UNAS IDEAS BÁSICAS SOBRE EL DISEÑO DE CLASES

Vimos algunas ideas del diseño de métodos (pseudocódigo) pero ¿de donde salen los métodos que quiero diseñar?, y por tanto, ¿De donde salen las clases que contiene esos métodos?. El diseño de métodos es mucho más simple que el diseño de clases.

El diseño de clases es un tema muy amplio y complejo, vamos a ver aquí por tanto, sólo unas pinceladas para ser conscientes de su necesidad. En general, cuanto más grande es el problema a resolver, precisaremos más clases, y por tanto, cuanto más grande es el problema a resolver, más importante es el diseño de clases. En nuestros mini programas, por tanto, no es un factor crítico pero hay que ir abordándolo paulatinamente.

¿Qué es el diseño de clases?. El proceso mediante el cual decidir cómo estructurar una aplicación en clases. Lógicamente si mi aplicación consiste en calcular la media de dos números, no es necesario ningún diseño ya que todo se resuelve en una única clase que hace la operación en un main() de 3 o 4 instrucciones.

Por lo tanto, el diseño de clases implica dos pasos:

1. obtener las clases de las que va a constar mi aplicación.
2. Para cada clase decidir su estructura (atributos y métodos de los que consta cada clase).

Para hacer esto hay múltiples e intrincados métodos, técnicas y herramientas. El UML es la herramienta principal para el diseño de clases, nos sirve para dibujar diseños pero no nos dice cómo diseñar.

¿Cómo sabemos si nuestro diseño es correcto?
Existen algunos síntomas que nos indican que el diseño de un sistema es bastante pobre:

- RIGIDEZ (las clases son difíciles de cambiar)

- FRAGILIDAD (es fácil que las clases dejen de funcionar)

- INMOVILIDAD (las clases son difíciles de reutilizar)

- VISCOSIDAD (resulta difícil usar las clases correctamente)

- COMPLEJIDAD INNECESARIA (sistema "sobrediseñado")

- REPETICIÓN INNECESARIA (abuso de "copiar y pegar")

- OPACIDAD (aparente desorganización)

EJEMPLO DE COMPLEJIDAD INNECESARIA

Vamos a ver ahora un ejemplo sencillo de COMPLEJIDAD INNECESARIA. Supón que necesitamos para una aplicación "Unidad4" guardar una lista de nombres de amigos

```
class Amigo{
   String nombre;
   int edad;
   int numeroDeHermanos;

   public Amigo(String nombre, int edad, int numeroDeHermanos) {
```

```
        this.nombre = nombre;
        this.edad = edad;
        this.numeroDeHermanos = numeroDeHermanos;
    }

}
public class Unidad4{


        public static void main (String[]args){

                Amigo[] amigos={new Amigo("juan",9,2),new Amigo("chu",8,1),new Amigo("fri",9,0)};

               String[] amigos2={"juan","chu","fri"};

           }
}
```

¿Qué enfoque es mejor, tener una clase *Amigo*, o directamente guardar los nombres en un array de Strings?. TODO DEPENDE DEL CONTEXTO. Es imposible obtener la solución adecuada sin tener claro el contexto. Si sólo quiero usar nombres de amigos la clase Amigo sobra ya que es una COMPLEJIDAD INNECESARIA.

EJEMPLO DE INMOVILIDAD(clase difícil de reutilizar)

Un ejemplo típico de mal diseño que dificulta la reutilización de clases (pero no el único caso) es mezclar en una única clase la entrada salida con el código que hace cálculos o en general que resuelve el corazón del problema.

El siguiente ejemplo tiene una clase Unidad4 que se dedica a la E/S y una clase Complejo que hace cálculos. Observa que la clase Complejo deliberadamente no tiene contacto directo con el usuario

```
class Complejo{
   double real;
   double imag;
   Complejo(double real,double imag){
      this.real=real;
      this.imag=imag;
   }

   public String convertirAString(){
      return this.real +(this.imag<0?" ":" + ")+this.imag+"i";
   }

   void acumular(Complejo b){
      this.real=this.real+b.real;
      this.imag=this.imag+b.imag;
   }
   Complejo sumar(Complejo b){
      double nuevaParteReal=this.real+b.real;
      double nuevaParteImaginaria=this.imag+b.imag;
      return new Complejo(this.real+b.real,this.imag+b.imag);
   }

}

class Unidad4{
   public static void main(String[] args){
      Complejo a = new Complejo(2.3,-8.9);
      Complejo b= new Complejo(2.0,2.0);
      Complejo c=a.sumar(b);
      System.out.println("a:("+a.convertirAString()+ ")  b:("+b.convertirAString()+")   a.sumar(b):( "+ c.convertirAString()+")");
      a.acumular(b);
      //a=a.sumar(b);//equivalente a a.acumular(b);
      System.out.println("a:("+a.convertirAString()+ ")  b:("+b.convertirAString()+")   a.acumular(b):( "+ a.convertirAString()+")");
   }
}
```

supón a continuación que hubiéramos escrito la función convertirAString() de la siguiente forma

```
public void  convertirAString(){
      System.out.println( this.real +(this.imag<0?" ":" + ")+this.imag+"i");
   }
```

Ahora, este método sólo funciona con la entrada/salida de consola, no se podría aprovechar directamente(sin retocar) la clase Complejo para una aplicación web o gráfica de escritorio, es decir, es difícil de reutilizar.
A pesar de lo dicho, hay muchas aplicaciones que se sabe que sólo van a ser de consola o de escritorio y vemos frecuentemente que se mezcla la entrada/salida con la lógica de la solución del problema y así se gana concisión y simplicidad, como siempre, **todo depende del contexto.**

DESARROLLO DE APLICACIONES EN CAPAS
Las aplicaciones complejas se organizan en capas lógicas en las que cada una tiene una función. Incluso en aplicaciones sencillas debe haber al menos una capa de negocio y otra de capa de presentación tal y como se ilustró en el ejemplo anterior

Dependiendo de la magnitud de la aplicación puede estar organizada en 2,3,4 o más capas. Puedes leer algo por ejemplo en https://www.ecured.cu/Arquitectura_en_Capas

Aplicaciones de dos capas: lógica de negocio y lógica de presentación.

Para lograr desarrollar aplicaciones mantenibles, extensibles y escalables tenemos que poder diferenciar y separar dos cuestiones de naturaleza muy diferentes que llamaremos "lógica del negocio" y "lógica de presentación". Es decir: una cosa es cómo la aplicación expone los resultados al usuario y cómo interactúa con este, y otra cosa muy distinta es cómo se procesan los datos para elaborar los resultados.
Generalmente, en toda aplicación, podemos distinguir dos componentes principales que llamaremos "frontend" y "backend". El frontend es el componente que interactúa directamente con el usuario. Le permite introducir datos, manda estos datos a procesar al backend, recoge el resultado del procesamiento y le muestra los resultados al usuario. El backend procesa los datos que recibe a través del frontend y le devuelve a este los resultados del proceso.
Visto así, el usuario interactúa con el frontend y este interactúa con el backend.
USUARIO<=>FRONTEND<=>BACKEND

En el ejemplo anterior
FRONT-END =>clase Unidad4
BACK-END=>clase Complejo

En aplicaciones complejas cada capa podrá constar de muchas clases, no sólo de una como en el ejemplo.
El siguiente ejemplo se utiliza la clase Complejo utilizada más arriba pero ahora Unidad4 no utiliza la consola para la entrada/salida. Gracias a la estructura por capas que utilizamos hemos conseguido entre otras cosas que la clase *Complejo* sea reutilizable(usarla en otro contexto sin modificarla) aunque cambie el sistema de entrada y salida.

```
import javax.swing.JOptionPane;

class Complejo{
   double real;
   double imag;
   Complejo(double real,double imag){
      this.real=real;
      this.imag=imag;
   }

   public String convertirAString(){
```

```
        return this.real +(this.imag<0?" ":" + ")+this.imag+"i";
    }

    void acumular(Complejo b){
        this.real=this.real+b.real;
        this.imag=this.imag+b.imag;
    }
    Complejo sumar(Complejo b){
        double nuevaParteReal=this.real+b.real;
        double nuevaParteImaginaria=this.imag+b.imag;
        return new Complejo(this.real+b.real,this.imag+b.imag);
    }

}

class Unidad4{
    public static void main(String[] args){
        Complejo a = new Complejo(2.3,-8.9);
        Complejo b= new Complejo(2.0,2.0);
        Complejo c=a.sumar(b);
        JOptionPane.showMessageDialog(null,"a:("+a.convertirAString()+ ")  b:("+b.convertirAString()+")   a.sumar(b):( "+ c.convertirAString()+")");
        a.acumular(b);
        //a=a.sumar(b);//equivalente a a.acumular(b);
        JOptionPane.showMessageDialog(null,"a:("+a.convertirAString()+ ")  b:("+b.convertirAString()+")   a.acumular(b):( "+ a.convertirAString()+")");
    }
}
```

Los principios SOLID

El diseño orientado a objetos es todo un mundo. Lo visto más arriba no son más que pinceladas escasas y desordenadas pero "entendibles" en este momento. Actualmente para verificar si un diseño es correcto se utilizan mucho los principios SOLID. Son demasiado técnicos para entenderlos en estos momentos pero está bien que te vaya sonando su nombre y que te des cuenta que el diseño de clases no es algo en absoluto trivial si no todo lo contrario, algo muy complejo.

Los 5 principios SOLID de diseño de aplicaciones de software son:

- S – Single Responsibility Principle (SRP)
- O – Open/Closed Principle (OCP)
- L – Liskov Substitution Principle (LSP)
- I – Interface Segregation Principle (ISP)
- D – Dependency Inversion Principle (DIP)

Como estamos introduciéndonos a realizar aplicaciones en dos capas, nos interesa analizar un poco el principio SRP.

El principio de responsabilidad única (Single Responsibility Principle (SRP))

El enunciado original indica que: *Un módulo debe tener una única razón para cambiar*

Esto actualizado a POO y haciendo referencia a Clase en lugar de módulo y simplificando podemos enunciarlo (aunque sabiendo que falta un poco de rigor) como:

Una clase debe de tener sólo una función(responsabilidad)

Como ya vimos, si hubiéramos escrito en la clase complejo

```
public void  convertirAString(){
      System.out.println( this.real +(this.imag<0?" ":" + ")+this.imag+"i");
    }
```

Estaríamos otorgándole a la clase Complejo dos funciones: gestionar el número complejo y gestionar la comunicación con el usuario. Ya discutimos que esto trae problemas de mantenimiento.

El principio de responsabilidad única y las arquitecturas en capas

Las arquitecturas en capas usan este principio para incluir las clases en una u otra capa. Una clase no debe mezclar responsabilidades de lógica de negocio con responsabilidades de lógica de presentación.

Observa que por supuesto, como ya indicamos, una capa puede tener muchas clases ya que dentro de una capa también se puede concluir por este principio que debe de haber varias clases.