

PROGRAMACIÓN ORIENTADA A OBJETOS

- **TEMA 1_INTRODUCCIÓN A POO:**

- La **abstracción**: consiste en la capacidad de descomponer un programa en subprogramas o funciones más sencillas, y un tipo de datos en otros más sencillos.
- Un **módulo**: se define como una entidad independiente de código que almacena un conjunto de procedimientos y funciones relacionados, junto con los datos que manipulan.
- El **encapsulamiento**: consiste en que los datos llevan asociados las funciones para manejarlos.
- La **ocultación de datos**: esto hace que los datos se tengan que modificar a través de operaciones o funciones asociadas a los mismos.
- Una **clase**: esta define las propiedades o estados de un tipo de objeto o entidad, y las operaciones admitidas para manipularlos, es decir, su comportamiento.
- Las **variables de un tipo de clase**: estas actúan como referencia a los objetos, de manera similar a los punteros. Los objetos de un tipo de clase, se denominan ejemplares.
- Un **programa orientado a objetos**: conjunto de clases que definen los tipos de objetos que van a existir en tiempo de ejecución, junto con un programa principal que define cuál es el primer objeto que se crea y cuyos métodos ejecutan.
- La **herencia**: mecanismo mediante el cual los objetos de una clase (**subclase**) disponen de los mismos atributos y métodos que la otra clase (**superclase**). Esto nos permite organizar el código y fomenta la reutilización de este.
- El **polimorfismo**: propiedad que permite que el tipo de una referencia varíe durante la ejecución de un programa orientado a objetos.
- La **vinculación dinámica**: esta consiste en la selección de la implementación de un método durante la ejecución de un programa orientado a objetos, y no durante la compilación.

• TEMA 2_DISEÑO BÁSICO DE CLASES:

- **Estrategia de WIRFS-BROCK:** esta es una **estrategia** utilizada para la identificación de los diferentes objetos que tenemos que utilizar en un programa. Esta no solo incluye el análisis gramatical de la especificación textual, sino también procesos específicos para la detección de superclases, colaboraciones, relaciones, etc.

Este método lleva a cabo unos filtros para eliminar, colapsar o usar como atributos aquellos candidatos que no son adecuados como objetos del sistema.

- Las **tarjetas CRC de Beck y Cunnigham:** es un método para la identificación y documentación de objetos. Esta combina el objeto (**clases**), con el de sus **responsabilidades y colaboraciones (CRC)**.
- Esto nos ayuda a un correcto nivel de abstracción (**def. T.1**) siendo conscientes de que no es necesario reflejar el mundo real y asegurándonos que los objetos tienen un **comportamiento diferencial** con los otros.
- **Representación de clases en UML:**
 - **Mínima:** solo es necesario el nombre de las clases. Corresponde a ¼.
 - **Esencial:** añadimos a las clases algunos atributos con sus tipos e incluir los métodos que no se refieren al acceso a los atributos. (2/4)
 - **Abreviada:** casi completa, le añadimos lo anterior más los getters&setters de los atributos. (3/4)
 - **Completa:** todos los atributos junto con los métodos que componen la clase, además de los valores que se pasan por parámetros y los que se devuelven en cada uno de los métodos. (4/4)
- **Atributos:** estas son las propiedades que tiene una clase. Es decir, las **características de objeto**.
- **Métodos:** operaciones que puede realizar un objeto.
- **Modificadores del acceso:** estos se utilizan para establecer la visibilidad de las clases, los atributos y los métodos.
 - **public(+):** nos indica que es visible desde cualquier clase, aunque este en un paquete diferente. (clases, atributos, métodos)
 - **private(-):** indica que sólo se puede acceder desde dentro de la clase. (atributos, métodos)
 - **protected(#):** indican que no será accesible desde otra clase, pero si podrá accederse a él por métodos de la clase además de las subclases que deriven. (atributos, métodos)
- **Encapsulación:** consiste en que los atributos de los objetos deben ser privados y para controlar su acceso, se lo tenemos que dar a través de métodos públicos.

Abreviada

Employee
-name : String
-hourlyWage : double
-hoursWorked : double
+getName()
+setName()
+getHourlyWage()
+setHourlyWage()
+getHoursWorked()
+setHoursWorked()
+getEarnings()

Mínima

Employee

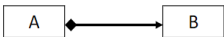


Completa

Employee
-name : String
-hourlyWage : double
-hoursWorked : double
+getName() : String
+setName(newName:string)
+getHourlyWage() : double
+setHourlyWage(newHourlyWage:double)
+getHoursWorked() : double
+setHoursWorked(newHoursWorked:double)
+getEarnings() : double

Esencial

Employee
-name : String
-hourlyWage : double
-hoursWorked : double
+getEarnings()

• TEMA 3_IMPLEMENTACION BÁSICA EN POO

- Una **instanciación**: proceso mediante el cual se declara un objeto.
- **Getters**: método que devuelve el valor de un atributo. Ejemplo: getName.
- **Setters**: método que modifica el valor del atributo. Ejemplo: setName.
- Los **parámetros**: estos se declaran cómo secuencias de forma separados por coma.
 - **Tipos primitivos**: se pasan **por valor** (crea una copia de los mismo al método y no se modifica en el cuerpo de este).
 - **Tipo objeto**: se pasan **por referencia** (se pasa al método una referencia a un objeto y éste puede ser modificado en el cuerpo del método).
- Un **constructor**: esto es un método que se utiliza para la inicialización de los atributos de un objeto.
- Constructor **"this"**: se utiliza dentro de un método de una clase, para referenciar a la instancia actual y permite:
 - Utilizar parámetros en los métodos con el mismo nombre que los atributos del objeto.
 - Realizar llamadas a otros constructores desde un constructor dado.
- Mecanismo **"cast"**: nos permite convertir tipos de expresiones para poder realizar cálculos que de otra manera no serían posibles.
- **Asociaciones**: esto es una relación entre dos clases. Encontramos:
 - Por **composición**: parte de él y no tiene sentido sin él. 
 - Por **agregación**: tiene sentido por libre o como parte del objeto. 
 - Por **asociación simple**: tiene relación pero no de **parte y todo**, es decir, puede vivir solo y ser utilizado por más objetos al mismo tiempo. 
- **Asociación unidireccional**: esta tiene una referencia a un objeto de la segunda clase, pero no al revés.
- **Asociación bidireccional**: cada clase tiene una referencia a un objeto de la otra clase.
- **Multiplicidad o cardinalidad** de las asociaciones: esta nos indica el número de instancias de una clase que pueden ser asociadas a una sola instancia de otra.

Representación	Significado
0..1	Cero o un ejemplar
0..* ó *	Cero o más ejemplares
1	Exactamente un ejemplar
1..*	Uno o más ejemplares

- **Asociaciones de multiplicidad 1:** estas se implementan con la utilización de los **atributos**.
- **Asociaciones de multiplicidad fija:** se implementan con la utilización de los **arrays**.
- **Asociaciones de multiplicidad N:** se implementan utilizando **contenedores** o colecciones.
- **Contenedores homogéneos:** todos los elementos pertenecen al mismo tipo de dato.
- **Contenedores heterogéneos:** los elementos pueden ser de distinto tipo de dato.
- **Contenedores de tipo List (lista):** su comportamiento es el de una estructura de recorrido secuencial con posibles elementos repetidos.
- **Contenedores de tipo Set (conjunto):** se comporta como una estructura lineal sin repeticiones de elementos, y donde el orden de recorrido no es relevante.
- **Contenedores de tipo Map (mapa o función):** se comporta como una tabla hash, es decir, los elementos se acceden en tiempo constante a través de una clave.
- **Herencia:** esta nos permite definir una relación entre dos clases. **(def. T.1)**
- **Herencia múltiple:** esta hereda atributos o métodos de varias clases.
- **Especialización:** esta nos permite que una clase A, sea definida como una especialización de otra clase B, más general. Como consecuencia, la clase A, hereda todo lo de B.
- Palabra clave **“super”**: permite a un objeto referenciarse a sí mismo, pero interpretándose como un objeto de la clase padre. En consecuencia, puede ejecutar versiones superiores de un método en la jerarquía de clases.
- **Jerarquía de herencia:** esta es la colección de todas las clases de una superclase común. Toda clase desciende directa o indirectamente de la clase *Object*.
- **Superclase:** es la clase de la que las demás heredan.
- **Subclase:** clase que hereda de otras clases como puede ser la superclase.
- Una **excepción:** es un problema que impide la continuación del método o bloque de procesamiento en el que se produce.
- Una **región protegida:** es una sección de código que puede producir excepciones, y que va acompañada usualmente del código necesario para gestionarlas. Estos suelen ser bloques **try-catch**.

TEMA 4 – DISEÑO AVANZADO DE CLASES

4.1 – IMPLEMENTACIÓN AVANZADA EN JAVA / IMPLEMENTACIÓN DE APLICACIONES

La documentación **JavaDoc** es un tipo de comentario HTML construido por que se ponen antes de la clase a comentar.

- Puede tener varias etiquetas (return, author...) que se especifican con un @etiqueta dentro del comentario

Paquetes: Agrupamiento de clases o interfaces relacionadas que proporciona protección de acceso y gestión de espacio de nombres. (Java tiene paquetes llamados API) que se llaman con import (Estructura modular de alto nivel)

- Algunos paquetes: Lang(funciones del lenguaje de progr.) | IO (lectura y escritura de archivos), Util (clases de utilidad como ordenadores)

Normas de nomenclatura de paquetes

- Minúsculas para no confundir con clases
- Las compañías tienen que poner nombres como *com.miempresa.mipaquete*
- Los paquetes de java empiezan por *java* o *javax*

***Los nombres no forman jerarquía Al importar paquete1 no estamos importando paquete1.paquete 2 . Para ello usamos *

Maneras de acceder a un paquete

“Declarando la función mediante el nombre del paquete” POCO USADO : `java.util.Vector v = new java.util.Vector();`

Importando el miembro: `import java.util.Vector;`

Importando el paquete: `import java.util.*;`

*Se recomienda crear las clases con carpetas y directorios para que al compilar, tengamos los archivos .class igual

Archivos JAR: (Creo que el profe no lo ha explicado)

Son archivos comprimidos en .zip controlados por la orden *jar* (Comando para crear .jar:)

```
jar cfm fich.jar META-INF/MANIFEST.MF *.class
```

Se habrá creado un directorio META-INF con un archivo MANIFEST.MF en el que podemos decir indicaciones del módulo como cuál es la clase principal

4.2 SOBRECARGA Y SOBRESCRITURA

Sobrecarga es la capacidad de un lenguaje de programación que permite nombrar con el mismo identificador (nombre) a distintas variables u operaciones (funciones)

Métodos:

- Restricción: El número y tipo de parámetros que se le pasa a la función
- Ventaja: 2 métodos que realizan lo mismo tienen el mismo nombre

Constructores (Método con el nombre de la clase que se ejecuta cuando se instancia la clase con `new ()`):

- `Super`

Operadores: Java permite utilizar operadores sobrecargados *ejemplo* `Syso("Tienes " + edad + " años");` *Normalmente no se puede sumar enteros y strings*

Sobrescritura: La diferencia es que tenemos que respetar todas las características que en la sobrecarga cambiábamos (número de parámetros, colocación, tipo de datos (normales u objetos)). *Def*: Volver a escribir un método que ya existía.

****La sobrescritura solo que puede realizar cuando heredamos las funciones de una clase padre, tenemos que utilizar el nombre de la función que heredamos pero podemos cambiarle el código "sobrescribir"**

```
1. class Vehiculo {
2.     public void frenar () {
3.         System.out.println("Frenamos de manera estándar");
4.     }
5. }
```

```
1. class MiCoche extends Vehiculo {
2.     public void frenar () {
3.         System.out.println("Frenamos un coche");
4.     }
}
```

4.3 Abstracto y final

Clase abstracta: Son aquellas clases que representan conceptos generales pero no pueden ser instanciadas (dan error por el compilador), se utilizan para crear clases hijo que las hereden

```
Public abstract class NombreClase{    }
```

Una clase es abstracta cuando tiene por lo menos un método abstracto, son declarados pero no tienen código sino “;” *ejemplo:* public abstract void funcion(int a);. Puede tener métodos no abstractos

Debemos escribir todas las cabeceras de las clases

Herencias y diseño UML

Def herencia: Una clase A hereda de una clase B cuando es una especificación de ella y extiende la información que almacena

Una clase que hereda de una clase abstracta puede ser abstracta también o incluir los métodos declarados

Diagrama UML: Un método está sin declarar (es abstracto) cuando está en cursiva, las clases que lo heredan estarán sin cursiva

Atributo final

Se utiliza para declarar constantes (variables que no vamos a poder alterar)

Sintaxis: final tipo nombre = valor;

******Para que un valor sea accesible desde todas las clases, tenemos que incluir public y static

Resultado public static final pi = 3.1415f;

Las clases declaradas como final no van a poder ser heredadas por otras clases para que no puedan sobrescribir el método

4.4 Modificadores de acceso

(palabras como public, private o static para asignar a las clases) |(Representación UML entre paréntesis)

“visibilidad”: Un elemento es visible cuando otros elementos lo pueden ver (clase, atributo o método)

Modificadores de clases:

- Por defecto: public
- Public: La clase será accesible desde cualquier parte del paquete

Modificadores de métodos:

- Public: El método será accesible desde dentro o fuera de la clase, hasta fuera del paquete (+)
- Private: El método solo es accesible dentro de la clase (-)
- Protected: El método solo podrá ser accedido dentro de la clase pero desde fuera se puede llamar a métodos que lo utilicen (#)
- Por defecto: public dentro del paquete

Modificadores en atributos (los símbolos en UML son iguales)

- Public, private y por defecto (igual que los métodos)
- Protected (no accesible fuera de la clase pero los métodos si que podrán acceder + herencias)

Variables estáticas:

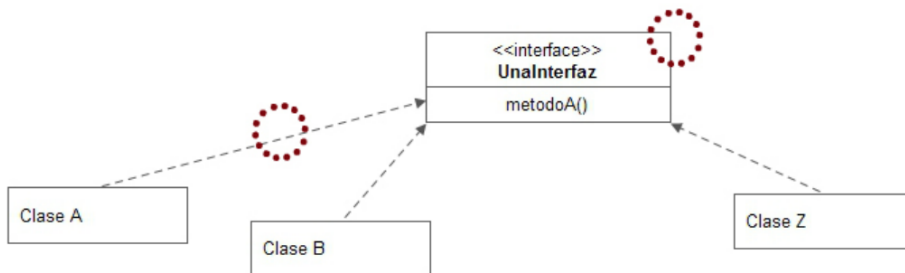
- Solo se reserva espacio una vez, un solo valor para todas
- Solo se usará cuando el valor tenga sentido global a todas las instancias de la clase
- No se necesita tener instanciada una clase para usar el método (conversiones Integer.parseInt());

4.5 Interfaces

En Java una clase solo puede heredar de otra clase (herencia múltiple) pero puede heredar de varias interfaces

Def: Una clase abstracta con todos sus métodos abstractos declarada interfaz por su palabra *interface*

*****Una interfaz no debería de tener atributos pero en Java se pueden utilizar (van a ser publicos, estáticos y constantes) pero es un error de diseño aunque esté correcto sintácticamente*****



Representación de interfaz → Relación de dependencia -----

El uso de las interfaces se hace mediante la palabra *implements*.

Si una clase hereda de varias interfaces que tienen la misma función, no va a haber problema ya que está sin instanciar.

TEMA 5- ARQUITECTURA E INTERFACES GRÁFICAS

5.1 GuiAwtSwing

El API de java son un conjunto de librerías que podemos utilizar en nuestros programas, estas librerías están formadas por paquetes que son clases relacionadas

Para utilizar la clase exacta, podemos poner la librería y paquete o podemos usar import para que el código sea más fácil de leer

Awt: Conjunto de clases que permiten la construcción de interfaces gráficas con los eventos generados por estas clases

- Contenedor: Elemento gráfico que puede contener componentes (ventana, diálogo, panel), se le considera componente también
- Componente: Elemento gráfico que aparece en la GUI (Graphical User Interface) (Botones, campos de texto, etiquetas, Lienzos de dibujo)

Swing: Def: Librería versión mejorada de AWT, soluciona problemas, mejora aspecto interfaces y portabilidad

Los componentes se crean mediante constructores con diferentes parámetros, estos componentes tienen métodos para cambiar ciertos aspectos

Contenedores:

- JFrame: Ventana tradicional de Windows
- JDialog: Ventana que sirve para mandar preguntas o advertencias, depende de un frame principal
- JPanel: Permite agrupar los elementos metiéndolos dentro de un frame
- Canvas: Permite dibujar figuras en este (polígono, círculos, líneas...)

-Para añadir elementos a un contenedor usamos el metodo add

```
<contenedor>.getContentPane().add(<componente>);
```

******Para cambiar aspectos de una ventana podemos usar setTitle(""), setBackground(Color.red), setSize(x,y);

Layouts: Se utilizan para controlar la posición dentro de un contenedor, tipos de layouts

```
<contenedor>.setLayout(<Layout>);
```

- FlowLayout: Sitúa los elementos de izq a der y de arriba abajo, es el default
 - This.setLayout(new FlowLayout);
 - This.add(<Componente>);
- BorderLayout: Sitúa los elementos en las direcciones *North, South, East, West, Center*
 - This.setLayout(new BorderLayout);
 - This.add(<componente>, "north");
- GridLayout: Divide los contenedores en un determinado número de filas y columnas iguales y sitúa a cada elemento en una de estas
 - *this.setLayout(new GridLayout(2,3));*
 - *This.add(<componente>);*
- GridBagLayout

- Divide al contenedor el filas y columnas de distinto tamaño
- CardLayout
 - Gestiona un conjunto de tarjetas para que solo una de estas sea visible a la vez
-

Gestión de eventos

- En la programación tradicional, el programa está constantemente revisando las acciones del usuario para ver si se ha pulsado algún botón
- En Java, existen modelos de eventos definidos, el programa es avisado automáticamente de estas acciones
- El modelo de eventos de Java tiene 2 elementos
 - Fuentes de eventos: Elementos sobre los que se producen los eventos
 - Escuchadores de eventos: Elementos que reciben las notificaciones de los eventos

Para asignar un escuchador a una fuente de eventos se utiliza:

```
<FuenteEvento>.add<EventListener>(<Escuchador>);
```

Diálogos (showConfirmDialog, showInputDialog, showMessageDialog)

Entrada/Salida consola (paquete java.io)

- Leer por consola:
 - Opción 1: crear un BufferedReader(new InputStreamReader(System.in));
 - Opción 2: Crear un Scanner y usar los métodos .nextLine(); .nextInt();

Excepciones

Durante la ejecución de un programa pueden haber errores, para manejarlos podemos hacer uso del try catch(excepción), la clase Exception incluye todas las excepciones posibles, se pueden hacer tantos bloques catch como excepciones se quieren capturar

- Se puede usar tanto con try catch, como con throws en la cabecera

Lectura de ficheros:

Apertura: File f = new File(fichero);

Lectura: BufferedReader lector = new BufferedReader(new FileReader(f));

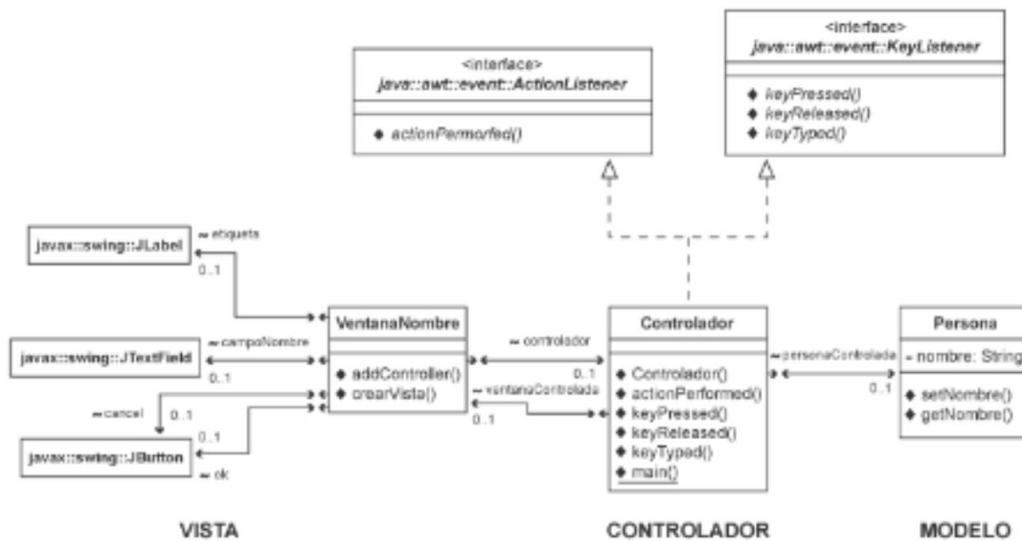
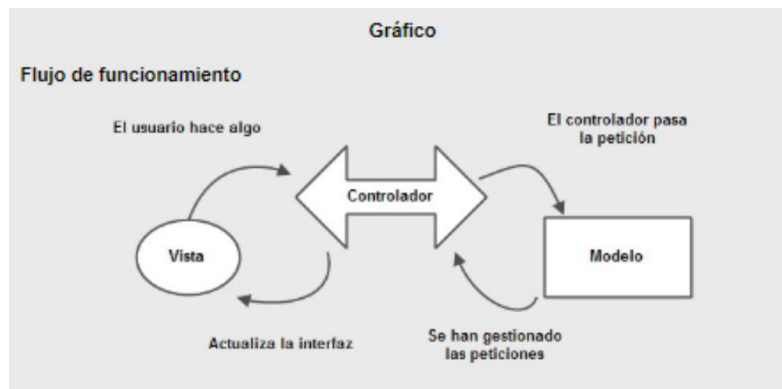
Try catch con código de leer línea (line = lector.readLine();) mientras que line != null

```
fileWriter = new PrintWriter (new BufferedWriter(new FileWriter("f.txt")));
fileWriter.println("");
```

5.2 IntroMVC MUY IMPORTANTE

MVC: Def: Modelo Vista Controlador, tiene como objetivo independizar la interfaz gráfica de la lógica de negocio, la idea es diseñar la aplicación con 3 capas para poder asignar más trabajadores y hacerlas independientes,

- Vista: Parte de la aplicación que vemos en la pantalla, todos los archivos de Java Swing
- Control: Funciones de intermediario entre la capa vista y la capa modelo, listeners,
- Modelo: Clases que se encargan de las operaciones, conectar base de datos, (javaBeans, son pequeñas clases para definir atributos)



Ejemplo MVC

5.3 Principios S.O.L.I.D. (<https://www.youtube.com/watch?v=2X50sKeBAcQ>)

Hay que saberlos, no hace falta saberlo de memoria, “explicar con 5 palabras”

HAY UN MONTÓN DE PREGUNTAS, SOBRE TODO LA D (INMERSIÓN DE DEPENDENCIAS, COMO SE INYECTA UNA DEPENDENCIA (SOBRE TODO EN EL REPOSITORIO))

<https://juan-garcia-carmona.blogspot.com/2012/11/solid-y-grasp-buenas-practicas-hacia-el.html>

Def: 5 Normas o recomendaciones que guían la forma de programar para que el código sea mantenible, aplicar cambios y errores y facilitar

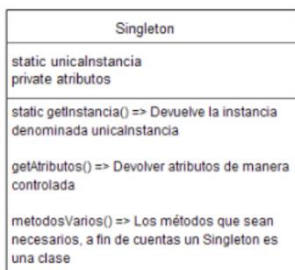
- Single Responsibility: Las clases deberían hacer una sola cosa para asegurar que la hacen muy bien
- Open closed: Una entidad software debe quedarse abierta para su expansión pero cerrada para su modificación
 - Nuestro sistema debería estar protegido, “para añadir funcionalidades deberíamos crear nuevo código y no cambiar el código que ya está creado”
 - Esto se puede realizar mediante herencia y polimorfismo
- Liskov substitution: Toda clase que sea hija de otra clase, debe poder utilizarse como si fuera la hija padre
- Interface Segregation: (Segregación de la interfaz), es mejor tener muchas clases pequeñas y especializadas que una clase enorme, si tienes pocas clases con muchas líneas de código, este va a ser muy lento y poco efectivo.
 - Ejemplo que hemos hecho: Tener distintas interfaces pequeñas (crear, leer, actualizar y borrar)
- Dependency Inversion: Los módulos de alto nivel no deberían depender de los módulos de bajo nivel, todos estos deberían depender de interfaces. Se basa en la abstracción, las implementaciones concretas deberían depender de capas abstractas para producir desacoplo de software.
 - Patrón strategy, decidir si vamos a usar una u otra

TEMA 6- PATRONES DE DISEÑO (*Design Patterns: Elements of Reusable Object Oriented Software*)

Patrón de diseño: Def: Solución en forma de diseño para resolver algún problema identificado. Se pueden clasificar en 3 grupos:

- Creacionales: Trabajan con las instancias de clases
 - Abstract Factory: Interfaz para crear familias de objetos relacionadas
 - Builder:
 - Factory
 - Prototype
 - Singleton
- Estructurales: Nos dan combinaciones entre clases y objetos para poder crear soluciones
- De comportamiento: Facilitan la labor de interacción entre clases

6.1 Singleton



Singleton: Def: Patrón de diseño que nos permite solo poder crear una instancia de nuestra clase en todo el código para evitar comportamientos extraños en nuestro código y desde varios dispositivos tener siempre la misma instancia

“La clase es la responsable de devolver la instancia de ella misma o crear una si no hay una creada ya”

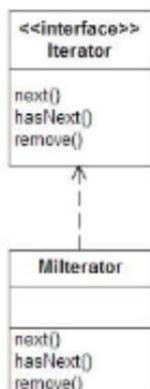
La clave de este patrón es crear un objeto llamado Singleton estático y privado y un método llamado getInstance() que nos va a devolver el contenido de la clase

If unicaInstancia == null{ unicaInstancia = new Singleton();} return unicaInstancia;

**Podemos incluir la palabra synchronized para que no se pueda volver a ejecutar el código de la clase hasta que se haya parado de ejecutar este mismo

Public static synchronized Singleton getInstance(){}

6.2 Iterator (no lo ha dado del todo, saber en qué consiste pero no implementar)



Iterator: Def: **Interfaz** que está construida para permitirnos iterar sobre colecciones de elementos, permitirnos eliminar elementos y ver sus métodos:

- hasNext() → Boolean que es true si quedan más elementos que iterar
- next() → Object que devuelve el siguiente elemento en la iteración
- remove() → void que elimina de la colección el último elemento iterado

Como es una interfaz, los elementos deben de tener función a excepción de remove que no tiene por qué si no se va a utilizar

6.3 Strategy → Un ejemplo puede ser nuestro repositorio

El patrón strategy se utiliza para otorgar distintas funcionalidades a una serie de objetos mediante el uso de interfaces, estas van a estar implementadas ya que vamos a instanciar a las clases hijo con clases abstractas que van a usar estas funciones.

```
1. abstract class SuperHeroe
2.     IVolar estrategiaVolar;
3.     IDisparar estrategiaDisparar;
4.
5.     realizarDisparo() {
6.         estrategiaDisparar.disparar();
7.     }
8.     ....
9. }
```

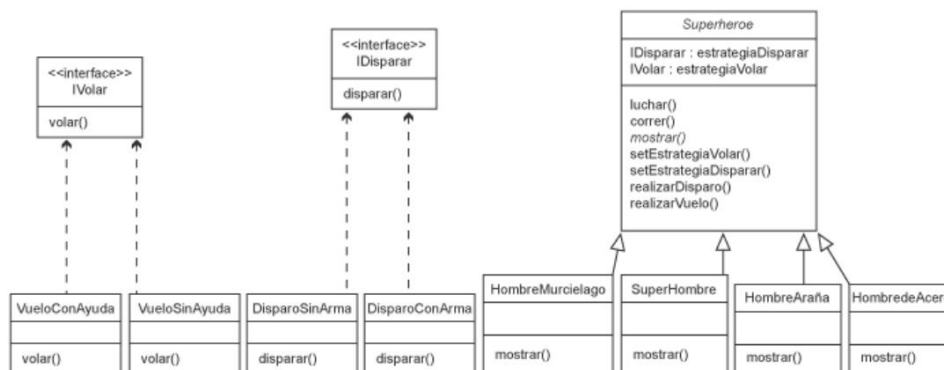
De esta manera, solo tenemos que cambiar una sola línea de código para poder cambiar la funcionalidad

```
1. estrategiaDisparar = new DisparoConArma();
```

Incluso se puede cambiar en tiempo de ejecución mediante un setter de estrategia disparar

```
1. void setEstrategiaDisparar (IDisparar eDisparar) {
2.     estrategiaDisparar = eDisparar
3. }
```

```
1. public static void main (String[] args) {
2.     SuperHeroe hombreAraña = new HombreAraña();
3.     hombreAraña.realizarVuelo();
4.     hombreAraña.realizarDisparo();
5.
6.     //Ahora cambiamos la estrategia de disparo
7.     hombreAraña.setEstrategiaDisparo (new DisparoConArma());
8.
9.     hombreAraña.realizarDisparo();
10. }
```



6.4 – Calidad Junit (entra solo teoría, nada de JUnit)

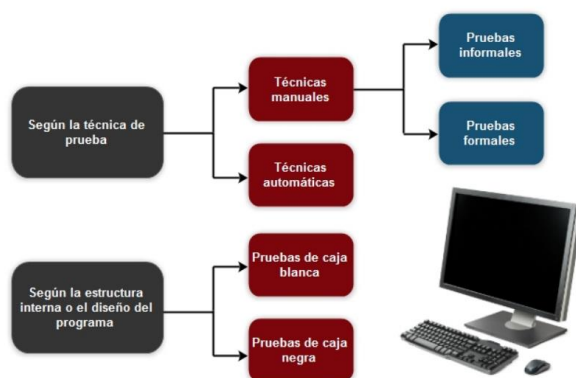
“Calidad”: *Def*: Entregar un producto cumpliendo las expectativas del cliente

Tenemos que definir los requisitos funcionales (qué debe hacer el programa) y no funcionales (restricciones del programa) y hacer pruebas para verificar su cumplimiento

- Pruebas funcionales: Aceptación, Regresión, alpha testing, beta testing
- Pruebas no funcionales: Rendimiento de carga, seguridad, usabilidad, mantenimiento

Estas pruebas deben empezar por los elementos más fáciles (funciones/métodos)

Técnicas de comprobación:



- Según la técnica de prueba
 - Técnicas manuales: Definidas y ejecutadas manualmente por expertos de prueba
 - Pruebas informales: Probadores usan la app sin plan predeterminado
 - Pruebas formales: Se definen guiones de prueba para ejecutar de manera minuciosa en busca de errores
 - Técnicas automáticas: Definidas por expertos, se ejecutan de manera automática por medio de una plataforma de pruebas
- Según la estructura interna o el diseño del programa: (Pueden ser automatizadas)
 - Pruebas de caja blanca: Validar la estructura de la aplicación buscando código sin utilizar estructura interna de datos...
 - Pruebas de caja negra: Verificar si el programa devuelve correctamente la salida del programa dependiendo de las entradas recibidas sin tener en cuenta el diseño/código

Pruebas unitarias: *Def*: Comprobación sistemática de clases utilizando el conjunto de datos de entrada y verificando que los resultados obtenidos son los esperados. (Pruebas automáticas de caja negra más extendidas). Deben cumplir lo siguiente

- Desatendida: Ejecutarse sin atención del usuario (ninguna intervención)
- Universal: Asumir configuraciones particulares o en datos que pueden variar de una prueba a otra
- Atómica: Comprobar la funcionalidad concreta del componente, función o clase

Funcionalidad de estas pruebas: simplificar la integración, documentar el código, fomentar la separación entre interfaz e implementación, localización y corrección de errores

Métodos Junit

- `assertArrayEquals(int[], int[]) → Verifica que los 2 arrays sean idénticos y registra el error`
- Para hacer pruebas, tenemos que utilizar la etiqueta `@Test`

```
import org.junit.* ;
import static org.junit.Assert.* ;

public class SumaListaPruebas {

    @Test
    public void pruebaSuma() {
        int[] valoresEsperados = {4,5,6,7};
        int[] valoresEntrada = {1,2,3,4};
        SumaLista listaEntrada = new SumaLista(valoresEntrada);
        listaEntrada.suma(3);
        assertArrayEquals(valoresEsperados, listaEntrada.lista);
    }
}
```

Normas de notación de las pruebas:

- La clase de pruebas se tiene para *Ejemplo* se tiene que llamar *EjemploPrueba*
- Para los métodos de prueba, tenemos que usar `metodo1Prueba1, metodo1Prueba2()`...
- Lo mejor es meter todas las pruebas de una clase en otra clase de pruebas
- Se puede sustituir “Prueba” por “Test”
- Los métodos de prueba tienen que ser públicos, sin parámetros y devolver void
- Una prueba unitaria tiene que ser pequeña y probar una sola funcionalidad
- Método AAA (*Arrange Preparar, Act Actuar, Assert Afirmar*):
 - Se preparan los datos deseados para el método, se ejecuta el método

```
1.  @Test
2.  public void testSuma() {
3.      // Preparar
4.      Calculadora calculadora = new Calculadora();
5.      // Actuar
6.      int res = calculadora.suma(1, 1);
7.      // Afirmar
8.      assertEquals(2, res);
9.  }
```

Condición	Explicación
<code>assertEquals([mensaje], A, B)</code>	Validar la igualdad de los objetos A y B, utilizando el método <code>equals()</code> .
<code>assertSame([mensaje], A, B)</code>	Validar que A y B son el mismo objeto, utilizando el operador <code>==</code> .
<code>assertNotSame([mensaje], A, B)</code>	Validar que A y B son objetos distintos, utilizando el operador <code>!=</code> .
<code>assertTrue([mensaje], A)</code>	Validar que la condición A es <i>true</i> .
<code>assertNull([mensaje], A)</code>	Validar que el objeto A es <i>null</i> .
<code>assertNotNull([mensaje], A)</code>	Validar que el objeto A no es <i>null</i> .
<code>assertEquals([mensaje], double A, double B, double delta)</code>	Validar que A y B no difieren entre sí más que un <i>delta</i> dado, mayor que cero.
<code>assertArrayEquals([mensaje], int[] A, int[] B)</code>	Validar la igualdad de los arrays A y B componente a componente.

- El parámetro mensaje es opcional pero recomendado, String
- La comparación entre números reales, en los float y double se hace con un parámetro delta que va a ser la diferencia entre uno y otro debido a la diferencia de valor por limitaciones de memoria
- Junit nos proporciona métodos para hacer que las pruebas se ejecuten antes o después de todas las pruebas

- Pruebas más avanzadas
 - `@Text(timeout=100)` → Test falla si el test tarda más de 100ms

Ejemplo

Etiquetas de pruebas:

`@RunWith(value=Parameterized.class)` → Se van a ejecutar tantas instancias de ejecución como parámetros creamos en `@Parameters`, hay que ponerlo antes de declarar la clase como abajo

`@Parameters` → Se pone justo encima de una función que devuelve `Collection<Object[]>` y devuelve un array dentro de la función `Arrays.asList()`

```
public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {

        { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }

    });
}
```

@Test

```
1.  import static org.junit.Assert.*;
2.  import org.junit.Test;
3.  import org.junit.runner.RunWith;
4.  import org.junit.runners.Parameterized;
5.  import org.junit.runners.Parameterized.Parameters;
6.  import java.util.*;
7.
8.  @RunWith(value=Parameterized.class)
9.  public class PruebaCalculadoraSuma {
10.     // Valores
11.     private double esperado, arg1, arg2;
12.     // Constructor con los parámetros necesarios
13.     public PruebaCalculadoraSuma(double esperado, double arg1, double arg2)
14.     {
15.         this.esperado = esperado;
16.         this.arg1 = arg1;
17.         this.arg2 = arg2;
18.     }
19.     // Método para definir el array de pruebas, anotado con @Parameters
20.     @Parameters
21.     public static Collection<Object[]> generarParametros() {
22.         Object[ ][ ] casos = { { 2, 1, 1 }, { 3, 2, 1 }, { 4, 3, 1 } };
23.         return Arrays.asList(casos);
24.     }
25.     // Prueba parametrizada
26.     @Test
27.     public void testSuma() {
28.         Calculadora calculadora = new Calculadora();
29.         double calculado = calculadora.suma(arg1, arg2);
30.         assertEquals(esperado, calculado, 0.0);
31.     }
32. }
```

Preguntas examen

"si yo tengo una clase a que depende de la clase b y voy a probar la clase a, que implica eso"

sol: si no he probado la clase b,c,d la clase a no va a poder funcionar porque alguna de sus dependencias falla, hay que probar todos los módulos, es mejor hacer las pruebas por separado

Refactorizar → Reestructurar nuestro programa pero sin añadir o quitar funcionalidad, modificar comportamientos

Serializar un objeto → "Un objeto es serializable si lo puedo almacenar en memoria en pseudocódigo binario / código de byte / código binario ¿En qué formato lo guardamos?. Tiene partes que compila a código binario y partes que no las compila

"Si tengo 3 instancias, que las guarde en el código binario de java"