Practica\_06: Juego del Bingo

**El juego del bingo consiste:**

1) Se tiene un bombo con 90 bolas numeradas de la 1 a la 90.

2) Se tienen una seríe de cartones con 15 números distintos cada uno, que se les reparte a los jugadores.

3) De forma aleatoria, se obtiene una bola del bombo (la cual se elimina del juego).

4) Los jugadores que forman la partida, y que juegan con un cartón cada uno, tienen que comprobar si el número que ha salido del bombo es uno de los que tiene su cartón, para asi tacharlo.

5) El primer jugador/jugadores que consigue tachar los 15 números de su cartón, es el ganador. No hay premio de línea.

**A) Estructura de la solución en Visual Studio.**

Nombre de la solución: Bingo

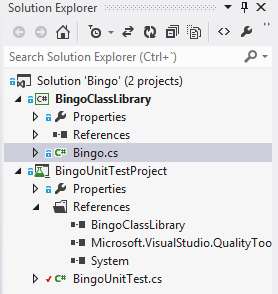
Proyecto de Libreria de clases: BingoClassLibrary (namespace BingoClassLibrary)

Archivo fuente: Bingo.cs

Proyecto de Test de Unidad: BingoUnitTestProject (namespace BingoUnitTestProject)

Archivo fuente: BingoUnitTest.cs

Referencia al proyecto BingoClassLibrary



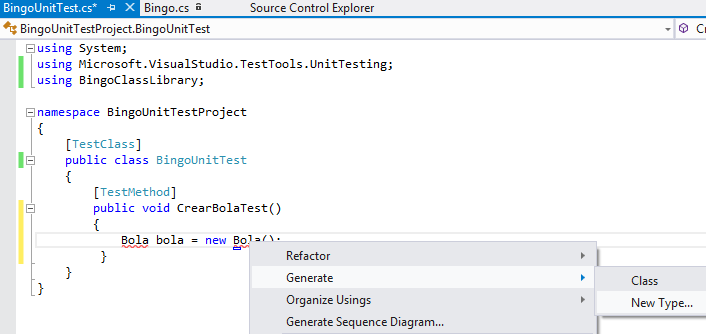
**B) Identificación de las clases y test de creación de objetos del juego**

Clase Bola.

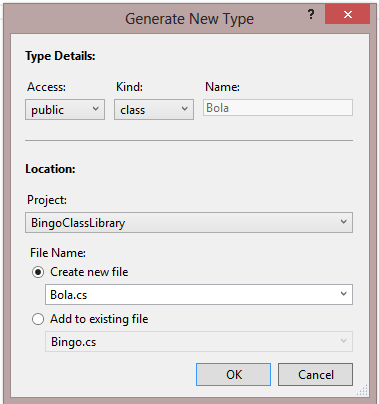
- Cada bola tiene un número comprendido entre el 1 y el 90.

- Lanzar una excepción ArgumentOutOfRangeException en caso contrario

*Escribimos el primer test: CrearBolaTest*



Botón derecho sobre la clase Bola (que no existe aún), seleccionamos, Generate y New Type .. aparece la siguiente pantalla:



Configurar la pantalla para que se cree un nuevo archivo Bola.cs, en el proyecto BingoClassLibrary (y no en el proyecto de test!!!).

Se observará que ha desaparecido el error debajo del nombre de la clase Bola, ya que se ha creado el archivo correspondiente, la declaración de la clase Bola.

Finalmente le añadimos la correspondiente instrucción Assert.IsNotNull, para comprobar que un objeto bola creado es no nulo.

[TestMethod]

public void CrearBolaTest()

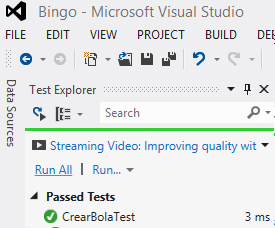
{

Bola bola = new Bola();

Assert.IsNotNull(bola);

}

Para comprobar si se pasa el test, compilamos la solución (F6) y en la ventana del Explorador de Test, pulsamos Run All. Despues de la ejecución, si aparece en verde, quiere decir que el test ha sido pasado satisfactoriamente, se puede seguir con el proceso ... Si no habrá que depurar el código hasta conseguir que se pase el test.



*Escribimos el segundo test: NumeroValidoBolaTest*

Cada bola tiene que tener un numero identificativo entre el 1 y el 90.

[TestMethod]

public void NumeroValidoBolaTest()

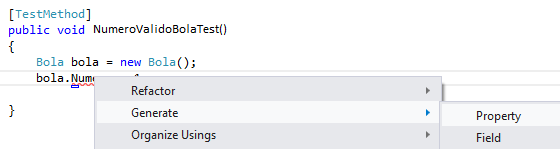
{

Bola bola = new Bola();

bola.Numero = 1;

}

Pulsamos botón derecho sobre la propiedad Numero de bola (que acabamos de escribir, marcada con un error ya que aún no existe) y seleccionamos Generate Property



Visual Studio habrá creado la propiedad Numero en la clase Bola, tal y como parece a continuación:

namespace BingoClassLibrary

{

public class Bola

{

public int Numero { get; set; }

}

}

El test se completaría con la instruccion Assert, que comprueba que el número asignado a una bola no cambia.

[TestMethod]

public void NumeroValidoBolaTest()

{

Bola bola = new Bola();

bola.Numero = 1;

Assert.AreEqual(bola.Numero, 1);

}

*Escribimos el tercer test: NumeroNoValidoBolaExcepcionTest*

¿que debería ocurrir, si una bola pretendiera tener un número no comprendido entre 1 y 90?

- Lanzar una excepción de argumento fuera de rango ...!!

[TestMethod]

[ExpectedException(typeof(ArgumentOutOfRangeException))]

public void NumeroNoValidoBolaExcepcionTest()

{

Bola bola = new Bola();

bola.Numero = 0;

}

Si ejecutáramos de nuevo los test, veríamos que este último no pasa. ¿por qué?, es evidente, no se ha lanzado la esperada excepción. ¿donde se tendría que haber lanzado?. En el momento de asignarle a una bola un número que no procede .. !!!

Hay que modificar el código para que cumpla con este nuevo requisito:

public class Bola

{

private int \_numero;

public int Numero

{

get { return \_numero; }

set

{

if (value < 1 || value > 90) throw new ArgumentOutOfRangeException();

else \_numero = value;

}

}

}

**C) Identificación de las clases y test de creación de objetos del juego**

Clase Bombo.

- El bombo, al principio del juego, contiene 90 bolas distintas de números comprendidos entre el 1 y el 90.

- Sacar de una bola aleatoria de entre las existentes dentro del bombo.

*Escribimos el primer test: CrearBomboTest*

[TestMethod]

public void CrearBomboTest()

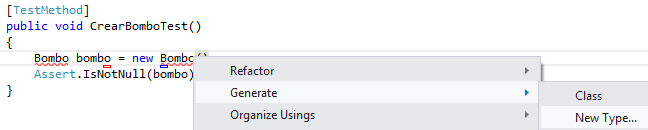
{

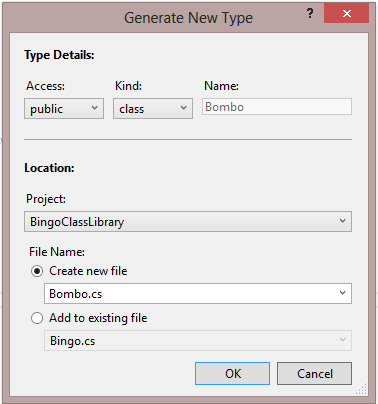
Bombo bombo = new Bombo();

Assert.IsNotNull(bombo);

}

pulsando el botón derecho sobre la clase Bombo (marcada como error), seleccionamos:





Visual Studio creará el archivo Bombo.cs en el proyecto BingoClassLibrary, con el siguiente contenido:

namespace BingoClassLibrary

{

public class Bombo

{

}

}

*Escribimos el segundo test:NumeroBolasBomboCreadoEsCeroTest*

Una vez creado el bombo, el numero de bolas que contiene es cero. Añadimos la propiedad NumeroBolas (tal cual hicimos anteriormente) a la clase Bombo. El test que representa ese comportamiento lo escribimos:

[TestMethod]

public void NumeroBolasBomboCreadoEsCeroTest()

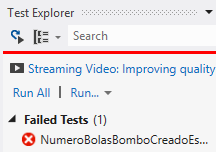
{

Bombo bombo = new Bombo();

Assert.AreEqual(bombo.NumeroBolas,0);

}

Si ejecutamos los test, veremos que este último no lo pasa



El problema está en que NumeroBolas no está inicializado a cero. Es necesario modificar el código para que se considere esa circunstancia. ¿cómo?. La formula que marca los cánones, es que la inclusión del código debe de ser mínimo, con el único objetivo de pasar el test. Según esto, tendríamos que declarar la propiedad NumeroBola de solo lectura, y que devolviera cero.

public class Bombo

{

public int NumeroBolas { get { return 0; } }

}

*Ahora si se pasan todos los test .. pero ¿por cuánto tiempo?.*

*Escribimos el siguiente test: MeterBolaBomboTest*

La condición inicial del juego es que en el bombo tiene que haber 90 bolas distintas, numeradas de la 1 a la 90. Para ello, lo primero que tendríamos que hacer es saber como meter una bola en el bombo, y comprobar que el número de bolas dentro del bombo se ha incrementado en uno.

[TestMethod]

public void MeterBolaBomboCreadoEsUnoTest()

{

Bombo bombo = new Bombo();

Bola bola = new Bola();

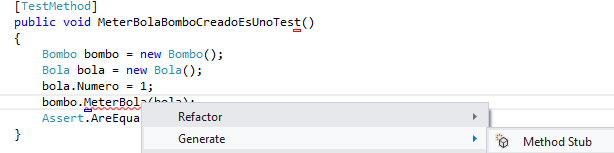
bola.Numero = 1;

bombo.MeterBola(bola);

Assert.AreEqual(bombo.NumeroBolas, 1);

}

Hay que crear el método MeterBola, que permite introducir una bola en el bombo. Pulsando el botón derecho sobre dicho método, seleccionamos generar Método, tal y como se muestra en la siguiente captura de pantalla.



Visual Studio modifica la clase Bombo según se muestra a continuación:

public class Bombo

{

public int NumeroBolas { get { return 0; } }

public void MeterBola(Bola bola)

{

throw new NotImplementedException();

}

}

Si ejecutaramos los test, veriamos que éste último no pasa ya que lo único que hace el método MeterBola es lanzar una excepción, no tiene ninguna interacción con la propiedad NumeroBolas de la clase Bombo.

Para pasar el test, podriamos hacer que el método MeterBola se limitara a establecer la propiadad NumeroBolas a 1. Pero en ese caso, cuando metieramos una segunda bola, el test fallaría ya que el NumeroBolas del bombo debería ser 2, y con la solución planteado siempre devolvería 1.

Para pasar el test, lo que podríamos hacer es incrementar la propiedad NumeroBolas en el método MeterBola, tal y como se muestra a continuación:

public class Bombo

{

public int NumeroBolas { get { return 0; } }

public void MeterBola(Bola bola)

{

NumeroBolas++;

}

}

Pero hay un problema la propiedad es de solo lectura, y el compilador no nos permite incrementarla. Hay que refactorizar..!!!

public class Bombo

{

private int \_numeroBolas;

public int NumeroBolas { get { return \_numeroBolas; } set { \_numeroBolas = value; } }

public void MeterBola(Bola bola)

{

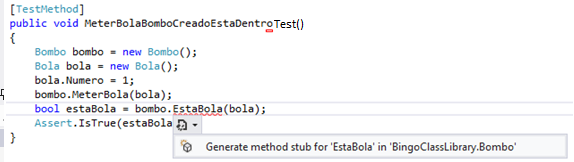
NumeroBolas++;

}

}

Con estas modificaciones, si se pasan los test ..!!

Pero es también necesario comprobar que la bola introducida está dentro del bombo. Para ello habría que escribir el test correspondiente:



Haciendo que el método generado EstaBola, devuelva true, pasariamos el test.

public bool EstaBola(Bola bola)

{

return true;

}

El problema es que siempre que preguntemos por si una bola está, obtendremos la misma respuesta verdadera. Lo cual nos lleva a tener que implementar un mecanismo para registrar dentro del bombo las bolas que se han metido.

las alternativas son multiples y variadas, desde utilizar arrays (ya sean de enteros o booleanos) a colecciones genéricas.

Para su implementación vamos a utilizar la clase HashSet, representa un conjunto de objetos, y permite operaciones para comprobar si un objeto pertenece al conjunto, y para operar entre conjuntos. Es justo lo que necesitamos ..!!!

public class Bombo

{

HashSet<Bola> \_bolas = new HashSet<Bola>();

private int \_numeroBolas;

public int NumeroBolas { get { return \_bolas.Count; } }

public void MeterBola(Bola bola)

{

\_bolas.Add(bola);

}

public bool EstaBola(Bola bola)

{

return \_bolas.Contains(bola);

}

}

Se pasan todos los test ...!!

Pero qué concepto de igualdad, que es el que se utiliza en la clase HashSet<bola> ???

Vamos a comprobarlo con el siguiente test:

[TestMethod]

public void MeterMismaBolaBomboCreadoEstaDentroTest()

{

Bombo bombo = new Bombo();

Bola bola = new Bola();

bola.Numero = 1;

bombo.MeterBola(bola);

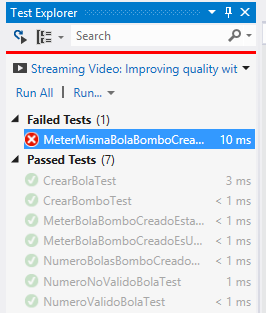
Bola mismaBola = new Bola();

mismaBola.Numero = 1;

bool estaBola = bombo.EstaBola(mismaBola);

Assert.IsTrue(estaBola);

}



La bola que tiene el mismo numero 1 no está dentro del bombo según el test.

Si ejecutaramos el siguiente test, comprobariamos que tambien falla.

[TestMethod]

public void DistintaBolaMismoNumeroTest()

{

Bola bola = new Bola();

bola.Numero = 1;

Bola otraBola = new Bola();

otraBola.Numero = 1;

Assert.AreEqual(bola,otraBola);

}

Eso quiere decir que tenemos que especificar el criterio de igualdad en la clase bola, para poder expresar que dos bolas son iguales si tienen el mismo número.

Hacemos que la clase Bola implemente el interface IEqualityComparer<Bola>, IEquatable<Bola>, sobre-escribimos los metodos de object (Equals y GetHashCode), y los operadores de igualdad y desigualdad. La clase Bola quedaría de la siguiente forma:

public class Bola:IEqualityComparer<Bola>,IEquatable<Bola>

{

private int \_numero;

public int Numero

{

get { return \_numero; }

set

{

if (value < 1 || value > 90) throw new ArgumentOutOfRangeException();

else \_numero = value;

}

}

public override bool Equals(object obj)

{

return Equals(obj as Bola);

}

public override int GetHashCode()

{

return this.Numero.GetHashCode();

}

public bool Equals(Bola x, Bola y)

{

if (ReferenceEquals(x,y)) return true;

if (ReferenceEquals(x, null) || ReferenceEquals(y, null)) return false;

else return (x.Numero == y.Numero);

}

public int GetHashCode(Bola obj)

{

return obj.Numero.GetHashCode();

}

public bool Equals(Bola other)

{

return Equals(this,other);

}

public static bool operator ==(Bola a, Bola b)

{

return a.Equals(b);

}

public static bool operator !=(Bola a, Bola b)

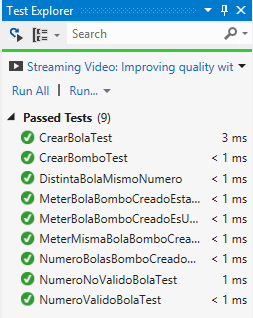
{

return !a.Equals(b);

}

}

Si ejecutamos todos los test, veremos que son pasados:



En el siguiente test, se mete la bola en el bombo y se modifica su valor desde fuera,seguidamente. La bola modificada no debe de estar dentro del bombo (a pesar de ser el mismo objeto, su valor ha cambiado).

[TestMethod]

public void MeterBolaBomboCreadoCambiarlaNoEstaDentroTest()

{

Bombo bombo = new Bombo();

Bola bola = new Bola();

bola.Numero = 1;

bombo.MeterBola(bola);

bola.Numero = 2;

bool estaBola = bombo.EstaBola(bola);

Assert.IsFalse(estaBola);

}

Test para sacar una bola existente del bombo.

public void MeterYSacarBolaBomboCreadoNoEstaDentroTest()

{

Bombo bombo = new Bombo();

Bola bola = new Bola();

bola.Numero = 1;

bombo.MeterBola(bola);

bombo.SacarBola(bola);

bool estaBola = bombo.EstaBola(bola);

Assert.IsFalse(estaBola);

}

Pero si la bola no está dentro del Bombo deberíamos obtener una excepción.

[TestMethod]

[ExpectedException(typeof(ArgumentOutOfRangeException))]

public void SacarBolaNoEstaBomboCreadoExcepcionTest()

{

Bombo bombo = new Bombo();

Bola bola = new Bola();

bola.Numero = 1;

bombo.SacarBola(bola);

}

Test para rellenar el bombo con todas las bolas, y comprobar que el numero de bolas en el bombo es 90.

[TestMethod]

public void MeterTodasBolasContadorBomboCreadoTest()

{

Bombo bombo = new Bombo();

for (int i = 0; i < 90; i++)

{

Bola bola = new Bola();

bola.Numero = i+1;

bombo.MeterBola(bola);

}

Assert.AreEqual(bombo.NumeroBolas,90);

}

Test para Meter y Sacar todas las bolas, comprobando que al final el número de bolas que queda dentro del bombo es cero.

[TestMethod]

public void MeterYSacarTodasBolasContadorBomboCreadoTest()

{

Bombo bombo = new Bombo();

for (int i = 0; i < 90; i++)

{

Bola bola = new Bola();

bola.Numero = i + 1;

bombo.MeterBola(bola);

}

for (int i = 0; i < 90; i++)

{

Bola bola = new Bola();

bola.Numero = i + 1;

bombo.SacarBola(bola);

}

Assert.AreEqual(bombo.NumeroBolas, 0);

}

¿Meter dos veces la misma bola?

[TestMethod]

public void MeterDosVecesMismaBolaContadorBomboCreadoTest()

{

Bombo bombo = new Bombo();

Bola bola = new Bola();

bola.Numero = 1;

bombo.MeterBola(bola);

bombo.MeterBola(bola);

Assert.AreEqual(bombo.NumeroBolas, 1);

}

¿Y sacar dos veces seguidas una bola introducida en el bombo?. Lanzará una excepción ..!!!

[TestMethod]

[ExpectedException(typeof(ArgumentOutOfRangeException))]

public void SacarDosVecesMismaBolaContadorBomboCreadoExcepcionTest()

{

Bombo bombo = new Bombo();

Bola bola = new Bola();

bola.Numero = 1;

bombo.MeterBola(bola);

bombo.SacarBola(bola);

bombo.SacarBola(bola);

}

Test para generar un numero aleatorio entre 1 y 90

[TestMethod]

public void GenerarNumeroAleatoriaEntre1y90Test()

{

Random ramdom = new Random(DateTime.Now.Millisecond);

int numeroAleatorio = ramdom.Next(1, 91);

Assert.IsTrue((numeroAleatorio >= 1 && numeroAleatorio <= 90));

}

Test para sacar una bola aleatoria de un bombo lleno.

[TestMethod]

public void SacarBolaAleatoriaDeBomboLlenoTest()

{

Random ramdom = new Random(DateTime.Now.Millisecond);

int numeroAleatorio = ramdom.Next(1, 91);

Bola bolaAleatoria = new Bola();

bolaAleatoria.Numero = numeroAleatorio;

Bombo bombo = new Bombo();

for (int i = 0; i < 90; i++)

{

Bola bola = new Bola();

bola.Numero = i + 1;

bombo.MeterBola(bola);

}

bombo.SacarBola(bolaAleatoria);

Assert.IsFalse(bombo.EstaBola(bolaAleatoria));

}

Test para generar una bola aleatoria (válida) de un bombo con bolas dentro (aleatorias). Se crea el método ElegirBola del bombo, que nos devuelve una bola existente dentro del bombo.

[TestMethod]

public void SacarBolaAleatoriaDeBomboAleatorioTest()

{

Bombo bombo = new Bombo();

Random ramdom = new Random(DateTime.Now.Millisecond);

int totalAleatorio = ramdom.Next(1, 91);

for (int i = 0; i < totalAleatorio; i++)

{

Bola bola = new Bola();

int numeroAleatorio = ramdom.Next(1, 91);

bola.Numero = numeroAleatorio;

bombo.MeterBola(bola);

}

int numeroAleatorioSacar = ramdom.Next(1, 91);

Bola bolaAleatoria = bombo.ElegirBola();

bombo.SacarBola(bolaAleatoria);

Assert.IsFalse(bombo.EstaBola(bolaAleatoria));

}

Hay que codificar el método ElegirBola(), con cuidado ya que se puede dar la circustancia de que el bombo esté vacio y se intente sacar una bola válida de él ..(algo que sería imposible y nos llevaria a un bucle infinito).

public Bola ElegirBola()

{

if (NumeroBolas <= 0) throw new ArgumentOutOfRangeException();

Random ramdom = new Random(DateTime.Now.Millisecond);

Bola bola = new Bola();

do

{

int numAleatorio = ramdom.Next(1, 91);

bola.Numero = numAleatorio;

}

while (!EstaBola(bola));

return bola;

}

El siguiente test, comprueba que ElegirBola cumple con las anteriores expectativas.

[TestMethod]

[ExpectedException(typeof(ArgumentOutOfRangeException))]

public void SacarBolaAleatoriaDeBomboVacioTest()

{

Bombo bombo = new Bombo();

Random ramdom = new Random(DateTime.Now.Millisecond);

int totalAleatorio = ramdom.Next(1, 91);

for (int i = 0; i < totalAleatorio; i++)

{

Bola bola = new Bola();

int numeroAleatorio = ramdom.Next(1, 91);

bola.Numero = numeroAleatorio;

bombo.MeterBola(bola);

bombo.SacarBola(bola);

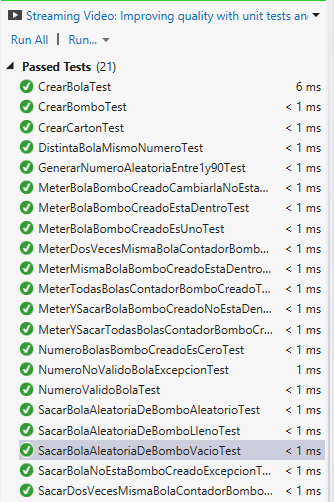
}

int numeroAleatorioSacar = ramdom.Next(1, 91);

Bola bolaAleatoria = bombo.ElegirBola();

bombo.SacarBola(bolaAleatoria);

}



**D) Identificación de las clases y test de creación de objetos del juego**

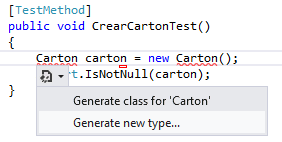
Clase Carton.

- El Carton, al principio del juego, contiene 15 números distintos de números comprendidos entre el 1 y el 90.

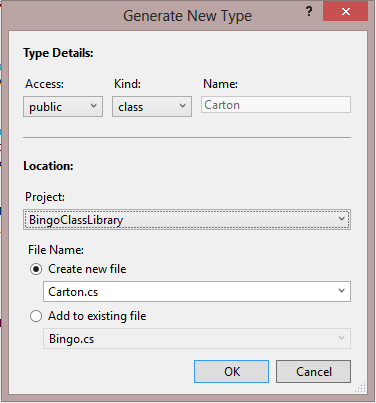
- Del cartón hay que ir borrando las números que corresponden a las bolas que salen del bombo.

- Cuando en un cartón se borran todos los números, ese cartón es el ganador y el juego termina (puede habar varios cartones ganadores).

Vamos a generar el primer test, para comprobar la correcta creación de un objeto Carton.



Sobre la clase Carton, generamos un nuevo tipo, al igual que hicimos anteriormente con las otras clases:



Tenemos que comprobar que un cartón recien creado tiene 15 numeros

[TestMethod]

public void CrearCartonTiene15NumerosTest()

{

Carton carton = new Carton();

Assert.AreEqual(carton.NumeroBolas,15);

}

Una vez creada la propiedad NumeroBolas en el cartón, comprobamos que el test falla. Necesitamos una estructura para almacenar los números. Utilizaremos, al igual que hicimos con el Bombo, un Hashset.

Si nos ponemos a pensar, no hay diferencia conceptual entre el bombo y el carton, salvo en el numero de elementos iniciales de cada uno al comienzo del juego.

La clase cartón, se muestra a continuación:

public class Carton

{

HashSet<Bola> \_bolas;

public int NumeroBolasSinTachar { get { return \_bolas.Count; } }

public Carton()

{

\_bolas = new HashSet<Bola>();

Rellenar();

}

private void Rellenar()

{

Random ramdom = new Random(DateTime.Now.Millisecond);

do

{

Bola bola = new Bola();

int numAleatorio = ramdom.Next(1, 91);

bola.Numero = numAleatorio;

\_bolas.Add(bola);

}

while (\_bolas.Count < 15);

}

public void TacharBola(Bola bola)

{

\_bolas.Remove(bola);

}

}

Vamos a comprobar que cuando un bombo está lleno de bolas completamente, y hay un cartón creado. Cuando se van sacando las bolas del bombo y a la vez tachando del cartón, al final cuando se sacan todas, el carton debe de tener todos los numeros tachados.

[TestMethod]

public void SacarTodasLasBolasDeBomboYTacharCartonTest()

{

Bombo bombo = new Bombo();

for (int i = 0; i < 90 ; i++)

{

Bola bola = new Bola();

bola.Numero = i + 1;

bombo.MeterBola(bola);

}

Carton carton = new Carton();

for (int i = 0; i < 90; i++)

{

Bola bola = new Bola();

bola.Numero = i + 1;

bombo.SacarBola(bola);

carton.TacharBola(bola);

}

Assert.AreEqual(carton.NumeroBolasSinTachar, 0);

}

El problema de la solución al modelado de la clase Cartón que hemos propuesto, es que no sabemos cuales eran los numeros originales del cartón, ya que al ir tachándolos los eliminabamos del Hashset.

Una solución sería hacer una copia de la estructura de almacenamiento (implica duplicidad de memoria), la alternativa es recalcular el NumeroBolasSinTachar para cada cartón, cada vez que se saque una bola ... (empeoraría el rendimiento).

Si optáramos por esta segunda solución, menejando conjuntos, seria realizar una operación de intersección entre Bombo y Cartón,y comprobar que es 0. En ese momento todas las bolas del bombo (contenidas en el cartón) ya han salido.

public class Carton2

{

HashSet<Bola> \_bolas;

public HashSet<Bola> Bolas

{

get { return \_bolas; }

set { \_bolas = value; }

}

public Carton2()

{

\_bolas = new HashSet<Bola>();

Rellenar();

}

private void Rellenar()

{

Random ramdom = new Random(DateTime.Now.Millisecond);

do

{

Bola bola = new Bola();

int numAleatorio = ramdom.Next(1, 91);

bola.Numero = numAleatorio;

\_bolas.Add(bola);

}

while (\_bolas.Count < 15);

}

public int NumeroBolasSinTachar(Bombo bombo)

{

return bombo.Bolas.Intersect(this.\_bolas).Count();

}

}

El test correspondiente:

[TestMethod]

public void SacarTodasLasBolasDeBomboYTacharCarton2Test()

{

Bombo bombo = new Bombo();

for (int i = 0; i < 90; i++)

{

Bola bola = new Bola();

bola.Numero = i + 1;

bombo.MeterBola(bola);

}

Carton2 carton = new Carton2();

for (int i = 0; i < 90; i++)

{

Bola bola = new Bola();

bola.Numero = i + 1;

bombo.SacarBola(bola);

}

Assert.AreEqual(carton.NumeroBolasSinTachar(bombo), 0);

}

Test para probar la dinámica del juego en ambos casos:

[TestMethod]

public void DinamicaJuego()

{

Bombo bombo = new Bombo();

bombo.Rellenar();

List<Carton> cartones = new List<Carton>();

List<Carton> cartonesGanadores = new List<Carton>();

for (int i = 0; i < 1000; i++)

{

cartones.Add(new Carton());

}

bool finJuego = false;

while (!finJuego)

{

var bola = bombo.ElegirBola();

bombo.SacarBola(bola);

foreach (var carton in cartones)

{

carton.TacharBola(bola);

if (carton.NumeroBolasSinTachar == 0) cartonesGanadores.Add(carton);

finJuego = (carton.NumeroBolasSinTachar == 0);

}

}

}

[TestMethod]

public void DinamicaJuego2()

{

Bombo bombo = new Bombo();

bombo.Rellenar();

List<Carton2> cartones2 = new List<Carton2>();

List<Carton2> cartones2Ganadores = new List<Carton2>();

for (int i = 0; i < 1000; i++)

{

cartones2.Add(new Carton2());

}

bool finJuego = false;

while (!finJuego)

{

var bola = bombo.ElegirBola();

bombo.SacarBola(bola);

foreach (var carton2 in cartones2)

{

if (carton2.NumeroBolasSinTachar(bombo) == 0) cartones2Ganadores.Add(carton2);

finJuego = (carton2.NumeroBolasSinTachar(bombo) == 0);

}

}

}

GRUPO DE TRABAJO.

Se le propone al alumno que basándose en el proyecto desarrollado **lo vuelva a construir** para incluir las siguientes modificaciones (junto con los correspondientes test de respaldo para verificar su correcta implementación):

* Utilizar una implementación de cartón alternativa basada en Lista<Bola>.
* El cartón tiene entre 9 y 21 números y divisible entre 3.
* Hay que Incluir el premio de LINEA.

*Los números del cartón se dividen en 3 subconjuntos iguales de números cada uno, que representan las líneas. El primer cartón (cartones) que consigue tachar los números de una línea gana el premio de línea y la partida sigue hasta conseguir el premio de bingo.*

TRABAJO AUTÓNOMO.

* Crear una especificación de conjuntos de números enteros con sus operaciones básicas basada en Arrays boleanos.
* Utilizar dicha especificación para implementar un cartón en el juego del Bingo.