Cuestiones Preguntadas en Examen

Describe encapsulamiento y ocultación de la información. ¿ Cuál es su utilidad?

El encapsulamiento consiste en que la información del tipo y sus operaciones se localizan en el mismo lugar.

En la ocultación de la información la información sobre la implementación se encuentra oculta al usuario.

Su utilidad es impedir que el usuario pueda modificar parte de la implementación o acceder a los módulos que no le interesan.

Describe los factores de calidad del software robustez y portabilidad. Indica también si son factores externos o internos.

La robustez es un factor externo y consiste en la capacidad del producto de comportarse con exactitud, de manera satisfactoria y precisa en todas las situaciones.

La portabilidad es un factor externo e interno y consiste en la capacidad del sistema de poder ejecutarse en un sistema operativo diferente o en un hardware distinto.

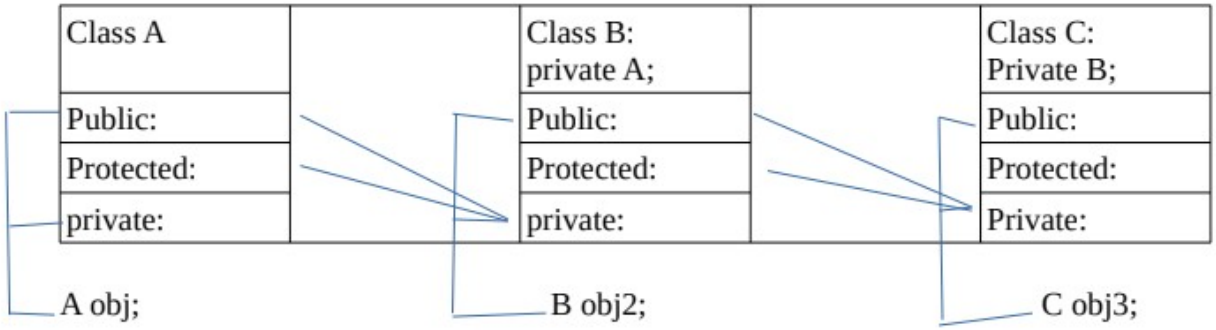
Factores de calidad del software

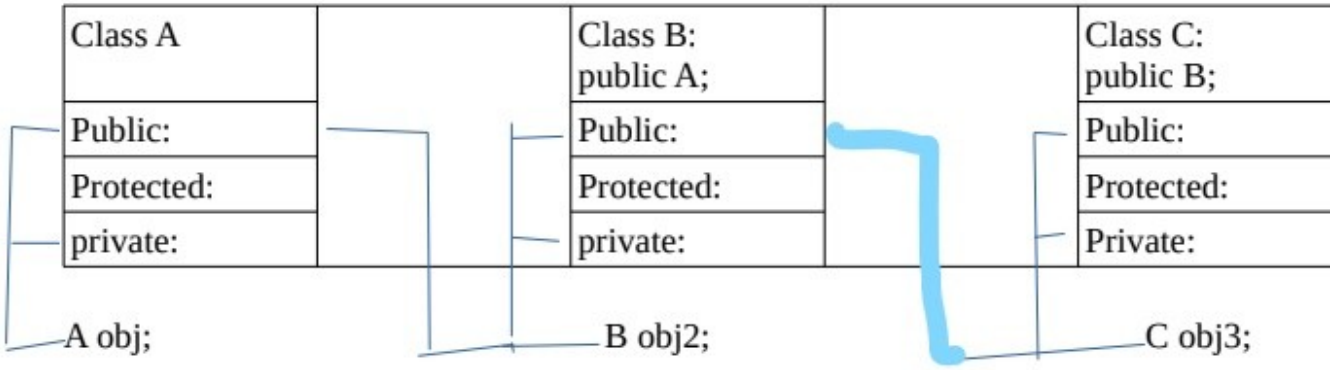
Nombre	**Tipo**	**Descripción**
Corrección	Externo	Capacidad del producto de realizar de forma adecuada su función, según los documentos de especificación y requerimientos
Robustez	Externo	Capacidad del producto de comportarse con exactitud, de manera satisfactoria y precisa en todas las situaciones.
Eficiencia	Externo e Interno	Conseguir que el programa realice correctamente, y de la mejor forma posible, su

		función
Integridad	Externo	El producto no debe corromperse por su utilización masiva o una gran acumulación de datos.
Facilidad de uso	Externo	Facilidad al introducir datos, manipular datos, etc
Portabilidad	Externo e Interno	Capacidad del producto de ejecutarse en un software o hardware distinto
Reutilidad/Reusabilidad	Interno	Capacidad del producto de ser reutilizado totalmente o en gran parte por otros productos
Compatibilidad	Externo	Facilidad de los programas de combinarse entre sí.
Mantenibilidad	Interno	Capacidad de encontrar y corregir un defecto en el software
Extensibilidad	Interno	Facilidad de adaptar el producto a cabios en la especificación de requisitos
Testable	Interno	Habilidad de validación del software
Seguridad	Externo	Capacidad del producto de proteger sus componentes de usos no autorizados y situaciones de pérdida de información
Accesibilidad	Externo	Acceso a la información sin

		limitación alguna por deficiencia, incapacidad o minusvalía
Oportunidad	Externo	Capacidad de un producto de ser lanzado al mercado
Economía	Externo	Costes del producto

Indica con flechas el acceso de las siguientes clases:





Codifica una función en C++ que intercambie el valor de 2 variables de tipo int, que recibe como parámetros utilizando referencias.

```
void intercambiar (int &a, int &b){
        int aux = a;
        a = b;
```

```
        b = aux;
    }
```

¿Qué es la programación con TAD y cuáles son las distintas operaciones de un TAD?

La programación con TAD es una división del sistema en partes diferenciadas y definir sus interfaces.

Se busca la claridad, la reducción de costes y la reutilización. Las operaciones de un TAD son:

a) Constructores: Permiten inicializar el objeto y sus datos internos.
b) Observadores: Permiten acceder en modo lectura a una cartacterística del objeto.
c) Modificadores: Permiten acceso de escritura a una característica del objeto.
d) Destructores: Llevan al objeto a su estado inicial descartando posibles efectos laterales improvistos.

¿ Qué son las referencias en C++? Indica sus utilidades y pon un ejemplo. ¿ Por qué es de gran utilidad el paso de objetos como referencia en C++?

Las referencias son alias que pueden darse a las variables o a los objetos de un programa. Sus utilidades son las siguientes:

1. Da eficiencia y rapidez.
2. Evita que a cada nuevo objeto declarado se le reserve un espacio de memoria y se le almacene una dirección.

Por ejemplo: bool asignar(int &a, Persona &P)

En C++ el paso de objetos de referencias es muy importante pues permite ahorrar memoria y tiempo de ejecución. Esto se debe a que al pasar objetos como referencia no se realizan copias.

Para un fichero de código fuente que se llama persona escribe el código correspondiente a las guardas de inclusión que se escribe al principio del fichero y el código de dichas guardas de inclusión que se escribe al final del fichero. El resto del contenido del fichero déjalo en blanco.

```
# ifndef PERSONA_H
# define PERSONA_H
/*
        Codigo
*/

#endif
```

Criterios para una buena descomposición modular:

Criterio	**Descripción**
Descomposición modular	Descomposición del software en subsistemas más sencillos de abordar.
Composición modular	Los elementos conforman otros sistemas, con elementos distintos de subsistemas diferentes
Comprensibilidad modular	Todo lector puede entender cada módulo sin conocer los otros, o examinando sólo unos pocos
Continuidad modular	Un cambio en los requisitos sólo produce cambios en un módulo o, en el peor de los casos, en un número reducido de módulos.
Protección modular	Los procesos realizados en un módulo sólo producen cambios en dicho módulo o, en el peor de los casos, a unos pocos módulos vecinos

Reglas generales para una buena descomposición modular

Regla	**Descripción**
Correspondencia directa	La estructura modular obtenida debe ser compatible con la estructura modular del dominio del problema
Pocas interfaces	Un módulo debe comunicarse con el menor número posible de módulos
Pequeñas interfaces	Los módulos deben intercambiar la menor información posible.
Interfaces explícitas	Las interfaces deben ser obvias a partir de su simple lectura
Ocultación de la información	El diseñador de cada módulo debe seleccionar un sobconjunto de propiedades del módulo como información oficial sobre el módulo, para ponerla a disposición de los autores de los módulos clientes.

Principios para una buena descomposición modular

Principio	**Descripción**
Unidades modulares lingüísticas	Los módulos deben corresponderse con las unidades sintácticas del lenguaje de programación utilizado.
Auto documentación	La información relativa al módulo forma parte del módulo.
Acceso uniforme	Todos los servicios proporcionados por un módulo debe estar disponibles a través de

	una notación uniforme.
Principio abierto-cerrado	Los módulos deben estar a la vez abiertos (facilita una posterior ampliación o modificación) y cerrados (si está disponible para ser usado e integrado en el sistema).
Elección única	Sólo un módulo puede acceder a una estructura de datos y elegir opciones según la estructura de datos utilizada en cada momento.

Describe herencia y los tipos de herencia que existen

La herencia es un proceso por el cuál una clase hereda las propiedades y funcionalidades de otras clases. Pueden ser de 2 tipos:

a) Herencia pública: Todos los miembros de la clase forman parte de las clases derivadas.
b) Herencia privada: Las clases derivadas no son tipos de la clase base.

Describe una variable de tipo const

Cont es un tipo de variable que no cambia su valor durante la ejecución del programa.

¿Qué tipo de herencia deberíamos usar para una hipotética clase Pila de una hipotética clase Lista, de modo que no se permita modificar la clase Pila usando funciones de la clase Lista? ¿Por qué?

La clase Lista dispone de operaciones y datos que permiten realizar cualquier operación sobre la clase Pila, de modo que un conjunto de ellas permiten montar una clase Pila.

Como no queremos que el usuario conozca esta información, de modo que pueda modificar su implementación, decimos que todos los elementos de la clase Lista son derivados de la clase Pila y sólo pueden ser utilizados por la clase Pila (no en las clases derivadas de la clase Pila).

Para poder utilizar esta clase Pila especial sólo se podrá utilizar la interfaz pública de la clase Pila. Por tanto, debemos utilizar una herencia privada.

¿ Qué es una función inline? Codifica un ejemplo.

Una función inline es una función que, dada su brevedad, debe realizarse en la declaración de la función.
Por ejemplo: inline void getValor(int v){ return v; }

¿ Qué es una función friend en C++? ¿ Cuándo debe utilizarse?

Una función friend es aquella que, no perteneciendo a una clase, puede acceder a la parte privada de la clase. Esta función debe usarse en 2 clases:

a) En una clase que utiliza la función.
b) En una clase que contiene en la parte privada los datos a los que se quiere acceder.

Define el código en C++ de una función entera que se llame f() y que reciba como los 3 parámetros siguientes y los muestre por pantalla:

a) I: Tendrá como valor por defecto 0.
b) j: Será un parámetro obligatorio.
c) K: tendrá como valor por defecto 1.

El orden de los parámetros I,j y k deberás fijarlo adecuadamente. Una vez hayas codificado la función enumera las distintas formas en las que puede invocarse la función f() según el número de parámetros que utilicen la llamada.

```
Int f(int j, int i=0, int K=1)
{
    std:: cout << ”Valor de j: ” << j << std::endl;
    std:: cout << “Valor de I: ” << I << std::endl;
```

```
    std::cout << "Valor de K: " << K << std::endl;
    return 1;
}
```

Las formas de invocar la función son las siguientes:

1. Si usamos un sólo parámetro: x = f(a);
2. Si usamos 2 parámetros: x = f(a,b);
3. Si utilizamos 3 parámetros: x = f(a,b,c);

¿ Qué características tienen las clases en las que es imprescindible el uso de constructores de copia?

a) Poseen funciones que se les pasan objetos como argumentos por valor.
b) Las funciones tienen un objeto como valor de retorno.

Definición, tipos y utilidad de las aserciones en C++.

Una aserción es un predicado incluido en el programa que siempre se cumple en dicho punto de flujo del programa.

Las aserciones pueden ser:

1. Precondiciones: Es un aserto al comienzo del código que determina aquello que se espera del conjunto de sentencias a ejecutar.
2. Postcondiciones: Es un aserto al final del código que describe el estado esperado al finalizar la ejecución.

Su utilidad es la siguiente:

1 Especificar programas.
2 Razonar la corrección de los programas.

¿ Qué es la clase raíz de un sistema software orientado a objetos y cuando se dice que un sistema software orientado a objetos está cerrado (cierre de un sistema software)?

La clase raíz es una clase de la cual derivan todas las demás clases del software. Un software está cerrado cuando el sistema contiene todas las clases requeridas por la clase raíz.

Define el concepto de invariante de clase y escribe 2 ejemplos. ¿ Cuándo es correcta una clase?

Una invariante de clase es una propiedad que hace que un objeto esté bien definido, caracterizado por su utilidad y su simpleza.

Algunos ejemplos son los siguientes:
a) class fecha: fecha_correcta
b) class Ruleta: banca_ > 0

Una clase es correcta cuando su implementación es consistente con las aserciones e invariantes.

Escribe todo lo necesario para definir la clase Persona con un constructor que recibe como parámetros opcionales el nombre y la edad, ambos de tipo string, con valores por defecto igual a "XX", y los asigna a variables privadas de dicha clase. Escríbelo todo en un único fichero que se llame persona y escribe todo el código desde el principio del fichero hasta el final: Todas las líneas necesarias para que compile perfectamente sin ningún error y con buena calidad del software.

```
// Fichero persona.h
# ifndef PERSONA_H
# define PERSONA_H
    class Persona
    {
        private:
            string nombre_, edad_, dni_, direccion_, localidad_, provincia_, pais_;
        public:
            Persona(string DNI, string nombre = "XX", string edad = "XX", string
            direccion, string localidad, string provincia, string     pais)
                {
                    DNI = dni_;
                    nombre = nombre_;
                    edad = edad_;
                    direccion = direccion_;
                    localidad = localidad_;
```

```
                provincia =provincia_;
                pais = pais_;
            }
        }
    # endif
```

Escribe una función en C++ de tipo void que se llame intercambia y que reciba como parámetros 2 variables usando referencias e intercambie sus valores.

```
Void intercambia (void &a, void &b)
{
    int aux = (int) a;
    (int) a = (int) b;
    (int) b = aux;
}
```

A lo largo de todo el curso y específicamente en el tema 3 hemos estudiado los múltiples beneficios de la reutilización. Comenta los que recuerdes.

La reutilización del software está tan presente que se tiende a que el desarrollo del software sea una industria basada en componentes.
Sus beneficios son los siguientes:

1. Rapidez del desarrollo.
2. Fiabilidad.
3. Eficiencia.
4. Menor mantenimiento.
5. Mayor consistencia del desarrollo.
6. Mejora de costes.

¿Qué es un patrón de diseño y cuándo se utilizan?

Un patrón de diseño es una solución reutilizable a problemas recurrentes de diseño de software.

Los patrones de diseño se utilizan para resolver problemas de diseño concretos en un determinado contexto.

Patrones de diseño

Nombre	**Tipo**	**Estructura**	**Aplicaciones**	**Consecuencias**	**Ejemplo**
Template Method	Estructural	Un método describe un comportamiento que se concreta en las clases derivadas.	Un mismo proceso se concreta de forma distinta en diferentes clases.	Modificabilidad	Templatemethod.cc
Parameterized Types	Estructural	Se define un nuevo tipo sin especificar los tipos de todos sus componentes	Genericidad	Genericidad	Plantillas de funcion y clase
Iterator	Comportamiento	Permite el acceso a los distintos elementos de un objeto secuencial	Colecciones Agregados Contenedores Listas Vectores	Sencillez de uso. Acceso uniforme para todas las colecciones. Independencia de la representación interna.	list <int>::iterator it;

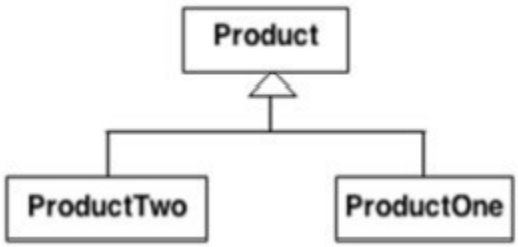
Observer	Comportamiento	Esquema de clases cooperantes que interaccionan entre sí.	Infinidad de aplicaciones con clases cooperantes de dicho tipo. Relación entre modelo-vista en el patrón de diseño MVC	Elementos independientes. Reutilizables por separado. Pueden añadir observadores nuevos.	Sujeto: Una base de datos. Observador: Hoja de cálculo de la base de datos.

Composite	Estructural	Jerarquías de objetos que se comportan de forma parecida con la misma interfaz.	Simplificar interfaces.	Simplificación. Interfaz sencilla. Acceso uniforme.	Menús de usuario. Menú de empleados.
Strategy	Comportamiento.	Define una familia de algoritmos intercambiables.	Habilitable al prever distintos comportamientos futuros. Estructuras de datos complejas que podrán implementarse en el futuro.	Posible mejora en el futuro de eficiencias y rendimiento. Permite estrategias de solución distintas a la original. Facilita ampliaciones	Intercambiar método de ordenación para un conjunto de datos.

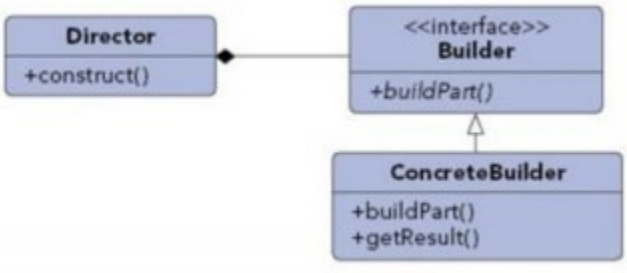
MVC	Estructural	Modelo: Objeto de la representaci ó n. Vista: Representaci ón. Controlador: Modo de racción ante la entrada.	Presente en casi todos los frameworks de desarrollo modernos. Aplicable a cualquier aplicación.	Simplifcaci ón del desarrollo. Separa desarrollos. Varias representacio nes para un mismo desarrollo.	
Builder	Creacional	Permite la creación de una variedad de objetos complejos desde un objeto puente.	Ayuda a construir objetos complejos. Constructor con muchos objetos. No puede construir el objeto en un solo paso. Definir objeto intermedio para ayuda a definir el objeto complejo. Muchos atributos internos. Complejo configurar bien el objeto.	Facilita a los clientes la creación de objetos complejos.	
Factory	Creacional	Creación de instancias de	Crear objetos diferentes de	Sencillez de creación de	

		una clase perteneciente s a una misma familia	una misma familia en una sola y sencilla llamada. Busca facilitar una futura funcionalidad	objetos. Sencillo ampliar el código con nuevas clases de la misma familia.	
Singleton	Creacional	Creación de un sólo elemento accesible desde cualquier punto del sistema.	Configurar aplicación.	Una única instancia. Simplifica uso datos globales.	Configuración global del sistema.

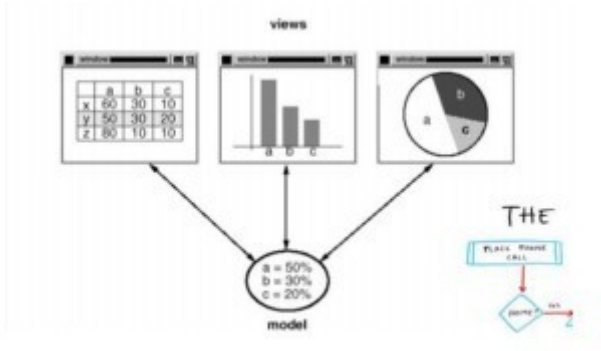
Patrón de diseño factory



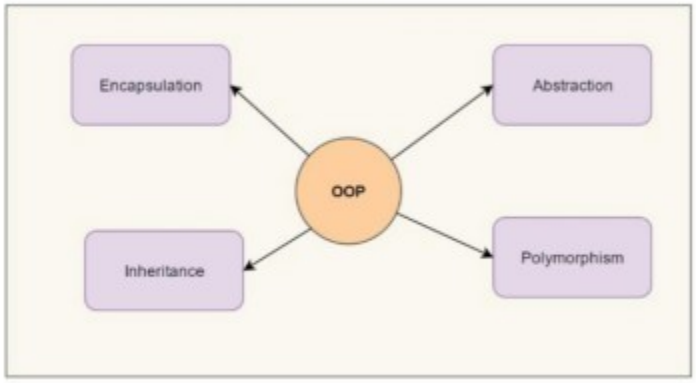
Patrón de diseño Builder



Patrón de diseño MVC



¿ A qué se corresponde la siguiente imagen? Describe sus elementos



La imagen representa los 4 pilares básicos de la programación orientada a objetos (OOP).

Sus elementos son:

1. Encapsulation (Encapsulamiento): La información interna permanece oculta al usuario.
2. Abstraction (Abstracción): Representa las clases y los objetos.
3. Inheritance (Herencia): ES una estructura de reutilización jerárquica específica de la OOP.
4. Polymorphism (Polimorfismo): Se produce cuando se dispone del mismo interfaz sobre objetos de distinto tipo.
5. OOP (Programación Orientada a Objetos): Tipo de programación que utiliza un ensamblado de clases en la construcción de programas informáticos.