**RESOLUCIÓN DEL EXAMEN:**

**Fase 1: Análisis**

- Lectura del enunciado (dos veces)

- Es un **nivel de abstracción alto de generalización** de lo que se pide.

Si vamos línea a línea, codificando lo que nos dice el enunciado, perdemos la visión global.

Eso no quiere decir que luego, en algún momento, después del análisis general, cuando se vayan a los detalles, se repase línea a línea el enunciado y se constate que se están cumpliendo todos los requisitos.

Pero en esta primera fase de análisis, nos olvidamos de los detalles y nos centramos en la visión general.

En el caso de este ejercicio: “Dentro de un main (que pertenecerá a una clase Java porque en Java siempre son clases), crearemos 9 boletos (el enunciado nos da los valores de los atributos de los objetos boletos así que no tenemos que pedir información al usuario ni por consola ni con componentes swing). Para cada boleto se comprueba si es correcto (hay una serie de condiciones que tiene que cumplir, CADA BOLETO) y si es correcto, lo insertamos en la base de datos. Después de la parte de la inserción, haremos una consulta general a la tabla y al final, eliminaremos todos los boletos.”

**Como el enunciado nos da la salida, podemos realizar esta reflexión/análisis también viendo la salida que nos ofrece.**

—> Es en este momento en el que podemos pensar en otros ejercicios que hemos hecho y consultarlos. Pero sólo después de saber qué tenemos que hacer, en un nivel alto de abstracción.

**Fase 2: Diseño**

- Con papel y boli se realiza el **diseño** hasta entender exactamente lo que pide el enunciado y **cómo y dónde lo vamos a codificar**.

- Pensar el diseño de clases (estamos en Java, siempre clases).

(1) Una **clase principal que tendrá el main**. Esta clase puede llamarse Loteria, AplicacionSorteo, Principal, etc.

- Construye los objetos sorteo con los valores del enunciado

- Crea el array

- Y para cada boleto del array, realizará las operaciones de inserción en base de datos.

- Por lo que hemos visto en clase, **la comprobación de si un registro boleto ya existe** en la base de datos (la jornada no puede ser la misma, es decir, el campo jornada actúa como PRIMARY KEY, el resto de campos da igual si se repiten) es haciendo una consulta a BD. Esta comprobación la haremos en el main.

- Dentro del main, después de insertar los boletos (o no, en función de errores), consulta la tabla y luego borra todos los registros.

(2) Otra **clase Sorteo** que tendrá los atributos, el constructor y los métodos.

Dibujar un objeto Sorteo ¿qué información va a tener? ¿qué acciones tiene asociadas?

En nuestro caso, tiene la acción imprimirSorteo() y también comprobarSorteo() —> ERROR común: No hace falta pasar parámetros porque estos métodos comprueban los atributos de su propio objeto.

imprimirSorteo es sacar por pantalla el valor de los atributos de dicho objeto (NO hacer un ”select \* from sorteos;”)

**comprobarSorteo() parece más complejo, podemos dejar el análisis, diseño y codificación de este método, para el final (pero tiene pinta que la comprobación está relacionada con los valores de los campos, nada que ver con BD).**

No obstante, es muy importante en esta fase, saber **qué devuelven los métodos** para poder utilizarlos correctamente en el flujo de instrucciones del main:

\* imprimirSorteo devuelve void porque simplemente imprime en pantalla los datos del propio boleto

\* comprobarSorteo devuelve un int que será 0 si el boleto es correcto y != 0 si es incorrecto.

—> Podemos volver a pensar en ejercicios ya hechos que realicen un diseño de clases similar.

**Fase 3: Codificación**

Tras el diseño, ya sabemos qué tenemos que **codificar**. Y lo vamos a hacer de manera **incremental**, poniendo **TRAZAS** y compilando y **ejecutando** nuestro código en cada mejora e incremento funcional en codificación.

Paso 1:

Creamos la clase principal con el main.

Creamos la clase **Sorteo con el constructor**.

Dentro del main, creamos los 9 objetos Sorteo, los metemos en un **array**.

Como cada operación/inserción en BD es de boleto, tendremos que **recorrer el array con un for** y ver si están bien creados. Es decir, crearemos un for y recorreremos el array.

TRAZAS (system.out en cada iteración del for para ver cada objeto creado) y EJECUCIÓN

**Si hubiéramos hecho esto, habríamos detectado errores en el constructor, en la clase …**

Paso 2:

En el main, hacemos que dentro del for, sea con una llamada al método de cada objeto Sorteo, que imprima su información (**imprimirSorteo()**).

Creamos dicho método en la clase Sorteo.

En el main, dentro del for, construimos el **string que almacena la sentencia insert** con los valores de cada objeto Sorteo. Creamos una TRAZA y ejecutamos.

Paso 3:

Seguimos en el for e incluimos la comprobación de que el boleto es correcto.

Creamos un método **comprobarSorteo()** en la clase Sorteo que siempre devuelva 0 y dejamos su implementación para el final.

Incluimos el código para **ejecutar la sentencia de INSERT en BD**.

Ponemos TRAZAS y EJECUTAMOS.

Podremos resolver problemas de inserción en BD … borramos desde la consola, los registros creados.

Paso 4:

Seguimos en el for. Ya insertamos boletos en BD. Ahora vamos a incluir la funcionalidad de “no insertar si el boleto ya existe en BD”.

Tenemos que saber si el boleto existe en BD. Así, la consulta Select debe incluir preguntar por un boleto con una jornada.

Incluimos la sentencia “Select \* from sorteos where jornada = sorteo[i].jornada”, la ejecutamos con executeQuery, recuperamos el resultSet, preguntamos en el if …

Ponemos TRAZAS y EJECUTAMOS.

En los resultados vemos que hay un registro que está repetido (el tercer sorteo). Debería reconocerlo en nuestro programa.

Paso 5:

(Dejamos comprobarSorteo() para más adelante)

Salimos del for y vamos a hacer la consulta de BD. Supone un “select \* from SORTEOS”.

Construimos la sentencia, la ejecutamos con executeQuery(), recogemos el resultTest, iteramos e imprimimos —> NADA COMPLEJO porque lo hemos hecho muchas veces.

Ponemos TRAZAS y EJECUTAMOS

Paso 6:

Nos ponemos con la funcionalidad DELETE. Sabemos los registros que hay en la tabla, por lo tanto, sabemos los que tiene que borrar. Creamos la sentencia, ejecutamos … el entero que devuelve executeUpdate nos dice los registros borrados. Es esta variable la que necesitamos imprimir por pantalla (NO un contador que nos ha contado el número de registros que se han insertado).

Ponemos TRAZAS y EJECUTAMOS.

Paso 7:

Ya podemos ponernos con el **método comprobarSorteo().**

Analizamos los requisitos y vemos cuales son los más fáciles de codificar: comprobar fecha, si números son > 50 o 0 …

Codificamos los supuestos más sencillos, ponemos TRAZAS y EJECUTAMOS.

Nos apoyamos en los resultados del enunciado para comprobar que se ejecuta correctamente.

Paso 8:

Antes de abordar el requisito de números repetidos, vemos cómo integramos en el main, la llamada al método comprobarSorteo(), en función del valor int que devuelva y los mensajes de error que escribe al comprobar el boleto. Y que además, si hay error, no inserta en BD. La lógica nos dice que si el boleto tiene un error, NO lo vamos a insertar.

Ponemos TRAZAS y EJECUTAMOS.

Paso 9:

Ya podemos abordar la codificación de la comprobación de números repetidos.

Está descrito en el enunciado: crearemos un array con los 5 números. Lo ordenamos y comprobamos solo números consecutivos.

Alguno de vosotros lo implementó de otra manera, comprobando en un if, las diferentes posibilidades. Es Válido.

Ponemos TRAZAS y EJECUTAMOS.

Paso 10:

Comprobamos que como la entrada es la misma que en el enunciado (es decir, los objetos que se crean son los mismos), la salida debería ser la misma (con respecto a registros insertados, la consulta y el borrado).

Depuramos “errores” .

Paso 11:

Mejoramos los mensajes de salida para que salgan igual que en el enunciado.

Paso 12:

Optimizamos el código. Sólo si tenemos claro que podemos ejecutar el código las veces que sean, con un resultado correcto, iríamos a ver si podemos optimizar alguna parte.

En este ejercicio, por ejemplo, los if’s anidados en el método comprobarSorteo(), o cómo se utiliza en el main, y pasar a usar un switch.

Pero solo haríamos esto si podemos comprobar ejecutando el programa que los cambios de optimización no cambian la funcionalidad.