

Módulo 5 – POO buenas prácticas de programación

Índice

Introducción

Breve reseña	3
Introducción	4
Nociones Básicas UML.....	6

Programación Orientada a Objetos

Objetos	10
Clases	13
Mensajes. Método. Variables e instancias.....	15
Constructores, Destructores y Garbage Collector.....	20
Relación entre clases y objetos	22

Fundamentos de POO

Fundamentos de POO	25
Abstracción	28
Encapsulamiento	31
Modularidad	32
Herencia	33
Polimorfismo	36
Tipificación, Concurrencia y Persistencia	38
Conclusiones POO	40

Análisis y Diseño Orientado a Objetos

Diseño de Software	42
¿Qué es ADOO?	44
Conceptos Importantes ADOO	46
Modelado de objetos – UML	48
Conceptos de modelado específico	50
Diagramas Estructurales	52

Diagramas de Comportamiento	55
Glosario - Recomendaciones UML	57
Design Patterns	
¿Qué son los Patrones de Diseño?	59
Design Patterns – Clasificación	60

Breve reseña

La velocidad con la que avanza la tecnología del hardware en los últimos años es muy grande, con lo que se ha logrado tener computadoras más poderosas, baratas y compactas. Pero el Software no ha tenido el mismo comportamiento, ya que, al desarrollar aplicaciones, es frecuente que se excedan los tiempos de entrega, así como los costos de los sistemas de información (tanto de desarrollo, como de mantenimiento), además de ser poco flexibles. Se han creado diferentes herramientas de ayuda al desarrollo, para lograr aumentar la productividad en el Software:

1. Técnicas como las del Diseño Estructurado y el desarrollo descendente (*topdown*)
2. Herramientas de Ingeniería de Software asistida por computadora conocida como CASE
3. Desarrollo de lenguajes de programación más poderosos como los lenguajes de cuarta generación ([4GL](#)) y los orientados a objetos (POO).
4. Diversas herramientas como gestión de proyectos, gestión de la configuración, ayuda en las pruebas, bibliotecas de clases de objetos, entre otras.

Haremos más énfasis en los lenguajes de programación, sobre todo en la Programación Orientada a Objetos (POO). Un lenguaje en términos generales, se puede entender como “... *sistemas de símbolos y signos convencionales que son aceptados y usados individual y socialmente, con el fin de comunicar o expresar sentimientos, ideas, conocimientos, etc., por ejemplo, el lenguaje natural o articulado, el corporal, el artificial o formal, los sistemas de señalamiento, el arte, entre muchos otros tipos...*” Existen también los lenguajes artificiales, que entre sus características, tenemos el que no es ambiguo y es universal, entre los que se encuentran los de las Matemáticas y los de Programación. En los lenguajes de programación (conjunto de sintaxis y reglas semánticas con el fin de comunicar ideas sobre algoritmos entre las computadoras y las personas), existen diferentes clasificaciones, una de ellas es la que a continuación se muestra:

- **Programación Imperativa**, donde el programa es una serie de pasos, realizando en cada uno de ellos un cálculo (como ejemplos están: Cobol, Fortran entre otros).
- **Programación Funcional**, el programa es un conjunto de funciones matemáticas que se combinan (como Lisp, Scheme, etc.).
- **Programación Lógica** (también conocida como declarativa), aquí el programa es una colección de declaraciones lógicas (ejemplo Prolog).
- **Programación Concurrente**, la programación consiste en un grupo de procesos corporativos, que llegan a compartir información ocasionalmente entre ellos (ejemplos LINDA y Fortran de alto rendimiento HPF 1995)
- **Programación guiada por eventos**, el programa consiste en un ciclo continuo que va a responder a los eventos generados aleatoriamente (orden no predecible), ya que dichos eventos son originados a partir de acciones del usuario en la pantalla (ejemplos JAVA y Visual Basic)




- **Programación Orientada a Objetos (POO)**, el programa está compuesto por varios objetos que interactúan entre ellos a través de mensajes, los cuales hacen que cambien su estado (ejemplos C++, Eiffel y JAVA)

Existen muchas formas de clasificar los paradigmas de programación incluyendo más o menos programas y niveles, por ejemplo:



Introducción

La Programación Orientada a Objetos (abreviada de ahora en más como POO) es un conjunto de reglas y principios de programación (o sea, un paradigma de programación) que busca representar las entidades u objetos del dominio (o enunciado) del problema dentro de un programa, en la forma más natural posible.

	<i>Es un paradigma de programación que usa objetos y sus interacciones, para diseñar aplicaciones y programas de computadora.</i>
	<i>Es una forma especial de programar, más cercana a como expresaríamos las cosas en la vida real.</i>
	<i>Se basa en la idea natural de la existencia de un mundo lleno de objetos, de modo que la resolución del problema se realiza en términos de objetos.</i>

En el paradigma de programación tradicional o estructurado, que es el que se usaba anteriormente, el programador busca identificar los procesos (en forma de subproblemas) que aplicados sobre los datos permiten obtener los resultados buscados. Esta forma de proceder no es en absoluto incorrecta: la estrategia de descomponer un problema en subproblemas es una técnica elemental de resolución de problemas que los programadores orientados a objetos siguen usando dentro del nuevo paradigma. Pero entonces, ¿qué es lo nuevo del paradigma de la POO?

La sola orientación a la descomposición en subproblemas no alcanza cuando el sistema es muy complejo. Se vuelve difícil de visualizar su estructura general y se hace complicado realizar hasta las más pequeñas modificaciones sin que estas reboten en la lógica de un número elevado de otras rutinas. Finalmente, se torna una tarea casi imposible replantear el sistema para agregar nuevas funcionalidades complejas que permitan que ese sistema simplemente siga siendo útil frente a continuas nuevas demandas.

La POO significó una nueva visión en la forma de programar, buscando aportar claridad y naturalidad en la manera en que se plantea un programa. Ahora, el objetivo primario no es identificar procesos sino identificar actores: las entidades u objetos que aparecen en el escenario o dominio del problema, tales que esos objetos tienen no sólo datos asociados sino también algún comportamiento que son capaces de ejecutar.

Pensemos en un objeto como en un robot virtual: el programa tendrá muchos robots virtuales (objetos de software) que serán capaces de realizar eficiente y prolijamente ciertas tareas en las que serán expertos, e interactuando con otros robots virtuales (objetos...) serán capaces de resolver el problema que el programador esté planteando.

En síntesis, podemos afirmar que, **“La programación Orientada a Objetos es una metodología que basa la estructura de los programas en torno a los objetos”**.

Los lenguajes de POO ofrecen medios y herramientas para describir los objetos manipulados por un programa. Más que describir cada objeto individualmente, estos lenguajes proveen una construcción (Clase) que describe a un conjunto de objetos que poseen las mismas propiedades y comportamientos.

Algunas ventajas del paradigma orientado a objetos son:

- . Es una forma más natural de modelar los problemas.
- . Permite manejar mejor la complejidad.
- . Facilita el mantenimiento y extensión de los sistemas.
- . Es más adecuado para la construcción de entornos GUI.
- . Fomenta el re-uso, con gran impacto sobre la productividad y la confiabilidad

Nociones Básicas UML

Antes de comenzar a entender los aspectos más importantes de POO, vamos a conocer un recurso que nos será útil durante todo el recorrido.

UML (Unified Modeling Language)

El lenguaje unificado de modelado (UML), sostiene la idea de que una imagen vale más que mil palabras. Por ello, nos permite crear y generar diagramas para forjar un lenguaje visual común en el complejo mundo del análisis y desarrollo de software. Estos diagramas también son comprensibles para los usuarios y quien desee entender un sistema.

Es importante aclarar que UML no es un lenguaje de programación, pero existen herramientas que se pueden usar para generar código en diversos lenguajes usando los diagramas UML. Además, UML guarda una relación directa con el análisis y el diseño orientados a objetos.

“No es una metodología, sino una notación para desarrollar modelos (...) UML es el estándar para visualizar, especificar, construir y documentar los artefactos de un sistema de software”

Ahora veamos algunos conceptos de modelado especificados por UML. El desarrollo de sistemas se centra en tres modelos generales de sistemas diferentes:

1. **Funcionales:** Se trata de diagramas de casos de uso que describen la funcionalidad del sistema desde el punto de vista del usuario.
2. **De objetos:** Se trata de diagramas de clases que describen la estructura del sistema en términos de objetos, atributos, asociaciones y operaciones.
3. **Dinámicos:** Los diagramas de interacción, los diagramas de máquina de estados y los diagramas de actividades se usan para describir el comportamiento interno del sistema.

Un software que nos permite realizar Diagramas en UML es Lucidchart, y puedes registrarte en el siguiente link: <https://www.lucidchart.com>

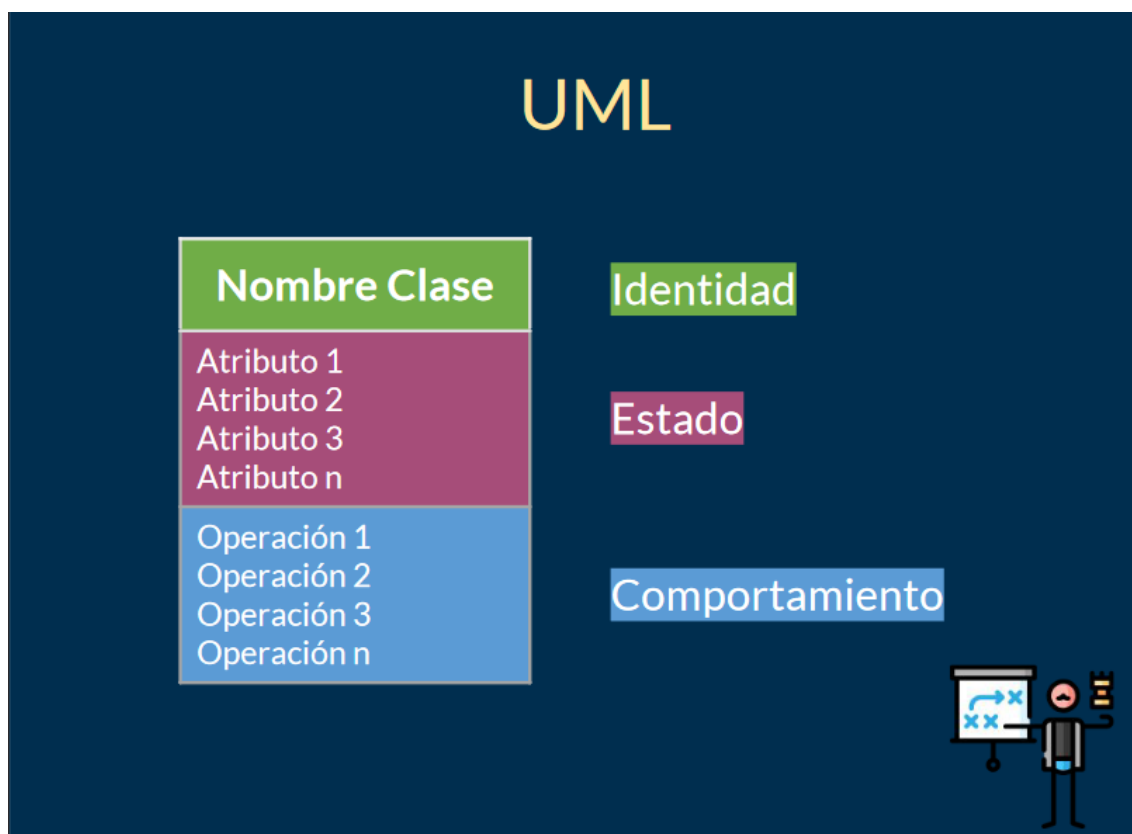
Más adelante visualizaremos con más detalle todas las características de UML. Por ahora, vamos a conocer 2 tipos de diagramas para entender la POO (los Diagramas de Clases, y los Diagramas de Secuencias)

Diagrama de Clases

El diagrama de clases es el diagrama UML más comúnmente usado, y la base principal de toda solución orientada a objetos. En el se presentan las clases dentro de un sistema, atributos y operaciones, y la relación entre cada clase. Las clases se agrupan para crear diagramas de clases al crear diagramas de sistemas grandes.

Los componentes básicos son:

1. **Sección superior:** Contiene el nombre de la clase. Esta sección siempre es necesaria, ya sea que estés hablando del clasificador o de un objeto.
2. **Sección central:** Contiene los atributos de la clase. Esta sección se usa para describir cualidades de la clase.
3. **Sección inferior:** Incluye operaciones de clases (métodos). Estas se organizan en un formato de lista y cada operación requiere su propia línea. Las operaciones describen cómo una clase puede interactuar con los datos.



Todas las clases pueden tener diferente Visibilidad, es decir diferentes niveles de acceso: Los más comunes son los Privados (-), y Protegidos (#)

Algunas clases pueden ser abstractas, porque en nuestro sistema no se crearía una instancia de ella. El nombre de una clase abstracta puede colocarse en Cursiva, o entre: <<nombre>>.

Además, hay diferentes relaciones entre clases:

1. **Herencia:** Es el proceso en el que una subclase o clase derivada recibe la funcionalidad de una superclase o clase principal, también se conoce como "generalización". Se simboliza mediante una línea de conexión recta con una punta de flecha cerrada que señala a la superclase.
2. **Asociación:** Es la relación predeterminada entre dos clases. Ambas clases están conscientes una de la otra y de la relación que tienen entre sí. Esta asociación se representa mediante una línea recta entre dos clases.
3. **Agregación:** Relación en donde un objeto derivado puede existir sin su objeto principal.
4. **Composición:** Relación en donde un objeto derivado no puede existir sin su objeto principal.
5. **Multiplidad:** Permite definir las relaciones numéricas (por ejemplo, 0..1, n, 0..*, 1..*, m..n)

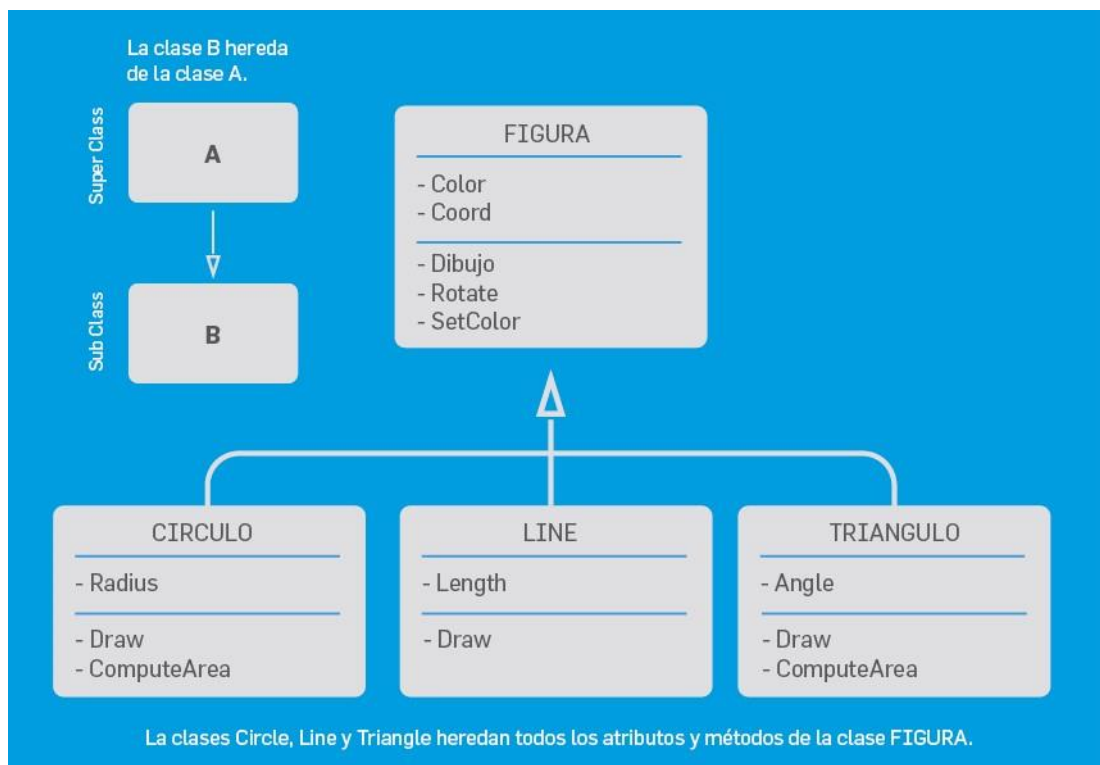


Diagrama de objetos

El diagrama de objetos muestra la relación entre objetos por medio de ejemplos del mundo real e ilustra cómo se verá un sistema en un momento dado.

Representa a un objeto instanciado (creado en tiempo de ejecución) dentro de una clase, estableciendo valores a los atributos. Dado que los datos están disponibles dentro de los objetos, estos pueden usarse para clarificar relaciones entre objetos.

Los componentes básicos son:

Objetos: Los objetos son instancias de una clase. Por ejemplo, si "Auto" es una clase, un 208 compacto de Peugeot es un objeto de una clase.

1. **Títulos de clase:** Los títulos de clases son los atributos específicos de una clase dada. Se pueden listar títulos de clases como elementos en el objeto o incluso en las propiedades del propio objeto (como el color).
2. **Atributos:** Los atributos de clases se representan por medio de un rectángulo con dos pestañas que indica un elemento de software.
3. **Enlaces:** Los enlaces son líneas que conectan dos figuras de un diagrama de objetos entre sí (debe nombrarse la interacción).

También, al igual que en el diagrama de clases, se puede agregar una tercera sección donde estén indicados los métodos u operaciones.

"El diagrama de objetos también es conocido como diagrama de instancias".

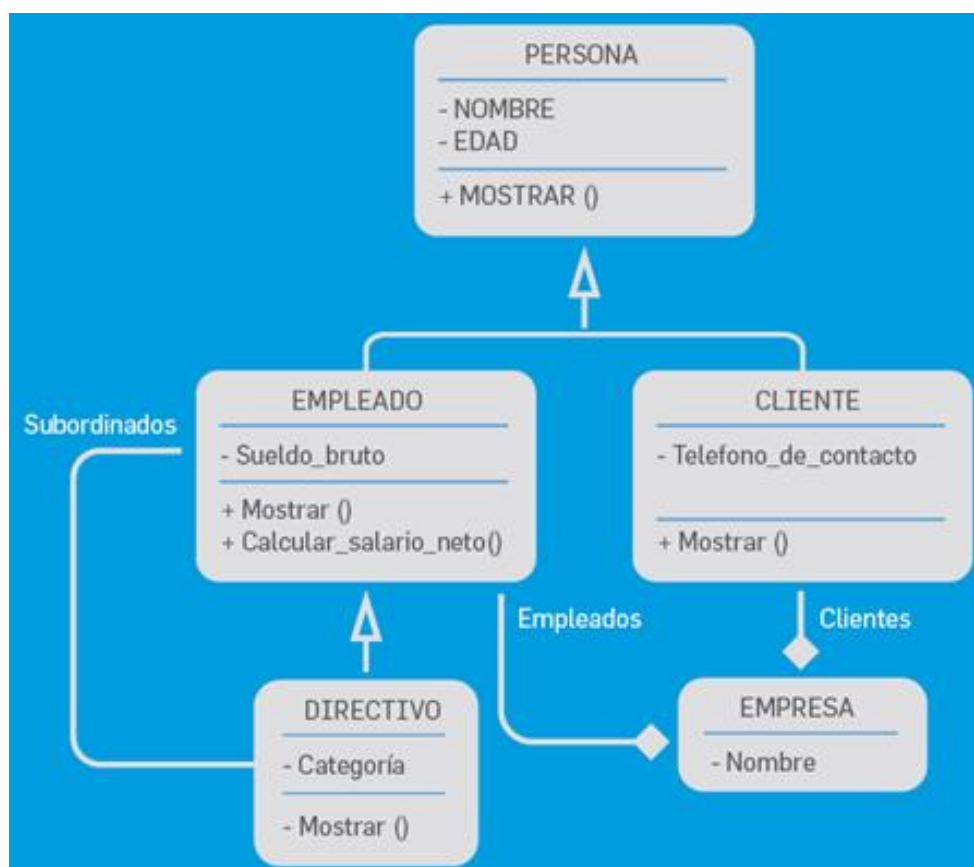


Diagrama de secuencias

El diagrama de secuencia es un esquema conceptual que permite representar el comportamiento de un sistema. Muestra cómo los objetos interactúan entre sí y el orden de la ocurrencia. Representan interacciones para un escenario concreto.

Los componentes básicos son:

1. Objetos: se representa del modo usual (como en el diagrama de objetos)
2. Líneas de vida: representa un participante individual en un diagrama de secuencia
3. Mensaje: el mensaje que va de un objeto a otro pasa de la línea de vida de un objeto al de otro. Hay diferentes tipos de mensajes como: simple, sincrónico, asincrónico.
4. Dimensiones: Horizontal, que es la disposición de los objetos, y vertical que muestra el paso del tiempo.

Objetos

Ya presentamos el funcionamiento básico de UML que nos ayudará a representar mejor algunos conceptos básicos de POO. Antes de desarrollar con profundidad cada uno de los conceptos, veamos algunas características de los Lenguajes Orientados a Objetos.

Según [Alan Kay](#) (1993), las características son seis:

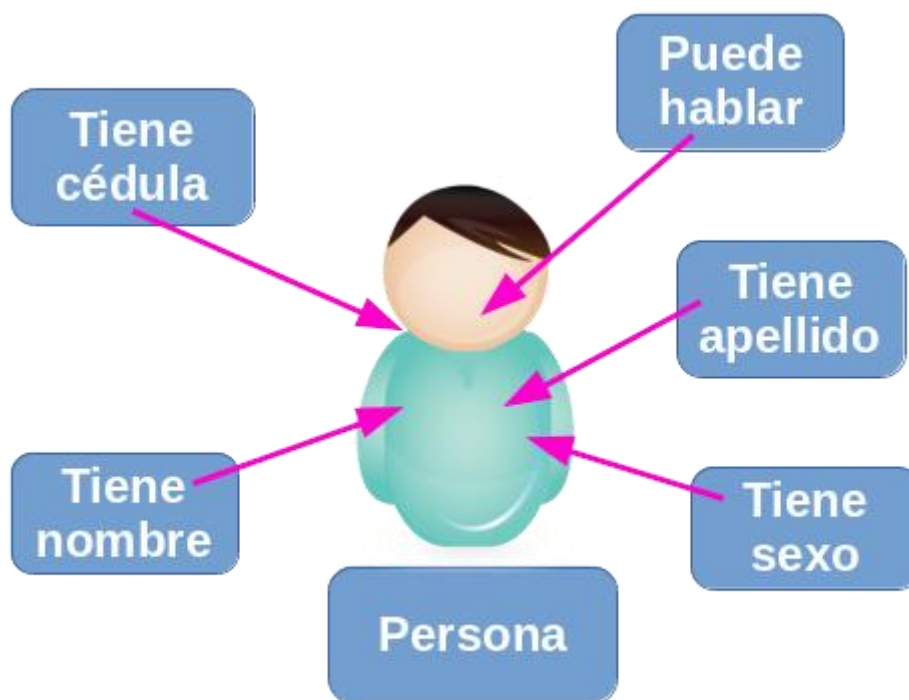
1. Todo es un objeto.
2. Cada objeto es construido a partir de otros objetos.
3. Todo objeto es instancia de una clase.
4. Todos los objetos de la misma clase pueden recibir los mismos mensajes (realizar las mismas acciones). La clase es el lugar donde se define el comportamiento de los objetos y su estructura interna.
5. Las clases se organizan en una estructura arbórea de raíz única, llamada jerarquía de herencia. Ejemplo: puesto que un círculo es una forma, un círculo siempre aceptará todos los mensajes destinados a una forma.
6. Un programa es un conjunto de objetos que se comunican mediante el paso de mensajes.

Ahora analicemos en profundidad cada término mencionado con más detalle...

Objetos

Los Objetos, se pueden definir como las unidades básicas de construcción, para la conceptualización, diseño o programación. Son instancias agrupadas en clases con características en común que son los atributos y procedimientos, conocidos como operaciones o métodos. También se puede decir, que un objeto es una abstracción encapsulada genérica de datos y los procedimientos para manipularlos.

"Un objeto es una cosa o entidad, que tiene atributos (propiedades) y de formas de operar sobre ellos que se conocen como métodos."



De una forma más simple se puede decir que un objeto es un ente que tiene características y comportamiento. Los objetos pueden modelar diferentes cosas como; dispositivos, roles, organizaciones, sucesos, ventanas (de Windows), iconos, etc.

Las características generales de los objetos son:

- Se identifican por un nombre o identificador único que lo diferencia de los demás.
- Poseen estados.
- Poseen un conjunto de métodos.
- Poseen un conjunto de atributos.
- Soportan el encapsulamiento.
- Tienen un tiempo de vida.
- Son instancias de una clase.

Organización de los objetos

Los objetos forman siempre una organización jerárquica, existiendo varios tipos de jerarquía, por ejemplo:

- Simples: cuando su estructura es representada por medio de un árbol (estructura de datos).
- Compleja: cualquier otra diferente a la de árbol.

Sea cual fuere la estructura se tienen en ella los siguientes tres niveles de objetos:

1. **La raíz de la jerarquía:** Es un objeto único, está en el nivel más alto de la estructura y se le conoce como objeto Padre, Raíz o Entidad.
2. **Los objetos intermedios:** Son los que descienden directamente de la raíz y que a su vez tienen descendientes (tienen ascendencia y descendencia) y representan conjuntos de objetos, que pueden llegar a ser muy generales o especializados, de acuerdo con los requerimientos de la aplicación.
3. **Los objetos terminales:** Son todos aquellos que tienen ascendencia, pero que no tienen descendencia.

Atributos de los objetos

Los atributos son las características del objeto (qué tiene, de qué consta), que se definen a través de tipos de datos o variables, los cuales se dividen en:

- **Tipos de Datos Primitivos (TDP):** nos sirven para representar tipos de datos como números enteros, caracteres, números reales, booleanos (v/f), etcétera. Una variable de tipo primitivo nos permite almacenar en ella un tipo primitivo como por ejemplo un valor numérico.
- **Tipos de Datos por Referencia:** indican que vamos a trabajar con instancias de clases, no con tipos primitivos. De esta manera, una variable de tipo referencia establece una conexión hacia un objeto, y a través de esta conexión podemos acceder a sus métodos y atributos.

Objeto: instancia de una clase



Clase Persona		Objeto Persona		Objeto Persona 2		Objeto Persona 3	
Atributos	nombre	Atributos	nombre = Leo	Atributos	nombre = Lucas	Atributos	nombre = Blaise
	edad		edad = 32		edad = 30		edad = 34
	colorRemera		colorRemera = Azul		colorRemera = Rojo		colorRemera = Amarilla
	colorPantalon		colorPantalon = Azul		colorPantalon = Negro		colorPantalon = Blanco
	colorBotines		colorBotines = Negro		colorBotines = Verde		colorBotines = Negro
	colorPelo		colorPelo = Negro		colorPelo = Rubio		colorPelo = Blanco
	colorPiel		colorPiel = Clara		colorPiel = Clara		colorPiel = Oscura

Clase Molde o Plantilla	Objeto 1	Objeto 2	Objeto 3

Más adelante desarrollaremos con mayor profundidad los tipos de variables y datos, viendo una clasificación más detallada. Se toma la convención de que, al utilizar estos atributos, se utilicen siempre con letras minúsculas y solo si se trata de más de una palabra, entonces será con mayúscula cada primera letra de cada palabra adicional y siempre juntas, sin espacios en blanco, por ejemplo: *colorDelPelo*

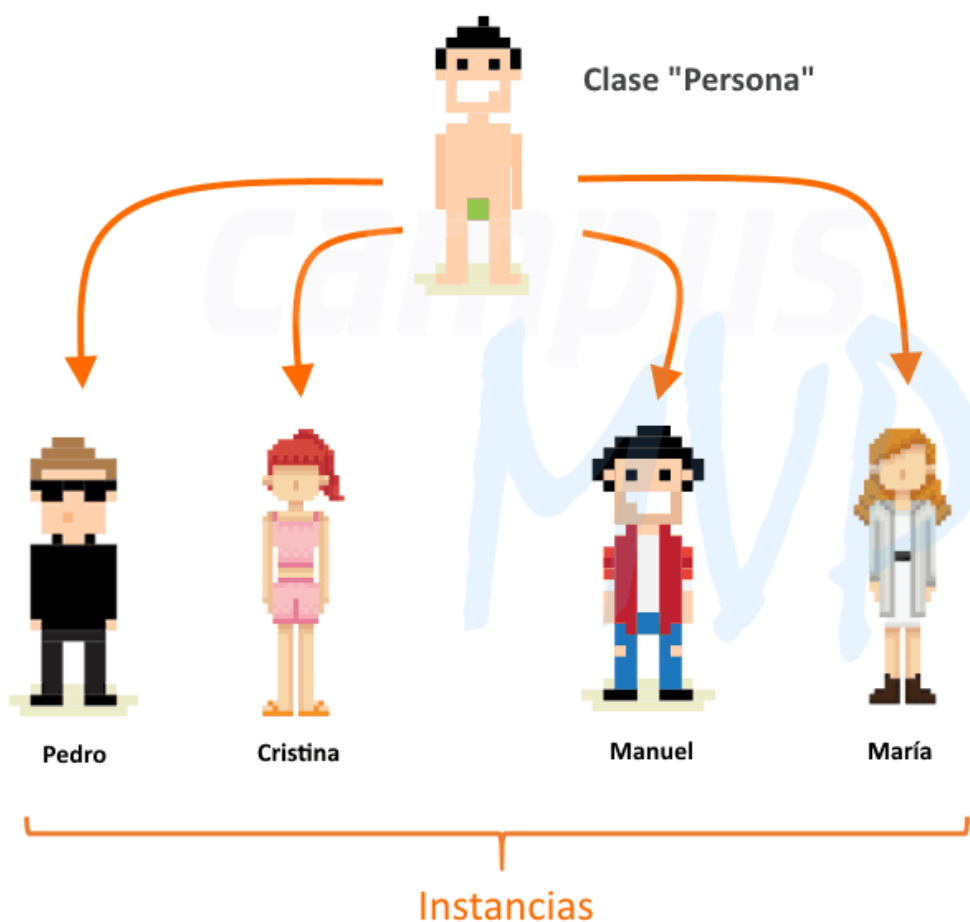
Clases


Es la generalización de un tipo específico de objetos, como el conjunto de características (atributos) y comportamientos de todos los objetos que componen a la clase. Una CLASE es una PLANTILLA mediante la cual se crean los diferentes objetos requeridos para la solución del problema. Los OBJETOS son instancias de las clases.

Las clases son a los objetos como los tipos de datos son a las variables.

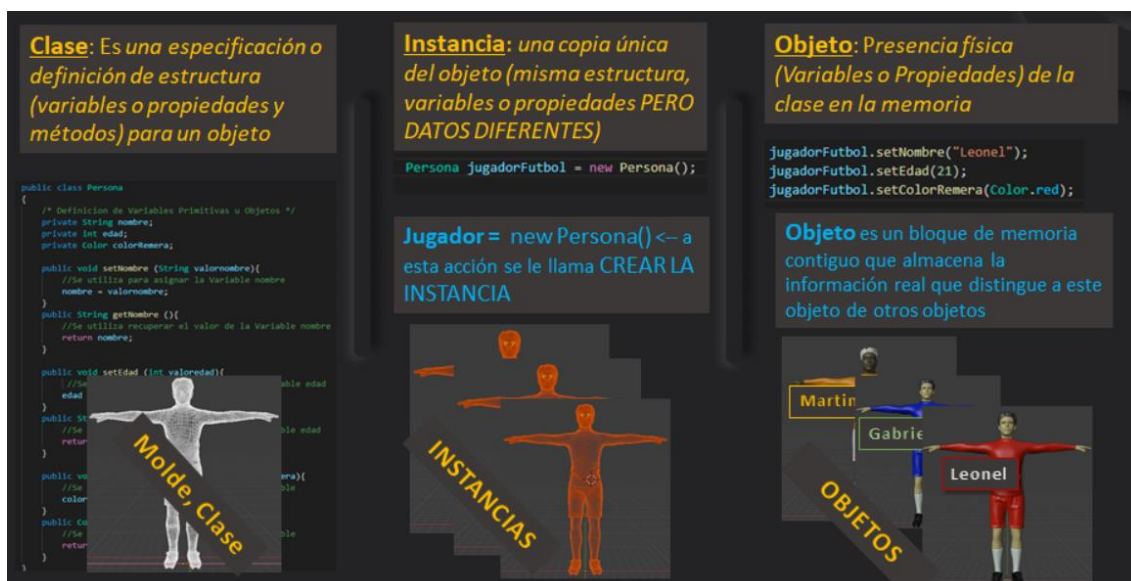
Ejemplo 1, la clase “Plumón” tiene todas las características (tamaño, color, grosor del punto, etc) y todos los métodos o acciones (pintar, marcar, subrayar, etc), que pueden tener todos los plumones existentes en la realidad. Un plumón en especial como un “Marcador” para discos compactos (CDs) de color negro y punto fino, es un objeto (o instancia) de la clase plumón.

Ejemplo 2: Se puede crear un objeto llamado Pedro. Este objeto es creado a partir de la clase Persona. Se puede crear otro objeto llamado: María el cual pertenece a la clase Persona. Significa que a partir de la clase se pueden crear los objetos que se deseen.



Clase		Características o Estado del Objeto	Comportamiento o Verbos del Objeto	<p>La clase de Software es el molde o es la Plantilla de un objeto</p> <p>Molde o Plantilla de Persona</p> 
Atributos	Métodos			
Los atributos son las características individuales que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades	Un método es un bloque de código que contiene una serie de instrucciones			
Clase Persona		Características o Estado del Objeto	Comportamiento o Verbos del Objeto	
Atributos	Métodos			
- nombre - edad - altura - ColorDePiel	+ agregarNombre() + recuperarNombre() + cambiarColorPelo() + caminar()			

¿Qué diferencia existe entre instancia y objeto?, veamos una comparativa en la siguiente imagen



Cuando se define una clase se especifica cómo serán los objetos de dicha clase, esto quiere decir, de qué variables y de qué métodos estará constituida. Las clases proporcionan una especie de plantilla o molde para los objetos, como podría ser el molde para hacer helados de hielo, los objetos que se crean son en sí los helados de hielo, estos helados van a tener como atributos (características), su color, tamaño, sabor, etc y como acciones o métodos, que se pueden congelar, derretir, etc.

Características generales de las clases:

- Poseen un nivel de abstracción alto.
- Se relacionan entre sí mediante jerarquías.
- Los nombres de las clases deben estar en singular.

Se tienen tres tipos de clase que son:

- Abstracta: es muy general (ejemplo: Animal).
- Común: es intermedia (ejemplo: Mamíferos).
- Final: es muy específica (ejemplo: GatoSiames).

La representación gráfica de una o varias clases se hará mediante los denominados Diagramas de Clase. Para los diagramas de clase se utilizará la notación que provee el Lenguaje de Modelación Unificado (UML, que vimos previamente) a saber:

Las clases se denotan como rectángulos divididos en tres partes. La primera contiene el nombre de la clase, la segunda contiene los atributos y la tercera los métodos.

Los modificadores de acceso a datos y operaciones, a saber: público, protegido y privado; se representan con los símbolos +, # y – respectivamente, al lado derecho del atributo. (+ *público*, # *protegido*, - *privado*).

Relaciones entre clases

Las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan relacionar entre sí, de manera que puedan compartir atributos y métodos sin necesidad de volver a escribirlos.

La posibilidad de establecer jerarquías entre las clases es una característica que diferencia esencialmente la programación orientada a objetos de la programación tradicional, debido fundamentalmente a que permite extender y reutilizar el código existente sin tener que volver a escribirlo cada vez que se necesite.

Los tipos de relaciones entre clases que veremos son:

- • **Herencia** (generalización / especialización o Es-un)
- • **Agregación** (todo / parte o Forma-parte-de)
- • **Composición** (Es parte elemental de)
- • **Asociación** (entre otras, la relación Tiene-un)
- • **Dependencia** (Usa un)

La herencia maneja una estructura jerárquica de clases o estructura de árbol (estructura de datos). En esta estructura cada clase tiene una sola clase padre, la cual se conoce como superclase y la clase hija de una superclase se conoce como subclase.

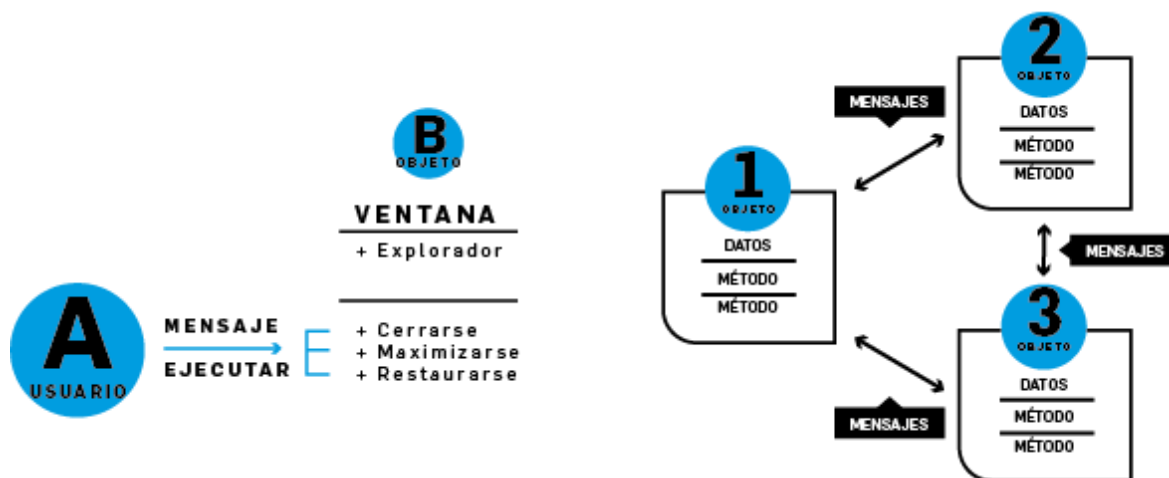
Superclase: se puede definir en términos sencillos como la clase padre de alguna clase específica y puede tener cualquier número de subclases.

Subclase: es la clase hija de alguna clase específica.

Mensajes. Método. Variables e instancias

Mensajes

En la programación orientada a objetos, los objetos descritos anteriormente se comunican a través de señales o mensajes, siendo estos mensajes los que hacen que los objetos respondan de diferentes maneras, por ejemplo, un objeto en Windows como una ventana de alguna aplicación, puede cerrarse, maximizarse o restaurarse (métodos) de acuerdo con el mensaje que le sea enviado. En otras palabras, un mensaje es una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos pudiendo o no llevar algunos parámetros.



Métodos

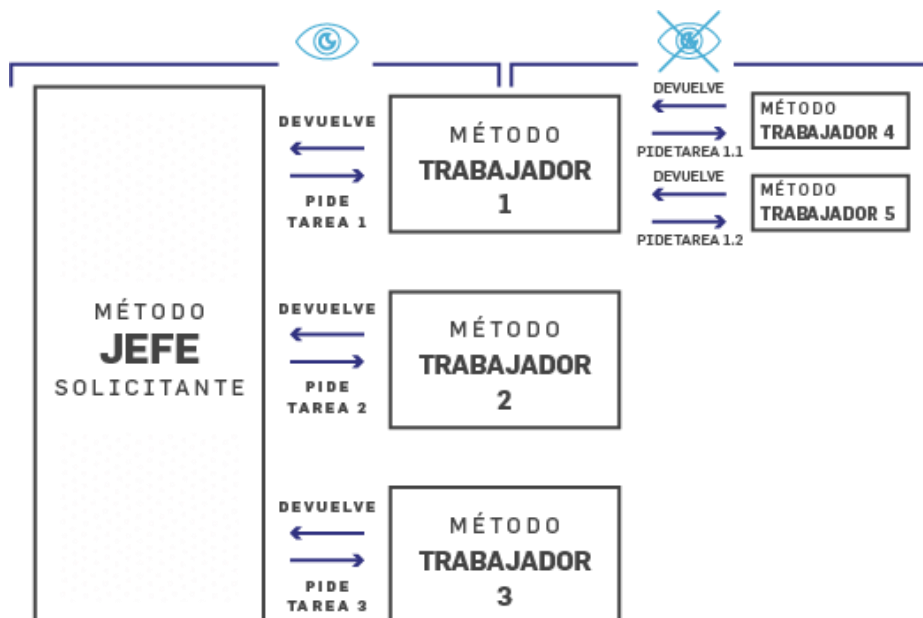
Los métodos (también conocidos como funciones o procedimientos en otros lenguajes) permiten al programador dividir un programa en módulos, por medio de la separación de sus tareas en unidades autónomas. Las instrucciones en los cuerpos de los métodos se escriben sólo una vez, y se reutilizan tal vez desde varias ubicaciones en un programa; además, están ocultas de otros métodos. Una razón para dividir un programa en módulos mediante los métodos es la metodología “divide y vencerás”, que hace que el desarrollo de programas sea más fácil de administrar, ya que se pueden construir programas a partir de piezas pequeñas y simples. Otra razón es la reutilización de software (usar los métodos existentes como si fueran bloques de construcción para crear nuevos programas). A menudo se pueden crear programas a partir de métodos estandarizados, en vez de tener que crear código personalizado. Una tercera razón es para evitar la repetición de código. El proceso de dividir un programa en métodos significativos hace que el programa sea más fácil de depurar y mantener.

Un método se invoca mediante una llamada, y cuando el método que se llamó completa su tarea, devuelve un resultado, o simplemente el control al método que lo llamó. Una analogía a esta estructura de programa es la forma jerárquica de la administración: Un jefe (el solicitante) pide a un trabajador (el método llamado) que realice una tarea y que le reporte (devuelva) los resultados después de completar la tarea. El método jefe, no sabe cómo el método trabajador, realiza sus tareas designadas. Tal vez el trabajador llame a otros métodos trabajadores, sin que lo sepa el jefe. Este “ocultamiento” de los detalles de implementación fomenta la buena ingeniería de software.

Los métodos interactúan con los mensajes, determinando cuáles son los que un objeto puede recibir. Haciendo una analogía con la programación procedural, se podría comparar con las funciones o subrutinas.

En el gráfico siguiente se muestra al método jefe comunicándose con varios métodos trabajadores en forma jerárquica. El método jefe divide las

responsabilidades entre los diversos métodos trabajadores. Observe que trabajador 1 actúa como “método jefe” de trabajador 4 y trabajador 5.



Un método cuenta con las siguientes partes:

- Nombre del método.
- Valor que regresa.
- Argumentos opcionales.
- Modificadores en la declaración del método (como los de visibilidad)
- Cuerpo del método.

Variables

Las variables son localidades de memoria, en las que se almacenan los datos, cada una tiene su propio nombre, tipo y valor. Los tipos de variables que vamos a ver son de:

- **Instancia:** se ocupan para definir los atributos de un objeto.
- **Clase:** son variables en las que sus valores son idénticos para todas las instancias de la clase
- **Locales:** son las variables que se declaran y se utilizan cuando se definen los métodos.

El nombre de la variable será algún identificador válido (de acuerdo con lo explicado anteriormente) y con él se hace la referencia a los datos que contienen la variable. El tipo de una variable indica qué valores puede manejar y a las operaciones qué se pueden hacer con ella, por ejemplo, si es una variable de tipo entero, se pueden realizar con ella las operaciones aritméticas, en tanto que, si fueran de tipo carácter, no se podrían hacer este tipo de operaciones. El valor que puede tomar es en sí la literal, que va a depender precisamente de su tipo.

Instancias

Hasta el momento se han abordado algunos aspectos concernientes a cómo se deben implementar las clases y, cómo expresar sus ATRIBUTOS y MÉTODOS.

Aun cuando todos los tipos de datos son clases, se denominarán objetos a las INSTANCIAS de las clases que excluyen los tipos de datos básicos del lenguaje. Cabe recordar que en el caso de los tipos de datos básicos sus valores pueden asociarse a la definición de variables y luego a través del operador de asignación, éstas cambiarían su estado.

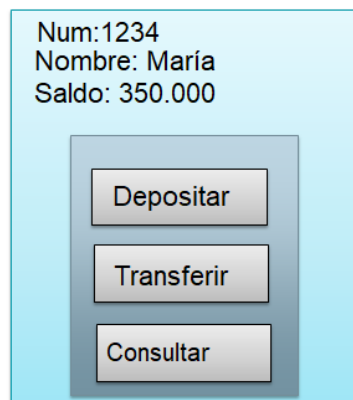
¿Es factible seguir esta misma idea para manejar objetos? Para manipular objetos o instancias de tipos de datos que no sean básicos también se utilizan variables y éstas utilizan semántica por referencia, solo que, con una diferencia con respecto a los tipos básicos, los objetos tienen que ser creados (al crear un objeto se reserva memoria para almacenar los valores de todos sus campos) o al menos contener la referencia de otro objeto creado con anterioridad.

Ejemplo de instancia de objetos

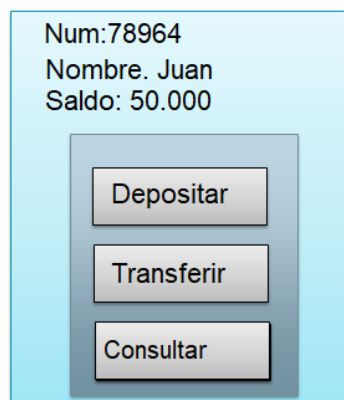
Clase: Cuenta corriente

Instanciación: Cuenta corriente A, B

Clase A,



Clase B



Declaración de variables de instancia

El estado de un objeto se representa por sus variables (conocidas como variables de instancia). Estas son declaradas dentro del cuerpo de la clase. A un objeto se le conoce como instancia de una clase y para crearlo se llama a una parte de líneas de código conocidas con el nombre constructor de una clase que tienen el mismo nombre de la clase. Una vez que ya no se ocupa el objeto, se ejecuta el recolector de basura (garbage collector). Estos conceptos de objeto, constructor y de recolector de basura, se abordarán más adelante con más detalle.

Constructores, Destructores y Garbage Collector

Constructores y destructores

Como ocurre con los objetos físicos, los objetos que se utilizan tienen un ciclo de vida definido. Este ciclo comienza con la instanciación del objeto y finaliza con su destrucción y limpieza por parte del Garbage Collector (es un proceso para reclamar el espacio de memoria de un objeto, cuando ya no exista ninguna referencia al objeto).

Como se comentó anteriormente, es un tipo específico de método, cuyo nombre es el mismo nombre que la clase. Y que es utilizado cuando se quieren crear objetos de esa clase. Es utilizado o ejecutado al crear e iniciar dicho objeto.

Puede existir más de un constructor y se conocen como constructores múltiples, con o sin parámetros. Si no existe algún constructor, se puede proporcionar uno por omisión, el cual inicializa las variables del objeto con los valores que se dan predeterminadamente.

Para ejecutar instrucciones durante cada una de estas dos operaciones, existen métodos específicos: el constructor y el destructor.

1. Un CONSTRUCTOR es un método especial que permite instanciar un objeto. Su nombre es idéntico al de la clase en la que se ha definido, y no tiene ningún tipo de retorno. Puede recibir de 0 a n parámetros.
2. Un DESTRUCTOR es, por tanto, otro método particular cuya firma está impuesta. Su nombre es el mismo que el de la clase, precedido por el carácter ~. Además, no puede recibir ningún parámetro y, como el constructor, no tiene ningún tipo de retorno.

La destrucción de un objeto puede ser una operación automática.

Pero ¿es posible crear instancias de la clase Persona si no se ha definido explícitamente el constructor? No es obligatorio definir CONSTRUCTORES y en el caso en que no se definan (como en el ejemplo anterior de la clase Persona) se brindará uno sin parámetros y de cuerpo vacío (constructor por defecto). Dicho CONSTRUCTOR sólo se ocupa de asignarle a todos los campos sus valores por defecto.

¿Qué sucede si los parámetros del constructor se denominan igual que los campos? ¿Dentro del cuerpo del constructor a quien se haría referencia: al campo o al parámetro?

La respuesta a estos interrogantes se puede obtener a partir de los conceptos de alcance (ámbito) y tiempo de vida de las variables.

1. **Tiempo de vida de las variables:** Es el tiempo durante el cual se puede hacer referencia a la variable y va desde su creación hasta que dejan de existir. Para el caso de los campos de los objetos, estos existen durante

toda la vida de los objetos. Pero en el caso de las variables locales (ejemplo, los parámetros), su tiempo de vida es más limitado pues dejan de existir después que finalice el bloque de instrucciones al que pertenece.

2. **Alcance o ámbito:** Este concepto está relacionado con la visibilidad de las variables y especifica desde qué código es posible hacer referencia a la variable.

El Garbage Collector (GC)

Supervisa la totalidad de instancias creadas a lo largo de la ejecución de la aplicación y marca como huérfano cada objeto que no se utiliza. Cuando el sistema necesita más memoria, el GC destruye aquellos objetos que considera que es posible destruir invocando a sus destructores, si los tienen.

Relación entre clases y objetos

Repasemos un poco algunas definiciones...

Un objeto es una INSTANCIA de una clase. Por lo tanto, los objetos hacen uso de los Atributos (variables) y Métodos (Funciones y Procedimientos) de su correspondiente Clase.

También lo podemos pensar como una variable de tipo clase. Por ejemplo: el objeto Cesar es un objeto de tipo clase: Persona.

Como se puede observar, un objeto a través de su CLASE está compuesto por 2 partes: Atributos o estados y Métodos que definen el comportamiento de dicho objeto a partir de sus atributos. Los atributos y los métodos pueden ser o no accedidos desde afuera dependiendo de la solución a plantear.

Por lo general los atributos siempre se ocultan al exterior y algunos métodos quedan visibles al exterior para convertirse en la interfaz del objeto (Encapsulamiento).

Algorítmicamente, las clases son descripciones netamente estáticas o plantillas que describen objetos. Su rol es definir nuevos tipos conformados por atributos y operaciones.

Por el contrario, los objetos son instancias particulares de una clase. Las clases son una especie de molde de fábrica, en base al cual son construidos los objetos. Durante la ejecución de un programa sólo existen los objetos, no las clases.

La declaración de una variable de una clase NO crea el objeto.

La creación de un objeto debe ser indicada explícitamente por el programador, de forma análoga a como inicializamos las variables con un valor dado, sólo que para los objetos se hace a través de un método CONSTRUCTOR.

Clases objetos métodos y variables de instancia

Comencemos con una analogía simple, para comprender el concepto de las clases y su contenido:

Supongamos que deseas conducir un automóvil y, para hacer que aumente su velocidad, debes presionar el pedal del acelerador. ¿Qué debe ocurrir antes de que puedas hacer esto? Bueno, antes de poder conducir un automóvil, alguien tiene que diseñarlo. Por lo general, un automóvil empieza en forma de dibujos de ingeniería, similares a los planos de construcción que se utilizan para diseñar una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador, para que el automóvil aumente su velocidad. El pedal “oculta” los complejos mecanismos que se encargan de que el automóvil aumente su velocidad, de igual forma que el pedal del freno “oculta” los mecanismos que disminuyen la velocidad del automóvil y por otro lado, el volante “oculta” los mecanismos que hacen que el automóvil de vuelta. Esto permite que las personas con poco o nada de conocimiento acerca de cómo funcionan los motores puedan conducir un automóvil con facilidad.

Desafortunadamente, no se pueden conducir los dibujos de ingeniería de un auto. Antes de poder conducir un automóvil, éste debe construirse a partir de los dibujos de ingeniería que lo describen. Un automóvil completo tendrá un pedal acelerador verdadero para hacer que aumente su velocidad, pero aun así no es suficiente; el automóvil no acelerará por su propia cuenta, así que el conductor debe oprimir el pedal del acelerador.

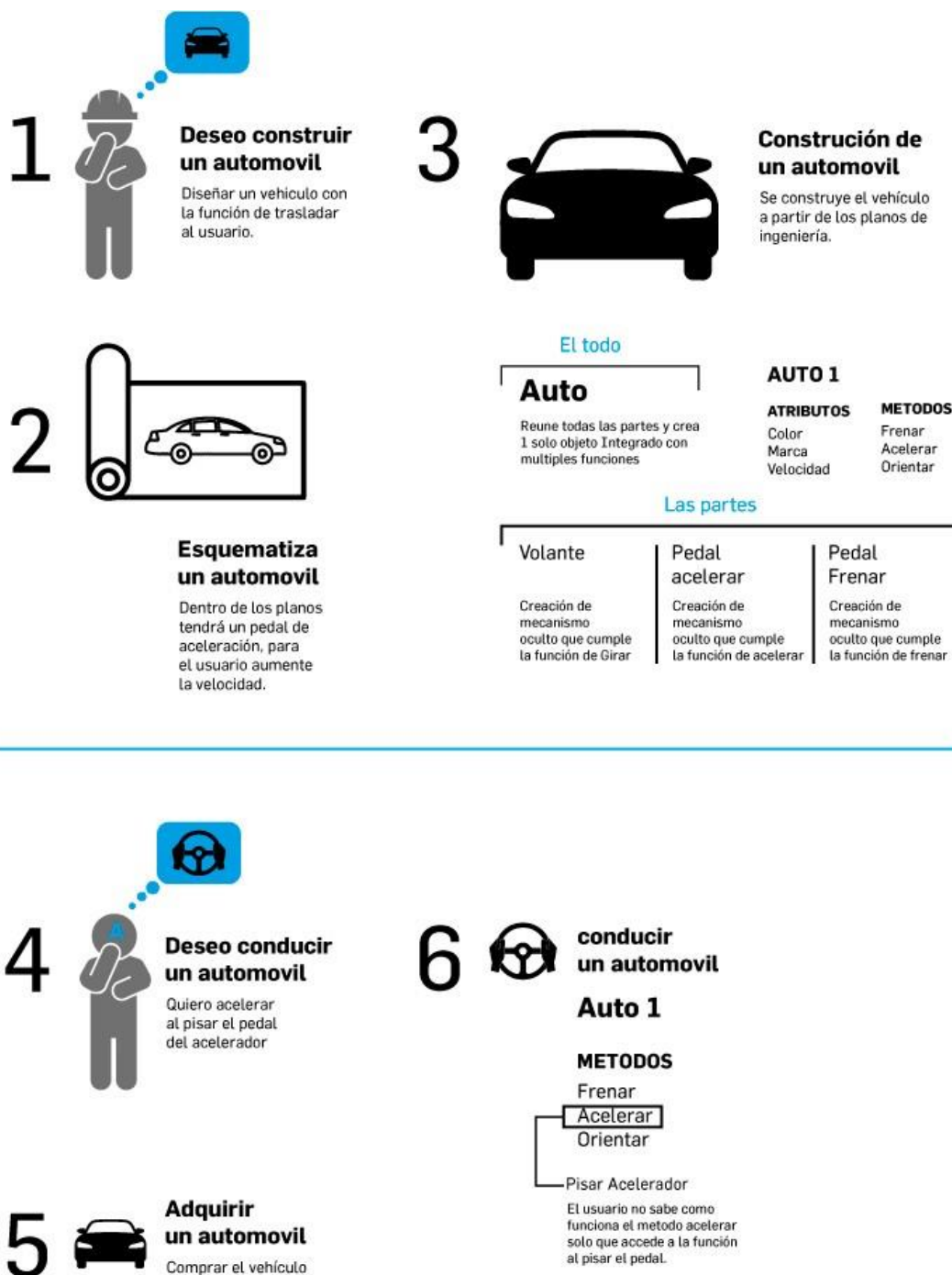
Ahora utilicemos nuestro ejemplo del automóvil para introducir los conceptos clave de programación:

Para realizar una tarea en una aplicación se requiere un método. El método describe los mecanismos que se encargan de realizar sus tareas; y oculta al usuario las tareas complejas que realiza, de la misma forma que el pedal del acelerador de un automóvil oculta al conductor los complejos mecanismos para hacer que el automóvil vaya más rápido. Empezamos por crear una unidad de aplicación llamada clase para alojar a un método, así como los dibujos de ingeniería de un automóvil alojan el diseño del pedal del acelerador. En una clase se proporcionan uno o más métodos, los cuales están diseñados para realizar las tareas de esa clase. Por ejemplo, una clase que representa a una cuenta bancaria podría contener un método para depositar dinero en una cuenta, otro para retirar dinero de una cuenta y un tercero para solicitar el saldo actual de la cuenta.

Así como no podemos conducir un dibujo de ingeniería de un automóvil, tampoco podemos “conducir” una clase. De la misma forma que alguien tiene que construir un automóvil a partir de sus dibujos de ingeniería para poder conducirlo, también debemos construir un objeto de una clase para poder hacer que un programa realice las tareas que la clase le describe cómo realizar. Como ya mencionamos antes, esta es una de las características de un lenguaje de programación orientado a objetos.

Cuando conduces un automóvil, si oprimes el pedal del acelerador se envía un mensaje al automóvil para que realice una tarea (hacer que el automóvil vaya más rápido). De manera similar, se envían mensajes a un objeto; cada mensaje se conoce como la llamada a un método, e indica a un método del objeto que realice su tarea.

Hasta ahora, hemos utilizado la analogía del automóvil para introducir las clases, los objetos y los métodos. Además de las capacidades con las que cuenta un automóvil, también tiene muchos atributos como: su color, el número de puertas, la cantidad de gasolina en su tanque, su velocidad actual y el total de kilómetros recorridos (es decir, la lectura de su odómetro). Al igual que las capacidades del automóvil, estos atributos se representan como parte del diseño en sus diagramas de ingeniería. Cuando se conduce un automóvil, estos atributos siempre están asociados con él. Cada uno mantiene sus propios atributos. Por ejemplo, cada conductor sabe cuánta gasolina tiene en su propio tanque, pero no cuánta hay en los tanques de otros automóviles.



De manera similar, un objeto tiene atributos que lleva consigo cuando se utiliza en un programa. Éstos se especifican como parte de la clase del objeto. Por ejemplo, un objeto tipo cuenta bancaria tiene un atributo llamado saldo, el cual representa la cantidad de dinero en la cuenta. Cada objeto tipo cuenta bancaria conoce el saldo en la cuenta que representa, pero no los saldos de las otras cuentas en el banco. Los atributos se especifican mediante las variables de instancia de la clase.

Fundamentos de POO

POO se ha convertido durante las pasadas dos décadas en el paradigma de programación dominante, y en una herramienta para resolver la llamada crisis del software. Algunos motivos son: por la escalabilidad, porque proporciona un modelo de abstracción que razona con técnicas que la gente usa para resolver problemas, y por el gran desarrollo de herramientas OO (IDEs, librerías,...) en todos los dominios.

“El paradigma orientado a objetos es una metodología de desarrollo de aplicaciones en la cual éstas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representan una instancia de alguna clase, y cuyas clases son miembros de jerarquías de clases unidas mediante relaciones de herencia” ([Grady Booch](#))

¿Qué es lo que cambia respecto a los paradigmas anteriores?

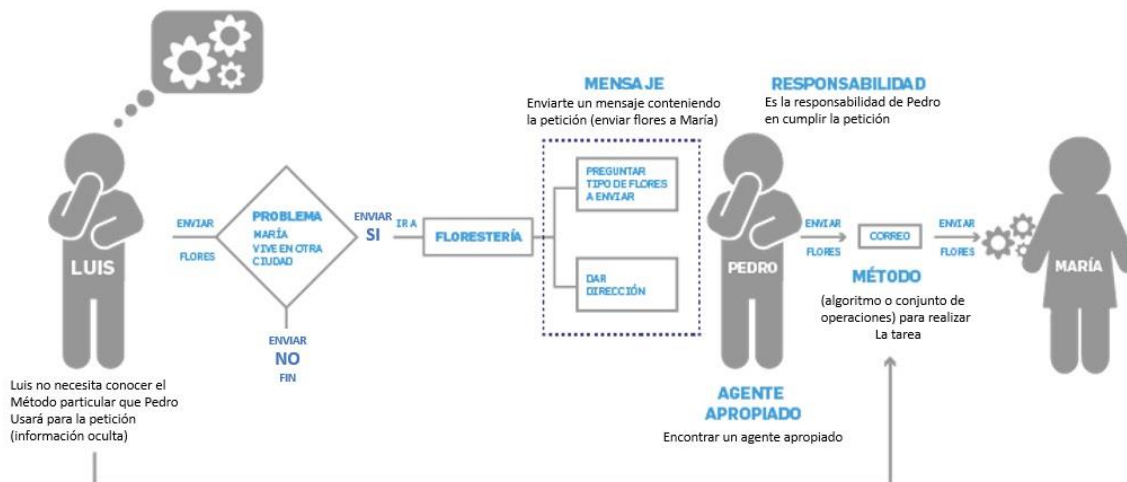
- El modo de organización del programa: en clases (datos + operaciones sobre datos)
- El concepto de ejecución del programa: paso de mensajes

No basta con utilizar un Lenguaje Orientado a Objetos para programar OO. Para eso hay que seguir un paradigma de programación Orientado a Objetos.

Veamos un ejemplo de la vida cotidiana: “Supongamos que Luis quiere enviar flores a María, que vive en otra ciudad”. Luis entonces irá a la floristería más cercana, que es administrada por Pedro, y le dirá a Pedro qué tipo de flores enviar a María, y la dirección de recepción.

Ahora analicemos el mecanismo que utilizó Luis para resolver el problema:

1. Encontrar un agente apropiado (Pedro)
2. Enviarle un mensaje conteniendo la petición (enviar flores a María)
3. Es la responsabilidad de Pedro cumplir esa petición
4. Para ello, es posible que Pedro de algún modo disponga de algún método (algoritmo o conjunto de operaciones) para realizar la tarea
5. Luis no necesita (ni le interesa) conocer el método particular que Pedro utilizará para la petición (esa información es oculta)



Así, la solución del problema requiere de la cooperación de varios individuos para su solución. La definición de problemas en términos de responsabilidades incrementa el nivel de abstracción y permite una mayor independencia entre objetos.

Repasemos algunos conceptos importantes:

Agentes y comunidades: Un programa OO se estructura como una comunidad de agentes que interaccionan (OBJETOS). Cada objeto juega un rol en la solución del problema. Cada objeto proporciona un servicio o realiza una acción que es posteriormente utilizada por otros miembros de la comunidad.

Mensaje y métodos: A un objeto se le envían mensajes para que realice una determinada acción. El objeto selecciona un método apropiado para realizar dicha acción y a este proceso se le denomina Paso de mensajes. Un mensaje se diferencia de un procedimiento/llamada a función en dos aspectos: en un mensaje siempre hay un receptor, lo cual no ocurre en una llamada a procedimiento, y la interpretación de un mismo mensaje puede variar en función del receptor del mismo (un nombre de procedimiento /función se identifica 1:1 con el código a ejecutar, mientras que un mensaje no).

Responsabilidades: El comportamiento de cada objeto se describe en términos de responsabilidades (hay mayor independencia entre los objetos). Otro término importante es el de Protocolo, el cual refiere al conjunto de responsabilidades de un objeto. Si hacemos una comparación entre POO y programación imperativa: “No pienses lo que puedes hacer con tus estructuras de datos. Pregunta a tus objetos lo que pueden hacer por ti”.

Objetos y clases: Un objeto es una encapsulación de un estado (valores de los datos) y comportamiento (operaciones). Otra manera de pensar en un objeto es como la instancia de una clase, además, los objetos se agrupan en categorías (CLASES). El método invocado por un objeto en respuesta a un mensaje viene determinado por la clase del objeto receptor.

Jerarquías de clases: En la vida real, mucho conocimiento se organiza en términos de jerarquías. Este principio por el cual el conocimiento de una categoría más general es aplicable a una categoría más específica se denomina generalización, y su implementación en POO se llama herencia. Por ejemplo, Pedro, por ser florista, es un administrador/vendedor (sabe vender y cobrar). Los vendedores son humanos (pueden hablar). Los humanos son mamíferos (Pedro respira oxígeno...). Las clases de objetos pueden ser organizadas en una estructura jerárquica de herencia. Una clase 'hijo' hereda propiedades de una clase 'padre' más alta en la jerarquía (más general)

Enlace de métodos: Instante en el cual una llamada a un método es asociada al código que se debe ejecutar. Este enlace puede ser estático (en tiempo de compilación), y dinámico (en tiempo de ejecución)

Luego de revisar las características generales de POO, podemos mencionar algunos más de manera resumida:

Polimorfismo: Es la capacidad de una entidad de referenciar elementos de distinto tipo en distintos instantes (Enlace dinámico)

Genericidad: Definición de clases parametrizadas que definen tipos genéricos. Lista<T>, donde T puede ser cualquier tipo.

Gestión de errores: Refiere al tratamiento de condiciones de error mediante excepciones

Aserciones: Son expresiones que especifican qué hace el software en lugar de como lo hace. Precondiciones (propiedades que deben ser satisfechas cada vez que se invoca un servicio), Postcondiciones (propiedades que deben ser satisfechas al finalizar la ejecución de un determinado servicio), e Invariantes (aserciones que expresan restricciones para la consistencia global de sus instancias).

Tipado estático: Es la imposición de un tipo a un objeto en tiempo de compilación. Se asegura en tiempo de compilación que un objeto entiende los mensajes que se le envían. Evita errores en tiempo de ejecución

Recogida de basura (garbage collection): Permite liberar automáticamente la memoria de aquellos objetos que ya no se utilizan.

Concurrencia: Permite que diferentes objetos actúen al mismo tiempo, usando diferentes threads o hilos de control.

Persistencia: Es la propiedad por la cual la existencia de un objeto trasciende la ejecución del programa. Normalmente implica el uso de algún tipo de base de datos para almacenar objetos.

Reflexión: es la capacidad de un programa de manipular su propio estado, estructura y comportamiento. En la programación tradicional, las instrucciones de un programa son "ejecutadas" y sus datos son "manipulados". Si vemos a las instrucciones como datos, también podemos manipularlas.

Cuando seguimos un Paradigma Orientado a Objetos, lo ideal es que el lenguaje proporcione el mayor número posible de las características mencionadas. Orientación a objetos no es una condición booleana (de verdadero o falso): un lenguaje puede ser 'más OO' que otro.

La meta de la Programación Orientada a Objetos (**Parámetros de calidad – Bertrand Meyer**)

La meta última del incremento de abstracción de la POO es MEJORAR LA CALIDAD DE LAS APLICACIONES. Para medir la calidad, Bertrand Meyer define unos parámetros de calidad:

PARÁMETROS EXTRÍNSECOS

Fiabilidad: Corrección (capacidad de los productos de software para realizar con exactitud sus tareas, tal y como se definen en las especificaciones) y Robustez (capacidad de los sistemas de software de reaccionar apropiadamente ante condiciones excepcionales).

Cuando nos referimos a Corrección, si un sistema no hace lo que se supone que debe hacer, de poco sirve todo lo demás. La corrección tiene que ver con el comportamiento de un sistema en los casos previstos por su especificación. La robustez caracteriza lo que sucede fuera de tal especificación.

PARÁMETROS INTRÍNSECOS

Modularidad: Extensibilidad (facilidad de adaptar los productos de software a los cambios de especificación. Simplicidad de diseño) y Reutilización (capacidad de los elementos del software de servir para la construcción de muchas aplicaciones diferentes. Las aplicaciones a menudo siguen patrones similares)

En definitiva, el objeto de POO es producir aplicaciones + fáciles de cambiar (MANTENIBILIDAD)

Abstracción

El Enfoque Orientado a Objetos se basa en cuatro principios que constituyen la base de todo desarrollo orientado a objetos. Estos son:

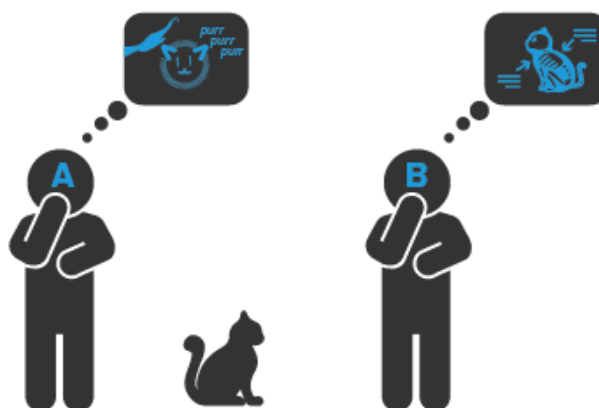
- Abstracción
- Encapsulamiento
- Modularidad
- Herencia

Y otros elementos a destacar (aunque no fundamentales):

- Polimorfismo
- Tipificación
- Concurrencia
- Persistencia

Ya hemos mencionado algunas características de estos principios, pero ahora desarrollémoslo con más detalle.

La abstracción se centra en las características esenciales de un objeto en relación a la perspectiva del observador.



Abstracción

¿Qué es la abstracción? Una abstracción denota las características esenciales de un objeto (datos y operaciones), que lo distingue de otras clases de objetos. Decidir el conjunto correcto de abstracciones de un determinado dominio, es el problema central del diseño orientado a objetos.

Otra forma de conceptualizar la Abstracción es pensarla como la supresión intencionada (u ocultación) de algunos detalles de un proceso o artefacto, con el fin de destacar más claramente otros aspectos, detalles o estructuras. En cada nivel de detalle cierta información se muestra y cierta información se omite. Por ejemplo: Diferentes escalas en mapas.

Mediante la abstracción creamos MODELOS de la realidad.

Abstracción es el proceso de representar entidades reales como elementos internos a un programa, La abstracción de los datos es tanto en los atributos, como en los métodos de los objetos. Por medio de la abstracción se puede tener una visión global del tema, por ejemplo en la clase PILA (Estructura de Datos, Últimas Entradas Primeras Salidas LIFO), se pueden definir objetos de este tipo, tomando únicamente las propiedades LIFO (del inglés Last Input First Output) de las PILAS, algunos atributos como lleno, vacío, el tamaño y las operaciones o métodos que se pueden realizar sobre estos objetos como PUSH (meter un elemento) o POP

(retirar un elemento) y no es necesario que el programador conozca o sea un experto en la Estructura de Datos con la que se implementa la PILA (como por ejemplo con una organización contigua o simplemente ligada, ni de los algoritmos que realizan el PUSH y el POP).

Los diferentes niveles de abstracción ofertados por un lenguaje dependen de los mecanismos proporcionados por el lenguaje elegido:

1. **Perspectiva funcional** (ensamblador – procedimientos – módulos)
2. **Perspectiva de datos** (paquetes – tipos abstractos de datos TAD)
3. **Perspectiva de servicios** (objetos – TAD, + paso de mensajes, + herencia. + polimorfismo)

La abstracción se centra en las características esenciales de un objeto en relación a la perspectiva del observador. Los mecanismos de abstracción son usados en el EOO para extraer y definir del medio a modelar, sus características y su comportamiento. Entre ellos podemos nombrar:

La GENERALIZACIÓN: es el mecanismo de abstracción mediante el cual un conjunto de clases de objetos es agrupado en una clase de nivel superior (Superclase), donde las semejanzas de las clases constituyentes (Subclases) son enfatizadas, y las diferencias entre ellas son ignoradas. En consecuencia, a través de la generalización la superclase almacena datos generales de las subclases. Las subclases almacenan sólo datos particulares.

La ESPECIALIZACIÓN: es lo contrario de la generalización, por ejemplo, la clase Médico es una especialización de la clase Persona, y a su vez, la clase Pediatra es una especialización de la superclase Médico.

La AGREGACIÓN: es el mecanismo de abstracción por el cual una clase de objeto es definida a partir de sus partes (otras clases de objetos). Mediante agregación se puede definir por ejemplo un computador, por descomponerse en: la CPU, la memoria y los dispositivos periféricos. El contrario de agregación es la DESCOMPOSICIÓN.

La CLASIFICACIÓN: consiste en la definición de una clase a partir de un conjunto de objetos que tienen un comportamiento similar. La EJEMPLIFICACIÓN es lo contrario a la clasificación, y corresponde a la instanciación de una clase, usando el ejemplo de un objeto en particular. La clasificación es el medio por el que ordenamos, el conocimiento ubicado en las abstracciones.

OCULTACIÓN DE INFORMACIÓN: Omisión intencionada de detalles de implementación tras una interfaz simple.

Las clases y objetos deberían estar al nivel de abstracción adecuado: ni demasiado alto ni demasiado bajo

Encapsulamiento

¿Qué es el encapsulamiento? El encapsulamiento en un sistema orientado a objeto se representa en cada clase u objeto, definiendo sus atributos y métodos con diferentes tipos de visibilidad (mediante los modificadores de acceso). Como mencionamos anteriormente, si usamos UML, podemos cambiar el tipo de visibilidad, o accesibilidad, usando:

- Público (+) Atributos o Métodos que son accesibles fuera de la clase. Pueden ser llamados por cualquier clase, aun si no está relacionada con ella.
- Privado (-) Atributos o Métodos que solo son accesibles dentro de la implementación de la clase.
- Protegido (#): Atributos o Métodos que son accesibles para la propia clase y sus clases hijas (subclases).

El encapsulamiento oculta los detalles de implementación de un objeto

Cada objeto está aislado del exterior, esta característica permite verlo como una caja negra, que contiene toda la información relacionada con ese objeto. Este aislamiento protege a los datos asociados a un objeto para que no se puedan modificar por quien no tenga derecho a acceder a ellos. Permite manejar a los objetos como unidades básicas, dejando oculta su estructura interna.

Retomemos el ejemplo anterior de una PILA: La PILA oculta una representación interna de datos que lleva el registro de los elementos que esperan actualmente en la línea, y ofrece operaciones a sus clientes (PUSH, agregar un elemento o POP, retirar un elemento). Al desarrollador no le preocupa la implementación de la PILA, simplemente dependen de que esta opere “como se indicó”. Cuando un cliente quiere agregar un elemento (PUSH), la PILA debe aceptarlo y colocarlo en algún tipo de estructura de datos interna. De manera similar, cuando el cliente desea retirar un elemento (POP), la pila debe retirar el elemento que haya estado más tiempo en la PILA.

El encapsulamiento da lugar al Principio de Ocultamiento. Esto implica que sólo los métodos de una clase deberían tener acceso directo a los atributos de esa clase, para impedir que un atributo sea modificado en forma insegura, o no controlada por la propia clase.

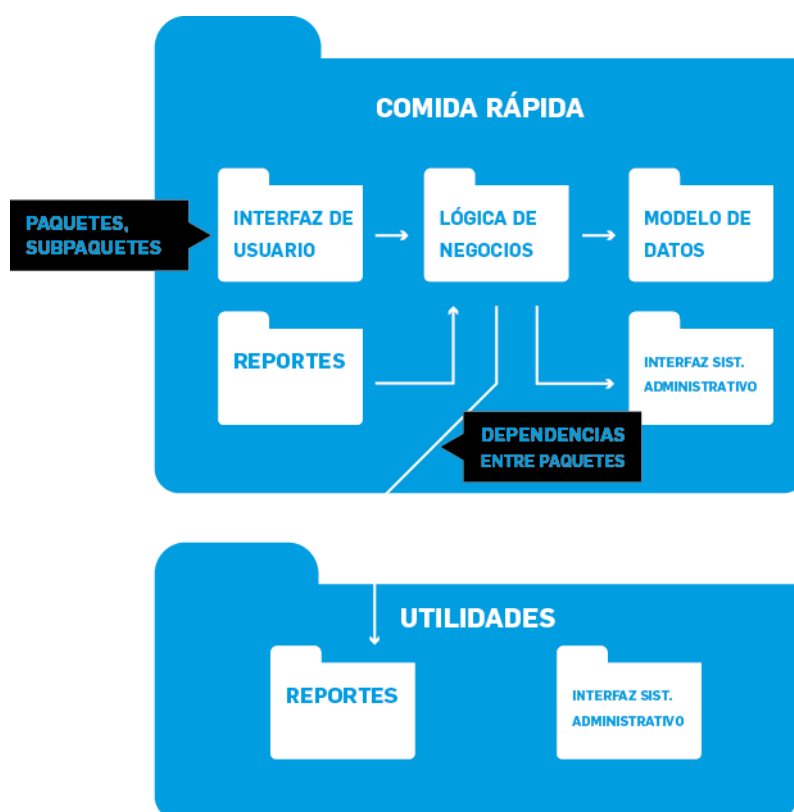
Diferencia entre Abstracción y encapsulamiento

1. **Abstracción:** Busca la solución en el diseño. Únicamente información relevante, Centrado en la ejecución.
2. **Encapsulamiento:** Busca la solución en la implementación. Ocultación de código para protegerlo. Centrado en la ejecución.

Modularidad

¿Qué es la modularidad? Es la propiedad que permite tener independencia entre las diferentes partes de un sistema. La modularidad consiste en dividir un programa en módulos o partes, que pueden ser compilados separadamente, pero que tienen conexiones con otros módulos. En un mismo módulo se suele colocar clases y objetos que guarden una estrecha relación. El sentido de modularidad está muy relacionado con el ocultamiento de información.

Empaqueta abstracciones en unidades discretas



La mayoría de los programas de cómputo que resuelven los problemas reales son mucho más extensos que los programas que hemos presentado. La experiencia ha demostrado que la mejor manera de desarrollar y mantener un programa extenso es construirlo a partir de pequeñas piezas sencillas, o módulos. Como mencionamos anteriormente, a esta técnica se le llama “divide y vencerás”.

Ahora, podemos mencionar cuáles son los principios de la Modularidad:

1. **Capacidad de descomponer un sistema complejo.** Recuerda el principio de «Divide y Vencerás», en este procedimiento se realiza algo similar, ya que descompones un sistema en subprogramas (recuerda llamarlos módulos), el problema en general lo divides en problemas más pequeños.
2. **Capacidad de componer a través de sus módulos.** Indica la posibilidad de componer el programa desde los problemas más pequeños completando y resolviendo el problema en general, particularmente cuando se crea

software se utilizan algunos módulos existentes para poder formar lo que nos solicitan, estos módulos que se integran a la aplicación deben de ser diseñados para ser reusables.

3. **Comprensión de sistema en partes.** El poder tener cada parte separada nos ayuda a la comprensión del código y del sistema, también a la modificación del mismo, recordemos que si el sistema necesita modificaciones y no hemos trabajado con módulos definitivamente eso será un caos.

Una de las razones por la cuál es útil hacerlo modular es debido a que podemos tener los límites bien definidos y es lo más valioso a la hora de leer el programa, recordemos una buena práctica en programación es hacerlo pensando en quien mantendrá el código.

Herencia

Ahora vamos a abordar con más detalle otra característica principal de la programación orientada a objetos: La herencia, que es una forma de reutilización del software en la que se crea una nueva clase absorbiendo los miembros de una clase existente. Además, se mejoran con nuevas capacidades, o con modificaciones en las capacidades ya existentes.

Con la herencia, los programadores vamos a ahorrar tiempo durante el desarrollo, al reutilizar software probado y depurado de alta calidad. Esto también aumenta la probabilidad de que un sistema se implemente con efectividad. Al crear una clase, en vez de declarar miembros completamente nuevos, podemos designar que la nueva clase herede los miembros de una clase existente. Esta clase existente como hemos mencionado antes se conoce como superclase (o clase base), y la nueva clase se conoce como subclase (o clase derivada). Cada subclase puede convertirse en la superclase de futuras subclases. Una subclase generalmente agrega sus propios campos y métodos. Por lo tanto, una subclase es más específica que su superclase y representa a un grupo más especializado de objetos. Generalmente, la subclase exhibe los comportamientos de su superclase junto con comportamientos adicionales específicos de esta subclase. Es por ello que a la herencia se le conoce algunas veces como especialización.

Jerarquía y Herencia

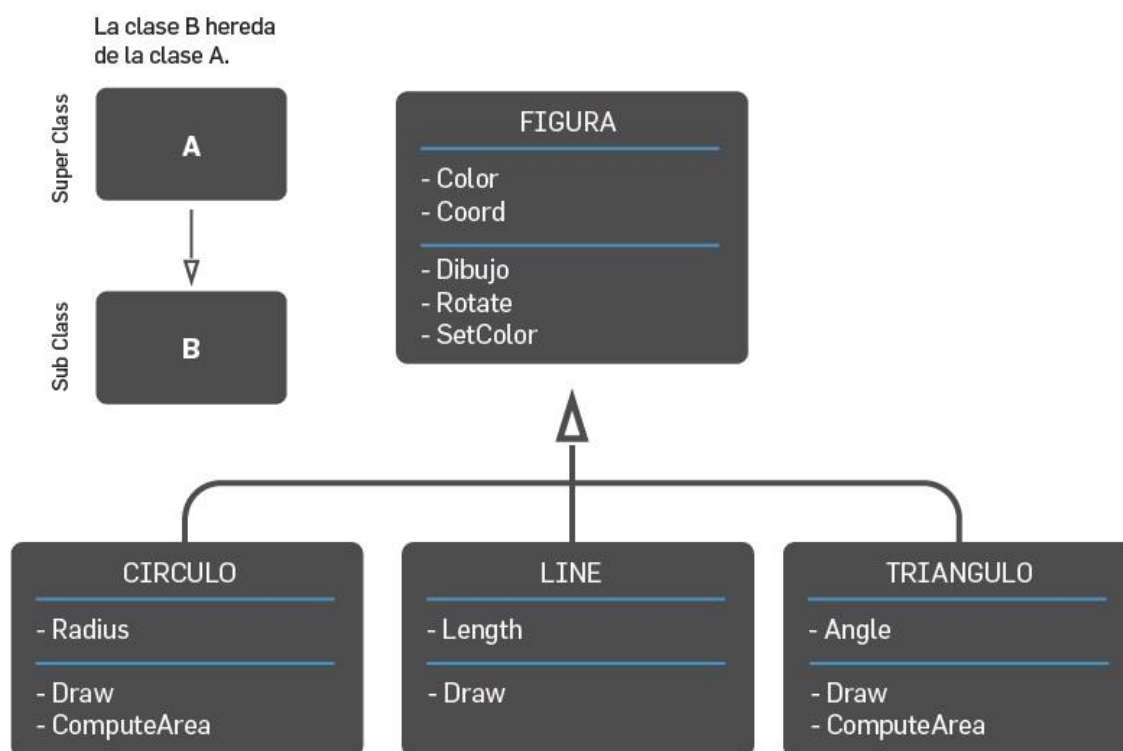
Si bien ya presentamos los términos super clase y subclase en el apartado de "Clases", veamos algunos puntos más. La jerarquía es una clasificación u ordenación de abstracciones. Las dos jerarquías más importantes en un sistema complejo son: su estructura de clases (la jerarquía "de clases") y su estructura de objetos (la jerarquía "de partes").

Las clases se encuentran relacionadas entre sí, formando una jerarquía de clasificación. La herencia es el medio para compartir en forma automática los métodos y los datos entre las clases y subclases de los objetos. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. Existe la herencia Simple y Múltiple:

1. **Herencia simple** (Una clase solo tiene una superclase. Un objeto pertenece a una sola clase)
2. **Herencia múltiple** (Una clase tiene 2 o más superclases. Un objeto pertenece a más de una clase)

Herencia simple

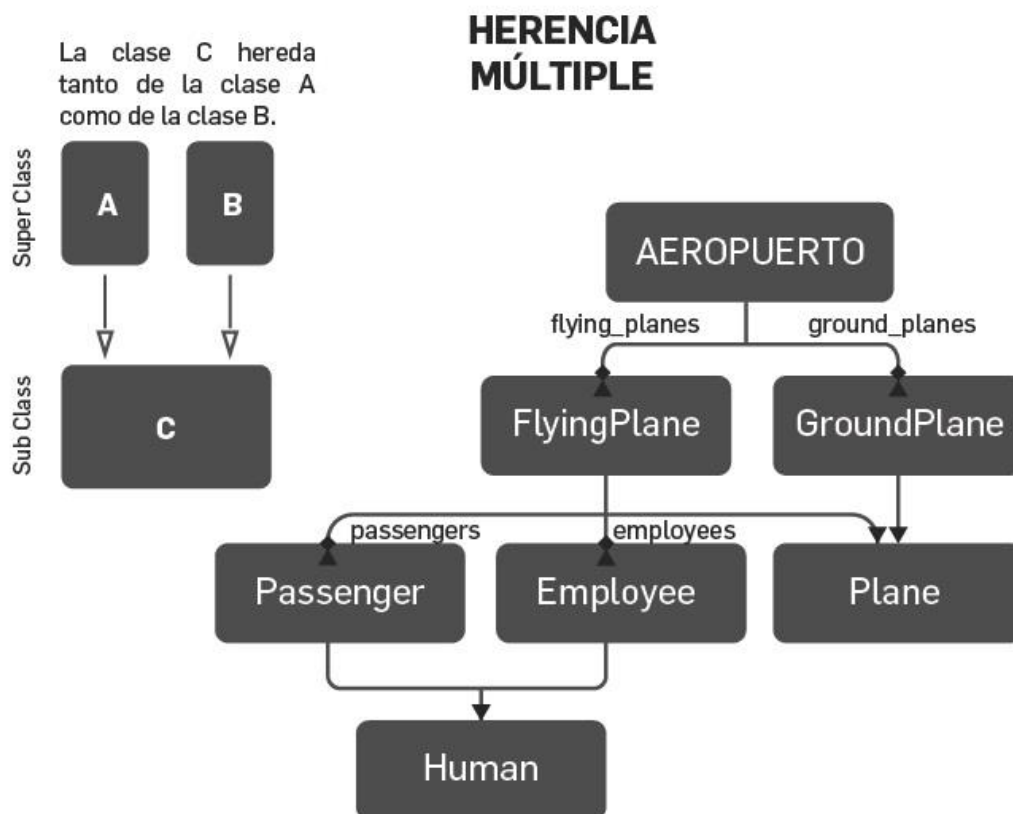
(Una clase solo tiene una superclase. Un objeto pertenece a una sola clase)



La clases Circle, Line y Triangle heredan todos los atributos y métodos de la clase FIGURA.

Herencia múltiple

(Una clase tiene 2 o más superclases. Un objeto pertenece a más de una clase)



En este ejemplo, las clases Passenger y Employee heredan de la clase Human y las clases FlyingPlane y GroundPlane heredan de la clase Plane.

Entonces, la superclase directa es la superclase a partir de la cual la subclase hereda en forma explícita. Una superclase indirecta es cualquier clase arriba de la superclase directa en la jerarquía de clases, la cual define las relaciones de herencia entre las clases. En el caso de la herencia simple, una clase se deriva de una superclase directa.

Para la reutilización del código, se crean nuevas clases, pero utilizando las clases ya existentes y la estructura jerárquica de clases que se mencionó anteriormente. Para ello se utilizan los siguientes dos métodos más frecuentemente:

- **Composición.** Donde se crean objetos a partir de clases existentes, la nueva clase se compone de objetos de clases ya existentes.
- **Herencia.** Es como en la vida real, alguien hereda, adquiere algún bien o característica genética de su padre, por ejemplo, se puede decir: esta persona heredó hasta la forma de caminar de su padre, en POO esta clase es como su clase padre, ya que conserva los atributos y los métodos de la clase padre, pudiendo “alterar” (por medio de los modificadores de acceso),

ya sea para quitar o agregar algunos de ellos en forma individual para esa clase en específico.

Pero, cuando hablamos de herencia ¿Qué tipo de relación hay entre clases y objetos?

Es necesario hacer una diferencia entre la relación “es un” y la relación “tiene un”. La relación “es un” representa a la herencia. En este tipo de relación, un objeto de una subclase puede tratarse también como un objeto de su superclase. Por ejemplo, un automóvil es un vehículo. En contraste, la relación “tiene un” identifica a la composición. En este tipo de relación, un objeto contiene referencias a objetos como miembros. Por ejemplo, un automóvil tiene un volante de dirección (y un objeto automóvil tiene una referencia a un objeto volante de dirección). Las clases nuevas pueden heredar de las clases ya existentes. Las organizaciones desarrollan sus propias bibliotecas de clases y pueden aprovechar las que ya están disponibles en todo el mundo. Es probable que algún día, la mayoría de software nuevo se construya a partir de componentes reutilizables estándar, como sucede actualmente con la mayoría de los automóviles y del hardware de computadora. Esto facilitará el desarrollo de software más poderoso, abundante y económico.

Polimorfismo

Ahora continuemos nuestro estudio de la programación orientada a objetos, explicando y demostrando el polimorfismo con las jerarquías de herencia:

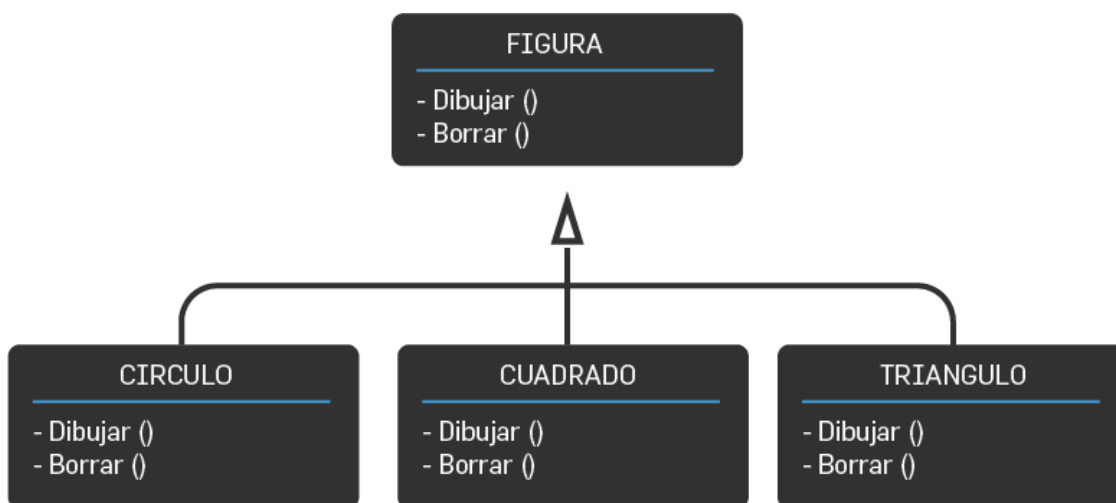
El polimorfismo nos permite “programar en forma general”, en vez de “programar en forma específica”. En especial, nos permite escribir programas que procesen objetos que compartan la misma superclase en una jerarquía de clases, como si todos fueran objetos de la superclase; esto puede simplificar la programación.

Consideremos el siguiente ejemplo de polimorfismo: Suponga que tenemos que crear un programa que simula el movimiento de varios tipos de animales para un estudio biológico. Las clases *Pez*, *Rana* y *Ave* representan los tres tipos de animales bajo investigación. Imaginemos que cada una de estas clases extiende a la superclase *Animal*, la cual contiene un método llamado *mover* y mantiene la posición actual de un animal, en forma de coordenadas x-y. Cada subclase implementa el método *mover*. Nuestro programa mantiene un arreglo de referencias a objetos de las diversas subclases de *Animal*. Para simular los movimientos de los animales, el programa envía a cada objeto el mismo mensaje una vez por segundo; a saber, *mover*. No obstante, cada tipo específico de *Animal* responde a un mensaje *mover* de manera única; un *Pez* podría nadar tres pies, una *Rana* podría saltar cinco pies y un *Ave* podría volar diez pies. El programa envía el mismo mensaje (es decir, *mover*) a cada objeto animal en forma genérica, pero cada objeto sabe cómo modificar sus coordenadas x-y en forma apropiada para su tipo específico de movimiento. Confiar en que cada objeto sepa cómo “hacer lo correcto” (es decir, lo que sea apropiado para ese tipo de objeto) en respuesta a la llamada al mismo método es el concepto clave del polimorfismo. El mismo mensaje (en este caso, *mover*) que se envía a una variedad de objetos

tiene “muchas formas” de resultados; de aquí que se utilice el término polimorfismo.

Con el polimorfismo podemos diseñar e implementar sistemas que puedan extenderse con facilidad; pueden agregarse nuevas clases con sólo modificar un poco (o nada) las porciones generales de la aplicación, siempre y cuando las nuevas clases sean parte de la jerarquía de herencia que la aplicación procesa en forma genérica. Las únicas partes de un programa que deben alterarse para dar cabida a las nuevas clases son las que requieren un conocimiento directo de las nuevas clases que el programador agregará a la jerarquía. Por ejemplo, si extendemos la clase Animal para crear la clase Tortuga (que podría responder a un mensaje mover caminando una pulgada), necesitamos escribir sólo la clase Tortuga y la parte de la simulación que crea una instancia de un objeto Tortuga. Las porciones de la simulación que procesan a cada Animal en forma genérica pueden permanecer iguales.

De manera resumida, el polimorfismo es una propiedad del EOO que permite que un método tenga múltiples implementaciones, que se seleccionan en base al tipo objeto indicado al solicitar la ejecución del método.



El polimorfismo operacional o sobrecarga operacional permite aplicar operaciones con igual nombre a diferentes clases o están relacionados en términos de inclusión. En este tipo de polimorfismo, los métodos son interpretados en el contexto del objeto particular, ya que los métodos con nombres comunes son implementados de diferente manera dependiendo de cada clase.

Clases diferentes (polimórficas) implementan métodos con el mismo nombre. Comportamientos diferentes, asociados a objetos distintos pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando.

Esta característica facilita la implementación de varias formas de un mismo método, con lo cual se puede acceder a varios métodos distintos, que tienen el mismo nombre. Existen dos tipos principales de polimorfismo, que son:

1. Por reemplazo: dos o más clase diferentes con el mismo nombre del método, pero haciéndolo de forma diferente.
2. Por sobrecarga: es el mismo nombre de método ocupado varias veces, ejecutándolo de diferente forma y diferenciándose solamente por el argumento o parámetro.

Cuando referíamos a los tipos de polimorfismo se pueden agregar los siguientes comentarios:

Un objeto solamente tiene la forma que se le asigna cuando es construido, pero se puede hacer referencia a él en forma polimórfica, ya que se puede referir a objetos de diferentes clases y debe de existir una relación de herencias entre clases. El Polimorfismo permite separar el “que” del “cómo” y permite distinguir entre tipos de objetos que son parecidos, esta diferencia se muestra través del comportamiento de los métodos que heredan de la clase padre.

Las funciones (métodos) polimórficas, son a las que no les interesa el tipo de variables que reciben.

Polimorfismo por herencia. Métodos virtuales

La palabra clave virtual se usa para modificar una declaración de método, propiedad (no estática) o evento y permitir que, cualquier clase que herede este método pueda reemplazarlo.

Polimorfismo por abstracción: Clase Abstracta.

Clase que no es posible crear instancias. Frecuentemente, están implementadas parcialmente o no están implementadas. Forzosamente se ha de derivar si se desea crear objetos de la misma.

Polimorfismo por interfaces: Interface.

Es un CONTRATO. Define propiedades y métodos, pero no implementación.

Tipificación, Concurrencia y Persistencia

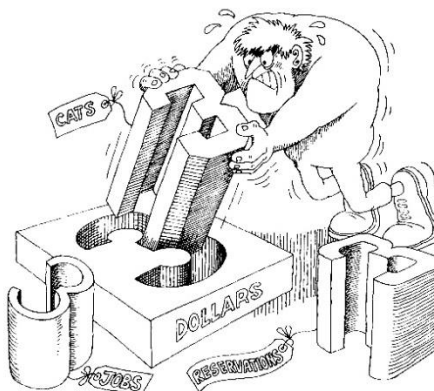
Tipificación

Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas. Es la caracterización precisa de propiedades estructurales o de comportamiento que comparten ciertas entidades.

De manera simple, tipificar es la imposición de una clase a un objeto, de tal modo que objetos de diferentes tipos no se puedan intercambiar, o se puedan intercambiar solo de forma restringida.

Un tipo es una caracterización precisa de las propiedades estructurales y de comportamiento que comparten una colección de entidades. De manera general puede considerarse que tipo y clase son sinónimos.

Existen lenguajes fuertemente tipificados y débilmente tipificados. Estos últimos soportan polimorfismo, mientras que los fuertemente tipificados no.



LA TIPIFICACION IMPIDE LA MEZCLA DE ENTIDADES

Concurrencia

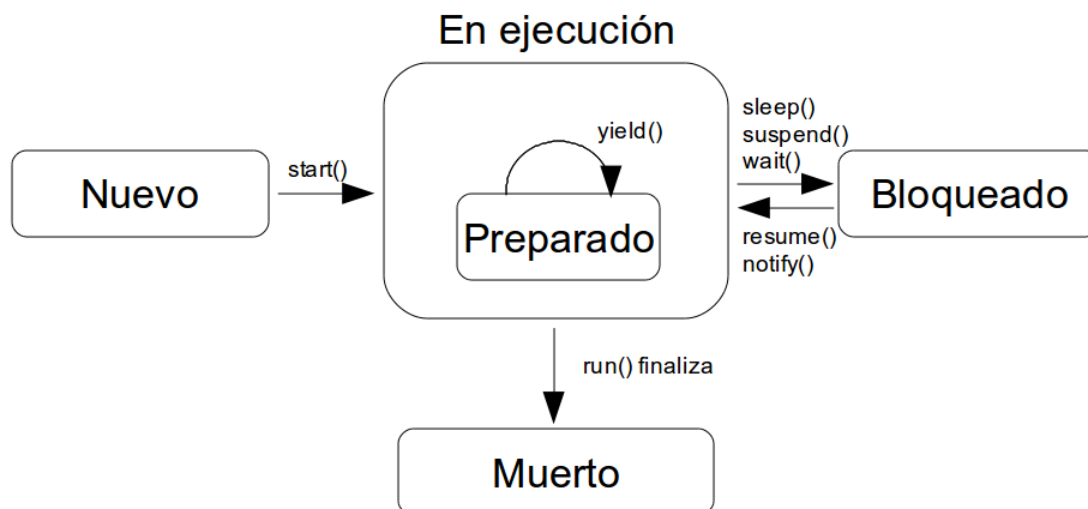
La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo. La concurrencia permite a dos objetos actuar al mismo tiempo. En estos casos es muy común tener que manejar varias acciones diferentes al mismo tiempo, para ello se utilizan procesos los cuales producen acciones dinámicas independientes dentro de un sistema.

En esto manejamos dos tipos de procesos: pesados y ligeros.

1. **Proceso Pesado:** Es aquel que es comúnmente manejado de forma autónoma por el sistema operativo, este tiene su propio espacio de direcciones.
2. **Proceso Ligero:** Existe dentro de un solo proceso del sistema operativo en compañía de otros procesos ligeros y comparten el mismo espacio de direcciones.

Recordemos que la Programación orientada a objetos se centraliza en la abstracción de datos, encapsulamiento y herencia, mientras que la concurrencia se centra en la abstracción de procesos y la sincronización.

Una vez que se tiene la concurrencia en un sistema, debemos de tomar en cuenta cómo los objetos activos se sincronizan con otros.



Persistencia

La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).

Conserva el estado de un objeto en el tiempo y en el espacio.

Dicho de otra manera, la persistencia es la acción de mantener la información del objeto de una forma permanente (guardarla), pero también debe de poder recuperarse dicha información para que pueda ser utilizada nuevamente.

La persistencia es el mecanismo que se usa para mantener información almacenada.

Para la persistencia los objetos podrían clasificarse en dos tipos: objetos transitorios y objetos persistentes.

1. **Transitorios:** Son aquellos que su tiempo de vida depende del espacio del proceso que lo creó.
2. **Persistentes:** Son aquellos que su estado es almacenado en un medio temporal para su posterior reconstrucción y utilización, por lo cual el objeto no depende del proceso que lo creó. Un ejemplo de la persistencia es aquel objeto que se crea para luego ser guardado en la base de datos.

Conclusiones POO

Lo que conocemos por P.O.O. no es un conjunto de rasgos añadidos a los lenguajes de programación. Más bien es un nuevo modo de organizar el pensamiento acerca del modo de descomponer problemas y desarrollar soluciones de programación.

La POO ve un programa como un conjunto de agentes débilmente acoplados (objetos). Cada objeto es responsable de un conjunto de tareas. La computación se realiza gracias a la interacción de estos objetos. Por tanto, en cierto sentido, programar consiste en simular un modelo del universo.

Un objeto es una encapsulación de un estado (valores de los datos) y comportamiento (operaciones). Así, un objeto es en muchos sentidos similar a un computador de propósito específico.

El comportamiento de los objetos viene dictado por su clase. Todos los objetos son instancias de alguna clase. Todas las instancias de la misma clase se comportarán de un modo similar (invocarán el mismo método) en respuesta a una petición similar.

La interpretación de un mensaje es decidida por el objeto, y puede diferir de una clase de objeto a otra.

Las clases pueden enlazarse unas a otras mediante la noción de jerarquías de herencia. En estas jerarquías, datos y comportamiento asociados con clases más altas en la jerarquía y pueden ser accedidas y usadas por clases que descienden de ellas.

El diseño de un programa OO es como organizar una comunidad de individuos. Cada miembro de la comunidad tiene ciertas responsabilidades. El cumplimiento de las metas de la comunidad como un todo viene a través del trabajo de cada miembro y de sus interacciones.

Mediante la reducción de la interdependencia entre componentes del software, la POO permite el desarrollo de sistemas de software reutilizables. Estos componentes pueden ser creados y testeados como unidades independientes, aisladas de otras porciones de la aplicación de software.

Los componentes reutilizables permiten al programador tratar con problemas a un nivel de abstracción superior. Podemos definir y manipular objetos simplemente en términos de mensajes, ignorando detalles de implementación. Este principio de "ocultación de información" ayuda en la comprensión y construcción de sistemas seguros. También favorece la mantenibilidad del sistema.

Se ha comprobado que a las personas con ciertos conocimientos tradicionales sobre computación les resulta más difícil aprender los nuevos conceptos que aporta la P.O.O. que a aquéllos que no saben nada, ya que el modo de razonar a la hora de programar es una metáfora del modo de razonar en el mundo real.

La P.O.O. está fuertemente ligada a la Ingeniería del Software, con el objetivo de conseguir aplicaciones de mayor calidad.

Diseño de Software

¿Qué es Diseño de Software?

El diseño de software es una actividad previa a la programación que consiste en pensar como dividir el sistema o proceso en módulos o en grupos de líneas de código más pequeños, esto permite tener un modelo de los elementos de software que necesito antes de empezar a programar. Por último, poder planificar las tareas pudiendo asignar a distintos programadores cada módulo o fragmento de código.

Dividir el sistema en elementos de software: el objetivo es no hacer un programa como un solo bloque de código. El objetivo es separar en partes que se comunican o relacionan para poder unirlos como un rompecabezas, esto que nos permite poder desarrollar diferentes partes en paralelo. Estas partes de programas las llamaremos Elementos de Software.

Elementos de Software:

1. **Módulos:** un módulo es una porción de un programa de ordenador. De las varias tareas que debe realizar un programa para cumplir con su función u objetivos, un módulo realizará, comúnmente, una de dichas tareas (o varias, en algún caso).
2. **Programas:** Un programa es un conjunto de pasos lógicos escritos en un lenguaje de programación que nos permite realizar una tarea específica.
3. **Procesos:** Un proceso es la ejecución de un programa, es decir, los datos e instrucciones están cargados en la memoria principal, ejecutándose o esperando a hacerlo.
4. **Paquetes:** Cuando escribimos código, es posible que el resultado de nuestro programa sea bastante grande. Además, podríamos tener cientos o miles de clases distintas y relacionadas entre sí. Así que es normal agrupar estas clases dentro de paquetes, para hacer más fácil el mantenimiento. Los paquetes pueden contener clases, pero también puede contener otros paquetes, y así integrar una jerarquía entre ellos.
5. **Componentes:** Un componente es una unidad modular de un programa software con interfaces y dependencias bien definidas que permiten ofertar o solicitar un conjunto de servicios o funcionales.

Asignar una función específica a cada Elemento de Software consiste en describir con precisión la función que cumple cada elemento de software. Las relaciones entre los elementos del diseño son:

1. **Herencia:** es una forma de reutilización de software en la que se crea una nueva clase absorbiendo los miembros de una clase existente.
2. **Composición:** implica que tenemos una instancia de una clase que contiene instancias de otras clases que implementan las funciones deseadas. Es decir, estamos delegando las tareas que nos mandan a hacer a aquella pieza de código que sabe hacerlas. El código que ejecuta esa tarea concreta está sólo en esa pieza y todos delegan en ella para ejecutar dicha tarea.

3. **Inyección de dependencia:** La inyección de dependencias es un patrón de diseño de software usado en la Programación Orientada a Objetos, que trata de solucionar las necesidades de creación de los objetos de una manera práctica, útil, escalable y con una alta versatilidad del código.
4. **Interfaces:** una interfaz (también llamada protocolo) es un medio común para que los objetos no relacionados se comuniquen entre sí. Estas son definiciones de métodos y valores sobre los cuales los objetos están de acuerdo para cooperar.

Ahora amplíemos un poco sobre algunos conceptos importantes ¿Qué es un módulo y cuál es su definición?

El concepto de módulo puede variar en diferentes documentaciones, pero pondremos varias definiciones que aplican para módulo:

1. Un módulo es una unidad de implementación de software que provee una funcionalidad definida o provee un conjunto de servicios definidos.
2. Un módulo es un conjunto de líneas de código agrupadas bajo un nombre, ese nombre puede ser reutilizado o referenciado en otros módulos de software.
3. Un módulo puede ser asignado a un programador para que diseñe o implemente según las definiciones solicitadas.

La estructura de un módulo podemos pensar en dividirlo en 2 partes:

Interface:

La interface es un conjunto mínimo de subrutinas que permite al módulo cumplir su función asignada, es decir cualquier tarea que el módulo deba hacer tiene que ser a través de las subrutinas provistas por la interface. Es el único punto de interacción con el módulo, es decir, cualquier parte del sistema debe acceder al módulo únicamente por la interface.

La interface para ser definida, debe asignarse un nombre, el tipo de datos de cada parámetro y el valor de retorno en caso que exista. La interface debe ser definida por el diseñador o arquitecto del sistema.

Implementación:

La implementación la deben hacer los programadores

¿Por qué dividir el sistema en módulos o partes más pequeñas?

El objetivo de dividir un sistema en módulos o partes más pequeñas es para tener un sistema “Diseñado para el Cambio”, el sistema estará sometido a cambios constantes por distintos factores al paso del tiempo, un Diseño en modulo permite agregar, modificar o reemplazar solamente el módulo sin afectar la funcionalidad del resto de los módulos o sistema. Los cambios dentro del sistema pueden ocurrir por muchos factores, pero listamos algunos:

1. Cambio a funcionalidades existentes ya programadas.
2. Cambiar la pantalla de usuario para agregarle un dato nuevo.
3. Cambiar una parte del cálculo que hace el sistema actualmente.
4. Cambios tecnológicos: (Necesidad de conectar con otros sistemas / Necesidad de acceder desde internet / Necesidad de acceder desde celulares / Necesidad de conectar a una nueva base de datos).
5. Cambios socioculturales: (Cambio de moneda / Cambio la forma de calcular impuestos / Cambio de idiomas o internacionalizar).

¿Cómo puedo pensar en diseñar en módulos?

No se puede saber con exactitud cuántos módulos necesitaran el sistema que vas a desarrollar, pero te compartimos que te ayudara a que módulos necesitaras.

Detectar ítem de cambio: Analizar el requerimiento recibido para detectar los posibles agentes de cambio que se observen.

Aprender los patrones de diseño: Los patrones de diseño son soluciones planteadas para reutilizar código o resolver problemas comunes y conocidos.

Para profundizar más en tema podemos recomendarte los libros de David L. Parnas, Robert Martins

¿Qué es ADOO?

Se han desarrollado metodologías, que tienen como una de sus funciones el lograr una mayor productividad en el desarrollo de los sistemas de información. Existen diferentes metodologías las cuales tienen diferentes fases y son precisamente dos de ellas en las que se enfatizará y son el Análisis y el Diseño. Tal vez no sea tan importante ver cuál es la mejor metodología, si no el conocer alguna y aplicarla, en general en la fase del Análisis se identifica el ¿qué se desea hacer?, es un Análisis detallado de los requerimientos del proyecto que se va a realizar, en el caso de la POO es la abstracción resumida y precisa de lo que debe de hacer el sistema deseado y en la del Diseño ¿cómo se va a hacer? para cumplir con esos requerimientos, que características de rendimiento hay que optimizar. Por último y no menos importante el paso siguiente sería la implementación del software.

El marco de desarrollo de una aplicación software estaría formado entonces por tres fases: análisis, diseño e implementación.

1. El análisis es la fase cuyo objetivo es estudiar y comprender el dominio del problema, es decir, el análisis se centra en responder al interrogante ¿QUÉ HACER?
2. El diseño, por su parte, dirige sus esfuerzos en desarrollar la solución a los requisitos planteados en el análisis, esto es, el diseño se haya centrado en el espacio de la solución, intentando dar respuesta a la cuestión ¿CÓMO HACERLO?

3. Por último, la fase de implementación sería la encargada de la traducción del diseño de la aplicación al lenguaje de programación elegido, adaptando por tanto la solución a un entorno concreto.

Como se ha comentado en el apartado anterior, la transición entre las fases de análisis y diseño en la orientación al objeto es mucho más suave que en las metodologías estructuradas, no habiendo tanta diferencia entre las etapas. Esto implica que es difícil determinar dónde acaba el Análisis Orientado a Objeto (AOO) y donde comienza el Diseño Orientado a Objeto (DOO), siendo la frontera entre ambas fases totalmente inconsistente, de forma que lo que algunos autores incluyen en el AOO otros lo hacen en el DOO. Esto conduce a que sea frecuente el uso de las siglas ADOO para hacer referencia a ambas fases de forma conjunta.

El objetivo del AOO es modelar la semántica del problema en términos de objetos distintos pero relacionados. Por su parte, el DOO conlleva reexaminar las clases del dominio del problema, refinándolas, extendiéndolas y reorganizándolas, para mejorar su reutilización y tomar ventaja de la herencia. El análisis se relaciona con el dominio del problema y el diseño con el dominio de la solución; por lo tanto, el AOO enfoca el problema en los objetos del dominio del problema y el DOO en los objetos del dominio de la solución.

Una vez realizada esta introducción al AOO y al DOO se puede proceder a dar una definición más concreta de ambos procesos.

Se puede definir AOO como el proceso que modela el dominio del problema identificando y especificando un conjunto de objetos semánticos que interaccionan y se comportan de acuerdo a los requisitos del sistema. En el AOO deben llevarse a cabo las siguientes actividades:

- La identificación de clases semánticas, atributos y servicios
- Identificación de las relaciones entre clases (generalizaciones, agregaciones y asociaciones).
- El emplazamiento de las clases, atributos y servicios.
- La especificación del comportamiento dinámico mediante paso de mensajes

Se puede definir DOO como el proceso que modela el dominio de la solución, lo que incluye a las clases semánticas con posibles añadidos, y las clases de interfaz, aplicación y utilidad identificadas durante el diseño. En el DOO deben llevarse a cabo las siguientes actividades:

- Añadir las clases interfaz, base y utilidad.
- Refinar las clases semánticas.

Como resumen final, se podría afirmar que el AOO y el DOO no deben verse como fases muy separadas, siendo recomendable llevarlas a cabo concurrentemente, así el modelo de análisis no puede completarse en ausencia de un modelo de diseño, ni viceversa. Uno de los aspectos más importantes del ADOO es la sinergia entre los dos conceptos.

Conceptos Importantes ADOO

En el caso de la POO, existen diferentes metodologías que consisten en construir un modelo (Representación de la realidad a través de diferentes variables) de un dominio de aplicación como:

OMT que es la Técnica del Modelado de Objetos, el cual a grandes rasgos cuenta con las siguientes cuatro fases: Análisis, Diseño del Sistema, Diseño de Objetos e Implementación.

UML (del que ya hablamos anteriormente), el cual es un Lenguaje Unificado de Modelado y es una representación gráfica que sirve para modelar sistemas orientados a objetos, ya que permite manejar los elementos descritos en los apartados anteriores (por ejemplo, los mensajes entre objetos, sincronización, etc.). Entre sus principales características se encuentran: su flexibilidad y que cuenta con muchos elementos gráficos.

Lenguajes orientados a objetos

Las técnicas, vistas anteriormente en las que se basa la programación orientada a objetos (como el encapsulamiento, abstracción, etc.) ya eran conocidas, desde hace ya varios años, sin embargo, no todos los lenguajes proporcionan todas las facilidades para escribir programas orientados a objetos. Existen discrepancias de cuáles deben de ser estas facilidades y se pueden agrupar en las siguientes:

1. Manejo de memoria automático, incorporándose el concepto del recolector de basura, con lo que la memoria utilizada por objetos cuya utilidad ha terminado es liberada por mecanismos propios del lenguaje, sin intervención del programador.
2. Abstracción de datos a través del lenguaje.
3. Estructura modular en objetos, tomando como base sus estructuras de datos.
4. Clases, Herencia y polimorfismo que puedan manipuladas a través del lenguaje.

Existen lenguajes como C++ y otros lenguajes, como OOCOBOL, OOLISP, OOPROLOG y Object REXX, han sido creados añadiendo extensiones orientadas a objetos a un lenguaje de programación ya existentes.

Una nueva tendencia en la abstracción de paradigmas de programación es la Programación Orientada a Aspectos (POA), pero esta metodología aún se encuentra en proceso, aunque cada vez más y más investigadores e incluso proyectos comerciales en todo el mundo ya la empiezan a adoptar.

Tipos de datos

Los tipos de datos son los siguientes tres:

1. **Primitivos.** - (Son unidades de información tales como caracteres, números o valores booleanos (lógicos).
2. **Clases del sistema.** - No son clases y no tienen métodos propios.
3. **Clases definidas por el usuario.** - Como su nombre lo indica son hechas por el usuario.

Los tipos primitivos son:

1. boolean.- Los cuales pueden tener valores de falso o verdadero (true/false).
2. char.-Es un carácter de 16 bits.
3. byte, short.- Cuenta con 8, 16,32 y 64 bits
4. int, long.- Tiene valores entero y enteros largos.
5. Float, double.- Son números de punto flotante (notación científica) de 32 y 64 bits.

Otra forma de entender la clasificación en tipos de datos es:

1. Tipos de datos primitivos: No son objetos) nos damos cuenta porque no tienen Propiedades, métodos, y no necesitan instanciarse. además, que comienzan con minúscula.
2. Tipos de datos que son Objetos: Los podemos reconocer porque podemos ver sus métodos y propiedades, estos objetos no necesitan ser INSTANCIADOS y se llaman objetos Estáticos. como por ejemplo los Integer y String, si bien es un tipo de datos similar a int notemos que empieza con mayúscula por ser una clase
3. También están los tipos de datos Objeto: Que SI NECESITAN SER INSTANCIADOS para poder ver sus métodos y propiedades. En la ayuda ya notamos que implementa otras clases.

¿Pero cómo vemos los métodos y propiedades de estos Objetos?

Para poder ver los métodos y propiedades debemos:

1. DECLARARLO (ya accedemos a sus métodos y propiedades)
2. INSTANCIARLO (creando una nueva instancia o nueva copia del objeto para poder usarlo) para poder utilizarlo

Ahora, es necesario que conozcamos otro concepto importante: Lenguajes tipados o tipificados

Los lenguajes tipificados (también llamados de tipado estático) son aquellos en los que una variable guarda siempre un mismo tipo de datos. En algunos lenguajes tipificados se exige al programador que declare el tipo de cada variable y en otros lo determina el compilador. Es bastante obvio que, si tienes que hacer un programa eficiente necesitas un lenguaje de tipado, porque hace un buen uso de la memoria RAM de tu computadora. ¿Pero, y con los lenguajes que no se usan para aplicaciones que necesiten esta eficiencia? Ahí es donde aparecen los lenguajes no tipados, donde no tienes que declarar explícitamente el tipo de dato que vas a usar, porque el lenguaje se hará cargo.

Entonces, la diferencia es que en uno tipado, tienes que manejar el tipo de dato de tus variables, mientras que en uno no tipado no es que no se manejan los tipos de datos, sino que dejas que el lenguaje de programación los maneje.

Cada uno tiene sus ventajas y desventajas, es obvio que al tener que manejar los tipos de datos, la programación es más compleja, pero es más eficiente, si no es tipado quizás tu aplicación no sea más rápida, pero será más fácil de programar.

Ahora preguntémonos ¿Qué paradigma es Javascript?, ¿qué paradigma es Java y que paradigma es angular? ¿Es tipado o no? Haciendo [clic aquí](#) encontrarás algunas respuestas.

Modelado de objetos - UML

Los objetos identificados en la etapa de análisis se plasman en diferentes modelos, combatiendo de esta forma la complejidad inherente del problema al que nos estamos enfrentando. Para la construcción de modelos de objetos, y modelos software en general, se debe contar con un lenguaje de modelado, es decir, un lenguaje para especificar, visualizar, construir y documentar los componentes de los sistemas software.

Anteriormente dimos una pequeña introducción a UML. Este lenguaje de modelado que se había convertido en estándar de facto de las notaciones en orientación a objeto.

El hecho de la estandarización de la notación de UML nos conduce hacia una notación única en la que expresar las abstracciones identificadas en análisis, así como sus relaciones. Además, esos mismos modelos serán refinados durante la fase de análisis y diseño hasta que estén preparados para su implementación en el lenguaje de programación elegido.

A la hora de realizar un modelo de un sistema software, éste debe hacerse desde diferentes puntos de vista, de forma que recojan tanto la dimensión estática y estructural de los objetos como su componente dinámica. Un lenguaje de modelado debe aportar una serie de mecanismos gráficos que permitan captar la esencia de un sistema desde diversas perspectivas. Estos mecanismos gráficos suelen recibir el nombre de diagramas en la mayoría de los lenguajes de modelado.

UML

Como ya hemos presentado antes, el Lenguaje Unificado de Modelado (UML) fue creado para forjar un lenguaje de modelado visual común y semántica y sintácticamente rico para la arquitectura, el diseño y la implementación de sistemas de software complejos, tanto en estructura como en comportamiento. Es comparable a los planos usados en otros campos y consiste en diferentes tipos de diagramas. En general, los diagramas UML describen los límites, la estructura y el comportamiento del sistema y los objetos que contiene.

Hay muchos paradigmas o modelos para la resolución de problemas en la informática, que es el estudio de algoritmos y datos. Hay cuatro categorías de modelos para la resolución de problemas: lenguajes imperativos, funcionales, declarativos y orientados a objetos (OOP). Como ya vimos, en los lenguajes orientados a objetos, los algoritmos se expresan definiendo 'objetos' y haciendo que los objetos interactúen entre sí. Esos objetos son cosas que deben ser manipuladas y existen en el mundo real. Pueden ser edificios, artefactos sobre un escritorio o seres humanos. Los lenguajes orientados a objetos dominan el mundo de la programación porque modelan los objetos del mundo real. UML es una combinación de varias notaciones orientadas a objetos:

1. Diseño orientado a objetos
2. Técnica de modelado de objetos
3. Ingeniería de software orientada a objetos

UML usa las fortalezas de estos tres enfoques para presentar una metodología más uniforme que sea más sencilla de usar. UML representa buenas prácticas para la construcción y documentación de diferentes aspectos del modelado de sistemas de software y de negocios.

El OMG define los propósitos de UML de la siguiente manera:

Brindar a arquitectos de sistemas, ingenieros y desarrolladores de software las herramientas para el análisis, el diseño y la implementación de sistemas basados en software, así como para el modelado de procesos de negocios y similares.

Hacer progresar el estado de la industria permitiendo la interoperabilidad de herramientas de modelado visual de objetos. No obstante, para habilitar un intercambio significativo de información de modelos entre herramientas, se requiere de un acuerdo con respecto a la semántica y notación.

UML cumple con los siguientes requerimientos:

1. Establecer una definición formal de un metamodelo común basado en el estándar MOF (Meta-Object Facility) que especifique la sintaxis abstracta del UML. La sintaxis abstracta define el conjunto de conceptos de modelado UML, sus atributos y sus relaciones, así como las reglas de combinación de estos conceptos para construir modelos UML parciales o completos.
2. Brindar una explicación detallada de la semántica de cada concepto de modelado UML. La semántica define, de manera independiente a la tecnología, cómo los conceptos UML se habrán de desarrollar por las computadoras.
3. Especificar los elementos de notación de lectura humana para representar los conceptos individuales de modelado UML, así como las reglas para combinarlos en una variedad de diferentes tipos de diagramas que corresponden a diferentes aspectos de los sistemas modelados.
4. Definir formas que permitan hacer que las herramientas UML cumplan con esta especificación. Esto se apoya (en una especificación independiente) con una especificación basada en XML de formatos de intercambio de

modelos correspondientes (XMI) que deben ser concretados por herramientas compatibles.

Para más información **Sitio oficial UML:** <https://www.uml.org/>

Conceptos de modelado específico

El desarrollo de sistemas se centra en tres modelos generales de sistemas diferentes:

1. **Funcionales:** Se trata de diagramas de casos de uso que describen la funcionalidad del sistema desde el punto de vista del usuario.
2. **De objetos:** Se trata de diagramas de clases que describen la estructura del sistema en términos de objetos, atributos, asociaciones y operaciones.
3. **Dinámicos:** Los diagramas de interacción, los diagramas de máquina de estados y los diagramas de actividades se usan para describir el comportamiento interno del sistema.

Conceptos orientados a objetos en UML

Los objetos en UML son entidades del mundo real que existen a nuestro alrededor. En el desarrollo de software, los objetos se pueden usar para describir, o modelar, el sistema que se está creando en términos que sean pertinentes para el dominio. Los objetos también permiten la descomposición de sistemas complejos en componentes comprensibles que permiten que se construya una pieza a la vez.

Estos son algunos conceptos fundamentales de un mundo orientado a objetos:

1. **Objetos:** Representan una entidad y el componente básico.
2. **Clase:** Plano/molde de un objeto.
3. **Abstracción:** Comportamiento de una entidad del mundo real.
4. **Encapsulación:** Mecanismo para enlazar los datos y ocultarlos del mundo exterior.
5. **Herencia:** Mecanismo para crear nuevas clases a partir de una existente.
6. **Polimorfismo:** Define el mecanismo para salidas en diferentes formas.

Tipos de diagramas UML

UML usa elementos y los asocia de diferentes formas para formar diagramas que representan aspectos estáticos o estructurales de un sistema, y diagramas de comportamiento, que captan los aspectos dinámicos de un sistema. Entonces, los modelos de sistemas se visualizan a través de dos tipos diferentes de diagramas: estructurales y de comportamiento.

Diagramas UML estructurales

Diagrama de clases: El diagrama UML más comúnmente usado, y la base principal de toda solución orientada a objetos. Las clases dentro de un sistema, atributos y operaciones, y la relación entre cada clase. Las clases se agrupan para crear diagramas de clases al crear diagramas de sistemas grandes.

Diagrama de objetos: Muestra la relación entre objetos por medio de ejemplos del mundo real e ilustra cómo se verá un sistema en un momento dado. Dado que los datos están disponibles dentro de los objetos, estos pueden usarse para clarificar relaciones entre objetos.

Diagrama de componentes: Muestra la relación estructural de los elementos del sistema de software, muy frecuentemente empleados al trabajar con sistemas complejos con componentes múltiples. Los componentes se comunican por medio de interfaces.

Diagrama de estructura compuesta: Los diagramas de estructura compuesta se usan para mostrar la estructura interna de una clase.

Diagrama de implementación: Ilustra el hardware del sistema y su software. Útil cuando se implementa una solución de software en múltiples máquinas con configuraciones únicas.

Diagrama de paquetes: Hay dos tipos especiales de dependencias que se definen entre paquetes: la importación de paquetes y la fusión de paquetes. Los paquetes pueden representar los diferentes niveles de un sistema para revelar la arquitectura. Se pueden marcar las dependencias de paquetes para mostrar el mecanismo de comunicación entre niveles.

Diagramas UML de comportamiento

Diagrama de secuencia: Muestra cómo los objetos interactúan entre sí y el orden de la ocurrencia. Representan interacciones para un escenario concreto.

Diagramas de actividades: Flujos de trabajo de negocios u operativos representados gráficamente para mostrar la actividad de alguna parte o componente del sistema. Los diagramas de actividades se usan como una alternativa a los diagramas de máquina de estados.

Diagrama de comunicación: Similar a los diagramas de secuencia, pero el enfoque está en los mensajes que se pasan entre objetos. La misma información se puede representar usando un diagrama de secuencia y objetos diferentes.

Diagrama de panorama de interacciones: Hay siete tipos de diagramas de interacciones. Este diagrama muestra la secuencia en la cual actúan.

Diagrama de máquina de estados: Similar a los diagramas de actividades, describen el comportamiento de objetos que se comportan de diversas formas en su estado actual.

Diagrama de temporización: Al igual que en los diagramas de secuencia, se representa el comportamiento de los objetos en un período de tiempo dado. Si hay un solo objeto, el diagrama es simple. Si hay más de un objeto, las interacciones de los objetos se muestran durante ese período de tiempo particular.

Diagrama de caso de uso: Representa una funcionalidad particular de un sistema. Se crea para ilustrar cómo se relacionan las funcionalidades con sus controladores (actores) internos/externos.

Diagramas Estructurales

Diagrama de Clases

Recientemente mencionamos que los diagramas de clases son los más comúnmente usados, y son la base principal de toda solución orientada a objetos.

Beneficios de los diagramas de clases

Los diagramas de clases ofrecen una serie de beneficios para toda organización:

1. Ilustrar modelos de datos para sistemas de información, sin importar qué tan simples o complejos sean.
2. Comprender mejor la visión general de los esquemas de una aplicación.
3. Expresar visualmente cualquier necesidad específica de un sistema y divulgar esa información en toda la empresa.
4. Crear diagramas detallados que resalten cualquier código específico que será necesario programar e implementar en la estructura descrita.
5. Ofrecer una descripción independiente de la implementación sobre los tipos empleados en un sistema que son posteriormente transferidos entre sus componentes.

Componentes básicos de un diagrama de clases

El diagrama de clases estándar está compuesto por tres partes:

1. Sección superior: Contiene el nombre de la clase. Esta sección siempre es necesaria, ya sea que estés hablando del clasificador o de un objeto.
2. Sección central: Contiene los atributos de la clase. Usa esta sección para describir cualidades de la clase. Esto solo es necesario al describir una instancia específica de una clase.
3. Sección inferior: Incluye operaciones de clases (métodos). Esto está organizado en un formato de lista. Cada operación requiere su propia línea. Las operaciones describen cómo una clase puede interactuar con los datos.

Modificadores de acceso a miembros

Todas las clases poseen diferentes niveles de acceso en función del modificador de acceso (visibilidad). A continuación, te mostramos los niveles de acceso con sus símbolos correspondientes: Público (+), Privado (-), Protegido (#), Paquete (~), Derivado (/), Estático (subrayado)

Alcance de los miembros

Hay dos alcances para los miembros: clasificadores e instancias. Los clasificadores son miembros estáticos, mientras que las instancias son las instancias específicas de la clase. Si estás familiarizado con POO, esto no es nada nuevo.

Componentes adicionales del diagrama de clases

En función del contexto, las clases de un diagrama de clases pueden representar los objetos principales, las interacciones en la aplicación o las clases que se programarán. Para responder la pregunta "¿Qué es un diagrama de clases en UML?", primero deberías comprender su composición básica.

1. **Clases:** Una plantilla para crear objetos e implementar un comportamiento en un sistema. En UML, una clase representa un objeto o un conjunto de objetos que comparte una estructura y un comportamiento comunes. Se representan con un rectángulo que incluye filas del nombre de la clase, sus atributos y sus operaciones. Al dibujar una clase en un diagrama de clases, solo se debe cumplimentar la fila superior. Las otras son opcionales y se usan si deseas agregar más detalles.
2. **Nombre:** La primera fila en una figura de clase.
3. **Atributos:** La segunda fila en una figura de clase. Cada atributo de una clase está ubicado en una línea separada.
4. **Métodos:** La tercera fila en una figura de clase. También conocidos como "operaciones", los métodos se organizan en un formato de lista donde cada operación posee su propia línea.
5. **Señales:** Símbolos que representan comunicaciones unidireccionales y asíncronas entre objetos activos.
6. **Tipos de datos:** Clasificadores que definen valores de datos. Los tipos de datos pueden modelar tanto enumeraciones como tipos primitivos.
7. **Paquetes:** Figuras diseñadas para organizar clasificadores relacionados en un diagrama. Se simbolizan con una figura de un gran rectángulo con pestañas.
8. **Interfaces:** Una recopilación de firmas de operaciones o de definiciones de atributo que define un conjunto uniforme de comportamientos. Las interfaces son similares a una clase, excepto por que una clase puede tener una instancia de su tipo, y una interfaz debe poseer, como mínimo, una clase para implementarla.
9. **Enumeraciones:** Representaciones de tipos de datos definidos por el usuario. Una enumeración incluye grupos de identificadores que representan valores de la enumeración.

10. **Objetos:** Instancias de una clase o clases. Los objetos se pueden agregar a un diagrama de clases para representar instancias prototípicas o concretas.
11. **Artefactos:** Elementos modelo que representan las entidades concretas de un sistema de software, como documentos, bases de datos, archivos ejecutables, componentes de software y más.
12. **Interacciones:** El término "interacciones" se refiere a múltiples relaciones y enlaces que pueden existir en diagramas de objetos y de clases. Algunas de las interacciones más comunes incluyen

Interacción de diagramas de clases de herencia

Como ya vimos anteriormente, Herencia refiere al proceso en el que una subclase o clase derivada recibe la funcionalidad de una superclase o clase principal, también se conoce como "generalización". Se simboliza mediante una línea de conexión recta con una punta de flecha cerrada que señala a la superclase.

Veamos un ejemplo: el objeto "Auto" heredaría todos los atributos (velocidad, números de pasajeros, combustible) y los métodos (arrancar(), frenar(), cambiarDirección()) de la clase principal ("Vehículo"), además de los atributos específicos (tipo de modelo, número de puertas, fabricante del auto) y métodos de su propia clase (Radio(), limpiaparabrisas(), aireacondicionado/calefacción()). La herencia se muestra en un diagrama de clases por medio de una línea continua con una flecha cerrada y vacía.

Asociación bidireccional: La relación predeterminada entre dos clases. Ambas clases están conscientes una de la otra y de la relación que tienen entre sí. Esta asociación se representa mediante una línea recta entre dos clases.

En el ejemplo anterior, la clase Auto y la clase Viaje están interrelacionadas. En un extremo de la línea, el Auto recibe la asociación de "autoAsignado" con el valor de multiplicidad de 0..1, de modo que cuando la instancia de Viaje existe, puede tener una instancia de Auto asociada a ella o no tener instancias de Autos asociadas a ella. En este caso, una clase CasaRodante separada con un valor de multiplicidad de 0..* es necesaria para demostrar que un Viaje puede tener múltiples instancias de Autos asociadas a ella. Dado que una instancia de Auto podría tener múltiples asociaciones "iniciarViaje", en otras palabras, un auto podría realizar múltiples viajes, el valor de multiplicidad se establece en 0..*

Asociación unidireccional: Una relación un poco menos común entre dos clases. Una clase está consciente de la otra e interactúa con ella. La asociación unidireccional se dibuja con una línea de conexión recta que señala una punta de flecha abierta desde la clase "knowing" a la clase "known".

Como ejemplo, en tu viaje por Arizona, podrías encontrarte con una trampa de velocidad donde un radar de tráfico registra la velocidad a la que conducías, pero no lo sabrás hasta que recibas la notificación por correo. Esto no está dibujado en la imagen, pero en este caso, el valor de multiplicidad sería 0..* en función de cuántas veces hayas conducido frente al radar de tráfico.

Diagrama de objetos

Diagrama de objetos Muestra la relación entre objetos por medio de ejemplos del mundo real e ilustra cómo se verá un sistema en un momento dado.

Elementos del diagrama de objetos

Los diagramas de objetos son sencillos de crear ya que se componen de objetos, representados por rectángulos, conectados mediante líneas. Echa un vistazo a los elementos principales de un diagrama de objetos.

1. **Objetos:** Los objetos son instancias de una clase. Por ejemplo, si "coche" es una clase, un Altima 2007 de Nissan es un objeto de una clase.
2. **Títulos de clases:** Los títulos de clases son los atributos específicos de una clase dada. En el diagrama de objetos de árbol genealógico, los títulos de clases incluyen nombre, género y edad de los integrantes de la familia. Se pueden listar títulos de clases como elementos en el objeto o incluso en las propiedades del propio objeto (como el color).
3. **Atributos de clases:** Los atributos de clases se representan por medio de un rectángulo con dos pestañas que indica un elemento de software.
4. **Enlaces:** Los enlaces son líneas que conectan dos figuras de un diagrama de objetos entre sí. El diagrama de objetos corporativo siguiente muestra cómo los departamentos están conectados al estilo del organigrama tradicional.

Aplicaciones del diagrama de objetos

A un desarrollador le resultarán útiles los diagramas de objetos en muchos de los casos. Dichos casos incluyen:

1. Revisión de una iteración específica de un sistema general.
2. Obtención de una vista de nivel alto del sistema que desarrollarás.
3. Prueba de un diagrama de clases que creaste para la estructura general del sistema, por medio de diagramas de objetos para casos de uso específicos.

Diagramas de Comportamiento

Diagrama de secuencias

Muestra cómo los objetos interactúan entre sí y el orden de la ocurrencia. Representan interacciones para un escenario concreto.

Los beneficios de los diagramas de secuencia

Los diagramas de secuencia pueden ser referencias útiles para las empresas y otras organizaciones. Prueba dibujar un diagrama de secuencia para:

1. Representa los detalles de un caso de uso en UML.
2. Modelar la lógica de una operación, una función o un procedimiento sofisticados.
3. Ve cómo los objetos y los componentes interactúan entre sí para completar un proceso.

4. Planificar y comprender la funcionalidad detallada de un escenario actual o futuro.

Componentes y símbolos básicos

Para comprender qué es un diagrama de secuencia, debes estar familiarizado con sus símbolos y componentes. Los diagramas de secuencia están formados por los siguientes elementos e íconos:

1. **Símbolo de objeto:** Representa una clase u objeto en UML. El símbolo objeto demuestra cómo se comportará un objeto en el contexto del sistema. Los atributos de las clases no deben aparecer en esta figura.
2. **Casilla de activación:** Representa el tiempo necesario para que un objeto finalice una tarea. Cuanto más tiempo lleve la tarea, más larga será la casilla de activación.
3. **Símbolo de actor:** Muestra entidades que interactúan con el sistema o que son externas al sistema.
4. **Símbolo de paquete:** Se usa en notación UML 2.0 para contener los elementos interactivos del diagrama. También conocida como "marco", esta figura rectangular tiene un pequeño rectángulo interior para etiquetar el diagrama.
5. **Símbolo de línea de vida:** Representa el paso del tiempo a medida que se extiende hacia abajo. Esta línea vertical discontinua representa eventos secuenciales que le ocurren a un objeto durante el proceso graficado. Las líneas de vida pueden comenzar con una figura rectangular etiquetada o un símbolo de actor.
6. **Símbolo de bucle de opción:** Se emplea para modelar escenarios del tipo "Si... entonces...", es decir, una circunstancia que solo sucederá en determinadas condiciones.
7. **Símbolo de alternativas:** Simboliza una decisión (que, por lo general, es mutuamente exclusiva) entre dos o más secuencias de mensajes. Para representar alternativas, emplea la figura rectangular etiquetada con una línea discontinua en su interior.
8. **Símbolos comunes de mensajes:** Usa los siguientes símbolos de mensaje y flechas para indicar cómo se transmite la información entre objetos. Estos símbolos pueden reflejar el inicio y la ejecución de una operación o el envío y la recepción de una señal.
 - Símbolo de mensaje asincrónico
 - Símbolo de mensaje sincrónico
 - Símbolo de mensaje de respuesta asincrónico
 - Símbolo de crear mensaje asincrónico
 - Símbolo de mensaje de respuesta
 - Símbolo de eliminar mensaje

Glosario – Recomendaciones UML

Glosario de términos de UML

Familiarízate con el vocabulario de UML, con esta lista extraída del documento UML, cuya finalidad es ayudar a quienes no son miembros de OMG a entender los términos comúnmente usados.

Compatibilidad con sintaxis abstracta: Los usuarios pueden mover modelos a través de diferentes herramientas, incluso si usan diferentes notaciones.

Metamodelo de almacén común (CWM): Interfaces estándares que se usan para permitir el intercambio de metadatos de almacén e inteligencia de negocios entre herramientas de almacén, plataformas de almacén y repositorios de metadatos de almacén en entornos heterogéneos distribuidos.

Compatibilidad con sintaxis concreta: Los usuarios pueden continuar usando una notación con la que estén familiarizados a través de diferentes herramientas.

Núcleo: En el contexto de UML, el núcleo comúnmente se refiere al "paquete central", que es un metamodelo completo particularmente diseñado para una alta reutilización.

Unidad de lenguaje: Consiste en una colección de conceptos de modelado estrechamente vinculados que proporciona a los usuarios la capacidad de representar aspectos del sistema en estudio según un paradigma o formalismo en particular.

Nivel 0 (L0): Nivel de cumplimiento inferior para la infraestructura UML - una sola unidad de lenguaje que hace posible el modelado de tipos de estructuras basadas en clases que se encuentran en los lenguajes más populares de programación orientados a objetos.

Meta Object Facility (MOF): Una especificación de modelado de OMG que brinda la base para las definiciones de metamodelos en la familia de lenguajes MDA de OMG.

Metamodelo: Define el lenguaje y los procesos a partir de los cuales formar un modelo.

Construcciones de metamodelos (LM): Segundo nivel de cumplimiento en la infraestructura UML - una unidad adicional de lenguaje para estructuras más avanzadas basadas en clases, usadas para construir metamodelos (por medio de CMOF), tales como el UML mismo. UML solo tiene dos niveles de cumplimiento.

Arquitectura dirigida por modelos (MDA): Un enfoque y un plan para lograr un conjunto coherente de especificaciones de tecnología dirigida por modelos.

Lenguaje de restricciones para objetos (OCL): Un lenguaje declarativo para describir reglas que se aplican al Lenguaje Unificado de Modelado. OCL complementa a UML proporcionando términos y símbolos de diagramas de flujo que son más precisos que el lenguaje natural, pero menos difíciles de dominar que las matemáticas.

Object Management Group (OMG): Es un consorcio sin fines de lucro de especificaciones para la industria de la computación, cuyos miembros definen y mantienen la especificación UML.

UML 1: Primera versión del Lenguaje Unificado de Modelado.

Lenguaje Unificado de Modelado (UML): Un lenguaje visual para especificar, construir y documentar los artefactos de los sistemas.

XMI: Una especificación basada en XML de formatos de intercambio de modelos correspondientes.

Recomendaciones

Aunque cada programador puede definir su propio estilo de código, una buena práctica es seguir el estilo utilizado por los diseñadores del lenguaje pues de seguir esta práctica será mucho más fácil analizar códigos de otros y a su vez que otros programadores analicen y comprendan nuestros códigos.

1. Evitar en lo posible líneas de longitud superior a 80 caracteres.
2. Indentar los bloques de código.
3. Los operadores se separarán siempre de los operandos por un carácter "espacio". La única excepción para esta regla es el operador de acceso a los componentes de objetos y clases (".") que no será separado.
4. Utilizar identificadores nemotécnicos, es decir, utilizar nombres simbólicos adecuados para los identificadores lo suficientemente autoexplicativos por sí mismos para dar una orientación de su uso o funcionalidad de manera tal que podamos hacer más claros y legibles nuestros códigos.
5. Los identificadores para constantes se especificarán siempre en mayúsculas.
6. Los identificadores de variables y parámetros se especificarán siempre comenzando en minúsculas, si se compone de más de un nombre, entonces el segundo comenzará con mayúscula.
7. Los identificadores de clases, métodos y propiedades se especificarán siempre comenzando en mayúsculas, si se compone de más de un nombre, entonces el segundo también comenzará con mayúscula.
8. Utilizar comentarios, pero éstos seguirán un formato general de fácil portabilidad y que no incluya líneas completas de caracteres repetidos. Los que se coloquen dentro de bloques de código deben aparecer en una línea independiente indentada de igual forma que el bloque de código que describen.

¿Qué son los Patrones de Diseño?

Breve reseña

¿Quién inventó los patrones de diseño? Esa es una buena pregunta, aunque algo imprecisa. Los patrones de diseño no son conceptos opacos y sofisticados, al contrario. Los patrones son soluciones habituales a problemas comunes en el diseño orientado a objetos. Cuando una solución se repite una y otra vez en varios proyectos, al final alguien le pone un nombre y explica la solución en detalle. Básicamente, así es como se descubre un patrón.

El concepto de los patrones fue descrito por Christopher Alexander en El lenguaje de patrones. El libro habla de un “lenguaje” para diseñar el entorno urbano. Las unidades de este lenguaje son los patrones. Pueden describir lo altas que tienen que ser las ventanas, cuántos niveles debe tener un edificio, cuan grandes deben ser las zonas verdes de un barrio, etcétera.

La idea fue recogida por cuatro autores: Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm. En 1995, publicaron Patrones de diseño, en el que aplicaron el concepto de los patrones de diseño a la programación. El libro presentaba 23 patrones que resolvían varios problemas del diseño orientado a objetos y se convirtió en un éxito de ventas con rapidez. Al tener un título tan largo en inglés, la gente empezó a llamarlo “el libro de la ‘gang of four’ (banda de los cuatro)”, lo que pronto se abrevió a “el libro GoF”.

Desde entonces se han descubierto decenas de nuevos patrones orientados a objetos. La “metodología del patrón” se hizo muy popular en otros campos de la programación, por lo que hoy en día existen muchos otros patrones no relacionados con el diseño orientado a objetos.

¿Qué es un patrón de diseño?

Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código.

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular. Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

A menudo los patrones se confunden con algoritmos porque ambos conceptos describen soluciones típicas a problemas conocidos. Mientras que un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, un patrón es una descripción de más alto nivel de una solución. El código del mismo patrón aplicado a dos programas distintos puede ser diferente.

Una analogía de un algoritmo sería una receta de cocina: ambos cuentan con pasos claros para alcanzar una meta. Por su parte, un patrón es más similar a un

plano, ya que puedes observar cómo son su resultado y sus funciones, pero el orden exacto de la implementación depende de ti.

¿Qué es el patrón MVC?

MVC era inicialmente un patrón arquitectural, un modelo o guía que expresa cómo organizar y estructurar los componentes de un sistema software, sus responsabilidades y las relaciones existentes entre cada uno de ellos. Su nombre, MVC, parte de las iniciales de Modelo-Vista-Controlador (*Model-View-Controller*, en inglés), que son las capas o grupos de componentes en los que organizaremos nuestras aplicaciones bajo este paradigma.

Es a menudo considerado también un patrón de diseño de la capa de presentación, pues define la forma en que se organizan los componentes de presentación en sistemas distribuidos.

Modelo:

En la capa Modelo encontraremos siempre una representación de los datos del dominio, es decir, aquellas entidades que nos servirán para almacenar información del sistema que estamos desarrollando. Por ejemplo, si estamos desarrollando una aplicación de facturación, en el modelo existirán las clases Factura, Cliente o Proveedor, entre otras.

La Vista:

Los componentes de la Vista son los responsables de generar la interfaz de nuestra aplicación, es decir, de componer las pantallas, páginas, o cualquier tipo de resultado utilizable por el usuario o cliente del sistema. De hecho, suele decirse que la Vista es una representación del estado del Modelo en un momento concreto y en el contexto de una acción determinada.

Controlador:

La misión principal de los componentes incluidos en el Controlador es actuar como intermediarios entre el usuario y el sistema. Serán capaces de capturar las acciones de este sobre la Vista, como puede ser la pulsación de un botón o la selección de una opción de menú, interpretarlas y actuar en función de ellas. Por ejemplo, retornando al usuario una nueva vista que represente el estado actual del sistema, o invocando a acciones definidas en el Modelo para consultar o actualizar información.

Design Patterns – Clasificación

¿En qué consiste el patrón?

La mayoría de los patrones se describe con mucha formalidad para que la gente pueda reproducirlos en muchos contextos. Aquí tienes las secciones que suelen estar presentes en la descripción de un patrón:

- El propósito del patrón explica brevemente el problema y la solución.

- La motivación explica en más detalle el problema y la solución que brinda el patrón.
- La estructura de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.
- El ejemplo de código en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.

Algunos catálogos de patrones enumeran otros detalles útiles, como la aplicabilidad del patrón, los pasos de implementación y las relaciones con otros patrones.

Clasificación de los patrones

Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad al sistema completo que se diseña. Nos gusta la analogía de la construcción de carreteras: puedes hacer más segura una intersección instalando semáforos o construyendo un intercambiador completo de varios niveles con pasajes subterráneos para peatones.

1. Los patrones más básicos y de más bajo nivel suelen llamarse idioms. Normalmente se aplican a un único lenguaje de programación.
2. Los patrones más universales y de más alto nivel son los patrones de arquitectura. Los desarrolladores pueden implementar estos patrones prácticamente en cualquier lenguaje. Al contrario que otros patrones, pueden utilizarse para diseñar la arquitectura de una aplicación completa.

Además, todos los patrones pueden clasificarse por su propósito:

1. **Los patrones creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
2. **Los patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
3. **Los patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

Patrones Creacionales

Los patrones creacionales proporcionan varios mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente.

Factory Method: es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

Prototype: es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

Abstract Factory: es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

Builder: es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

Singleton: es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Patrones Estructurales

Patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.

Adapter: es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

Bridge: es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

Composite: es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

Decorator: es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

Facade: es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

Flyweight: es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.

Proxy: es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

Patrones de Comportamiento

Patrones de comportamiento se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

Chain of Responsibility: es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

Command: es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

Iterator es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

Mediator: es un patrón de diseño de comportamiento que te permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

Memento: es un patrón de diseño de comportamiento que te permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

Observer: es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

State: es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.

Strategy: es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

Template Method: es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclasses sobrescriban pasos del algoritmo sin cambiar su estructura.

Visitor: es un patrón de diseño de comportamiento que te permite separar algoritmos de los objetos sobre los que operan.

Conclusiones y críticas

¿Por qué debería aprender sobre patrones?

La realidad es que podríamos trabajar durante años como programadores sin conocer un solo patrón. Mucha gente lo hace. Incluso en ese caso, podrías estar implementando patrones sin saberlo. Así que, ¿por qué dedicar tiempo a aprenderlos?

Los patrones de diseño son un juego de herramientas de soluciones comprobadas a problemas habituales en el diseño de software. Incluso aunque nunca te encuentres con estos problemas, conocer los patrones sigue siendo de utilidad, porque te enseña a resolver todo tipo de problemas utilizando principios del diseño orientado a objetos.

Los patrones de diseño definen un lenguaje común que puedes utilizar con tus compañeros de equipo para comunicarlos de forma más eficiente. Podrías decir: “Oh, utiliza un singleton para eso”, y todos entenderían la idea de tu sugerencia. No habría necesidad de explicar qué es un singleton si conocen el patrón y su nombre.

Crítica de los patrones

¿Por qué siempre es importante conocer todas las perspectivas?, veamos los aspectos que pueden ser negativos con el uso de Patrones de diseño. Da la sensación de que todos los holgazanes han criticado ya los patrones de diseño. Veamos los argumentos más habituales contra el uso de los patrones.

Resultados deficientes para un lenguaje de programación débil

Normalmente, la necesidad por los patrones surge cuando la gente elige un lenguaje de programación o una tecnología que carece del nivel necesario de abstracción. En este caso, los patrones se convierten en un artilugio que otorga al lenguaje unas súper habilidades muy necesitadas.

Por ejemplo, el patrón Strategy puede implementarse con una simple función anónima (lambda) en la mayoría de lenguajes de programación modernos.

Soluciones ineficientes

Los patrones intentan sistematizar soluciones cuyo uso ya es generalizado. Esta unificación es vista por muchos como un dogma, e implementan los patrones “al pie de la letra”, sin adaptarlos al contexto del proyecto particular.

Uso injustificado

Si lo único que tienes es un martillo, todo te parecerá un clavo.

Este es el problema que persigue a muchos principiantes que acaban de familiarizarse con los patrones. Una vez que aprenden sobre patrones, intentan

aplicarlos en todas partes, incluso en situaciones en las que un código más simple funcionaría perfectamente bien.