PRÁCTICA 2B

Aitor Melero Picón, Arturo Morcillo Penares

UAM Ampliación de Programación

PRÁCTICA 2b:

En esta parte de la memoria explicaremos el porqué de la elección de clases realizada para permitir entender las decisiones de diseño.

- Coordenada.

En esta clase definimos una coordenada. Una coordenada toma dos valores enteros, uno para la posición x y otro para la posición y.

No hay ningún método en el que podamos destacar nada de su implementación. Simplemente es una clase que utilizaremos para abstraer mejor la clase *Provincia*.

- Provincia.

Una *provincia* corresponde a un *rectángulo*.

Un *rectángulo* está definido por dos *coordenadas*: La Superior izquierda y la inferior derecha.

En cuanto a los métodos, podemos destacar:

- bool sonVecinos (Provincia p): Comprueba si las provincias tienen frontera de la misma forma que lo hacemos en haskell. Mostramos una parte del código.

```
/*Caso primero*/
if (c11.X == c22.X + 1 || c12.X == c21.X -1 || c11.X == c22.X -1 || c12.X == c21.X + 1)
{
    /*Intersecciones*/
    if (c12.Y >= c21.Y && c12.Y <= c22.Y) // Si la de abajo es menor que la de arriba y mayor que la de abajo
    {
        return true;
    }
    if (c11.Y <= c22.Y && c11.Y >= c21.Y) // Si la de arriba es mayor que la de abajo y menor que la de arriba
    {
        return true;
    }

    if (c22.Y >= c11.Y && c22.Y <= c12.Y) // Si la de abajo es menor que la de arriba y mayor que la de abajo
    {
        return true;
    }
    if (c21.Y <= c12.Y && c21.Y >= c11.Y) // Si la de arriba es mayor que la de abajo y menor que la de arriba
    {
        return true;
    }
}
```

En este caso, primero comprobamos si las coordenadas X están adyacentes. Si lo están, comprobamos si coinciden algunas coordenadas Y.

- bool overlap (Provincia p): Comprueba si las dos provincias tienen alguna coordenada en común. Si la tienen, es que están superpuestas.

```
public bool overlap(Provincia provincia)
{
    /*A lo bruto, si coincide una coordenada es que se superponen*/
    for (int i1 = this.arribaIzquierda.X; i1 < this.abajoDerecha.X ; i1++)
    {
        for (int i2 = provincia.arribaIzquierda.X; i2 < provincia.abajoDerecha.X ; i2++)
        {
            for (int j1 = this.arribaIzquierda.Y; j1 < this.abajoDerecha.Y ; j1++)
            {
                for (int j2 = provincia.arribaIzquierda.Y; j2 < provincia.abajoDerecha.Y ; j2++)
                {
                    if (i1 == i2 && j1 == j2)
                    {
                        return true;
                    }
                }
            }
        }
    }
    return false;
}
```

T

Resulta obvio que si coincide al menos una casilla es que superponen. De modo que comprobamos todas las casillas para verificar si se da el caso.

- Mapa.

Un mapa corresponde a un conjunto de provincias. Cuando lo creamos comprobamos que ninguna provincia se solapa. De suceder esto, lanzaríamos una *overlapException.*

En cuanto a los métodos, podemos destacar:

- addProvincia (Provincia p): Añade una provincia a nuestro mapa. Antes comprueba que no se superpone con ninguna y, de hacerlo, lanzaría una *overlapException*.

```
public void addProvincia(Provincia provincia)
{
    /*Comprobacion superposicion*/
    foreach (Provincia pAux in this.provincias)
    {
        if (pAux.overlap(provincia))
        {
            throw new OverlapException("[ERROR] Superposicion");
        }
    }

    if (!this.provincias.Contains(provincia))
    {
        this.provincias.Add(provincia);
    }
}
```

- List<Provincia> getFronteras(Provincia provincia): Permite obtener todas las *fronteras* en el *mapa* de una determinada *provincia*.

- Dictionary<Provincia,Color> coloreado(): Devuelve este diccionario que resuelve el problema del coloreado para todos los colores disponibles. La resolución del problema consta de dos partes que se aplican para cada *provincia*:

```
/*Recorremos todas las provincias en el mapa*/
foreach (Provincia provincia in this.provincias)
{
    /*Lista para los colores que no puede contener*/
    List<Color> colores = new List<Color>();

    /*Recorremos todos los vecinos de la provincia*/
    vecinos = this.getFronteras(provincia);
    foreach (Provincia vecino in vecinos)
    {
        /*Compruebo si ya la he anyadido*/
        if (diccionario.ContainsKey(vecino))
        {
            /*Si la he anyadido antes lo anyado a la lista de colores*/
            Color color = diccionario[vecino];
            if (!colores.Contains(color))
            {
                colores.Add(color);
            }
        }
    }
    /*El color de la provincia es el que no tengan sus vecinos*/
```

En primer lugar, obtenemos una lista con los colores de todos los vecinos. Esos colores son los que no puede tener nuestra *provincia*.

```
        }
        /*El color de la provincia es el que no tengan sus vecinos*/
        /*Iteramos la enumeracion Color y buscamos el primero que no está en colores*/
        var coloresPosibles = Enum.GetValues(typeof(Color));
        foreach (Color c in coloresPosibles)
        {
            if (!colores.Contains(c))
            {
                diccionario.Add(provincia, c);
                break;
            }
        }
    }

    return diccionario;
}
```

Tras saber que color no puede tener nuestra provincia, simplemente le asignamos un color que no se encuentre entre ellos de entre todos los colores posibles.

- Dictionary<Provincia,Color> coloreadoResticted(List<Color> colores): Devuelve este diccionario que resuelve el problema del coloreado para los colores que le pasamos.

```
    }
    /*El color de la provincia es el que no tengan sus vecinos*/
    /*Iteramos los colores posibles y buscamos el primero que no está en colores*/
    foreach (Color c in coloresPosibles)
    {
        if (!colores.Contains(c))
        {
            diccionario.Add(provincia, c);
            break;
        }
    }
```

Su funcionamiento es similar al método anterior, con la única diferencia de que al comprobar que color asignar lo hace con la lista que hemos pasado como argumento.

- Mosaico.

Un mosaico corresponde a nuestro “lienzo”. Está formado por dos enteros, las filas y las columnas y una lista de strings. El Constructor más empleado es el siguiente:

```
public Mosaico(List<string> oldMosaico, int filas, int columnas, Provincia provincia, Color color)
{
    /*Comprobacion por si la provincia se sale del mosaico*/
    if (provincia.arribaIzquierda.X < 1 || provincia.arribaIzquierda.Y < 1
        || provincia.arribaIzquierda.X > columnas || provincia.arribaIzquierda.Y > filas)
    {
        throw new OutOfLimitsException("[ERROR] Nos salimos de los limites");
    }

    if (provincia.abajoDerecha.X < 1 || provincia.abajoDerecha.Y < 1
        || provincia.abajoDerecha.X > columnas || provincia.abajoDerecha.Y > filas)
    {
        throw new OutOfLimitsException("[ERROR] Nos salimos de los limites");
    }

    List<string> newMosaico = new List<string>();
    /*Clonacion del tablero*/
    foreach (string aux in oldMosaico)
    {
        newMosaico.Add((string)aux.Clone()); // Así es una copia y no la misma referencia
    }
```

Primero realizamos las comprobaciones pertinentes y copiamos el anterior mosaico.

```
    /*Cambiamos la provincia*/
    for (int i = provincia.arribaIzquierda.Y; i <= provincia.abajoDerecha.Y; i++)
    {
        var newMosaicoAux = new StringBuilder(newMosaico[i - 1]);
        for (int j = provincia.arribaIzquierda.X; j <= provincia.abajoDerecha.X; j++)
        {
            newMosaicoAux[j - 1] = getCaracter(color);
        }
        newMosaico[i - 1] = newMosaicoAux.ToString();
    }

    this.mosaico = newMosaico;
    this.filas = filas;
    this.columnas = columnas;
}
```

Tras eso, cambiamos las casillas de la provincia introducida al color introducido. Nótese que en *Mosaico* no hacemos comprobación de si son *vecinos* o no. Esto es debido a que *Mosaico* solamente se encarga de representar lo que le pidamos siempre que no produzca solapamientos.

En cuanto a los métodos, podemos destacar:

- char getCaracter (Color color): Obtiene la letra inicial de un color.

- Múltiples creadores que cambian casillas o añaden provincias de un determinado color. Estos pueden lanzar la excepción *OutOfLimitsException.*

- Region.

Una región corresponde a un conjunto de provincias y, por tanto, está formada por una lista de provincias.

```
public Region(List<Provincia> provincias)
{
    /*bucle 1*/
    foreach (Provincia p in provincias)
    {
        bool vecino = false;
        /*Bucle 2*/
        foreach (Provincia pAux in provincias)
        {
            if (!(p == pAux))
            {
                if (p.overlap(pAux))
                {
                    throw new OverlapException("[ERROR] Superposicion");
                }
                if (p.sonVecinos(pAux))
                {
                    vecino = true;
                }
            }
        }

        if (!vecino && provincias.Count > 1)
        {
            throw new RegionException("[ERROR] Una provincia no tiene vecino en la region");
        }
    }

    this.provincias = provincias;
}
```


En su creacion puede lanzar una *OverlapException* si dos provincias se superponen; o una *RegionException* si una provincia no hace frontera con ninguna otra provincia de la región.

En cuanto a los métodos, podemos destacar:

- addProvincia(Provincia provincia): que añade una provincia a la región. También puede lanzar una *OverlapException* o una *RegionException*.

```
public void addProvincia(Provincia provincia)
{
    bool vecino = false;
    /*Comprobacion superposicion*/
    foreach (Provincia pAux in this.provincias)
    {
        if (pAux.overlap(provincia))
        {
            throw new OverlapException("[ERROR] Superposicion");
        }
        if (!vecino)
        {
            throw new RegionException("[ERROR] Una provincia no tiene vecino en la region");
        }
    }

    if (!this.provincias.Contains(provincia))
    {
        this.provincias.Add(provincia);
    }
}
```

- MapaRegion.

Un MapaRegion es igual a un mapa, pero implementa los métodos para regiones y no para provincias.

En cuanto a los métodos, podemos destacar:

- List<Region> getFronteras(Region region).

```
public List<Region> getFronteras(Region region)
{
    List<Region> listaRegiones = new List<Region>();
    /*Iteramos todas las regiones del mapa*/
    foreach (Region regionAux in this.regiones)
    {
        if (regionAux != region)
        {
            /*Iteramos todas las provincias en cada region del mapa*/
            foreach (Provincia provinciaAux1 in regionAux.provincias)
            {
                /*Iteramos todas las provincias de la region a comprobar*/
                foreach (Provincia provinciaAux2 in region.provincias)
                {
                    /*Si son vecinos*/
                    if (provinciaAux1.sonVecinos(provinciaAux2))
                    {
                        /*Si no lo hemos anyadido antes*/
                        if (!listaRegiones.Contains(regionAux))
                        {
                            listaRegiones.Add(regionAux);
                        }
                    }
                }
            }
        }
    }

    return listaRegiones;
```

El funcionamiento se basa en que tenemos que obtener todas las *regiones* que tienen una *provincia* que sea *frontera* con una *provincia* de la *región* revisada.

- Dictionary<Region, Color> coloreado(): Posee un funcionamiento similar a su primo en Mapa.

```
/*Lista para los colores que no puede contener*/
List<Color> colores = new List<Color>();

/*Recorremos todos los vecinos de la provincia*/
vecinos = this.getFronteras(region);
foreach (Region vecino in vecinos)
{
    /*Compruebo si ya la he anyadido*/
    if (diccionario.ContainsKey(vecino))
    {
        /*Si la he anyadido antes lo anyado a la lista de colores*/
        Color color = diccionario[vecino];
        if (!colores.Contains(color))
        {
            colores.Add(color);
        }
    }
}
/*El color de la provincia es el que no tengan sus vecinos*/
```

Primero obtiene los colores de todas las regiones vecinas.

```
    }
}
/*El color de la provincia es el que no tengan sus vecinos*/
/*Iteramos la enumeracion Color y buscamos el primero que no está en colores*/
var coloresPosibles = Enum.GetValues(typeof(Color));
foreach (Color c in coloresPosibles)
{
    if (!colores.Contains(c))
    {
        diccionario.Add(region, c);
        break;
    }
}
}
```

Luego le asigna un color que no estuviera entre los colores de las regiones vecinas.

- Dictionary<Region, Color> coloreadoRestricted()

Apartados a y b:

Estos dos apartados fueron realizados al mismo tiempo, ya que los rectángulos los tratamos como provincias. Una vez podemos obtener las fronteras de las provincias solo tenemos que solucionar el problema de coloreado.

Para esto, aplicamos una serie de bucles muy sencillos. De forma resumida comprobamos si les hemos asignado un color a los vecinos. La provincia que estamos comprobando tendrá un color que no hayamos visto anteriormente (ver funciones *coloreado* y *coloreadoRestricted*).

A la hora de implementar el mosaico, la función realizada consiste en crear un nuevo mosaico tomando de base el primer mosaico y con las mismas filas y columnas. Tras esto, cambia el color de las coordenadas correspondientes a la provincia introducida.

```
{
    /*Creamos las provincias*/
    Provincia hu = new Provincia(new Coordenada(4, 6), new Coordenada(5, 8));
    Provincia ca = new Provincia(new Coordenada(6, 8), new Coordenada(8, 9));
    Provincia se = new Provincia(new Coordenada(6, 5), new Coordenada(10, 7));
    Provincia ma = new Provincia(new Coordenada(9, 8), new Coordenada(16, 9));
    Provincia co = new Provincia(new Coordenada(11, 6), new Coordenada(16, 7));
    Provincia ja = new Provincia(new Coordenada(13, 3), new Coordenada(16, 5));
    Provincia gr = new Provincia(new Coordenada(17, 4), new Coordenada(19, 8));
    Provincia al = new Provincia(new Coordenada(20, 6), new Coordenada(22, 8));

    /*Creamos el mapa y le metemos todas las provincias*/
    Mapa mapa = new Mapa(new List<Provincia>());
    mapa.addProvincia(hu);
    mapa.addProvincia(ca);
    mapa.addProvincia(se);
    mapa.addProvincia(ma);
    mapa.addProvincia(co);
    mapa.addProvincia(ja);
    mapa.addProvincia(gr);
    mapa.addProvincia(al);

    /*Creamos el mosaico*/
    Mosaico mosaico = new Mosaico(11, 24);
    Console.WriteLine(mosaico);
    Console.WriteLine(" ");
    Console.WriteLine(" ");
```

Primero inicializamos todos los datos que vamos a necesitar.

```
    List<Color> colorsitos = new List<Color>();
    colorsitos.Add(Color.Verde);
    colorsitos.Add(Color.Azul);
    colorsitos.Add(Color.Rojo);

    Dictionary<Provincia, Color> diccionario = mapa.coloreadoRestricted(colorsitos);

    foreach (Provincia provincia in mapa.provincias)
    {
        mosaico = new Mosaico(mosaico.mosaico, mosaico.filas, mosaico.columnas, provincia, diccionario[provincia]);
        Console.WriteLine(mosaico);
        Console.WriteLine(" ");
        Console.WriteLine(" ");
    }

    Console.ReadLine();

}
```

Para concluir resolviendo el problema.

Apartado c:

Para poder realizar este apartado, a la funcionalidad anterior le añadimos manejo de excepciones en las funciones que hacía falta. Estas han sido mencionadas previamente.

En el *Main* de ejemplo simplemente imprimimos por pantalla la excepción encontrada.

```
Provincia hu = new Provincia(new Coordenada(4, 6), new Coordenada(5, 8));
Provincia ca = new Provincia(new Coordenada(6, 8), new Coordenada(8, 9));
Provincia se = new Provincia(new Coordenada(6, 5), new Coordenada(10, 7));
Provincia ma = new Provincia(new Coordenada(9, 8), new Coordenada(16, 9));
Provincia co = new Provincia(new Coordenada(11, 6), new Coordenada(16, 7));
Provincia ja = new Provincia(new Coordenada(13, 3), new Coordenada(16, 5));
Provincia gr = new Provincia(new Coordenada(17, 4), new Coordenada(19, 8));
Provincia al = new Provincia(new Coordenada(20, 6), new Coordenada(22, 8));

Console.WriteLine("Comprobamos la excepcion de provincia incorrecta:");
try
{
    Provincia provinciaIncorrecta = new Provincia(new Coordenada(22, 8), new Coordenada(20, 6));
} catch (IncorrectProvincia e)
{
    Console.WriteLine("Excepcion de provincia incorrecta");
    Console.WriteLine(" ");
    Console.WriteLine(" ");
}
```

Primero creamos las *provincias* y una que sabemos que *es incorrecta*.

```
Provincia provinciaSuperposicion1 = new Provincia(new Coordenada(2, 8), new Coordenada(16, 9));
Provincia provinciaSuperposicion2 = new Provincia(new Coordenada(3, 8), new Coordenada(7, 9));
Console.WriteLine(provinciaSuperposicion1);
Console.WriteLine(provinciaSuperposicion2);
Console.WriteLine("Anyadimos la primera");
try
{
    mapa.addProvincia(provinciaSuperposicion1);
}
catch (OverlapException e)
{
    Console.WriteLine("Excepcion de superposicion para la primera provincia");
}
Console.WriteLine("Anyadimos la segunda");
try
{
    mapa.addProvincia(provinciaSuperposicion2);
}
catch (OverlapException e)
{
    Console.WriteLine("Excepcion de superposicion para la segunda provincia");
}
Console.WriteLine(" ");
```

Tras crear el *Mapa*, comprobamos si coge bien las *provincias* que fallan.

```
Console.WriteLine("Comprobamos la excepcion de las provincias que se estan fuera del mosaico");
Console.WriteLine("Creamos un mosaico de tamanyo insuficiente");
Mosaico mosaico = new Mosaico(2, 3);

try
{
    Mosaico mosaicoAux = new Mosaico(mosaico.mosaico, mosaico.filas, mosaico.columnas, hu, Color.Rojo);
} catch (OutOfLimitsException e)
{
    Console.WriteLine("Error fuera de los limites");
}


Console.WriteLine(" ");


Console.ReadLine();
```

Para finalizar con las *provincias* de fuera del *mosaico*.

Apartado d:

En este apartado no fue necesario cambiar nada. Simplemente añadimos la funcionalidad pidiendo entrada de datos y mostrando la salida adecuada.

Mostramos como vamos añadiendo *rectángulos y una excepción*. Una diferencia notable respecto al código implementado en la p2a es que cuando ocurre una excepción los *rectángulos* anteriores se mantienen. Para esto, tenemos dos *bucles anidados*:

```
/*****************************************************************************/
/*****************************Pedir rectangulos*******************************/
/*****************************************************************************/
/*Obtenemos las coordenadas del rectangulo*/
Console.WriteLine("Introduzca la coordenada x1 (x superior izquierda):");
String x1Aux = Console.ReadLine();
int x1 = Convert.ToInt32(x1Aux);

Console.WriteLine("Introduzca la coordenada y1 (y superior izquierda):");
String y1Aux = Console.ReadLine();
int y1 = Convert.ToInt32(y1Aux);

Console.WriteLine("Introduzca la coordenada x2 (x inferior derecha):");
String x2Aux = Console.ReadLine();
int x2 = Convert.ToInt32(x2Aux);

Console.WriteLine("Introduzca la coordenada y1 (y inferior derecha):");
String y2Aux = Console.ReadLine();
int y2 = Convert.ToInt32(y2Aux);

/*Lo metemos en el mapa*/
```

El bucle en el que pedimos los *rectángulos*.

```
/*************************************************************************/
/*****************************Pedir colores*******************************/
/*************************************************************************/

/*Creamos la lista de colores que vamos a pasa*/
List<Color> listaColores = new List<Color>();

/*Peticion de cosas de colores*/
string comprobacionColor = "S";
while (comprobacionColor.Equals("S"))
{

    Console.WriteLine("Introduzca un color entre el Rojo, Verde, Azul, Naranja y Morado:");
    string color = Console.ReadLine();

    /*Comprobamos que color ha elegido*/
    if (color.Equals("Verde") || color.Equals("VERDE") || color.Equals("verde"))
    {
        if (!listaColores.Contains(Color.Verde))
        {
            listaColores.Add(Color.Verde);
        }
    }
    if (color.Equals("Azul") || color.Equals("AZUL") || color.Equals("azul"))
    {
        if (!listaColores.Contains(Color.Azul))
```

El bucle en el que pedimos los *colores*.

Apartado e:

Fueron creadas las clases *Region y MapaRegion*.

Los cambios más relevantes fueron:

→ getVecinos: Obtiene todos los vecinos de todas las provincias de la *región*, pero descarta las *provincias* que pertenecen a la *región*.

→ coloreado: Funciona de manera similar, pero comprobamos todas las *Provincias* de las distintas *regiones* como si fueran una sola.

→ coloreadoRestricted: Igual que la anterior.

En el main:

```
/*Todo en tres regiones*/
List<Provincia> lista1 = new List<Provincia>();
lista1.Add(al);
lista1.Add(gr);
lista1.Add(ma);
Region region1 = new Region(lista1);
// Console.WriteLine(region1);

List<Provincia> lista2 = new List<Provincia>();
lista2.Add(ja);
lista2.Add(co);
lista2.Add(se);
Region region2 = new Region(lista2);
// Console.WriteLine(region2);

List<Provincia> lista3 = new List<Provincia>();
lista3.Add(hu);
lista3.Add(ca);
Region region3 = new Region(lista3);
// Console.WriteLine(region3);

List<Provincia> lista4 = new List<Provincia>();
lista4.Add(pAux);
Region region4 = new Region(lista4);
// Console.WriteLine(region4);

/*Lista de regiones*/
List<Region> regiones = new List<Region>();
regiones.Add(region1);
regiones.Add(region2);
regiones.Add(region3);
regiones.Add(region4);
```

Creamos varias *regiones* (4 en total).

```
    MapaRegion mapa = new MapaRegion(regiones);

    Dictionary<Region,Color> diccionarioColores = mapa.coloreado();

    /*Creamos el mosaico. De momento de 20x20*/
    Mosaico mosaico = new Mosaico(20, 30);

    foreach(Region r in regiones)
    {
        foreach(Provincia p in r.provincias)
        {
            mosaico = new Mosaico(mosaico.mosaico, mosaico.filas, mosaico.columnas, p, diccionarioColores[r]);
        }
    }

    Console.WriteLine(mosaico);
    Console.ReadLine();

}
```

Realizamos el *problema del coloreado por regiones* y dibujamos las provincias de cada *región* con el color de su *región*.

Diagrama de clases:

En nuestro diagrama de clases no se ve una correlación entre las clases ya que son independientes. A la hora de interactuar unas con otras es debido a que utilizan parámetros que son miembros de otras clases