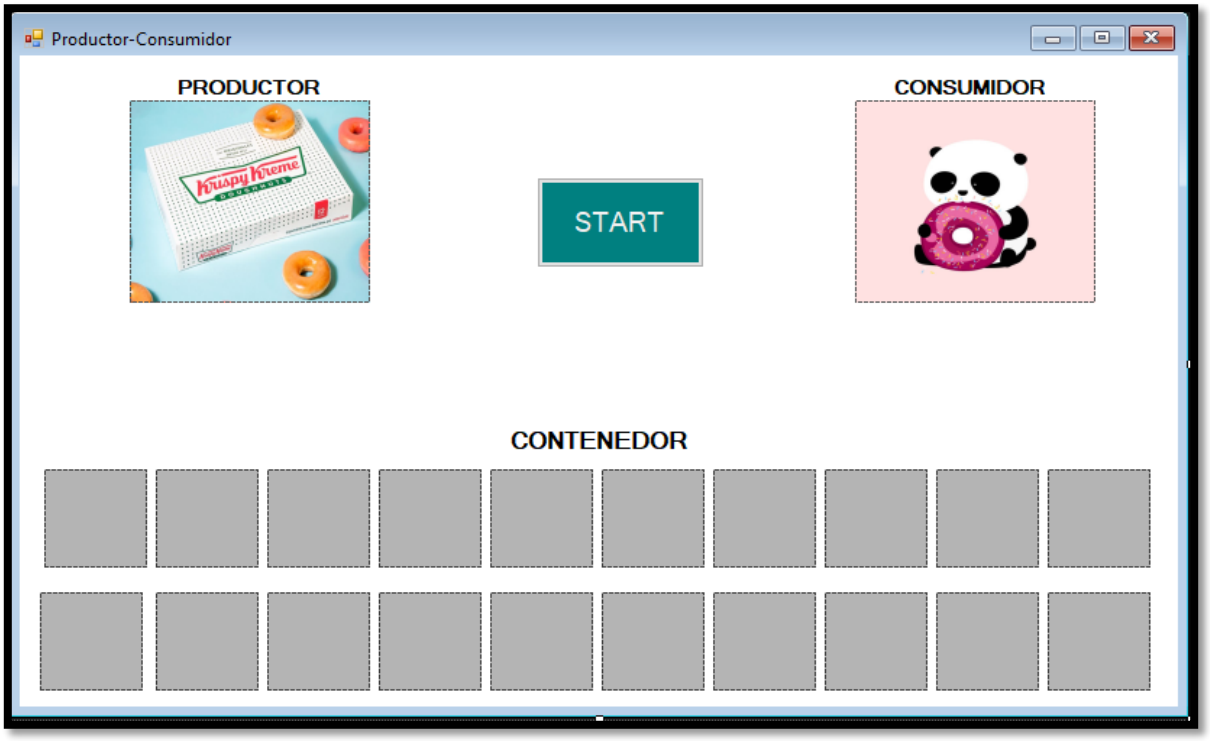
ESTRUCTURA

Este programa se divide en 4 partes, la primera es el “Form()” o ventana principal, donde se encuentra la interfaz grafica y se muestra el funcionamiento del programa.



Dentro del código de la interfaz tenemos el método “ChangeStateLabel()”, este método se encarga de cambiar el estado visual de las etiquetas que muestran el estado del productor y del consumidor en la interfaz de usuario.

```
3 referencias
public void ChangeStateLabel(int c, int p)
{
    switch (p)
    {
        case Constants.WORKING:
            producerState.Text = "PRODUCIENDO";
            producerState.BackColor = Color.Purple;
            producerState.Refresh();
            break;
        case Constants.TRYING:
            producerState.Text = "LLENO";
            producerState.BackColor = Color.MediumVioletRed;
            producerState.Refresh();
            break;
        default:
            producerState.Text = "DESCANSANDO";
            producerState.BackColor = Color.Plum;
            producerState.Refresh();
            break;
    }
    switch (c)
    {
        case Constants.WORKING:
            consumerState.Text = "CONSUMIENDO";
            consumerState.BackColor = Color.Purple;
            consumerState.Refresh();
            break;
        case Constants.TRYING:
            consumerState.Text = "VACIO";
            consumerState.BackColor = Color.MediumVioletRed;
            producerState.Refresh();
            break;
        default:
            consumerState.Text = "DESCANSANDO";
            consumerState.BackColor = Color.Plum
                ;
            consumerState.Refresh();
            break;
    }
}
```

```
2 referencias
public void SetImage(int index, int turn)
{
    Bitmap bmp = new Bitmap("..\\..\\fullDonnut.png");
    PictureBox pb = new PictureBox();

    switch (index)
    {
        case 0: pb = pictureBox1; break;
        case 1: pb = pictureBox2; break;
        case 2: pb = pictureBox3; break;
        case 3: pb = pictureBox4; break;
        case 4: pb = pictureBox5; break;
        case 5: pb = pictureBox6; break;
        case 6: pb = pictureBox7; break;
        case 7: pb = pictureBox8; break;
        case 8: pb = pictureBox9; break;
        case 9: pb = pictureBox10; break;
        case 10: pb = pictureBox11; break;
        case 11: pb = pictureBox12; break;
        case 12: pb = pictureBox13; break;
        case 13: pb = pictureBox14; break;
        case 14: pb = pictureBox15; break;
        case 15: pb = pictureBox16; break;
        case 16: pb = pictureBox17; break;
        case 17: pb = pictureBox18; break;
        case 18: pb = pictureBox19; break;
        case 19: pb = pictureBox20; break;
    }

    if (turn == Constants.PRODUCER_TURN)
    {
        pb.Image = bmp;
    }
    else
    {
        if (pb.Image != null)
        {
            pb.Image = null;
        }
    }
}
```

Después tenemos el método “SetImage()”, este método asigna imágenes a los PictureBox en la interfaz de acuerdo al índice proporcionado y el turno especificado (ya sea del productor o del consumidor).

```
public async void start()
{
    Consumer consumer = new Consumer();
    Producer producer = new Producer();
    int moves, i;
    do
    {
        moves = container.NextMoves();

        for (i = 0; i < moves; ++i)
        {
            await Task.Delay(1000);
            consumer = container.GetConsumer();
            producer = container.GetProducer();

            if (container.GetCurrentTurn() == Constants.CONSUMER_TURN)
            {
                if (consumer.getState() == Constants.WORKING)
                {
                    if (container.setAction(consumer.getCurrentPos(), consumer.Consume()))
                    {
                        SetImage(consumer.getCurrentPos(), Constants.CONSUMER_TURN);
                        consumer = container.GetConsumer();
                        consumer.setCurrentPos(consumer.getCurrentPos() + 1);
                    }
                    if (consumer.getCurrentPos() == Constants.CONTAINER_SIZE)
                    {
                        consumer.setCurrentPos(0);
                    }
                }
                if (consumer.getState() == Constants.TRYING)
                {
                    ChangeStateLabel(consumer.getState(), producer.getState());
                    break;
                }
            }
            else
            {
                if (producer.getState() == Constants.WORKING)
                {
                    if (container.setAction(producer.getCurrentPos(), producer.Produce()))
                    {
                        SetImage(producer.getCurrentPos(), Constants.PRODUCER_TURN);
                        producer = container.GetProducer();
                        producer.setCurrentPos(producer.getCurrentPos() + 1);
                    }
                    if (producer.getCurrentPos() == Constants.CONTAINER_SIZE)
                    {
                        producer.setCurrentPos(0);
                    }
                }
                if (producer.getState() == Constants.TRYING)
```

En el método “start()” inicia la simulación del productor y el consumidor, creando una instancia de “Consumer” y “Producer”, dentro de un bucle infinito se realizan movimientos en el contenedor y se actualizan las posiciones del productor y del consumidor. Tambien se manejan los eventos de tiempo de espera usando “Task.Delay()” para simular el tiempo entre movimientos y por ultimo se actualiza el estado visual de la interfaz de acuerdo a las acciones realizadas por el productor y el consumidor.

La clase “Producer” proporciona funcionalidades básicas para representar y manipular un producto en el sistema de simulación Productor-Consumidor, además de proporcionar métodos como “setState”, “getState”, “setCurrentPos” y “getCurrentPos” para establecer y obtener el estado y la posición actual del productor, respectivamente.

Algo muy similar ocurre en la clase “Consumer”, implementado los mismos métodos que la clase “Producer”, por lo que podemos omitir el enseñarla para no alargar demasiado el reporte.

Por último, tenemos la clase “Container”, la cual encapsula la lógica principal del sistema de simulación de Productor-Consumidor:

```
static class Constants
{
    public const int CONTAINER_SIZE = 20;
    public const int SLEEPING = 0;
    public const int WORKING = 1;
    public const int TRYING = 2;
    public const int PRODUCER_TURN = 1;
    public const int CONSUMER_TURN = 2;
}
```

En la clase “Constants” se guardan los valores fijos como el tamaño del contenedor, los estados posibles de los productores y consumidores, y los turnos de los mismos.

```
class Container
{
    private Producer producer;
    private Consumer consumer;
    private int currentTurn;
    private Random turnRandom;
    private Random amount;
    private bool[] buffer;

    public Container()
    {
        buffer = new bool[Constants.CONTAINER_SIZE];
        producer = new Producer();
        consumer = new Consumer();
        currentTurn = 0;
        turnRandom = new Random();
        amount = new Random();
        for (int i = 0; i < buffer.Length; ++i)
        {
            buffer[i] = false;
        }
    }
```

La clase “Container()” representa el contenedor que contiene los productos y maneja la lógica de turno entre productores y consumidores, tambien guarda sus variables locales para identificar productor, consumir, el turno actual y generar números aleatorios que servirán más adelante.

```
2 referencias
public bool setAction(int pos, bool action)
{
    if (action == buffer[pos])
    {
        if (action)
        {
            producer.setState(Constants.TRYING);
        }
        else
        {
            consumer.setState(Constants.TRYING);
        }
        return false;
    }
    buffer[pos] = action;
    return true;
}
```

El método “setAction” establece una acción en una posición específica del buffer y actualiza el estado del productor o el consumidor en función de la acción realizada.

```
1 referencia
public int NextMoves()
{
    int p, a;
    p = turnRandom.Next() % 100;
    a = amount.Next(3, 7);

    if (p % 2 == 0)
    {
        currentTurn = Constants.PRODUCER_TURN;
    }
    else
    {
        currentTurn = Constants.CONSUMER_TURN;
    }

    if (currentTurn == Constants.PRODUCER_TURN)
    {
        if (ProductsCount() != Constants.CONTAINER_SIZE)
        {
            producer.setState(Constants.WORKING);
            consumer.setState(Constants.SLEEPING);
        }
        else
        {
            producer.setState(Constants.TRYING);
            consumer.setState(Constants.SLEEPING);
        }
    }
```

Y por último el método “NextMoves” determina el numero de movimientos que ocurrirán en el contenedor en el próximo turno, además se generan números aleatorios para determinar si el próximo turno es del productor o del consumidor, asi como la cantidad de movimientos que ocurrirán.