Programación II - COMISIÓN 01 - Universidad de General Sarmiento

- Trabajo Práctico Integrador -

2do. Cuatrimestre 2022

SEGUNDA PARTE

Profesores José Nores - Cristian Russo

Alumnos Cristian Pereyra López DNI 30387628

Diana Milagros Bogado DNI 43326653

INTRODUCCIÓN

En el presente trabajo práctico planteamos una posible solución para el análisis e implementación del **Sistema AlbumDelMundial**. En el mismo intentamos utilizar todos los conocimientos, conceptos y herramientas aprendidos en clase, no de forma deliberada, sino con fines utilitarios y pragmáticos para la mejor resolución de todos los problemas que fuimos encontrando en el desarrollo del trabajo. Procuramos utilizar y sacar provecho en todo momento de los pilares de la POO que vimos en la cursada;

ENCAPSULAMIENTO*:* *intentamos que en todo momento que los datos de los objetos queden de forma privada dentro de éste, solamente accesibles mediante métodos de consulta y en cada caso que así se requiera, que el retorno sea lo mínimo e indispensable necesario para resolución motivo de la consulta.*

ABSTRACCIÓN*:* *nos propusimos identificar aquellos atributos y acciones o métodos propios de un elemento que deseemos representar. Con esta información pudimos crear TADs, diagramas de clases, y eventualmente decidir y hacer Clases Abstractas, por ejemplo.*

HERENCIA: *mediante la herencia pudimos derivar automáticamente métodos y datos entre clases, subclases y objetos. Esto nos permitió reutilizar código, además de hacerlo más limpio y legible.*

POLIMORFISMO: *nos permitió generalizar el funcionamiento de los objetos que modelamos sin tener que forzar una implementación concreta, pudiendo estandarizar las llamadas a los métodos y que, de haber sobrescritura, se automatice la elección del correcto comportamiento según corresponda.*

Así también la utilización de Tecnologías Java como For Each, Iterator, StringBuilder, o estructuras de datos como HashMap, Array, ArrayList, etc, las cuales implementamos según la necesidad circunstancial y las ventajas que nos proveía su elección y uso.

La concepción de nuestro diseño está basada en lograr la mayor cohesión entre los métodos de cada clase buscando que estén fuertemente relacionados entre sí, delegando la responsabilidad de cada proceso a cual clase corresponda, y el menor acoplamiento posible, o sea el menor conocimiento entre las clases posible para poder funcionar y teniendo responsabilidades bien definidas, siendo más fáciles de entender por separado, reutilizables y adaptables a diferentes usos.

EJEMPLOS DE CONCEPTOS UTILIZADOS

- **HERENCIA**
  Utilizamos herencia en distintas clases;

```
public class AlbumTradicional extends Album
public class AlbumExtendido extends AlbumTradicional
public class AlbumWeb extends Album
public class FiguritaTop10 extends Figurita
```

Cada clase hereda de su clase padre los métodos y atributos en común con su clase padre, luego internamente agrega los que faltan o sobrescribe si es necesario.

- **POLIMORFISMO Y SOBREESCRITURA**

estaPegada en Clase albumTradicional (hereda de Clase Abstracta album)

```
public boolean estaPegada(Figurita figurita) {

        Figurita[] pais = selecciones.get(figurita.consultarPais());

        if(pais[figurita.consultarNumJugador()-1]!=null) {

                return true;

        }

        return false;

    }
```

estaPegada en la Clase albumExtendido (extiende de la Clase Álbum Tradicional):

```
public boolean estaPegada(FiguritaTop10 figurita) {
        Figurita[] mundial = mundiales.get(figurita.consultarPais());
            FiguritaTop10 top = (FiguritaTop10) figurita;
            if(mundial[top.consultarBalon()]!=null) {
                return true;
        }
        return false;
    }
```

En todo momento que se llame al método album.estaPegada,mediante polimorfismo el sistema, de acuerdo a la instancia del álbum y de la Figurita, decidirá cuál es el método correcto que deberá utilizar.

consultarValor() en Clase Figurita:

```
protected double consultarValor() {
        return this.valorBase;
    }
```

consultarValor() en clase FiguritaTop10 (extiende de Clase Figurita):

```
@Override
    protected double consultarValor(){
        if (esBalonOro(this.balon)) {
        return this.consultarValor() * 1.2 ;
        }
        return this.consultarValor() * 1.1;
    }
```

En todo momento que se llame al método figurita.consultarValor() ,mediante polimorfismo el sistema, de acuerdo a la instancia de la figurita, decidirá cuál es el método correcto que deberá utilizar.

- **SOBRECARGA**

```
protected boolean pegarFigurita(Figurita figurita){

      Figurita [] pais = selecciones.get(figurita.consultarPais());

            if (pais[figurita.consultarNumJugador()-1] == null) {

                pais[figurita.consultarNumJugador()-1] = figurita;

                this.albumLleno = albumLleno();

                return true;

            }

        return false;

    }

protected String pegarFigurita(FiguritaTop10 figurita){

        String pegada ="";

        FiguritaTop10 top = (FiguritaTop10) figurita;

        Figurita [] pais = mundiales.get(top.consultarMundial());

        if (pais[top.consultarBalon()] == null) {

            pais[top.consultarBalon()] =  top;

            pegada =(top.toString());

        }

        this.albumLleno = albumLleno();

        return pegada;

    }
```

La clase albumExtendido hereda de su clase padre albumTradicional el método pegarFigurita(Figurita Figurita), y además implementa pegarFigurita(FiguritaTop10 figurita).

- **INTERFACES**

```
public class AlbumDelMundial implements InterfazPublicaAlbumDelMundial
```

Nuestra clase creada AlbumDelMundial implementa la interfaz InterfazPublicaAlbumDelMundial con la implementación de las firmas dadas.

- **CLASES ABSTRACTAS**

```
public abstract class Album
```

Creamos una clase abstracta, que obviamente no puede ser instanciada, en la cual englobamos todos los atributos y métodos en común necesarios entre las clases AlbumTradicional, AlbumExtendido y AlbumWeb.

IREP (Invariante de Representación)

- **AlbumDelMundial**

- Para toda entrada la clave debe ser igual a participante.dni de todas las entradas en el diccionario participantes.

- Con la elección de hashmap como estructura de datos nos aseguramos que cada participante vinculado a participante.dni sea único

- **Participante**

- El dni debe ser un entero entre 0 y 99.999.999.

- El nombre de usuario no debe ser vacío.

- album tiene que ser un objeto del tipo Album y debe ser único en cada instancia de la clase Participante.

- El arrayList coleccion contiene solamente ejemplares del tipo Figurita y FiguritaTop10, los cuales pueden ser repetidas. (motivo por el cual implementamos esta estructura de datos)

- **AlbumTradicional / AlbumExtendido / AlbumWeb**

    - El código del álbum debe ser entero positivo único para cada álbum. (de esto nos aseguramos creando una variable de clase utilizada como contador e incrementándola en cada creación de un álbum)

    - Para toda entrada la clave debe ser igual a figurita.pais en todas las entradas en el diccionario selecciones.

    - Con la elección de hashmap como estructura de datos nos aseguramos que cada selección creada a partir de la lista paisesParticipantes sea única.

    - albumLleno se iniciará en False con cada creación de una instancia de un tipo de álbum, y solamente podrá ser True si ningún elemento del Array Figurita es null, recíprocamente, no podrá ser False si todos los elementos del Array Figurita son distintos de null.

- **AlbumTradicional**

    - El String Premios sólo podrá tener valor null si fue sorteado. Recíprocamente, será distinto a null si aún no fue sorteado.

- **AlbumExtendido**

    - Para toda entrada la clave debe ser igual a figurita.mundial en todas las entradas en el diccionario mundiales.

    - Con la elección de HashMap como estructura de datos nos aseguramos que cada mundial ingresado a partir de la lista listadoMundialesTop10 sea única.

- **AlbumWeb**

  - El codPromocional debe ser un entero positivo del formato correcto.
  - codPromocional sólo podrá tener valor null si fue usado. Recíprocamente, será distinto a null si aún no fue usado.

- **Figurita**

  - codIdentificacion es un entero positivo único, (de esto nos aseguramos creando una variable de clase utilizada como contador e incrementándola en cada creación de una figurita.
  -
  - El país debe estar incluido en la lista paisesParticipantes de la clase Fabrica.

  - numeroJugador debe ser un entero en el rango de a 1 a 12.

  - El valor base debe ser un entero positivo, igual a la suma entre numeroJugador y el ranking del dato pais registrado en la clase Fabrica.

- **FiguritaTop10**

  - mundial no puede ser vacío y debe pertenecer a la lista listadoDeMundialesTop10.

  - balon solamente puede tomar el valor entero 0 o 1, atribuyendo el valor 0 a balón de oro, y 1 a balón de plata.

  - Solamente puede haber dos instancias de la clase con su atributo mundial igual y de haberlas, el valor de balonOro debe ser distinto.

Desarrollo

Luego de la devolución de la primer parte del trabajo práctico y habiendo leído y analizado la segunda parte, nos dedicamos a modificar y ajustar los TADs que habíamos hecho a partir del primer enunciado; tuvimos que descartar acciones y TADs que vimos que no eran esenciales para resolver los problemas planteados (como por ejemplo, el TAD paisSeleccion). Con lo filtrado, realizamos un nuevo diagrama de clases marcando las relaciones entre clases esta vez sí reconociendo atributos y métodos en común entre éstas para decidir cómo y de qué manera era lógico y conveniente utilizar clases abstractas (por ejemplo, la clase abstracta Album), y plantear relaciones de herencia entre ellas (por ejemplo; AlbumExtendido que extiende de AlbumTradicional). Además, decidimos aprovechar la interfaz InterfazPublicaAlbumDelMundial dada por la cátedra implementandola en la clase AlbumDelMundial. También empezamos a tener en claro qué estructuras de datos iban a ser correctas para el mejor funcionamiento posible del sistema (HashMap para registrar los participantes que debian ser únicos utilizando su DNI como key, ArrayList en la colección de figuritas del participante las cuales eran objetos, quizás repetidos, que se iban a agregar y eliminar frecuentemente, Arrays en selecciones en donde su longitud iba a ser inmutable, etc.). Al terminar ésta primera parte de análisis y abstracción, empezamos a escribir código.

En nuestro codeo, primero probando los métodos aislados, y eventualmente testeandolos haciendoles interactuar, nos fuimos dando cuenta qué acciones debian pulirse y qué métodos auxiliares habíamos pasado por alto o eran necesarios para hacer el código más declarativo, legible y limpio. Fue muy interesante y provechoso ir aprendiendo detalles de casteos, ventajas de herencia y polimorfismo, mientras los utilizabamos para mejorar nuestro código haciéndolo más eficiente y elegante

(clave el acompañamiento y guía de los profesores en esto, además de la investigación por nuestra cuenta).

La siguiente etapa, una vez que logramos que el código cliente corra sin errores, fue empezar a correr los test y empezar a corregir acciones y código para pasarlos. Buena enseñanza fue el concepto que nos daban los profesores de no crear métodos solamente con el fin de pasar los test, sino intentar hacer los métodos necesarios bien hechos y que eso derive en los resultados lógicos esperados.

La última etapa, con el código cliente corriendo sin errores y los test pasados, fue hacer una última depuración. Principalmente borrar el código repetido que teníamos e intentar reducir lo escrito a lo mínimo posible. Acá fue quizás donde más tuvimos que intentar poner nuestra forma de pensar en lo conceptual trabajado en esta materia (particularmente en el proceso de intercambiar figurita); en cómo hacer interactuar los tres métodos entre sí, en cómo era mejor delegar las responsabilidades de cada acción a cada clase en lugar de hacerlo en el sistema principal, y en cómo entender que lo más óptimo era buscar la forma de que los objetos se envíen mensajes entre sí, entre otras cosas. También en esta etapa pudimos reconocer una situación excepcional en un test:

```
@Test
    public void t10_intercambiarFiguritasConAlbumVacio_DevuelveTrue() {
        // Para este caso se registraron 2 tradicionales
        // y ambos compraron figuritas.

assertTrue(sistema.intercambiarUnaFiguritaRepetida(dniConAlbumTradicional)
);
    }
```

Este test tiene la particularidad que excepcionalmente la figurita que se intenta intercambiar tiene un valor más alto que las disponibles para intercambio del otro usuario registrado, por lo tanto aún cuando el participante que desea intercambiar tiene su album vacio y le serviría cualquier figurita para pegarla, la comparación de valores hace que excepcionalmente el método devuelva False en un promedio aproximado de una vez cada seis testeos.

Analisis Complejidad Método buscarFiguritaRepetida

```
@Override
    public int buscarFiguritaRepetida(int dni) {
        Participante participante = validarYBuscarParticipante(dni);
        return participante.buscarFiguritaRepetida();
    }
```

Método de complejidad O(1)

```
private Participante validarYBuscarParticipante(int dni) {
        if (participantes.containsKey(dni)) {
            return participantes.get(dni);
        }
        throw new RuntimeException("El participante no esta registrado");
    }
```

Método de complejidad O(1) (La búsqueda de un valor utilizando su clave en la estructura de datos HashMap es de complejidad O(1)).

```
protected int buscarFiguritaRepetida() {

        if (coleccion.size() == 0) {

            return -1;

            }

        return coleccion.get(0).consultarNumeroIdentificador();

    }
```

Método de complejidad O(1)

```
protected int consultarNumeroIdentificador() {

        return this.numeroIdentificador;

    }
```

Método de complejidad O(1)

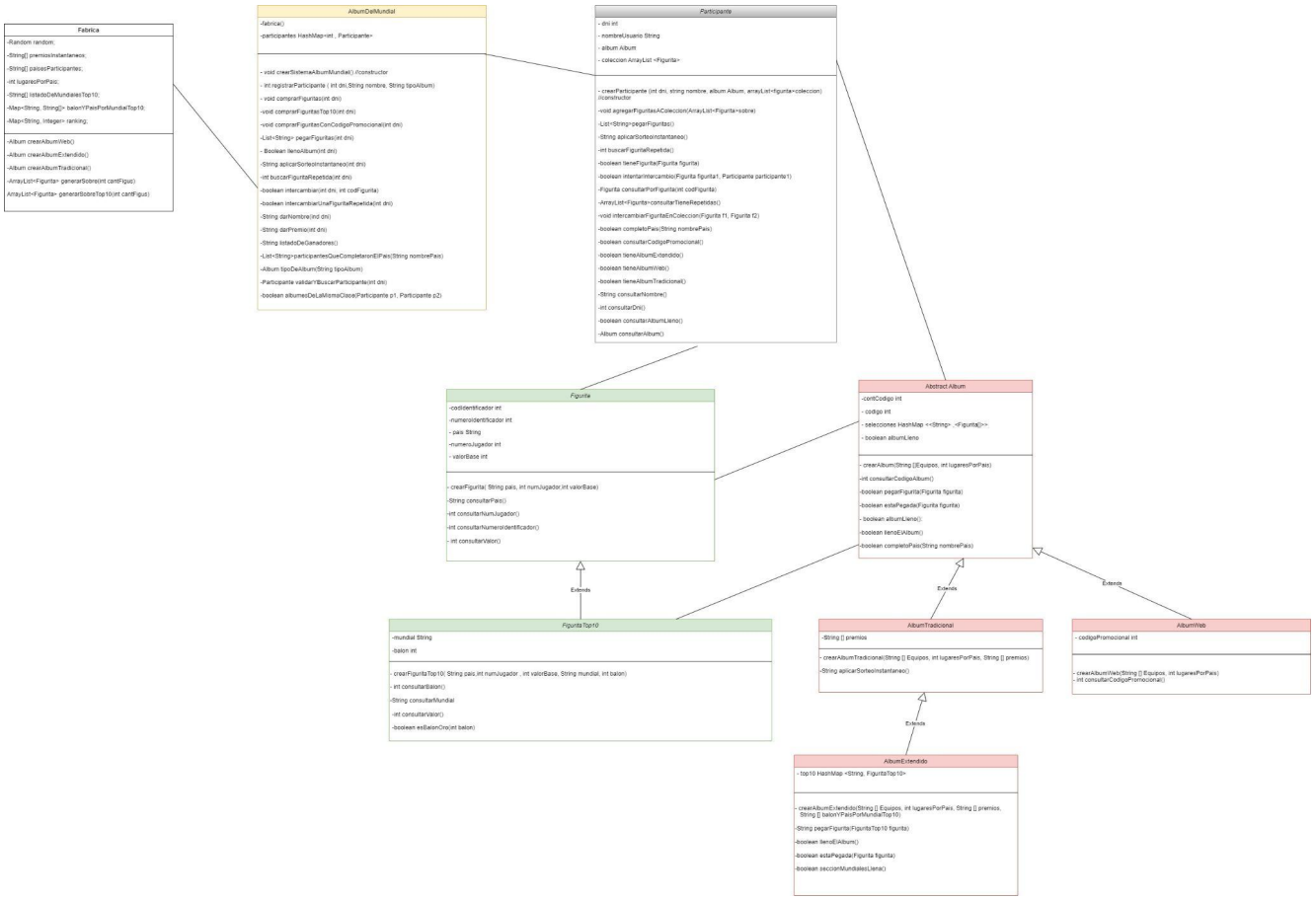
En nuestro diseño cada vez que se pega una figurita, se actualiza la colección del participante removiendo las que son pegadas, asi que siempre que se necesite una figurita es suficiente al participante que retorne la primera figurita de su *ArrayList coleccion*, sin necesidad de recorrerlo y hacer chequeos adicionales.

Por Álgebra de Órdenes:

O(1) + O(1) + O(1) + O(1) = O(max(1 , 1 , 1 ,1)) Regla 2

O(1) Regla 2

Diagrama de Clases Final



[Link al diagrama de clases en drawio]

Conclusiones

En la resolución de este trabajo práctico pudimos desarrollar y aplicar varios conceptos aprendidos en clase, sobre todo ejercitar el pensamiento abstracto e intentar despegarnos de paradigmas previos que tenemos adquiridos de previos acercamientos a la Programación Orientada a Objetos (POO). Encontramos el trabajo desafiante en su justa medida, si bien encontramos problemas en donde quizás nos parecía que no nos alcanzaba con nuestros conocimientos para resolverlos, con la guía de los profesores, investigación propia, y prueba error, progresivamente pudimos ir determinando cuál era la mejor opción para encontrar soluciones. También nos deja el aprendizaje de cómo aprovechar e implementar de forma adecuada y eficiente conceptos aprendidos de POO como herencia, polimorfismo, encapsulamiento, etc, como también de estructuras de datos antes desconocidas para nosotros, como HashMap, HashSet, etc., o tecnologías Java que jamás habíamos usado, como StringBuilder o Iterator, además de nuestra primer experiencia de testeo utilizando jUnit, lo cual más allá de enseñarnos y obligarnos a depurar nuestro código, nos hizo tener que desarrollar métodos y acciones generales que sumen a realizar su función, el cumplir con lo pedido en cada test individual.

En cuanto a intentar adoptar buenas prácticas, tratamos en todo momento de hacer un código limpio, declarativo, y, en la medida de nuestras habilidades, elegante.

Como líneas finales, nos gustaría decir que encontramos el trabajo práctico y la cursada en general, interesante y provechosa, principalmente por la enseñanza de métodos de razonamiento y abstracción que fueron nuevos para nosotros y que son imposibles de adquirir en cursos de programación o simil.