 151x^{98} + 28x^{97} + 230x^{96} + 110x^{95} + 75x^{94} + 109x^{93} + 243x^{92} + 235x^{91} + 47x^{90} + 59x^{89} + 197x^{88} + 35x^{87} + 76x^{86} + 64x^{85} + 3x^{84} + 232x^{83} + 166x^{82} + 133x^{81} + 210x^{80} + 173x^{79} + 158x^{78} + 236x^{77} + 172x^{76} + 11x^{75} + 68x^{74} + 80x^{73} + 94x^{72} + 22x^{71} + 192x^{70} + 236x^{69} + 17x^{68} + 236x^{67} + 17x^{66} + 236x^{65} + 17x^{64} + 236x^{63} + 17x^{62} + 236x^{61} + 17x^{60} + 236x^{59} + 17x^{58} + 236x^{57} + 17x^{56} + 236x^{55} + 17x^{54} + 236x^{53} + 17x^{52} + 236x^{51} + 17x^{50} + 236x^{49} + 17x^{48} + 236x^{47} + 17x^{46} + 236x^{45} + 17x^{44} + 236x^{43} + 17x^{42} + 236x^{41} + 17x^{40} + 236x^{39} + 17x^{38} + 236x^{37} + 17x^{36} + 236x^{35} + 17x^{34} + 236x^{33} + 17x^{32} + 236x^{31} + 17x^{30} + 236x^{29} + 17x^{28} + 236x^{27} + 17x^{26} + 236x^{25} + 17x^{24}$$

El exponente de cada término está compuesto de la siguiente forma: $(2)^{1}$.

(Número de bloques de datos) + (Número de palabras de código de corrección de errores) – 1

[1] Numero para tomar referencia en la implementación.

Esto es, 194 + 24 - 1 = 217. Por lo tanto, el primer término del polinomio es $38x^{217}$.

Para la implementación de la Versión 8, referente a éste paso los únicos cambios realizados se hicieron en el código fuente. La estructura de este código es la misma al ya explicado anteriormente, por lo tanto, solo se verán reflejados los comentarios de las líneas de código modificadas.

```
class Paso2_2
    {
        string strPaso2_2 = "";
        public Paso2_2()
        { }

        public string hazPaso2_2(string strPaso6)
        {
            string aux = strPaso6.Replace(" ", "");
            string[] paso6Array = new string[(aux.Length / 8)];
            convertidorDec num = new convertidorDec();
            int[] numDec = new int[(aux.Length / 8)];
            int j, k = 0;
            for (int i = 0; i < paso6Array.Length; i++)
            {
                j = 0;
                while (j < 8 && k < aux.Length)
                {
                    paso6Array[i] = paso6Array[i] + aux[k];
                    k++;
                    j++;
                }
            }

            for (int i = 0; i < paso6Array.Length; i++)
            {
                numDec[i] = num.convierteADec(paso6Array[i]);
            }

            for (int i = 0; i < numDec.Length; i++)
            {//Ciclo encargado de generar el polinomio del mensaje donde los coeficientes
             //de obtienen de "numDec" y los exponentes de la operación
             //"(194 + (24 - i)-1)" (1).

                strPaso2_2 = strPaso2_2+numDec[i] + "x " + (194+(24 - i)-1);
                if (i < numDec.Length - 1) strPaso2_2 = strPaso2_2 + " + ";
            }
            return strPaso2_2;
        }
    }
```

Código 16 – Modificaciones a la Clase referente al Paso 2 de la segunda parte del algoritmo.

Una vez hechas las mencionadas modificaciones en el código fuente, el Forms ya diseñado en la primera etapa (`Form2daPart : Form`) nos muestra el siguiente resultado (Figura 6):

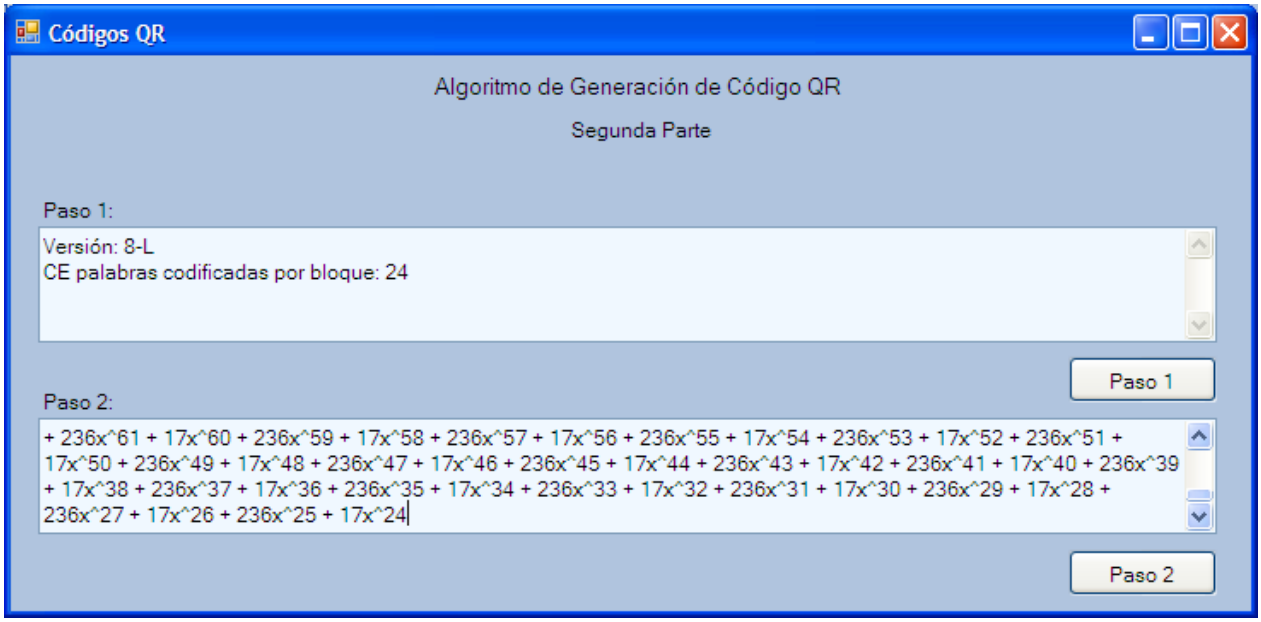

Figura 6 - Captura de la aplicación donde se aprecia la funcionalidad de los Pasos de la segunda parte del algoritmo de generación de Códigos QR con las modificaciones hechas.

CONCLUSIONES.

En la elaboración del presente proyecto se logró sintetizar automáticamente parte del algoritmo de generación de códigos QR en una biblioteca en lenguaje C#. Pues se consiguió delimitar el tipo de datos y capacidad de caracteres a codificar en el código QR resultante de la biblioteca. Se determinó el nivel de corrección de errores que contendría el código QR. También, se estableció el tamaño del código QR que se generaría. Posteriormente, se diseñó una biblioteca de código abierto basada en el algoritmo de código QR, en sus primeras dos etapas, donde dicha biblioteca se implementó y con ello se realizaron las respectivas pruebas codificando información de entrada, mostrando la variabilidad de capacidad que ésta puede tener. Así mismo, se diseñó una aplicación para comprobar la funcionalidad de la biblioteca, la cual a su vez se implementó, logrando así resultados satisfactorios en comparación a la propuesta hecha en el material de consulta.

Cabe mencionar que debido a que el algoritmo de generación de Códigos QR va más allá de un solo proyecto terminal, el presente no tuvo los alcances suficientes como para que se pudiera llegar a la parte final del algoritmo donde se genera el Código QR en una imagen en formato PNG, dicha parte del algoritmo se comenta en la siguiente sección de Trabajo a Futuro.

TRABAJO A FUTURO.

Posteriormente, a los pasos realizados a lo largo del presente proyecto se debe sumar el análisis, diseño e implementación de los que a continuación se mencionan con el propósito de alcanzar los objetivos propuestos originalmente, pues se estarían acompletando los pasos que componen el algoritmo de generación de Códigos QR.

SEGUNDA PARTE DEL ALGORITMO DE GENERACIÓN DE CÓDIGOS QR.

PASO 3: CREAR UN POLINOMIO GENERADOR

En éste paso se va a crear el polinomio generador. En donde se tiene que dividir el polinomio del mensaje (Paso 2) por el polinomio generador [8] para crear las palabras componentes del código de corrección de errores.

Es importante mencionar que el polinomio generador es usualmente conocido como campo finito o campo de Galois **¡Error! No se encuentra el origen de la referencia.**]. Los códigos QR utilizan un campo de Galois que tiene 256 elementos, lo que significa, que los números que se ocupan están en un rango entre máximo 255 y 0 como mínimo.

Para realizar la división de polinomios, se varía entre términos de α (alfa) y una notación de enteros. El alfa y valores enteros provienen de tablas de logaritmos y antilogaritmos. Por lo tanto, el polinomio generador tiene la forma $(x - \alpha)(x - \alpha^2)\ldots(x - \alpha^k)$, donde k es el número de palabras requeridas de código corrección de errores menos 1. Donde, posteriormente es necesario multiplicar todos aquellos (x - α) términos hasta que se tenga el polinomio generador.

TERCERA PARTE DEL ALGORITMO DE GENERACIÓN DE CÓDIGOS QR.

COLOCACIÓN DE MÓDULOS EN MATRIZ

Una vez codificados los datos, lo que resta es elegir el mejor patrón de máscara. Los patrones de máscara están definidos en el estándar de código QR. Donde, hay definidos 8 patrones.
Un código QR en particular puede tener ciertos patrones o rasgos que hacen que sea difícil, para los escáneres de QR, el poder leer precisión el código QR. Es decir, por ejemplo, si los módulos del mismo color se producen cerca, un escáner de QR podría tener problemas para leerlos con precisión, esto de acuerdo a reglas que más adelante se les hará mención.

Como se mencionó en un principio en los objetivos propuestos para el presente proyecto, el resultado final de la ejecución del algoritmo de generación de Códigos QR es una imagen, la cual se puede ver como una matriz donde en cada intersección se observan cuadros blancos y negros del código QR como módulos en lugar de píxeles.

Cada código QR debe incluir patrones de función, que son estructuras que deben ser colocadas en áreas específicas del código de QR para asegurar que los lectores de códigos QR puedan identificar y orientar correctamente el código de decodificación. En la Imagen 2 [10] se muestra que son los patrones de función y dónde se deben situar.

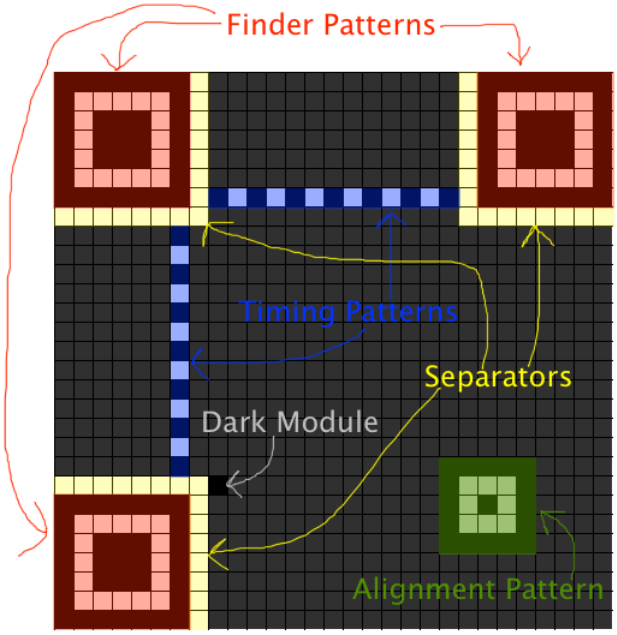

Imagen 2 – Ubicación de los patrones de función.

Descripción[1]:

Patrones del buscador: (*Finder Patterns*) son tres bloques en las esquinas del código QR, en la parte superior derecha, superior izquierda e inferior izquierda.

Separadores: (*Separators*) son módulos blancos colocados a los lados de los patrones del buscador.

Patrones de alineación: (*Alignament Pattern*) son similares a los patrones del buscador, pero más pequeños y se colocan a lo largo del código. Se utilizan de la versión 2 en adelante y sus posiciones dependen de la versión del código QR.

Patrones de tiempo: (*Timing patterns*) son módulos alternantes entre blanco y negro (líneas punteadas) que conectan los patrones del buscador.

Módulo oscuro: (*Dark Module*) es un módulo único negro que se coloca siempre al lado del patrón del buscador ubicado en la parte inferior izquierda.

COLOCACIÓN DE LOS BITS DE DATOS.

Colocados los patrones de función, ya mencionados, se procede a realizar la distribución de los bits de datos en la misma matriz. Los cuales se colocan en un patrón particular.

Patrón de colocación

Primeramente, los bits de datos se colocan en la parte inferior derecha de la matriz, posteriormente, se va avanzando hacia arriba en una columna, que es de 2 módulos de ancho. Una vez que se alcanza la parte superior de la columna, la siguiente columna empieza consecutivamente a la izquierda de la columna anterior y continúa hacia abajo. Si en el proceso de colocación se topa con un patrón de función o zona reservada, el bit de datos se coloca en el siguiente módulo no utilizado. En la Imagen 3 [12] se muestra el patrón de colocación de los bits de datos en el código QR.

[1] Para consultar como se debe colocar cada patrón de función, de acuerdo a la Versión de Código QR a implementar, *véase* [11]

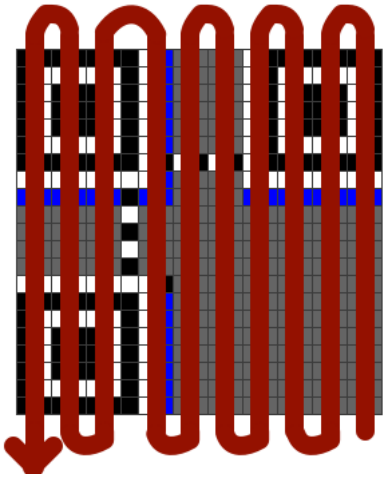

Imagen 3 – Patrón de colocación de bits de datos.

- ENMASCARADO DE DATOS

En la matriz del código QR pueden existir ciertos patrones que dificulten la lectura que realizan los escáneres de código QR. En respuesta a ello, la especificación del código QR precisa ocho patrones de máscara[1], cada uno de los cuales altera el código QR de acuerdo con un patrón particular. En base a esto, se debe determinar cuál da como resultado un código QR con el menor número de atributos indeseables.

El enmascaramiento consiste en cambiar el color del módulo, es decir, si se trata de un modulo de luz (blanco), éste será cambiado a obscuro (negro) y viceversa. Es importante mencionar que los patrones de máscara sólo se deben aplicar a los módulos de datos y a los módulos de corrección de errores. Es decir, no se van a enmascarar los patrones de función, ni las áreas reservadas (área de información del formato y área de información de la versión[2]).

Ahora bien, la determinación se realiza mediante la evaluación de cada matriz, enmascarando en base a las siguientes cuatro reglas de penalización:

- **Primera regla**: por cada conjunto de cinco o más módulos del mismo color en una fila o columna, otorga una sanción al código QR.
- **Segunda regla**: por cada área de 2x2 del mismo color, en los módulos de la matriz, le otorga una sanción al código QR.
- **Tercera regla**: si existen patrones que coinciden a los patrones del buscador (*Finder Patterns*), otorga una sanción mayor al código QR.
- **Cuarta regla**: si más de la mitad de los módulos son de color oscuro o claro, le otorga una sanción al código QR con una mayor penalización para poderlos diferenciar de forma más efectiva.

El proceso de enmascaramiento y el de evaluación de cada una de éstas reglas es explicado con más detalle en “Enmascaramiento de datos” (*Data Masking* [13]).

[1] Datos, que al realizárseles cierta operación, permiten extraer otros datos determinados.
[2] Se explican en el apartado: Información de formato y versión [14].

- INFORMACIÓN DE FORMATO Y VERSIÓN

Finalmente, para la creación de un código QR, el último paso es crear las cadenas de formato y de versión y posteriormente, colocarlos en el lugar correcto en el código QR, esto mediante la adición de los módulos en áreas particulares del código que se ha dejado en blanco en los pasos anteriores. Los píxeles de Formato se encargan de identificar el nivel de corrección de errores y patrón de la máscara que se utiliza en este código QR. Los pixeles versión se encargan de codificar el tamaño de la matriz de QR y sólo se utilizan en grandes códigos QR. Para adquirir más información acerca de este paso, consulte [14].

Añadir la Zona Muda

Antes de generar el código resultante de la implementación del algoritmo de generación del mismo, es importante tener en cuenta que una de las especificaciones del código QR demanda que la matriz QR este rodeada por una zona de silencio, que es un área de 4 módulos de ancho, los cuales son módulos de luz (blancos).

Después de mencionados los últimos puntos a tomar en cuenta, para la generación de códigos QR, en la Imagen 4 [14] se muestra la codificación en la versión 1-Q de la cadena "Hello World", codificado en modo alfanumérico.

Imagen 4 – Código QR.

ANEXOS

ANEXO 1

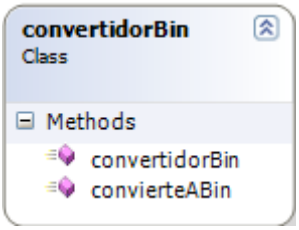

Diagrama de Clase

```
public class convertidorBin
{
    public convertidorBin() //Constructor de la clase.
    { }

    public string convierteABin(int longDatos)
    {
        int num;
        string bin = String.Empty, binc = String.Empty; //El valor de estos campos es
           //la cadena de longitud cero, "".

        num = longDatos;

        if(longDatos == 0) //Si se recibe un solo carácter y es 0.
        {
            binc += "0";
        }
        else if (longDatos == 1) //Si se recibe un solo carácter y es 1.
        {
            binc += "1";
        }
        else
        {
            do //Ciclo do-while que se cumple al menos una vez para convertir de
               //decimal a binario.
            {
                if (num % 2 == 1)
                    bin += "1";
                else
                    bin += "0";
                num /= 2;
            } while (num != 1);
            bin += "1";  //Siempre se le agrega un 1 al final.

            for (int i = 1; i <= bin.Length; i++) //Ciclo for para invertir el número
               //que ha quedado al revés.
               binc += bin[bin.Length - i];
        }

        return binc; //Retorna el valor del número decimal en binario.
    }
}
```

ANEXO 2

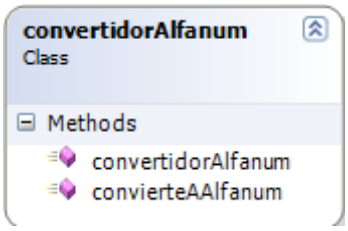

Diagrama de Clase

```
public class convertidorAlfanum
{
    public convertidorAlfanum() //Constructor.
    { }

    public int [] convierteAAlfanum(string datos)
    {
        int [] datosEnAlfa = new int [datos.Length]; //Arreglo de la misma longitud
           //que el número de caracteres de entrada.
        datos = datos.ToUpper(); //Solo letras mayúsculas.
        string datoChar;

        for (int i = 0; i < datos.Length;i++)
        {
            if(char.IsLetter(datos[i])) //Solo para las letras.
            {
                datosEnAlfa[i] = Math.Abs(Encoding.ASCII.GetBytes(datos)[i] - 55);
                   //Guarda en el arreglo el valor alfanumérico de cada carácter que
                   //es letra.
            }
            if (char.IsNumber(datos[i]))
            {
                //Guarda en el arreglo el valor alfanumérico de cada carácter que es
                //número.
                datoChar=Convert.ToString(datos[i]);
                datosEnAlfa[i] = int.Parse(datoChar);
            }
            //Si el carácter es igual al de la condición guarda en el arreglo su
            //valor alfanumérico.
            //Para caracteres especiales.
            if (datos[i] == ' ') datosEnAlfa[i] = 36;
            if (datos[i] == '$') datosEnAlfa[i] = 37;
            if (datos[i] == '%') datosEnAlfa[i] = 38;
            if (datos[i] == '*') datosEnAlfa[i] = 39;
            if (datos[i] == '+') datosEnAlfa[i] = 40;
            if (datos[i] == '-') datosEnAlfa[i] = 41;
            if (datos[i] == '.') datosEnAlfa[i] = 42;
            if (datos[i] == '/') datosEnAlfa[i] = 43;
            if (datos[i] == ':') datosEnAlfa[i] = 44;
        }

        return datosEnAlfa; //Regresa en un arreglo el valor alfanumérico de cada
           carácter recibido.
    }

}
```

ANEXO 3

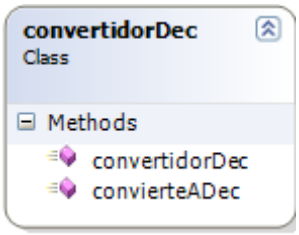

Diagrama de Clase

```
public class convertidorDec
{
    public convertidorDec() //Constructor de la clase.
    { }

    public int convierteADec(string dato) //Recibe la cadena a convertir.
    {
        int numDec,j=dato.Length-1;
        double num=0;
        for (int i = 0; i < dato.Length; i++)
        { //Ciclo for encargado de la conversión de la cadena binaria de entrada a su
          //valor decimal.
            if (dato[i] == '1')
            {
                num = (num + Math.Pow(2, j)); //De acuerdo a la posición en el arreglo
                    //será el valor en la potencia.
                j--;
            }
            else
            {
                j--;
            }
        }
        numDec = (int)num;

        return numDec; //Retorna el valor de lo recibido en decimal.
    }
}
```

BIBLIOGRAFÍA.

[1] Thonky, "QR Code Tutorial Part 1: Generating your binary string**".** [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/part-1-encode-data/ , 2012.

[2] Thonky, "QR Code Tutorial Part 2: Generating Error Correction Code Words". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/part-2-error-correction/, 2012.

[3]QR Code.com, Denso Wave INCORPORATED, "The maximum data capacity for each version". [En línea]. Disponible: http://www.qrcode.com/en/vertable1.html, 2012.

[4] Thonky, "Table of Alphanumeric Values". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/alphanumeric-table/ , 2012.

[5]Paginas Galegas, "generador de códigos QR". [En línea]. Disponible: http://www.paxinasgalegas.es/generadorQR.aspx , 2012.

[6] Thonky, "Error Correction Code Words and Block Information". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/error-correction-table/, 2012.

[7] Bernard Sklar, "Reed-Solomon Codes". [En línea]. Disponible: http://ptgmedia.pearsoncmg.com/images/art_sklar7_reed-solomon/elementLinks/art_sklar7_reed-solomon.pdf, 2012.

[8] Thonky, "Error Correction Coding", en su apartado: "Step 9: Divide the Message Polynomial by the Generator Polynomial". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/error-correction-coding/#step-3-understand-the-galois-field, 2013.

[9] Thonky, "Error Correction Coding", en su apartado: "Step 3: Understand The Galois Field" y "Step 4: Understand Galois Field Arithmetic". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/error-correction-coding/, 2013

[10] Thonky, "Module Placement in Matrix", en su apartado: "Overview of Function Patterns". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/module-placement-matrix/, 2013.

[11] Thonky, "Module Placement in Matrix", en sus apartados: "Step 1: Add the Finder Patterns" al "Step 5: Add the Dark Module and Reserved Areas". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/module-placement-matrix/, 2013.

[12] Thonky, "Module Placement in Matrix", en sus apartados: "Step 6: Place the Data Bits". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/module-placement-matrix/, 2013.

[13] Thonky, "Data Masking". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/data-masking/, 2013.

[14] Thonky, "Format and Version Information". [En línea]. Disponible: http://www.thonky.com/qr-code-tutorial/format-version-information/, 2013.