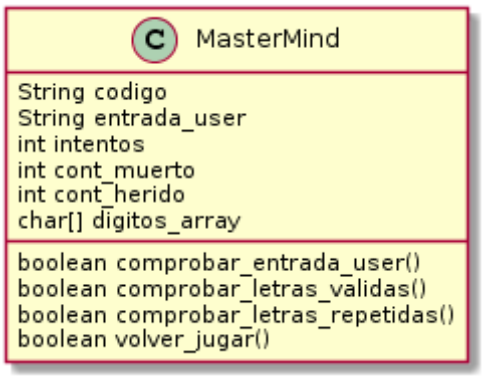
Práctica MASTERMIND

El objetivo principal de esta práctica consiste en el análisis, diseño e implementación del juego MasterMind. Esto se va a ir haciendo mediante distintos pasos o iteraciones con los cuales podremos ir viendo el progreso y las soluciones a nuestros propios errores, a medida que vamos haciendo más completo y más efectivo nuestro código.

Iteración 1.

Después de haber realizado una primera versión del juego MasterMind, y tras haber visto en clase los conceptos de un código de calidad, puedo identificar diferentes errores en distintos aspectos del código. El diagrama UML de la primera versión del juego es el que se muestra a continuación.



El principal error, del cual surgen los demás, es que solo planteé una única clase, en la cual incluí todo lo necesario para que el ejercicio cumpliera su objetivo. Este hecho obviamente hace que el código tenga poca calidad ya que, al estar toda la funcionalidad en una misma clase, resulta ser poco fluido. Esto es consecuencia directa de no haber realizado previamente el análisis y diseño previo correctamente.

El código resulta frágil, ya que al estar todo en la misma clase, resulta complicado hacer pruebas de una sola parte. Por esta misma razón, el código es rígido.

Además de la calidad del código, faltaba una parte que no conseguí terminar en la primera parte del trabajo, y es la de generar aleatoriamente la combinación secreta que debe ser adivinada por el jugador.

Iteración 2.

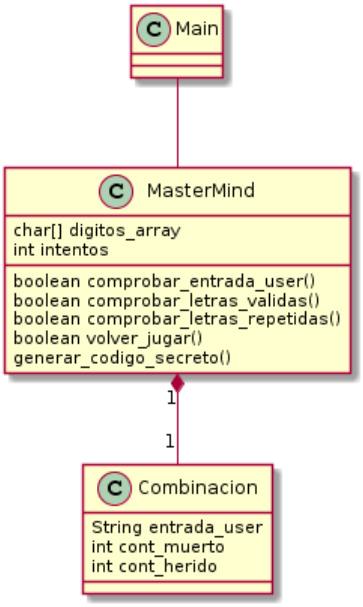
Tras haber identificado los anteriores errores, es hora de intentar corregirlos y mejorar así la calidad de nuestro código. Para ello, debemos realizar un análisis y diseño previo identificando las posibles clases con sus respectivos atributos y métodos, acorde a lo que sabemos de la funcionalidad del juego.

De esta forma, podemos identificar las clases:

- “menú/main”: tiene la función de controlar el inicio de la partida y llamar a la clase MasterMind.

- “MasterMind”: incluye toda la lógica del juego y el control del mismo

- “Combinación”: sirve para almacenar la combinación propuesta por el usuario, así como la cantidad de “muertos” y “heridos” en cada una de las propuestas.



Iteración 3.

En esta iteración, se pretende solventar los errores que se tuvieron en la anterior, centrándonos en la calidad del código y su mantenibilidad.

Por ello, analizamos y diseñamos un diagrama en el que se expresara de una manera más completa y real lo que conocíamos hasta el momento del funcionamiento del juego. Así, pasamos a realizar los siguientes cambios:

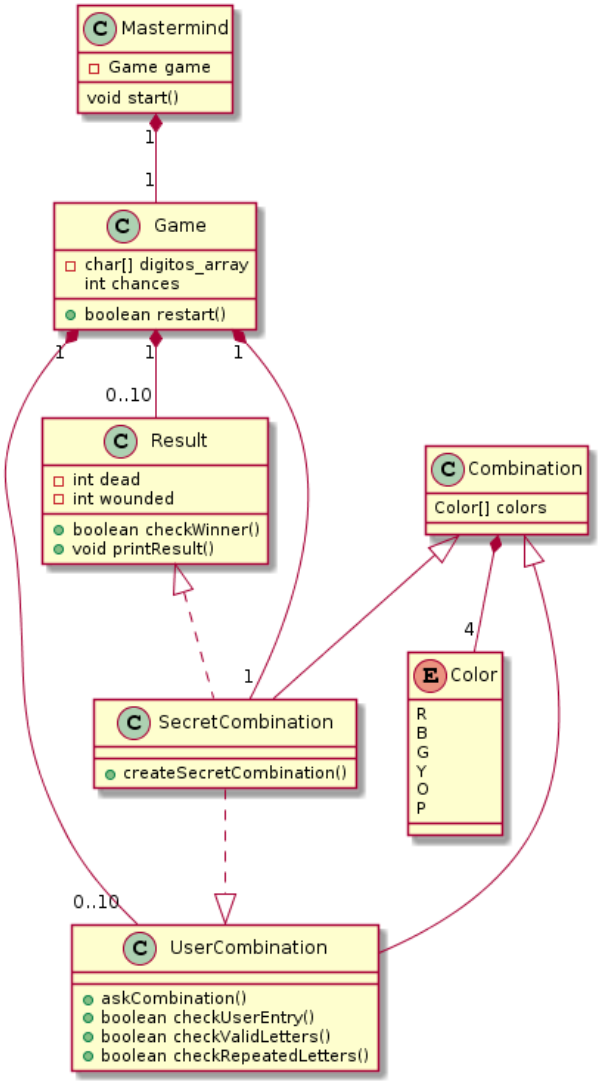
- La clase “Main” no aportaba ninguna funcionalidad añadida por lo que se decidió no utilizarla, pasando a ser la clase “Game” la clase principal o más importante, que contiene las funcionalidades para llevar a cabo el desarrollo del programa.
- Se crea una clase inicial “MasterMind” que actúa como iniciador del programa, cuya única función es llamar a la clase “Game” y a su método jugar.
- Como vemos en el segundo diagrama UML, solo tenemos en cuenta una clase “Combination”. Esto no es correcto, ya que, al existir dos tipos de combinaciones, la secreta y la que propone el usuario, se pueden obtener dos clases diferentes que comparten algunos atributos y métodos y tienen los suyos propios, por lo que encontramos una herencia con la clase padre “Combination”, y las hijas “SecretCombination “y “UserCombination”.
- En el código de la iteración anterior, la combinación de colores que propone el usuario la almacenaba en una variable String, lo cual me dificultaba el procesamiento de los caracteres de colores que introdujo el usuario, haciendo el código viscoso. Por esto, se plantea la implementación de una clase “Color” (enumerado) que contenga los posibles valores de colores que se indican en el

enunciado del juego. De esta forma, la clase “Combination” tendría como atributo un array de Color de longitud 4, para formar así la propuesta del jugador.

- Además, nos damos cuenta que es necesario tener una clase capaz de guardar el “Result” de la combinación propuesta por el jugador.
- Por último, reordenamos los métodos que antes estaban en la clase “MasterMind” en cada una de las clases a las que pertenecen, para que todo esté más claro y con cohesión.

Con estos cambios, el código quedaría algo más ordenado y estaría dividido en distintas clases haciendo más pequeño y específico el contenido de cada una de ellas, con lo que conseguiríamos un código algo más fluido. También es un poco más fuerte, ya que, si queremos hacer una prueba en un método de una clase, resulta más sencillo que antes.

Como podemos comprobar, los nombres usados para las clases y sus atributos y métodos también han cambiado su formato e idioma, para adecuarnos así a los estándares fijados, lo cual permite que el código sea correctamente legible y comprensible por todos. El diagrama UML correspondiente a esta iteración es el siguiente:

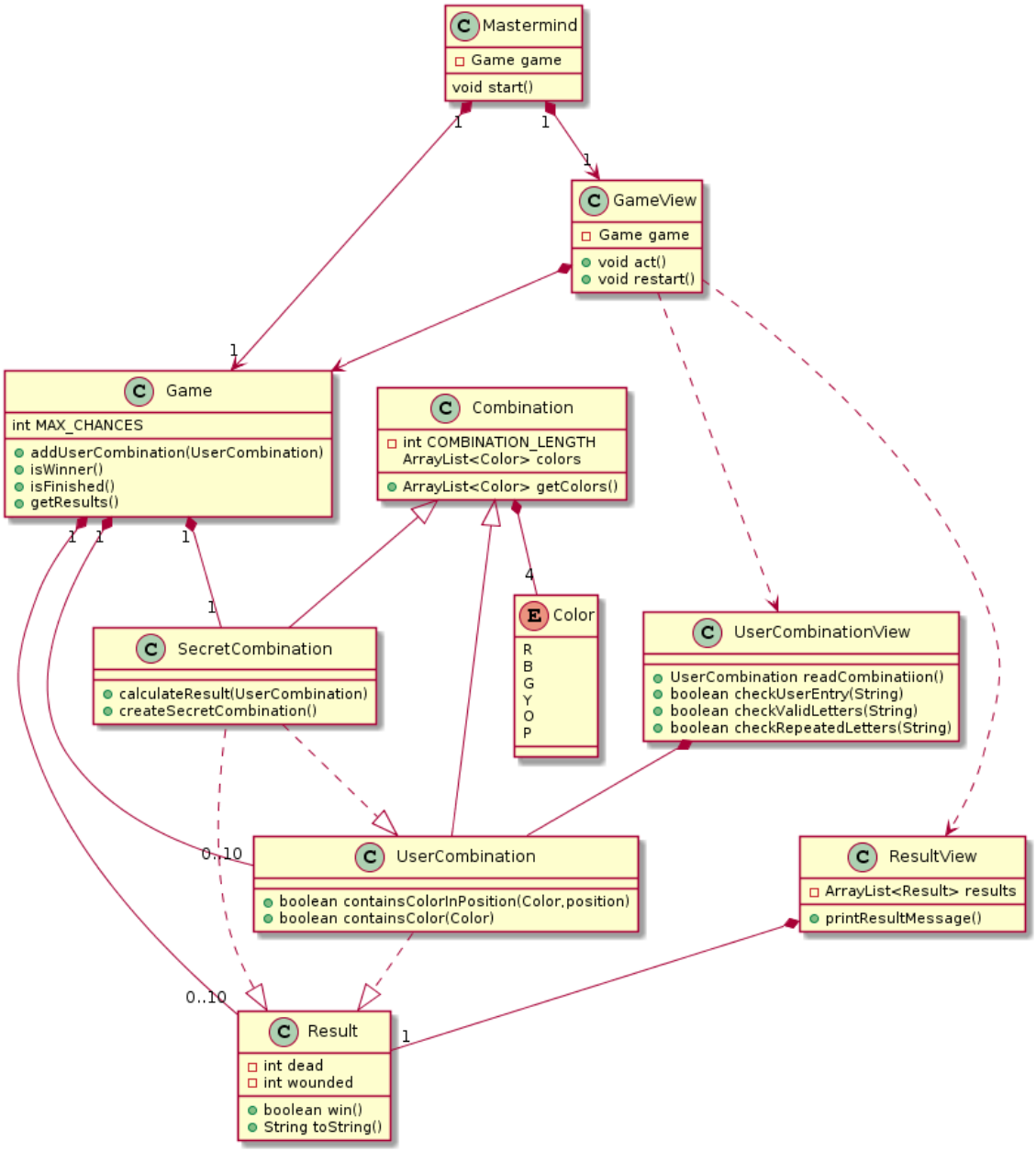


Iteración 4.

En esta iteración pretendemos implementar el modelo – vista, para poder diferenciar las funcionalidades de cada parte, consiguiendo así una mayor modularización y la separación de asuntos. Esta técnica es buena para poder adaptar nuestro código a posibles nuevas o diferentes funcionalidades que, en el caso de no estar el código divido, sería demasiado complicado y tedioso adaptarlo.

Se crean las vistas (Views) relacionadas con las clases Result, UserCombination y Game. En estas clases Vistas se gestiona lo necesario para la petición de datos al usuario y la salida de texto por consola. Por esto, debemos reorganizar el código que estaba en las clases anteriores para ponerlo de acuerdo al modelo-vista.

El diagrama UML resultante de esta iteración es:



Iteración 5

Con esta última iteración queremos conseguir que nuestro código se adapte al modelo-vista-controlador (MVC), siendo esta la forma más correcta, modularizada y organizada para completar un proyecto software. Con este modelo, las funciones y responsabilidades de las clases del proyecto están debidamente separadas y cada una de ellas responde a una función. Logramos así una buena práctica en nuestro código.