TEMA 1: INTRODUCCIÓN A LA POO

POO

Concepto más importante: Programación con tipos abstractos de datos.

Abstracción

- Concepto clave de la POO.
- Desarrollo de software: Manejar la complejidad de un programa ocultando detalles innecesarios en cada etapa del desarrollo del software.

Encapsulamiento

- Significado: La información referente a la definición del tipo y sus operaciones se encuentran en el mismo lugar.
- Ventajas:
  1. Evitar la modificación del estado de un objeto de modo inadecuado.
  2. Protege de operaciones no permitidas sobre los objetos.
  3. Simplifica la comprensión del objeto.
  4. Si se modifican los datos internos del objeto, no afectará a ningún programa que use dicho objeto.

Ocultación de la información

- Significado: La información acerca de la implementación se encuentra oculta al usuario.

Utilidad de encapsulado y ocultación de la información: Impedir al usuario modificar la implementación o acceder a parte de los módulos que no le interesan.

Factores de calidad del software

- **Corrección** (Factor externo): Capacidad del producto de realizar de forma adecuada su función, según los documentos de especificación y requerimientos.
- **Robustez** (Factor externo): Capacidad del producto de comportarse con exactitud, de manera satisfactoria y precisa en todas las situaciones.
- **Eficiencia** (Factor externo e interno): Se trata de conseguir que el programa realice correctamente, y de la mejor forma posible, su función.
- **Integridad** (Factor externo): El producto no debe corromperse por su utilización masiva o una gran acumulación de datos.
- **Facilidad de Uso** (Factor externo): Facilidad al introducir datos, interpretar datos, etc
- **Portabilidad** (Factor externo e interno): Capacidad del producto de ejecutarse en un hardware o sistema operativo diferente.
- **Reutilidad/Reusabilidad** (Factor interno): Capacidad del producto de ser reutilizado en su totalidad o en gran parte por otros productos.
- **Compatibilidad** (Factor externo): Facilidad de los programas de combinarse entre sí.
- **Mantenibilidad** (Factor interno): Capacidad de encontrar y corregir un defecto en el software.
- **Extensibilidad** (Factor interno): Facilidad de adaptar el producto a cambios en la especificación de requisitos.
- **Testable** (Factor interno): Habilidad de poder validarse el software.
- **Seguridad** (Factor externo): Capacidad del producto de proteger sus componentes de usos no autorizados y situaciones de pérdida de información.

- **Accesibilidad** (Factor externo): Acceso a la información sin limitación alguna por razones de deficiencia, incapacidad o minusvalía.
- **Oportunidad** (Factor externo): Capacidad de un sistema de ser lanzado al mercado.
- **Economía** (Factor externo): Costes del producto.

TEMA 2: PROGRAMACIÓN CON TAD

Concepto: División del sistema en partes diferenciadas y definir sus interfaces. Se busca la claridad, la reducción de costes y la reutilización.
Criterios para una buena descomposición modular

1. **Descomposición modular**: Descomposición del software en subsistemas más sencillos de abordar.
2. **Composición modular**: Los elementos conforman otros sistemas, con elementos distintos distintos de subsistemas distintos.
3. **Comprensibilidad modular**: Todo lector puede entender cada módulo sin conocer los otros, o examinando sólo unos pocos.
4. **Continuidad modular**: Un cambio en los requisitos sólo provoca cambios en un módulo o en un número reducido de ellos.
5. **Protección modular**: Los procesos realizados en un módulo sólo afectan a dicho módulo o a unos pocos módulos vecinos.

Reglas generales para una buena descomposición modular

1. **Correspondencia directa**: La estructura modular obtenida debe ser compatible con la estructura modular del dominio del problema.
2. **Pocas interfaces**: Un módulo debe comunicarse con el menor número posible de módulos.

3. **Pequeñas interfaces**: Los módulos deben intercambiar la menor información posible.
4. **Interfaces explícitas**: Las interfaces deben ser obvias a partir de su simple lectura.
5. **Ocultación de la información**: El diseñador de cada módulo debe seleccionar un subconjunto de propiedades del módulo como información oficial sobre el módulo, para ponerla a disposición de los autores de módulos clientes.

Principios para una buena descomposición modular

1. **Unidades modulares lingüísticas:** Los módulos deben corresponderse con las unidades sintácticas del lenguaje de programación utilizado.
2. **Auto documentación:** La información relativa al módulo forme parte del propio módulo.
3. **Acceso uniforme:** Todos los servicios ofrecidos por un módulo deben estar disponibles a través de una notación uniforme.
4. **Principio abierto-cerrado:** Los módulos deben ser a la vez abiertos (para facilitar la posterior ampliación, modificación...) y cerrados (si está disponible para ser usado e integrado en el sistema).
5. **Elección única:** Sólo un módulo puede acceder a una estructura de datos y elegir opciones según la estructura de datos utilizada en cada momento.

Excepciones: Mecanismo de los lenguajes de programación actuales. Se utilizar para notificar situaciones de error o situaciones excepciones al usuario.

Operaciones con TAD

1. **Constructores:** Inicialización del objeto y sus datos internos.
2. **Observadores:** Acceso de lectura a una característica del objeto.
3. **Modificadores:** Acceso de escritura a una característica del objeto.

4. **Destructores:** Llevan al objeto a su estado inicial descartando posibles efectos laterales imprevistos.

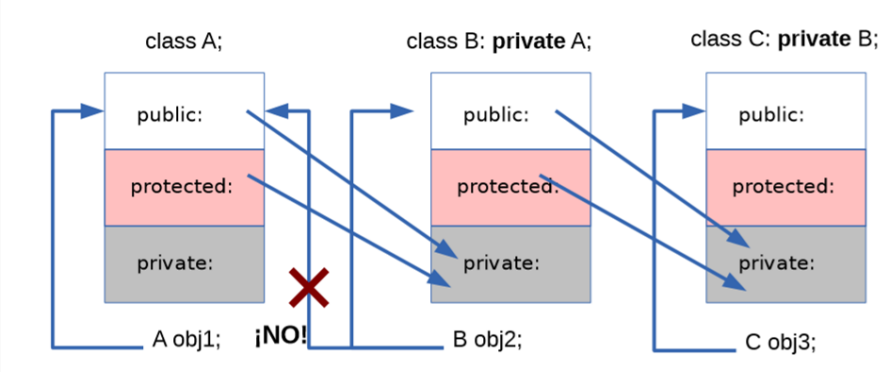
Conceptos del lenguaje C++

- **g++:** Es GCC (GNU compiler collection).
- **Const:** Sirve para declarar una constante con atributos invariables durante la ejecución.
- **Parámetros por defecto:** Lista de parámetros que necesita una función.
- **Función inline:** Función de optimización de funciones breves. Debe hacerse en la declaración de la clase.
  - Ejemplo: inline void getDado(int valor) { dado1_ = valor;}
- **Referencias:** Son alias que pueden darse a un objeto o una variable.
  - Al definirse siempre deben indicar la variable de referencia.
  - Utilidades:
    - Da eficiencia y rapidez.
    - Evita que a cada nuevo objeto declarado se le reserve un espacio de memoria y se almacene una dirección para dicho objeto.
  - Paso de objetos como referencia permite ahorrar memoria y tiempo de ejecución.
  - No se realizan copias al pasar parámetros como referencia.
  - Ejemplo de una función de nombre <asignar>, de tipo booleana, con parámetros por referencia un entero y un objeto de clase persona:
    - bool asignar(int &a, Persona &P);
- **Clase:** Es un objeto con atributos y funcionalidades.
  - Características de las clases con imprescindible uso de constructores de copia:
    - Poseen funciones que se le pasan objetos como argumentos por valor.
    - Las funciones tienen un objeto como valor de retorno.

- ↘ Clase raíz: Es una clase de la cual derivan todas las demás clases del sistema software.
- ↘ Clase correcta: Cuando su implementación es consistente con las aserciones e invariantes.
- ↘ Invariante de clase: Es una propiedad que hace que un objeto esté bien definido, caracterizado por su utilidad y su simpleza.
  - ✦ Ejemplo: class fecha: fecha_correcta
  - ✦ Ejemplo: class Ruleta: banca>0

⑩ **Herencia:** Proceso por el cual una clase hereda las propiedades y funcionalidades de otras clases.

- ↘ Pública: Todos los miembros de la clase base pasan a formar parte de las clases derivadas.
- ↘ Privada: Las clases derivadas no son tipos de la clase base.



- ↘ **¿Qué tipo de herencia deberíamos usar para derivar una clase B de una clase A, de modo que no se permita modificar la clase B usando funciones de la clase A? ¿Por qué?**
  - ✦ La clase a dispone de operaciones y datos que permiten realizar cualquier operación sobre la clase A, de modo que un grupo de ellas permiten montar una clase B. Por tanto, la clase B puede construirse por medio de la clase A.
  - ✦ Como no queremos que el usuario conozca esta información y puede modificar la implementación, decimos que todos los elementos de la

clase A son privados en B y sólo pueden ser usados en la clase B (no en las clases derivadas de B).

- ✦ Para poder usar esta clase B especial, sólo se podrá utilizar el interfaz público de la clase B.
- ✦ Por tanto, deberíamos usar una herencia privada.

- ❿ **STL** (Standard Template Library)**:** Contiene funciones, algoritmos, iteradores y contenedores.
- ❿ **Funciones friend:** Es una función que puede acceder a la parte privada de una clase, sin ser miembro de dicha clase.
  - ↘ Debe declararse en las 2 clases:
    - ✦ Clase que utiliza la función.
    - ✦ Clase que contiene en la parte privada los datos a los que se quiere acceder.
- ❿ **Aserciones:**
  - ↘ Definición: Predicado incluido en el programa que siempre se cumple en dicho punto de flujo del programa.
  - ↘ Tipos de aserciones:
    - ✦ Precondición: Es un aserto al comienzo del código. Determina qué se espera del conjunto de sentencias que siguen a ser ejecutadas.
    - ✦ Postcondición: Es un aserto al final del código. Describe el estado que se espera alcanzar al final de la ejecución.
  - ↘ Utilidad:
    - ✦ Especificar programas.
    - ✦ Razonar la corrección de los programas.

  - ↘ Ejemplo: Función Insertar()
    - ✦ Llama a la función Abortar() si el resultado es false.
    - ✦ Muestra el nombre y la línea del archivo fuente.
    - ✦ #define ERROR // evita la ejecución de la función Insertar()

Tema 3: Patrones de diseño

Concepto: Soluciones reutilizables a problemas recurrentes de diseño de software.

- Está compuesto por objetos, clases y las relaciones entre ellos.
- Se utilizan para resolver problemas de diseño concretos en un determinado contexto.
- Clasificación:
  1. Comportamiento: Interacciones de objetos, algoritmos…
  2. Estructural: Estructuras de clase entre muchos objetos dispares.
  3. Creacional: Compuesto de objetos instantáneos.

- Template Method
  - Ejemplo: - template-method.cc
  - Tipo: Estructural.
  - Estructura: Un método describe un comportamiento que se concreta en las clases derivadas.
  - Aplicaciones: Un mismo proceso se concreta de forma diferente en distintas clases.
  - Consecuencias: Modificabilidad.
- Parameterized types
  - Ejemplo: Plantillas de función y de clase en C++
  - Tipo: Estructural.
  - Estructura: Se define un nuevo tipo sin especificar los tipos de todos sus componentes.
  - Aplicaciones: Genericidad.
  - Consecuencias: Genericidad.
- Iterator
  - Ejemplo: list<int>::iterator it;

- ↘ Estructura: Permite el acceso a los distintos elementos de un objeto secuencial.
- ↘ Tipo: Comportamiento.
- ↘ Aplicaciones: Colecciones, agregados, contenedores, listas, vectores...
- ↘ Consecuencias:
  - ✦ Sencillez de uso.
  - ✦ Acceso uniforme para todas las colecciones.
  - ✦ Independencia de la representación interna.

❿ Observer

- ↘ Tipo: Comportamiento.
- ↘ Estructura: Es un esquema de clases cooperantes que interaccionan entre sí. Hay un sujeto con datos y observadores que se nutren de dichos datos.
- ↘ Ejemplo:
  - ✦ Sujeto: Una base de datos.
  - ✦ Observador: Hoja de cálculo de la base de datos.
- ↘ Aplicaciones:
  - ✦ Infinidad de aplicaciones tienen clases cooperantes de este tipo.
  - ✦ Relación entre modelo y vista en el patrón de diseño MVC.
- ↘ Consecuencias:
  - ✦ Ambos elementos son independientes.
  - ✦ Se pueden reutilizar por separado.
  - ✦ Se pueden añadir observadores nuevos.

❿ Composite:

- ↘ Tipo: Estructural.

- ↘ Estructura: Son jerarquías de objetos que se comportan de forma parecida con la misma interfaz.
- ↘ Aplicaciones: Simplificar interfaces.
- ↘ Consecuencias:
  - ✦ Simplificación.
  - ✦ Interfaz sencilla.
  - ✦ Acceso uniforme.
- ↘ Ejemplos: Menús de usuario, empleados de una empresa…

❿ Strategy:

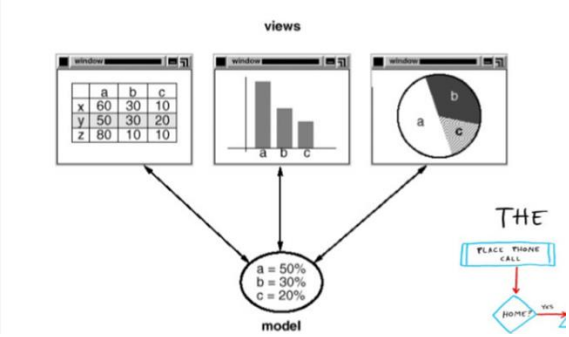
- ↘ Tipo: Comportamiento.
- ↘ Estructura: Define una familia de algoritmos intercambiables.
- ↘ Aplicaciones:
  - ✦ Podemos habilitarlo al prever distintos comportamientos en un futuro.
  - ✦ Estructuras de datos complejas que podrán implementarse en un futuro de otras formas.
- ↘ Consecuencias:
  - ✦ Posibilidad de mejorar eficiencias y rendimientos en el futuro.
  - ✦ Permitir otras estrategias de solución distintas a la propuesta inicial.
  - ✦ Facilitar ampliaciones.
- ↘ Ejemplo: Intercambiar el método de ordenación para un conjunto de datos.

❿ MVC

- ↘ Tipo: Estructural.
- ↘ Estructura:
  - ✦ Modelo: Objeto de la aplicación.
  - ✦ Vista: Representación.
  - ✦ Controlador: Define el modo de reacción ante la entrada.
- ↘ Aplicaciones:
  - ✦ Presente en casi todos los frameworks de desarrollo modernos.
  - ✦ Se puede aplicar a cualquier aplicación.
- ↘ Consecuencias:

- ✦ Simplificación del desarrollo.
- ✦ Separar desarrollos.
- ✦ Varias presentaciones para un mismo modelo.

↘ Ejemplo:



❿ Builder:

↘ Tipo: Creacional.

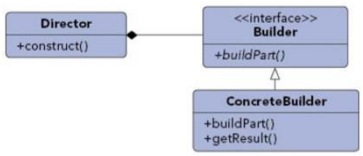
↘ Estructura: Permite la creación de una variedad de objetos complejos desde un objeto puente.

↘ Aplicaciones:

- ✦ Ayuda en la construcción de objetos complejos.
- ✦ Muchos atributos internos.
- ✦ Constructor con muchos parámetros, que deben cumplir ciertas condiciones entre ellos.
- ✦ Resulta complejo configurar bien el objeto.
- ✦ No se puede construir el objeto en un solo paso.
- ✦ Definir objeto intermedio que ayude con la definición del objeto complejo.

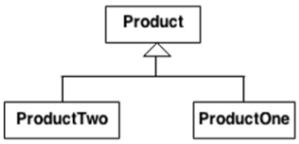
↘ Consecuencias: Facilita a los clientes la creación de objetos complejos.

↘ Ejemplo:



❿ Factory:

- ↘ Tipo: Creacional.
- ↘ Estructura: Permite la creación de instancias de una clase pertenecientes a una misma familia.
- ↘ Aplicaciones:
  - ✦ Crear objetos diferentes de una misma familia en una sola y sencilla llamada.
  - ✦ Se busca facilitar añadir futura funcionalidad.
- ↘ Consecuencias:
  - ✦ Sencillez de creación de objetos.
  - ✦ Sencillez de ampliación del código con nuevas clases de la misma familia.
- ↘ Ejemplo:



❿ Singleton:

- ↘ Tipo: Creacional.
- ↘ Estructura: Se basa en la creación de únicamente un elemento accesible desde cualquier punto del sistema.
- ↘ Aplicaciones:
  - ✦ Configurar una aplicación.
  - ✦ Cuando se usan variables globales pueden meterse en una struct global, pero la clase es más flexible.
- ↘ Consecuencias:
  - ✦ Garantiza una única instancia.
  - ✦ Simplifica el uso de datos globales.

↘ Ejemplo: Configuración global del sistema.

Sistema software cerrado (cierre de sistema software): Sistema que contiene todas las clases que necesita la clase raíz.

Tema 4: Reutilización

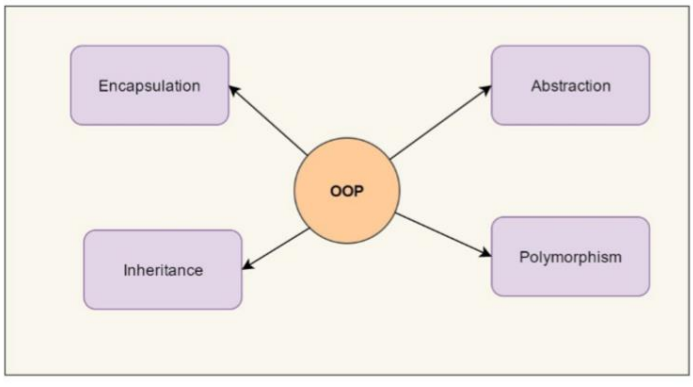
Utilidad: Está tan presente la reutilización de software que se tiende hacia que el desarrollo de software sea una industria basada en componentes.

Beneficios:

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

Tema 5: Tecnología OO

Los 4 pilares de la OOP



Four Pillars of Object Oriented Programming

- Abstraction (Abstracción): Las clases y los objetos.
- Encapsulation (Encapsulamiento): Ocultar la visión interna del objeto.

- Inheritance (Herencia): Estructura de reutilización jerárquica específica de la POO.
- Polymorphism (Polimorfismo): Ocurre cuando se dispone del mismo interfaz sobre objetos de distinto tipo.
- OOP (Programación Orientada a Objetos): Programación que utiliza un ensamblado de clases en la construcción de programas informáticos.