

Diseño del modelo conceptual bajo el paradigma orientado a objetos

Breve descripción:

En este componente formativo podrá abordar saberes relacionados con el Lenguaje de Modelado Unificado, diagramas de clase y el uso de herramientas Case que le servirán para crear modelos fundamentales para visualizar, especificar y construir “software”. Así mismo, se desarrollarán temáticas relacionadas a la implementación en el lenguaje de programación Java, de clases, objetos, atributos, constructores, métodos, herencia y relaciones.

Tabla de contenido

| | |
|---|----|
| Introducción..... | 1 |
| 1. Introducción a UML | 4 |
| 2. Diagramas de clase | 6 |
| 2.1. Control de acceso | 8 |
| 2.2. Asociaciones | 9 |
| 2.3. Herencia..... | 13 |
| 2.4. Agregación | 20 |
| 2.5. Composición | 24 |
| 3. Herramientas CASE..... | 28 |
| 4. Características del lenguaje de programación orientada a objetos | 33 |
| 5. Implementación de clases y objetos | 37 |
| 5.1. Atributos y métodos de la clase..... | 40 |
| 5.2. Constructores y destructores..... | 45 |
| 5.3. Métodos de acceso y modificadores | 48 |
| 5.4. Sobrecarga de métodos (“overload”)..... | 53 |
| 6. Comunicación entre clases | 55 |
| 6.1. Implementación de asociaciones | 55 |
| 6.2. Implementación de composición/agregación | 57 |

| | |
|---------------------------------------|----|
| 6.3. Implementación de herencia | 59 |
| Síntesis | 65 |
| Material complementario..... | 66 |
| Glosario..... | 68 |
| Referencias bibliográficas | 69 |
| Créditos..... | 70 |

Introducción

Para la construcción de “software” se hace evidente la importancia de la utilización de modelos ya que representan el aspecto y la dirección de una necesidad. Esa “necesidad” puede estar en un estado de desarrollo o en un estado de planeación. Los diseñadores del modelo deben realizar un levante de los requerimientos del producto y dichos requerimientos pueden incluir áreas tales como funcionalidad, “performance” y confiabilidad. El modelo se divide en un número de vistas, cada una de las cuales describe un aspecto específico del producto o sistema en construcción. Un modelo permite documentar la estructura y conducta de un sistema antes de que sea codificado. En el siguiente video se expone la importancia de la programación orientada a objetos (POO) para modelar soluciones de “software”.

Video 1. Diseño del modelo conceptual bajo el paradigma orientado a objetos



[Enlace de reproducción del video](#)

Síntesis del video: Diseño del modelo conceptual bajo el paradigma orientado a objetos

La programación basada en objetos es el estilo de programación que soporta principalmente la encapsulación e identidad de objetos.

La programación orientada a objetos (POO) se implementa cuando se aplican las características de abstracción, encapsulación, herencia y polimorfismo. La POO encierra datos (atributos) y métodos (comportamientos) que hacen parte del objeto y que se encuentran relacionados entre sí.

Los objetos tienen la propiedad de ocultar la información. Lo que quiere decir que, aunque los objetos pueden saber cómo comunicarse entre sí, a través de interfaces bien definidas, por lo general a los objetos no se les permite saber cómo se implementan otros objetos. Por ejemplo, con toda seguridad es posible conducir una moto de manera efectiva sin conocer los detalles de cómo funcionan internamente los sistemas del motor, la transmisión y el escape.

De acuerdo con Deitel & Deitel (2004), lo anterior es la razón de que el ocultamiento de información sea tan importante para la buena ingeniería de “software” en los lenguajes de programación y, por ende, para la modelación de soluciones.

Este componente formativo también dará un recorrido a las características del lenguaje de programación orientada a objetos, las implementaciones de clases y objetos, y en su interior se reconocerán los atributos, métodos de las clases, los constructores y destructores, los métodos de acceso y modificación, junto con la

sobrecarga de métodos. Esto permitirá conocer con la comunicación entre clases desde su implementación de asociaciones, composición / agregación y la implementación de herencia.

1. Introducción a UML

UML es una técnica para la especificación de sistemas en todas sus fases. Nació en 1994 cubriendo los aspectos principales de todos los métodos de diseño antecesores. Los padres de UML son Grady Booch, autor del método Booch, James Rumbaugh, autor del método OMT, e Ivar Jacobson, autor de los métodos OOSE y Objectory. La versión 1.0 de UML fue liberada en enero de 1997 y ha sido utilizada con éxito en sistemas construidos para toda clase de industrias alrededor del mundo: hospitales, bancos, comunicaciones, aeronáutica, finanzas, etc. (Holt, R., 2009).

UML por sus siglas en inglés (“Unified Modeling Language”), lenguaje unificado de modelado, es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; permite visualizar, especificar, construir y documentar sistemas con una perspectiva orientada a objetos (Rumbaugh et al., 2004).

Figura 1. Usos del modelado



| Descripción de la imagen: Usos del modelado |
|---|
| <p>El modelado se utiliza en la construcción de “software” principalmente para:</p> <ul style="list-style-type: none">• Comunicar la estructura de un sistema complejo.• Comprender de una manera más clara lo que se construye.• Especificar el comportamiento que se requiere en el sistema.• Identificar oportunidades de simplificación y reutilización. |

El UML está compuesto por diversos elementos gráficos que se combinan para conformar diagramas. Entre los diagramas que existen en UML, se tienen el diagrama de clase que hace parte de los diagramas de estructura, los cuales enfatizan los elementos que deben existir en el sistema de modelado.

2. Diagramas de clase

Un diagrama de clases es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargarán del funcionamiento y la relación entre uno y otro. En un diagrama de clases se pueden distinguir principalmente dos elementos: clases y sus relaciones (Flores Cueto & Bertolotti Zuñiga, 2013).

El objetivo de un diagrama de clase es representar las clases que conforman el modelo de un determinado sistema, el diagrama de clase se construye y se refina durante las fases de análisis y diseño, y se toma como referencia en la implementación del sistema.

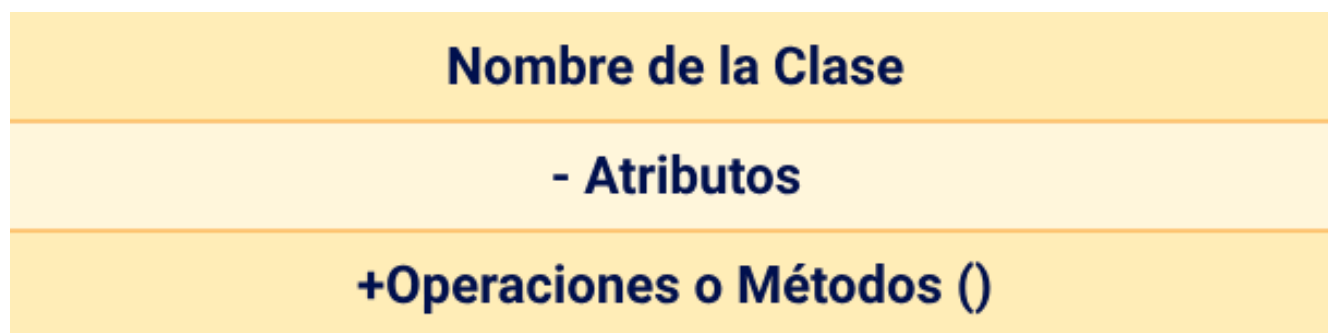
Hay tres perspectivas diferentes en las cuales se pueden utilizar los diagramas de clase:

- **Conceptual.** El diagrama de clase representa los conceptos en el dominio del problema que se está estudiando.
- **Especificación.** El diagrama de clase refleja las interfaces de las clases, pero no su implementación. Aquí las clases aparecen más cercanas a los tipos de datos, ya que un tipo representa una interfaz que puede tener muchas implementaciones diferentes.
- **Implementación.** Esta vista representa las clases tal cual aparecen en el entorno de implementación.

Los elementos que componen un diagrama de clase son:

- **Clase.** Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una casa, un auto, una cuenta corriente, etc.).
- **Atributos.** Los atributos o características de una clase.
- **Métodos.** Los métodos u operaciones de una clase son la forma en cómo esta interactúa con los datos.

En UML, una clase es representada por un rectángulo que posee tres divisiones:



En donde:

- El rectángulo superior contiene el nombre de la clase.
- El rectángulo intermedio contiene los atributos (o variables de instancia) que caracterizan a la clase.
- El rectángulo inferior contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno.

Los atributos y las clases tienen características que definen el grado de comunicación y visibilidad de ellos con el entorno a lo que llamamos controles de acceso.

2.1. Control de acceso

El control de acceso o la visibilidad define la accesibilidad para los atributos o métodos de la clase. Los atributos y métodos pueden ser privados, protegidos o públicos y se representan con cada uno de los símbolos que muestra la siguiente tabla:

Tabla 1. Símbolos de modificadores de acceso

| | |
|----------------|----------|
| Público | + |
| Privado | - |
| Protegido | # |

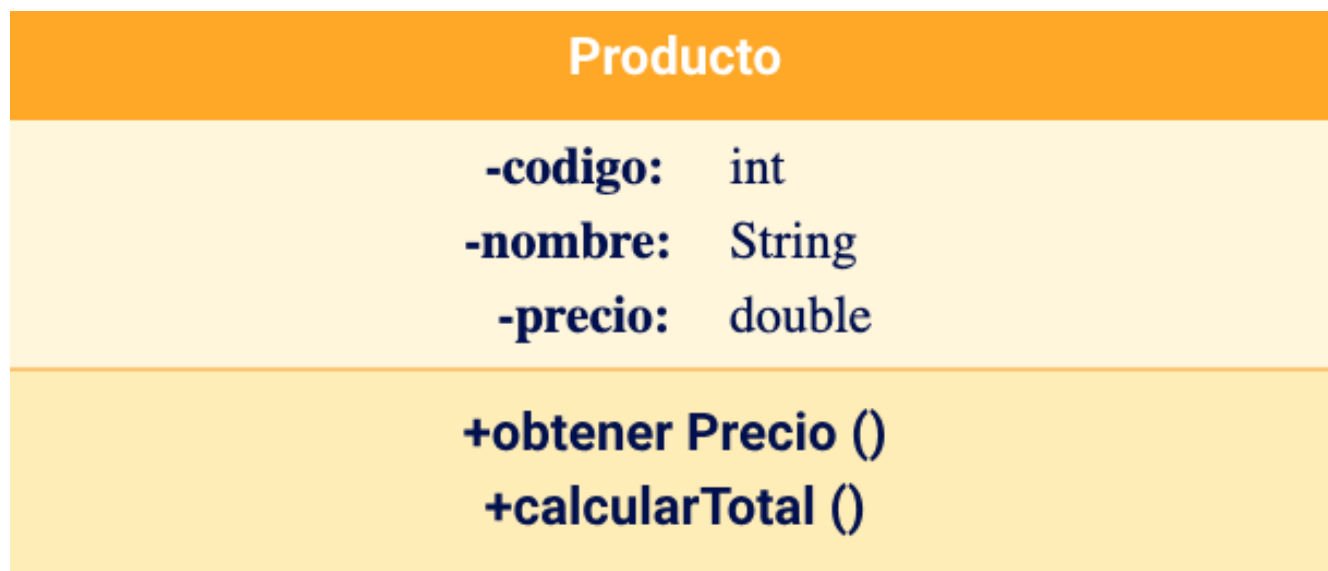
Fuente: SENA (2021)

- **Público.** Se define un atributo o método público cuando se quiere que sean vistos o accesibles por todas las clases.
- **Privado.** Se define un atributo o método privado cuando se quiere que sean visibles o accesibles únicamente por la misma clase en que están.
- **Protegido.** Se define un atributo o método protegido cuando solo se quiere tener acceso a él desde la propia clase que lo define y las que heredan de él.

Por ejemplo, se puede representar una clase denominada **Producto** que contiene 3 atributos (**código de tipo entero**, **nombre de tipo “String”** y **precio de tipo “double”**) y 2 métodos (**obtenerPrecio** y **calcularTotal**), de la siguiente manera:

Se observa que todos los atributos en esta clase son privados porque tienen el símbolo – y los métodos son públicos porque tienen el símbolo +

Figura 2. Representación de la clase Producto en UML



Fuente: SENA (2021).

Relaciones entre clases. Las relaciones existentes entre las distintas clases nos indican cómo se comunican los objetos de esas clases entre sí:

Existen diferentes tipos de relaciones:

- Asociación (conexión entre clases).
- Dependencia (relación de uso).
- Generalización (relaciones de herencia).

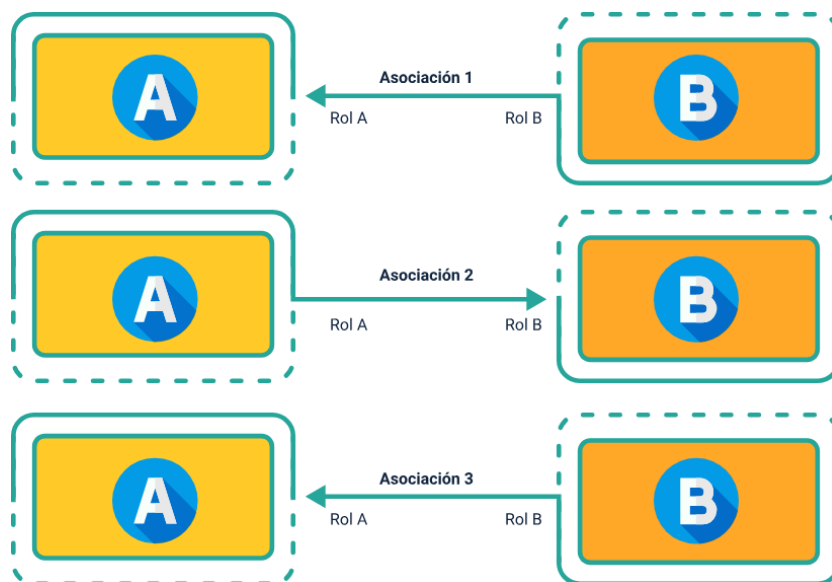
2.2. Asociaciones

Se le conoce como relación de asociación a las conexiones entre clases. Este tipo de relación se representa gráficamente como una línea que une las dos clases y tiene las siguientes características:

- a. Nombre de la asociación. (Es obligatorio establecer el nombre de la asociación).
- b. Cada clase tiene asociado un rol, el cual se interpreta como el papel que juega dicha clase en el contexto de la asociación. Los roles son opcionales y no se utilizan cuando el nombre de la clase expresa correctamente el rol de esta en el contexto de la relación de asociación.
- c. **Navegabilidad:** en las relaciones de asociación se puede establecer si el vínculo es unidireccional o bidireccional. Gráficamente se representan con puntas de flecha.
- d. **Multiplicidad:** representan el número de instancias de la clase que pueden ser partícipes en el proceso de asociación con respecto a una instancia particular de la otra clase vinculada en la relación. (Schmuller, 2001)

A continuación, se presentan algunos ejemplos gráficos.

Figura 3. Ejemplo de asociación



Fuente: SENA (2021).

En el ejemplo anterior, se pueden observar 3 diferentes ejemplos de asociación entre la clase A y la clase B, en todos los ejemplos la Clase A asume el rol A en el contexto de asociación, mientras que la clase B asume el rol B en el contexto de la relación.

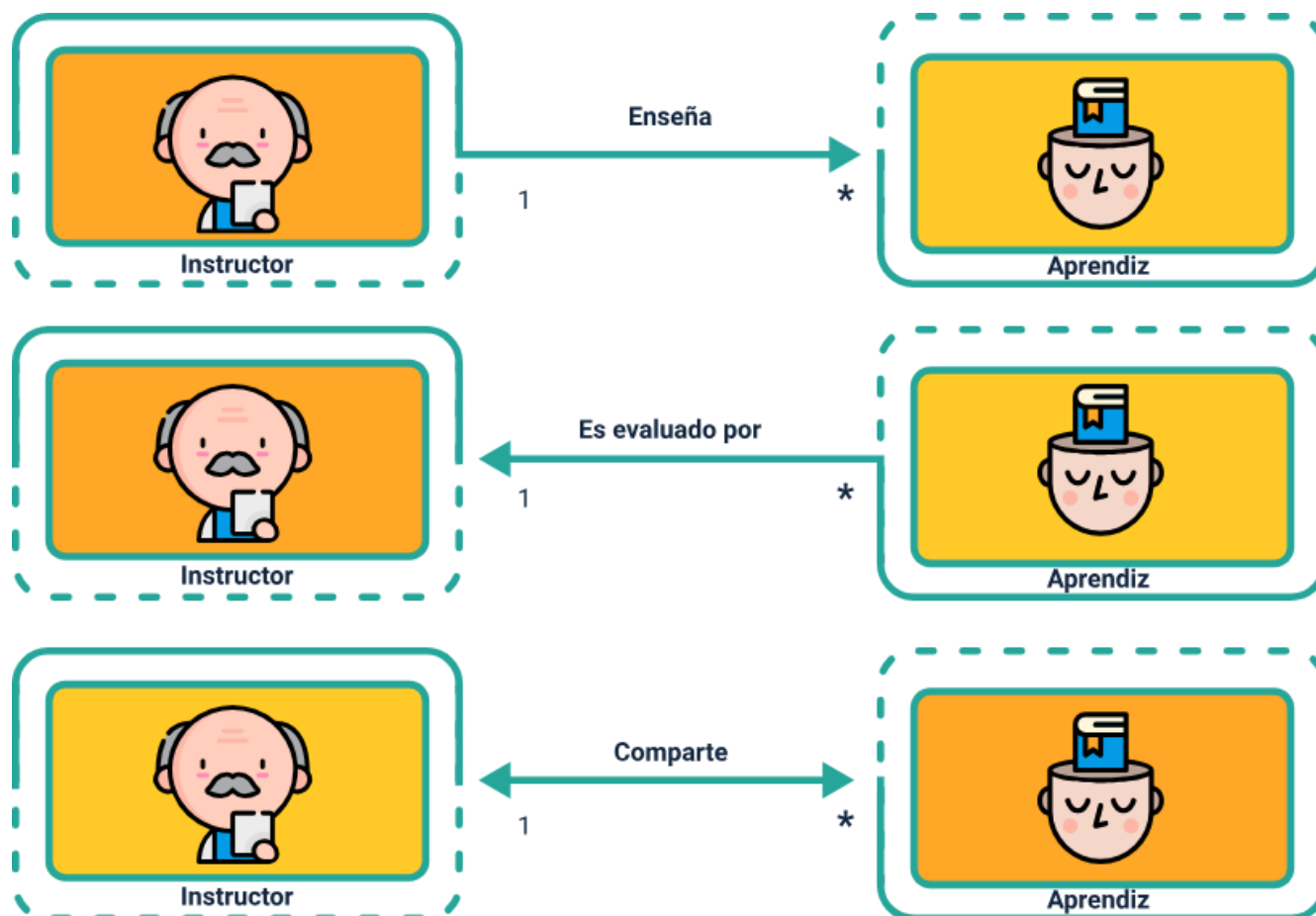
Adicionalmente, se puede observar que los dos primeros ejemplos tienen navegabilidad unidireccional, lo cual representa que la relación se establece en un solo sentido. En el primer caso la Asociación 1 vincula la clase B como origen y la clase A como destino, en el segundo caso se vincula con la Asociación 2 a la clase A como origen con la clase B como destino y en el último ejemplo el vínculo es bidireccional, es decir en los dos sentidos, desde A hacia B y desde B hacia A.

Una relación de asociación debe tener como mínimo tres elementos: (I) nombre de la asociación, (II) navegabilidad y (III) multiplicidad. Respecto a la multiplicidad esta representa la cantidad numérica de instancias de las clases que participan en la relación, aunque este puede ser cualquier número normalmente se acostumbra a utilizar el valor de 1 o * cuando son varias instancias las participantes.

Para facilitar el proceso de entendimiento y lectura se debe tener en cuenta la navegabilidad. Siempre se lee tomando como referencia un ejemplar de la clase desde donde se origina la asociación respecto al valor **N** de instancias del otro extremo de la asociación, este valor N es la multiplicidad en el otro extremo.

Por ejemplo, se puede representar una clase denominada **Figura** que contiene dos atributos (lado1 y lado2) y 3 métodos (método constructor **Figura**, método área y método perímetro), de la siguiente manera:

Figura 4. Ejemplo de variaciones de asociación entre clases



Fuente: SENA (2021)

En el ejemplo anterior se presentan tres variaciones de relaciones de asociación entre la clase instructor y la clase aprendiz, en todos los tres casos la navegabilidad varía dependiendo de lo que se está representando en la relación por medio de su nombre. En el primer caso es una relación de asociación llamada Enseña con navegabilidad desde Instructor hacia Aprendiz indicando que **un** instructor enseña a **varios** aprendices.

En el segundo caso el segundo ejemplo se lee de la siguiente forma: **un** aprendiz es evaluado por **un** instructor. En el último ejemplo de esta gráfica se representa una relación bidireccional, es decir, se hacen dos lecturas desde cada extremo, en este caso **un** instructor comparte con **muchos** aprendices y **un** aprendiz comparte con **un** instructor.

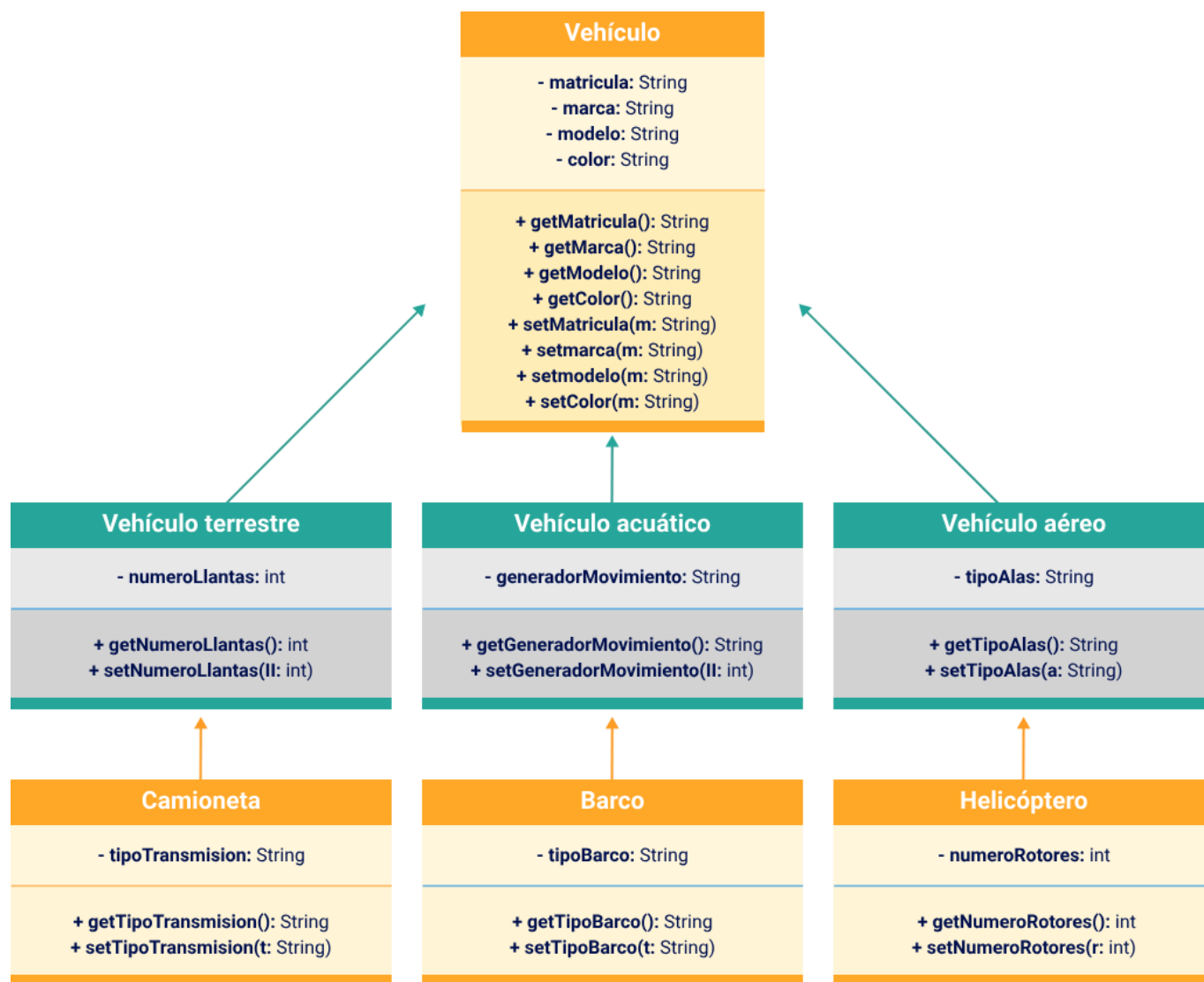
2.3. Herencia

La herencia es una de las relaciones más comunes en la práctica del paradigma orientado a objetos y, como se mencionó en el componente formativo anterior, se puede construir relaciones partiendo desde conceptos (clases) generales a conceptos (clases) específicas (especialización) o desde conceptos específicos hacia conceptos generales (generalización) (Schmuller, 2001).

Es un tipo especial de asociación donde se tienen clases principales o superclases (las más generales en la relación) y clases secundarias o subclases (las clases más especializadas). Adicionalmente, este tipo de relación tiene implícito el nombre **es un** o **es una** y también, implícitamente, es una relación que vincula a un ejemplar de tipo superclase y un ejemplar de tipo subclase; por lo anterior, no se colocan nombre ni multiplicidad a este tipo de relación.

En una relación de herencia las subclases heredan las características (atributos) y los comportamientos (métodos) de las superclases.

Figura 5. Ejemplo de herencia



Fuente: SENA (2021).

Descripción de la imagen: Ejemplo de herencia

Teniendo en cuenta los niveles que componen un diagrama: clase (fila superior), atributo (fila intermedia) y métodos u operaciones (fila inferior), se describe cada una de las siete clases, distribuidas en tres niveles.

Nivel inferior:

Clase: Camioneta

Atributo: **-tipoTransmision:** String

Métodos y operación:

+ getTipoTransmision(): String

+ setTipoTransmision(t: String)

Clase: Barco

Atributo: **-tipoBarco:** String

Métodos y operación:

+ getTipoBarco(): String

+ setTipoBarco(t: String)

Clase: Helicóptero

Atributo: **-numeroRotores:** int

Métodos y operación:

+ getNumeroRotores(): int

+ setNumeroRotores(r: int)

Nivel medio:

Clase: Vehículo terrestre

Atributo: **-numeroLlantas:** int

Métodos y operación:

+ getNumeroLlantas(): int

+ setNumeroLlantas(l1: int)

Clase: Vehículo acuático

Atributo: **-generadorMovimiento:** String

Métodos y operación:

+ getGeneradorMovimiento(): String

+ setGeneradorMovimiento(l1: int)

Clase: Vehículo aéreo

Atributo: **-tipoAlas:** String

Métodos y operación:

+ getTipoAlas(): String

+ setTipoAlas(a: int)

Nivel superior:

Clase: Vehículo

Atributos:

- matrícula: String

- **marca:** String

- **modelo:** String

- **color:** String

Métodos y operación:

+ **getMatrícula():** String

+ **getMarca():** String

+ **getModelo():** String

+ **getColor():** String

+ **setMatricula(m:** String)

+ **setmarca(m:** String)

+ **setmodelo(m:** String)

+ **setColor(m:** String)

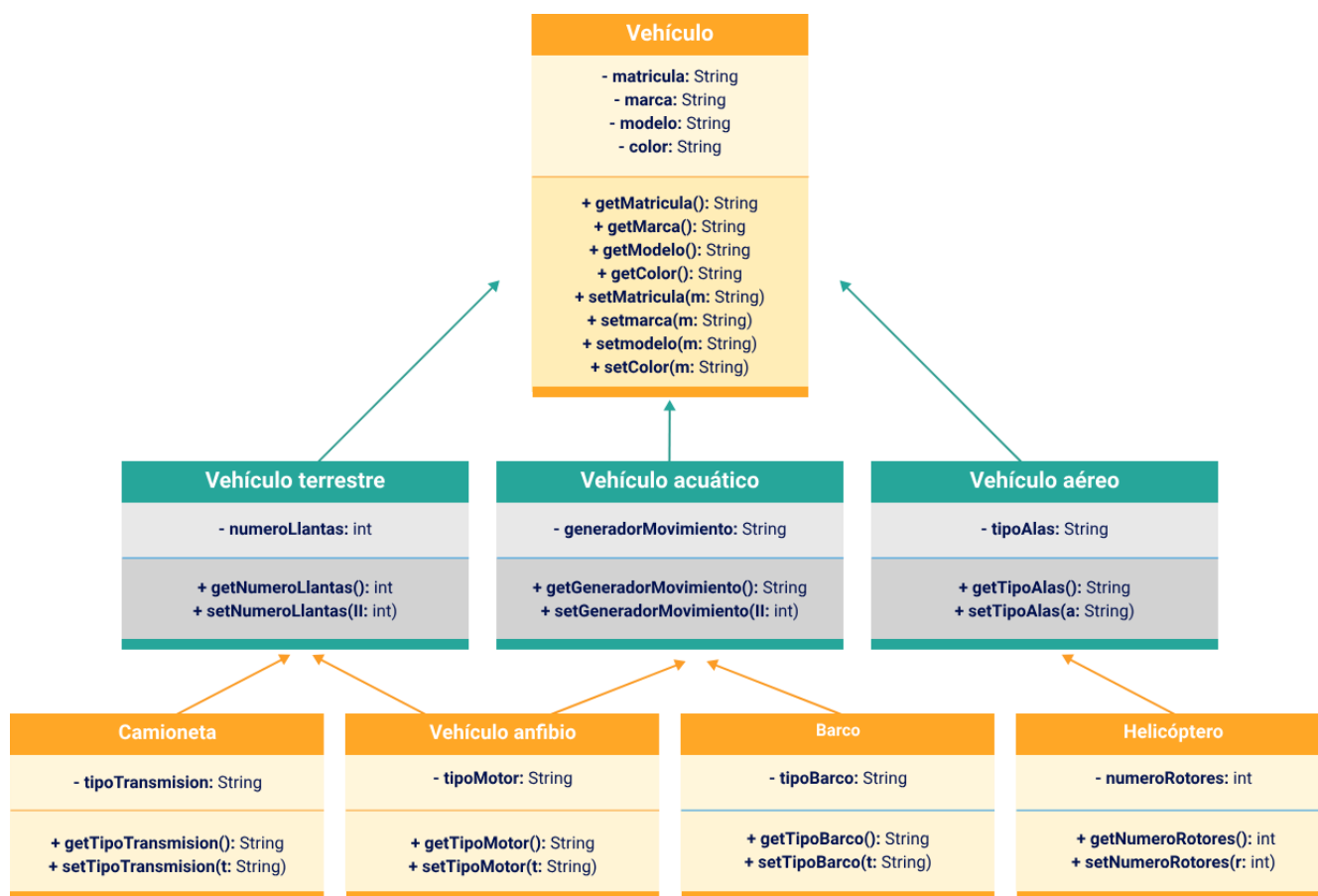
En la figura anterior, se representa una relación de herencia entre siete (7) diferentes clases. Las clases supertipo están en los niveles superiores de la jerarquía y representan las clases más generales cuyas definiciones serán heredadas por las subclases. En las relaciones de herencia se pueden presentar varios niveles, en cada nivel existirá la correspondiente superclase y subclases. Note que no existe nombre en las relaciones porque implícitamente es una relación de tipo es **un** o es **una** que se lee desde las subclases hacia las superclases y siempre en relaciones de **uno a uno**.

En el ejemplo, **una** Camioneta es **un** VehiculoTerrestre, el cual a la vez es **un** Vehículo. El VehiculoTerrestre hereda todas las características y comportamientos

del Vehículo y a su vez la clase Camioneta hereda todas las características y comportamientos de la clase VehiculoTerrestre. De manera indirecta la clase Camioneta está heredando las características y comportamientos de la clase Vehículo por estar en la misma línea de herencia. La clase Barco no está heredando las características de la clase VehiculoTerrestre ya que no está en la misma línea de herencia.

Existen dos tipos de herencia dependiendo de la cantidad de clases supertipo directas asociadas, si una clase hereda características de **una sola supertipo**, es una relación de **herencia simple**; si una clase hereda características de más de **una clase supertipo**, es una relación de **herencia múltiple**.

Figura 6. Ejemplo de herencia múltiple



Descripción de la imagen: Ejemplo de herencia múltiple

Continuando con el ejemplo anterior, se presenta el mismo esquema pero esta vez se agrega otra clase en el nivel inferior:

Clase: Vehículo anfibio

Atributo: **-tipoMotor:** String

Métodos y operación:

+ getTipoMotor(): String

+ setTipoMotor (t: String)

Luego, las clases inferiores se relacionan de manera múltiple al nivel medio: la camioneta y el vehículo anfibio se relaciona con el vehículo terrestre; el vehículo anfibio y el barco se relacionan con el vehículo acuático; y, finalmente, el helicóptero se relaciona con el vehículo aéreo.

En el ejemplo de herencia múltiple, se puede observar un diagrama de clases donde se presentan varias relaciones de herencia, para el caso de la clase **VehiculoAnfibio** se presenta un caso de herencia múltiple ya que esta clase está heredando características de dos clases supertipo directas diferentes: **VehiculoTerrestre** y **VehiculoAcuatico**.

Solo algunos lenguajes de programación admiten la implementación de la herencia múltiple, entre los más conocidos encontramos: Python, Perl y C++. Sin embargo, otros lenguajes orientados a objetos implementan características similares a la herencia múltiple por medio de interfaces.

Visibilidad en las relaciones de herencia

Para determinar la visibilidad tanto de los atributos como de los métodos de una clase, se usan los identificadores de acceso: “private”, “public” y “protected”. En el caso de las relaciones de herencia se debe ser cuidadoso ya que se puede producir resultados no deseados si no se tiene en cuenta las siguientes reglas:

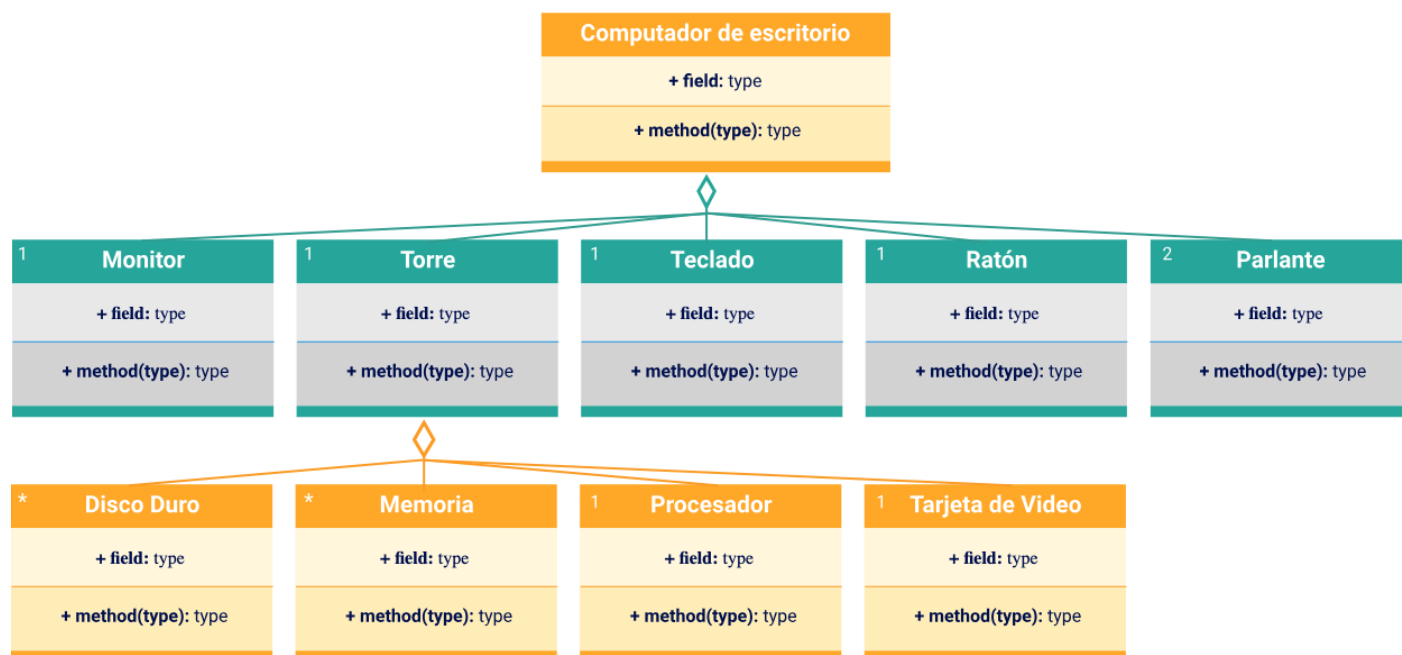
- Un atributo o método privado en una supertipo es inaccesible en la subclase.
- Un atributo o método público sigue siendo público en las subclases.
- Un atributo o método protegido sigue siendo protegido en las subclases si hay más niveles de herencia o funciona como privado en otro caso.

2.4. Agregación

La agregación es un tipo de relación especial, donde se representa que un conjunto de clases conforma un todo, es decir, existen clases agregadas que representan el todo y se constituyen a partir de un conjunto de clases componentes (Schmuller, 2001).

Este tipo de relación se representa con una línea que une la clase agregada junto a sus clases componentes; ahora, del lado de la clase agregada se tiene un rombo que representa la relación de agregación. Esta relación no tiene nombre explícito, ya que se entiende que hay clases agregadas y clases componentes que la forman. Es necesario establecer la multiplicidad del lado de los componentes.

Figura 7. Ejemplo de relación de agregación



Fuente: SENA (2021).

Descripción de la imagen: Ejemplo de relación de agregación

Clase: Computador de escritorio

Atributo: **+ field: type**

Métodos y operación:

+ method(type): type

De ahí se desprenden las siguientes clases:

Clase: Monitor

Atributo: + **field**: type

Métodos y operación:

+ **method(type)**: type

Clase: Torre

Atributo: + **field**: type

Métodos y operación:

+ **method(type)**: type

Clase: Teclado

Atributo: + **field**: type

Métodos y operación:

+ **method(type)**: type

Clase: Ratón

Atributo: + **field**: type

Métodos y operación:

+ **method(type)**: type

Clase: Parlante

Atributo: + **field**: type

Métodos y operación:

+ method(type): type

De la clase de la Torre a su vez se desprenden las siguientes clases:

Clase: Disco Duro

Atributo: **+ field: type**

Métodos y operación:

+ method(type): type

Clase: Memoria

Atributo: **+ field: type**

Métodos y operación:

+ method(type): type

Clase: Procesador

Atributo: **+ field: type**

Métodos y operación:

+ method(type): type

Clase: Tarjeta de video

Atributo: **+ field: type**

Métodos y operación:

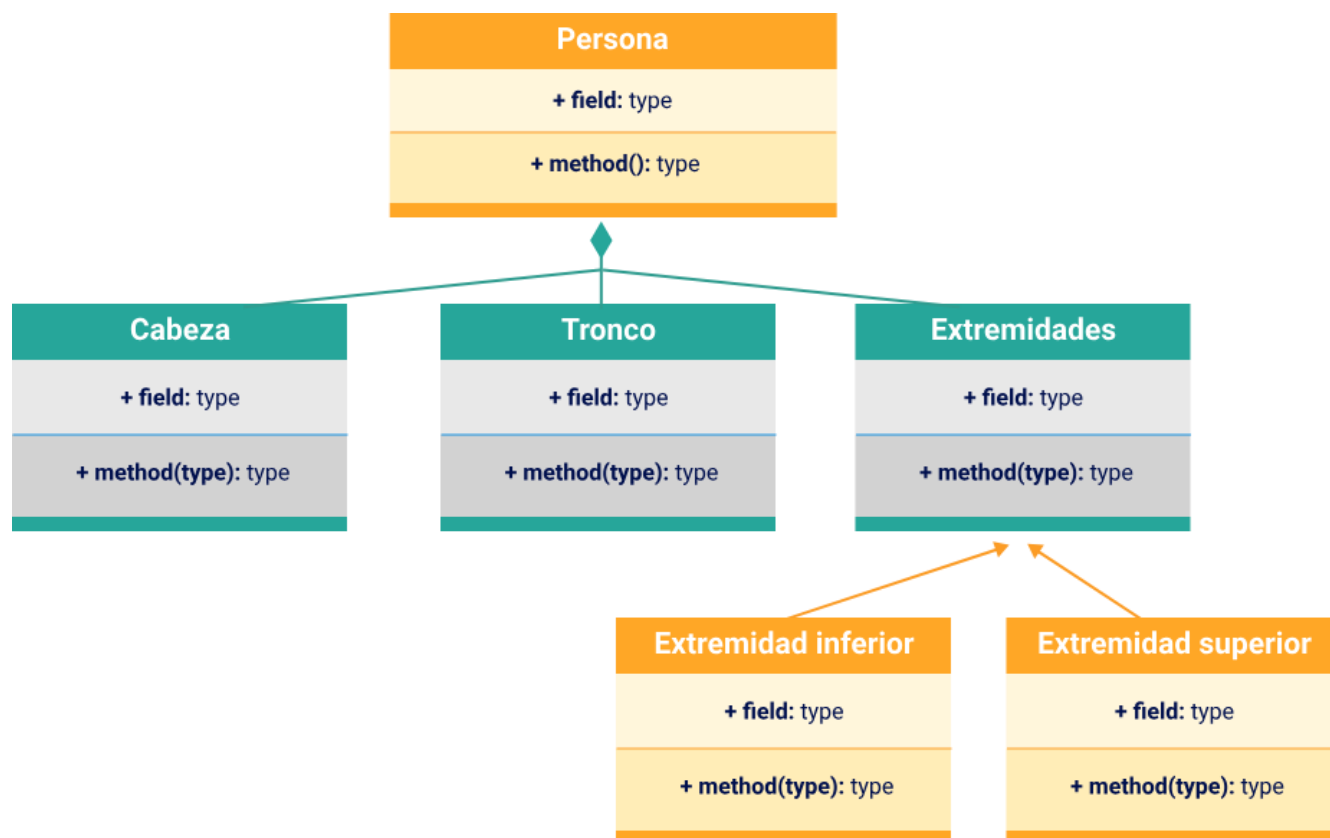
+ method(type): type

En la representación anterior, se muestra un ejemplo de relación de agregación; como se puede observar la clase Computador de escritorio se compone de un monitor, una torre, un teclado, un ratón y dos parlantes. A su vez la clase Torre se compone de varios discos duros, varias memorias, un procesador y una tarjeta de video. Los elementos que son componentes en este diagrama pueden ser componentes de otras clases, además una instancia de los componentes puede ser reemplazado por otra fácilmente, razón por la cual la relación que tienen con la clase agregada no es tan fuerte y por eso es una agregación.

2.5. Composición

Es un tipo especial de relación de agregación en el que las clases componentes no pueden formar parte de otra relación de agregación, es decir, son exclusivos de la composición establecida. Para diferenciar una relación de composición de una relación de agregación el rombo se pinta completamente de negro en su interior. (Schmuller, 2001)

Figura 8. Ejemplo de relación de composición



Fuente: SENA (2021).

| Descripción de la imagen: Ejemplo de relación de composición |
|--|
| <p>Clase: Persona</p> <p>Atributo: + field: type</p> <p>Métodos y operación:</p> <p>+ method(type): type</p> |

De ahí se desprenden tres clases:

Clase: Cabeza

Atributo: **+ field:** type

Métodos y operación:

+ method(type): type

Clase: Tronco

Atributo: **+ field:** type

Métodos y operación:

+ method(type): type

Clase: Extremidades

Atributo: **+ field:** type

Métodos y operación:

+ method(type): type

De esta última clase se heredan dos clases:

Clase: Extremidad inferior

Atributo: **+ field:** type

Métodos y operación:

+ method(type): type

Clase: Extremidad superior

Atributo: **+ field:** type

Métodos y operación:

+ method(type): type

En la figura anterior, se observa un ejemplo de composición. En este caso lo que se representa es una relación mucho más fuerte que la agregación ya que los elementos que componen la clase agregada no pueden formar parte de otra clase.

3. Herramientas CASE

De acuerdo con Pressman (1998), tiempo atrás la ingeniería del “software” era principalmente una actividad manual, utilizando las herramientas únicamente en las etapas finales del proceso. Los ingenieros de “software” en la actualidad hacen notar que necesitan herramientas más variadas, ya que las herramientas manuales no satisfacen las demandas de los sistemas basados en computadoras (Pressman,1998).

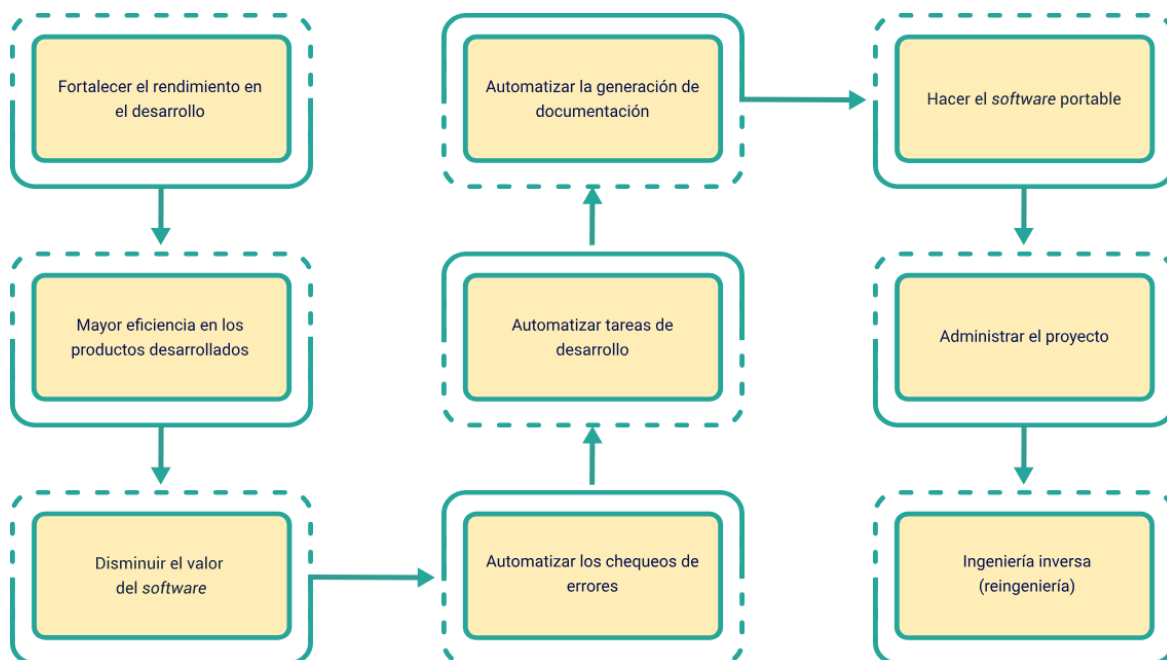
Las herramientas Case son un complemento de la caja de herramientas de un ingeniero del “software” que le proporciona la posibilidad de automatizar actividades manuales, ayudan a asegurar que la calidad sea algo diseñado antes de llegar a construir el producto.

CASE es una sigla, que corresponde a las iniciales de: “Computer Aided software Engineering”; y en su traducción al español significa Ingeniería de “Software” Asistida por Computación.

La tecnología CASE corresponde a la ingeniería de “software” apoyado por computadora. Es un conjunto de herramientas de “software” para automatizar las tareas del desarrollo del “software”, la idea es facilitar un conjunto integrado de herramientas que enlazan y automatizan las fases del ciclo de vida del desarrollo del “software”. (Cuevas,1991).

Según Pressman (1998) los objetivos de las herramientas Case son:

Figura 9. Objetivos de la herramienta CASE



Descripción de la imagen: Objetivos de la herramienta CASE

Los seis objetivos de la herramienta CASE son:

- Fortalecer el rendimiento en el desarrollo.
- Mayor eficiencia en los productos desarrollados.
- Disminuir el valor del “software”.
- Automatizar los chequeos de errores.
- Automatizar tareas de desarrollo.
- Automatizar la generación de documentación.
- Hacer el “software” portable.
- Administrar el proyecto.
- Ingeniería inversa (reingeniería)

Clasificación de las herramientas CASE

Las herramientas CASE en función de las fases del ciclo de vida que abarcan, se pueden agrupar de la forma siguiente:

- **Herramientas integradas, I-CASE** (“Integrated CASE”, CASE integrado). Abarcan todas las fases del ciclo de vida del desarrollo de sistemas. Son llamadas también CASE “workbench”.
- **Herramientas de alto nivel, U-CASE** (“Upper CASE” - CASE superior) o “front-end”. Orientadas a la automatización y soporte de las actividades desarrolladas durante las primeras fases del desarrollo: análisis y diseño.
- **Herramientas de bajo nivel, L-CASE** (“Lower CASE” - CASE inferior) o “back-end”. Dirigidas a las últimas fases del desarrollo: construcción e implantación.
- **Juegos de herramientas o “Tools”-CASE**, son el tipo más simple de herramientas CASE. Automatizan una fase dentro del ciclo de vida. Dentro de este grupo se encontrarán las herramientas de reingeniería, orientadas a la fase de mantenimiento.

A continuación, se presentan ejemplos de las herramientas CASE por ámbitos:

- **Análisis y Diseño**
 - Workbench
 - GUI Design Studio
 - DIA
 - StarUML
- **Diseño de Archivos y Base de Datos**
 - Power Designer

- Oracle Designer
- EasyCase
- **Programación**
 - NetBeans
 - Eclipse/Omondo

Para profundizar en las herramientas CASE, se sugiere revisar el material complementario sobre esta temática.

Los diagramas de clase se diagraman con las herramientas que hacen parte del grupo de **Herramientas de alto nivel, U-CASE**, entre las que encontramos “StarUML”, “Lucichart”, “ArgoUML”, “MagicDraw”.

El manejo de cualquiera de estas herramientas es sencillo, a continuación, se mostrarán los pasos básicos para el manejo de “StarUML”.

Video 2. Diagramas de clases: “StarUML”



[Enlace de reproducción del video](#)

Síntesis del video: Diagramas de clase: “StarUML”

“StarUML” es un modelador de “software” sofisticado, destinado a respaldar el modelo ágil y conciso. Para su instalación puede ingresar a <https://www.staruml.io> y hacer clic sobre la opción de “Download”. Después de instalado el “software”, se muestra la información de la ventana inicial. En la parte inferior izquierda se encuentra la caja de herramientas que contiene las clases y los tipos de relación para los diagramas de relación de clases.

Ahora para crear la clase, se presiona clic sobre la opción “Class” y clic sobre el área. Para agregar atributos y métodos, se presiona clic derecho sobre la clase y luego sobre la opción “add”. Después de creadas las clases con sus atributos y métodos se puede seleccionar la relación que requiera el diagrama de clases.

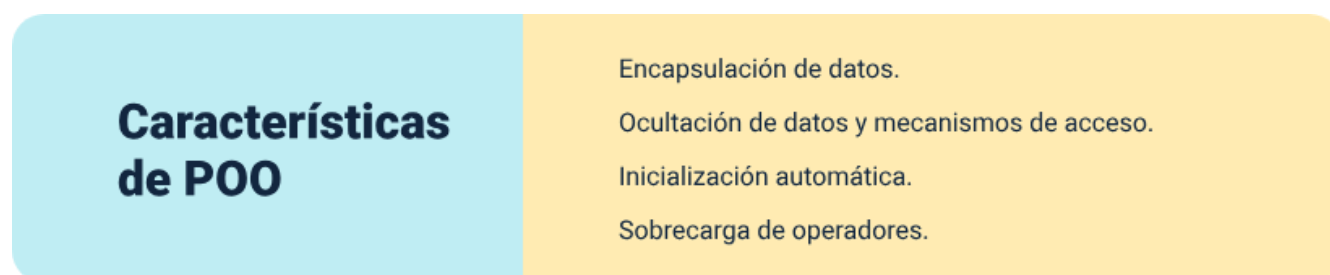
Para profundizar en la creación de los diagramas de clase en “StarUML”, se sugiere revisar en el material complementario el video sobre esta temática.

4. Características del lenguaje de programación orientada a objetos

Aunque la POO se puede implementar en lenguajes como C y PASCAL, se hace muy pesada y confusa cuando es un programa de gran tamaño. Utilizar un lenguaje que esté especialmente diseñado para soportar los conceptos de POO, se vuelve más adecuado para implementarlos. Ya que dependiendo de las características que soportan los lenguajes, se pueden clasificar en las dos categorías siguientes, en lenguajes de programación basados en objetos y en lenguajes de programación orientados a objetos.

La programación basada en objetos es el estilo de programación que soporta principalmente encapsulación e identidad de objetos y sus características principales se centran en:

Figura 10. Características de POO



SENA (2021)

Se dice que los lenguajes que no soportan programación con objetos son lenguajes de programación basados en objetos; es decir, que no soportan la herencia ni la ligadura dinámica. Un ejemplo podría ser el lenguaje de programación Ada que además de tener estas características, es un lenguaje multipropósito.

La programación orientada a objetos incorpora todas las características de la programación basada en objetos, junto con dos adicionales, a saber: herencia y ligadura dinámica. Además, se caracteriza por la siguiente sentencia:

Figura 11. Características de programación orientada a objetos



SENA (2021).

Entre los lenguajes que soportan estas características se incluyen C++, Smalltalk, Object Pascal y Java, aunque existe un gran número de lenguajes de programación basados en objetos y orientados a objetos.

En este componente se realizará la implementación de los conceptos de la POO en el lenguaje de programación Java, el cual es un lenguaje orientado a objetos desarrollado por Sun Microsystems presentado en el año 1995. Java traduce el código fuente Java en instrucciones que son interpretadas por la máquina virtual de Java (JVM, Java Virtual Machine). A diferencia de C y C++ en los que está inspirado, este es un lenguaje interpretado (Andaluza & Garcilaso, 2012).

Sus características principales son:

Figura 12. Características Java



SENA (2021).

Descripción de la imagen: Características Java

Las características Java son:




- **Sencillo:** es fácil de entender, elimina la complejidad de otros lenguajes.
- **Orientado a objetos:** al ser un paradigma orientado a objetos, la filosofía de programación orientada a objetos facilita la creación y mantenimiento de programas.
- **Independiente de la arquitectura y portable:** al compilar un programa en Java, el código resultante es un tipo de código binario conocido como Java Bytecode. Este código es interpretado por diferentes computadoras

de igual manera. Como el código compilado de Java es interpretado, un programa compilado de Java puede ser utilizado por cualquier computadora que tenga implementado el intérprete de Java.

- **Robusto:** Java simplifica la gestión de la memoria.
- **Multitarea:** Java puede ejecutar diferentes líneas de código al mismo tiempo.
- **Dinámico:** en Java no es necesario cargar completamente el programa en memoria, sino que las clases compiladas pueden ser cargadas bajo demanda en tiempo de ejecución.

Además de contar con las anteriores características, existen distintos entornos de desarrollo de aplicaciones Java, este tipo de productos ofrecen al programador un ambiente de trabajo integrado para facilitar el proceso completo de desarrollo de aplicaciones; estos productos se denominan IDE (“Integrated Development Environment”). Entre los más conocidos se encuentran:

Tabla 2. Entornos de desarrollo Java

| ID | Logo | Enlace |
|----------|---|---|
| NetBeans |  | https://netbeans.apache.org/download/index.html |
| Eclipse |  | http://www.eclipse.org/downloads/ |
| JBlue |  | https://bluej.org/versions.html |

SENA (2021).

5. Implementación de clases y objetos

Los principales elementos en un lenguaje de programación son las clases, debido a que se usan para todas las operaciones lógicas en un desarrollo. Se puede decir que son estructuras o plantillas que representan objetos del mundo real.

Por su parte, los objetos pueden ser cosas, lugares o personas y son de tipo físico o conceptuales. En general las clases poseen propiedades, comportamientos y relaciones con otras clases.

Java determina en sus estándares de codificación convenciones para nombres de clases, métodos y atributos. Para las clases se define lo siguiente:

- La primera letra debe ser mayúscula.
- Utilizar nomenclatura camelCase (por ejemplo: un nombre compuesto por varias palabras: clase "CuentaAhorros").
- Los nombres deben ser sustantivos.

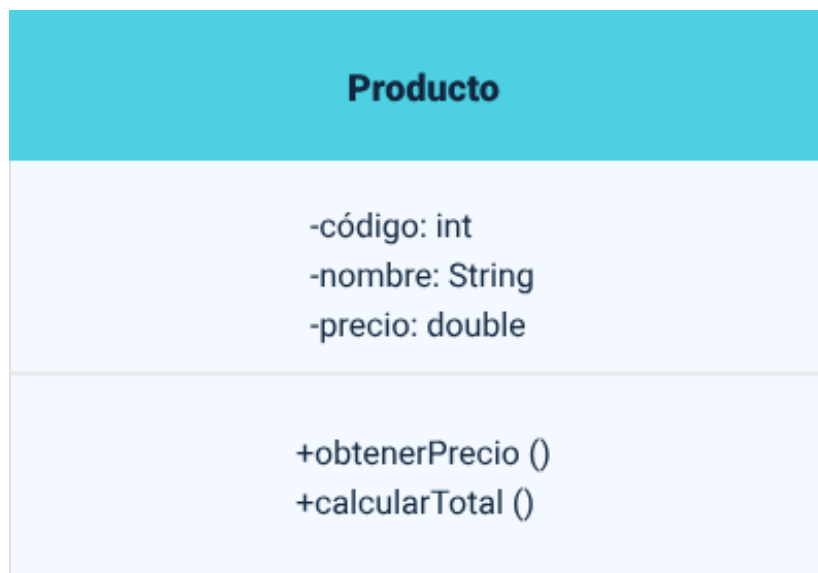
Tabla 3. Estructura de una clase en Java

| Estructura de una clase en Java | Ejemplo |
|---|---|
| <p>Una clase en java se compone por las siguientes declaraciones:</p> <ul style="list-style-type: none"> • Paquete • Comentarios • Definición de clases • Atributos • Constructores • Métodos | <p>Estructura:</p> <pre>public class NombreDeLaClase {}</pre> <p>Donde la palabra public indica que el alcance de la clase será público.</p> <p>La palabra class indica que es una clase.</p> <p>Y las llaves {} se usan para agrupar los bloques de código.</p> |

SENA (2021).

Basado en lo anterior, a continuación, se relaciona un ejemplo sobre el diseño de la clase de producto con sus atributos y métodos:

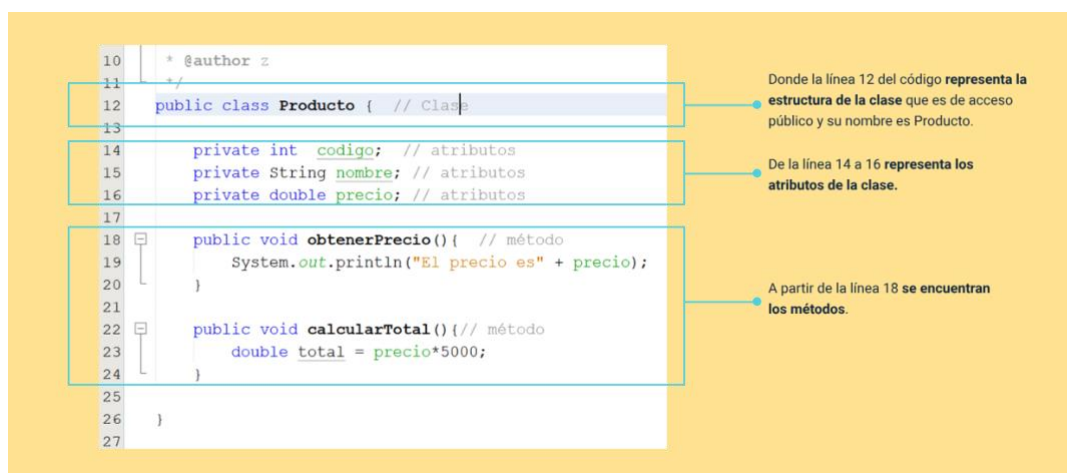
Figura 13. Diagrama clase producto



SENA (2021).

En Java este diseño se representaría de la siguiente manera:

Figura 14. Implementación de la clase Producto en Java



SENA (2021).

Descripción de la imagen: Implementación de la clase Producto en Java

La estructura que presenta contiene:

En la línea 12 del código **representa la estructura de la clase** que es de acceso público y su nombre es Producto.

```
public class Producto { // Clase
```

De la línea 14 a 16 **representa los atributos de la clase.**

```
Private int codigo; // atributos
```

```
Private String nombre; // atributos
```

```
Private double precio; // atributos
```

Luego, a partir de la línea 18 **se encuentran los métodos.**

```
public void obtenerPrecio () { // método
```

```
    System.out.println("El precio es" + precio);
```

```
}
```

```
public void calcularTotal () { // método
```

```
}
```

La creación de un objeto en Java es equivalente a decir que se está instanciando una clase. Cuando se hace la instancia se comparten los atributos y métodos de esta. Una sola clase puede tener varias instancias.

Tomando como ejemplo la clase **Producto** al instanciar un objeto estamos dando valores a sus atributos código, nombre y precio y accediendo a sus métodos **obtenerPrecio** y **calcularTotal**.

Para instanciar un objeto de una clase en Java se utiliza la palabra reservada **“new”** y se crea el objeto del tipo de la clase así:

```
Producto p = new  
Producto ();
```

Cuando instanciamos el objeto p este puede acceder a los métodos y atributos de la clase Producto que en este caso serían obtenerPrecio y calcularTotal así:

```
p.obtenerPrecio ();  
p.calcularTotal ();
```

5.1. Atributos y métodos de la clase

Los atributos de una clase son definidos según esta sintaxis: **[modifVisibilidad] tipo nombreVariable [= valorInicial];**

modifVisibilidad: el modificador de visibilidad indica desde qué parte del código se puede acceder a las variables o a los métodos y puede ser:

- **“public”**. Indica que los atributos son accesibles desde cualquier lugar del programa. No hay restricciones en el alcance de los miembros de datos públicos.
- **“Private”**. Indica que los atributos solo son accesibles dentro de la clase en la que se declaran.

- **“Protected”**. Indica que los atributos son accesibles dentro del mismo paquete o sub-clases en paquetes diferentes. Al heredar sí se puede usar desde la clase derivada.

tipo: es el tipo de la variable, pudiendo ser un tipo básico o un objeto.

A continuación, se relacionan los tipos de datos que existen en Java:

Tabla 4. Tipos de datos en Java. Números enteros

| Tipo | Formato | Descripción |
|---------|---------|---------------------|
| “byte” | 8 bit | Entero de un “byte” |
| “short” | 16 bit | Entero corto |
| “int” | 32 bit | Entero |
| “long” | 64 bit | Entero largo |

Tabla 5. Tipos de datos en Java. Números reales

| Tipo | Formato | Descripción |
|----------|---------|-----------------|
| “float” | 32 bit | Flotante simple |
| “double” | 64 bit | Flotante doble |

Tabla 6. Tipos de datos en Java. Cadenas de texto

| Tipo | Formato | Descripción |
|----------|---------|------------------|
| “char” | 16 bit | Un solo carácter |
| “String” | textos | Textos |

Tabla 7. Tipos de datos en Java. Otros

| Tipo | Formato | Descripción |
|-----------|-----------------|----------------------------------|
| "boolean" | "True or false" | Valor booleano falso o verdadero |

nombreVariable: es el nombre que daremos a la variable, el cual debe cumplir con las siguientes convenciones:

- Por convención, en Java, los nombres de las variables empiezan con una letra minúscula y si es una palabra compuesta la primera letra en **Mayúscula** con nomenclatura **camelCase**.
- No pueden tener espacios ni iniciar con un número, ni caracteres especiales.
- No puede ser el mismo que una palabra clave.
- No deben tener el mismo nombre que otras variables cuyas declaraciones aparezcan en el mismo ámbito.

A continuación se presentan algunos ejemplos:

Ejemplo:

- **primerNombre**
- **apellidoCliente**
- **edad**

[= valorInicial]; permite inicializar la variable con un valor.

Ejemplo:

Se permite definir más de una variable, separándolas por coma, por ejemplo:

public int a = 5, b, c = 4;

Ejemplos de declaración de atributos o variables:

- publicintedad=5;
- privatechargenero='M';
- privatedoubleestatura;

Nota: todos los atributos terminan en punto y coma.

Los métodos se definen la siguiente manera:

[modifVisibilidad] tipo nombreMetodo (listaParámetros) {}

En los atributos y métodos para las clases también existen:

- **Modificador de visibilidad.** Se aplican las mismas normas que para los atributos, se debe indicar el modificador de acceso, este puede ser público, privado o protegido.
- **Tipo.** Es el tipo de valor devuelto por el método, pudiendo ser “int”, “char”, “String”, etc.
- **Nombre del método.** Es el nombre que tendrá el método y debe cumplir con las siguientes convenciones:

Los nombres de los métodos deberían ser verbos, todo el verbo en minúscula. Se pueden agregar sustantivos con la primera letra en mayúscula. Ejemplo: revisarCuenta ()

- **Lista de parámetros.** Dentro de los paréntesis se escriben los parámetros, los parámetros son los atributos que recibe el método para poder operar correctamente, puede o no tener parámetros.
- **Cuerpo del método {}.** Para finalizar todo método debe llevar un par de llaves, que es donde se escribe el código a ejecutar. (Nota: a diferencia de los atributos, los métodos no finalizan con punto y coma).

A continuación, se presentan algunos ejemplos de métodos:

Ejemplo metodo:

```
public int obtenerEdad ()  
{  
    return edad;  
}
```

Este es un método público que retorna un valor entero, cuyo nombre es **obtenerEdad**, no recibe parámetros y devuelve la edad.

Ejemplo método:

```
public void cambiarEdad (int laEdad)  
{  
    edad = laEdad;  
}
```

Este es un método público que no retorna ningún valor porque tiene la palabra reservada **void**, su nombre es **cambiarEdad** y recibe un parámetro de tipo entero, en el

cuerpo del método asigna a la variable edad el valor del parámetro que viene almacenado en la variable **laEdad**.

5.2. Constructores y destructores

El constructor de una clase es un método estándar para inicializar los objetos de esa clase, a su vez es un destructor de un método que se invoca automáticamente cuando el objeto se destruye.

Se habla de un constructor cuando se instancia un objeto y es necesario inicializar sus variables con valores correctos. Un constructor es un método perteneciente a la clase, que posee unas características especiales:

- El nombre del constructor es igual que el nombre de la clase.
- No tiene ningún retorno.
- Se pueden crear varios, siempre y cuando tengan diferentes atributos.

Dentro del código de un constructor generalmente suelen existir inicializaciones de variables y objetos, para conseguir que el objeto sea creado con dichos valores iniciales.

Para definir los constructores se emplea la siguiente sintaxis:

- **modifVisibilidad:** se aplica las mismas normas que para atributos y métodos que debe indicar si son “public”, “private” o “protected”.

```
[modifVisibilidad] nombreConstructor (listaParámetros)
{
}
```


- **nombreConstructor:** debe coincidir con el nombre de la clase.
{} El constructor posee un par de llaves, dentro de las cuales estará el código que se ejecutará al ser llamado el método.
- **listaParámetros:** es la lista de los parámetros que tomará la función separados por comas y definidos cada uno de ellos como: tipo nombreParámetro
Ejemplo: (int a, int b)

Figura 15. Ejemplo de implementación de constructor en Java

```
// @author z
```

```
public class Producto { // Clase
    private int codigo; // atributos
    private String nombre; // atributos

    // Constructor
    public Producto(int codigo, String nombre) {
        this.codigo = codigo;
        this.nombre = nombre;
    }
}
```

En el ejemplo de la clase **Producto** se crea el constructor **Producto** que tiene el mismo nombre de la clase, con el modificador de visibilidad público y su lista de parámetros, el cual recibe los atributos y se los asigna a la variable de la clase.

La palabra reservada “**this**” hace referencia a los miembros de la propia clase en el objeto actual; es decir, “**this**” se refiere al objeto actual sobre el que está actuando

un método determinado y se utiliza siempre que se quiera hacer referencia al **objetoactual** de la clase.

Un constructor sin parámetros es llamado **constructor vacío**. Si no se define ningún constructor, el compilador crea un constructor por defecto. Esto permite crear objetos con “**new Clase()**” sin argumentos, aunque no se defina ningún constructor.

Figura 16. Ejemplo implementación de constructor vacío o por defecto en Java

```
// @author z

public class Producto { // Clase
    private int codigo; // atributos
    private String nombre; // atributos

    // Constructor vacio
    public Producto() {
    }

    // Constructor por parametros
    public Producto(int codigo, String nombre) {
        this.codigo = codigo;
        this.nombre = nombre;
    }
}
```

Un **destructor** es un método opuesto a un constructor, este método en lugar de crear un objeto lo destruye, liberando la memoria de la computadora para que pueda ser utilizada por alguna otra variable u objeto.

En Java no existen los destructores, esto es gracias al recolector de basura de la máquina virtual de Java. Como su nombre lo dice, el recolector de basura recolecta todas las variables u objetos que no se estén utilizando y que no haya ninguna

referencia a ellos por una clase en ejecución, liberando así automáticamente la memoria de nuestra computadora.

Aunque Java maneja de manera automática el recolector de basura, el usuario también puede decir en qué momento Java pase el recolector de basura con la instrucción.

```
System.gc ();
```

5.3. Métodos de acceso y modificadores

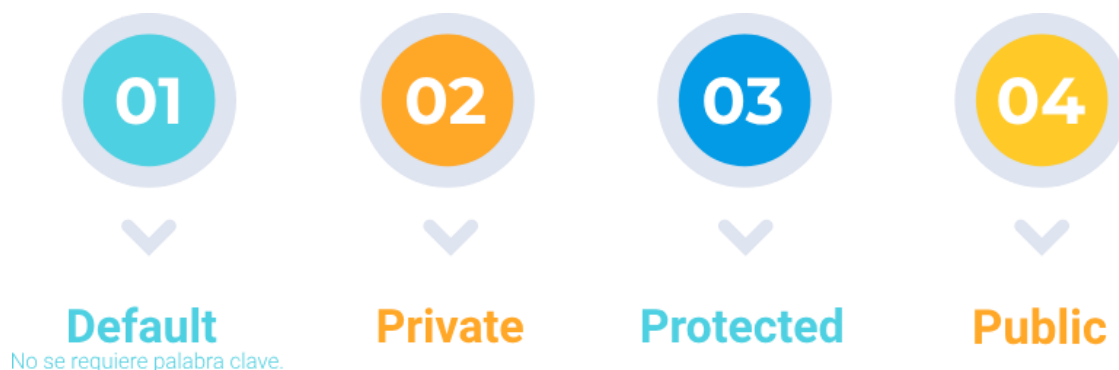
De métodos y modificadores, hay dos tipos básicos esenciales de miembros de la clase: público (“public”) y privado (“private”).

A los métodos y modificadores se puede acceder libremente a un miembro público, mediante un código definido fuera de su clase. Se puede acceder a un miembro privado solo por otros métodos definidos por su clase. Es a través del uso de miembros privados que el acceso está controlado.

Restringir el acceso a los miembros de una clase es una parte fundamental de la programación orientada a objetos, ya que ayuda a evitar el mal uso de un objeto; no obstante, al permitir el acceso a datos privados solo a través de un conjunto de métodos bien definidos, puede evitar que se asignen valores incorrectos a esos datos.

Como su nombre indica, los modificadores de acceso en Java ayudan a restringir el alcance de una clase, constructor, variable, método o miembro de datos.

Hay cuatro tipos de modificadores de acceso disponibles en Java:



Modificador de acceso por defecto (“default”)

Cuando no se especifica ningún modificador de acceso para una clase, método o miembro de datos, se dice estar teniendo modificador de acceso default por defecto.

Los miembros de datos, clase o métodos que no se declaran utilizando ningún modificador de acceso; es decir, que tengan un modificador de acceso predeterminado, solo son accesibles dentro del mismo paquete.

Ejemplo:

```
class Producto {  
  
    void mostrar ()  
  
    {  
  
        System.out.println(" Hola Mundo!");  
  
    }  
  
}
```

La clase Producto tiene modificador de acceso default al igual que el método mostrar, esto quiere decir que no podrán ser accedidas desde otro paquete.

Modificador de acceso privado (“private”)

El modificador de acceso privado se especifica con la palabra clave “private”. Los métodos o los miembros de datos declarados como privados solo son accesibles dentro de la clase en la que se declaran. Además:

- Ninguna de las otras clases del mismo paquete podrá acceder a los miembros.
- Las clases e interfaces no se pueden declarar como privadas (“private”).

Ejemplo:

```
public class Producto {  
  
private void mostrar ()  
  
{  
  
System.out.println(" Hola Mundo!");  
  
}  
  
}
```

En este ejemplo tenemos el método mostrar con el modificador privado, lo que quiere decir que si se llama a este método de otra clase no se visualizará.

Modificador de acceso protegido (“protected”)

El modificador de acceso protegido se especifica con la palabra clave protected. Además:

- Los métodos o miembros de datos declarados como `protected` son accesibles dentro del mismo paquete o sub-clases en paquetes diferentes.

Ejemplo:

```
public class Producto {  
  
    protected void mostrar ()  
  
    {  
  
        System.out.println(" Hola Mundo!");  
  
    }  
  
}
```

Modificador de acceso público (“public”)

El modificador de acceso público se especifica con la palabra clave `public`.

Además:

- El modificador de acceso público tiene el alcance más amplio entre todos los demás modificadores de acceso.
- Las clases, métodos o miembros de datos que se declaran públicos son accesibles desde cualquier lugar del programa.
- No hay restricciones en el alcance de los miembros de datos públicos.

Ejemplo:

```
public class Producto {
```

```
public void mostrar ()  
  
{  
  
    System.out.println(" Hola Mundo!");  
  
}  
  
}
```

Siguiendo con el ejemplo de la clase Producto, en este caso al declarar el método mostrar como público permite acceder desde cualquier parte al método mostrar.

Métodos accesoros

Son los métodos que permiten obtener o modificar los atributos de un objeto y son de dos tipos “**get**” y “**set**”.

Los métodos “**get**” y “**set**”, son simples métodos que usamos en las clases para mostrar (“**get**”) o modificar (“**set**”) el valor de un atributo. El nombre del método siempre será “**get**” o “**set**” y a continuación el nombre del atributo, su modificador siempre es “**public**”, ya que se quiere mostrar o modificar desde fuera de la clase. Por ejemplo, **getNombre** o **setNombre**.

Ejemplo:

Sintaxis método get:

```
public tipo_dato_atributo getAtributo (){  
  
    return atributo;  
  
}
```

Ejemplo:

Sintaxis método set:

```
public void setAtributo (tipo_dato_atributo variable){  
  
this.atributo = variable;  
  
}
```

Figura 17. Ejemplo implementación de métodos “get” y “set” en Java

```
// @author z  
  
public class Producto { // Clase  
    private int codigo; // atributos  
    private String nombre; // atributos  
  
    public int getCodigo() { // método get del atributo codigo  
        return codigo; //obtiene el valor del atributo codigo  
    }  
  
    public void setCodigo(int codigo) { // metodo set del atributo codigo  
        this.codigo = codigo; // cambia el valor del atributo codigo  
    }  
  
    public String getNombre() { // método get del atributo nombre  
        return nombre; //obtiene el valor del atributo nombre  
    }  
  
    public void setNombre(String nombre) { // metodo set del atributo nombre  
        this.nombre = nombre; // cambia el valor del atributo nombre  
    }  
}
```

En el ejemplo se muestra la clase Producto con sus atributos codigo y nombre, por cada uno de estos atributos encontramos sus métodos “get” y “set” para obtener y cambiar el valor de cada uno de ellos.

5.4. Sobrecarga de métodos (“overload”)

Permite definir más de un constructor o método con el mismo nombre, con la condición de que no puede haber dos de ellos con el mismo número y tipo de

parámetros. En resumen, la sobrecarga permite declarar métodos que se llamen igual pero que reciban parámetros diferentes (no puede haber 2 métodos con el mismo nombre y los mismos parámetros), por esta razón lo que define a qué método se ingresa, son los argumentos que se envían como parámetros.

Figura 18. Tipos de datos en Java

```
// @author z
public class Producto { // Clae
    private int codigo; // atributos
    private String nombre; // atributos
    private double precio; // atributos

    public Producto() {
    }

    public Producto(int codigo, String nombre, double precio) {
        this.codigo = codigo;
        this.nombre = nombre;
        this.precio = precio;
    }

    public void descontarProducto(double precio, String nombre){
        double descuento = precio * 0.10;
        System.out.println("El producto " + nombre + "tiene descuento");
    }

    public void descontarProducto(int precio){
        double descuento = precio * 0.10;
    }
}
```

SENA (2021).

En el ejemplo se muestra una sobrecarga de constructores y una de métodos. Existe sobrecarga en los constructores Producto, ya que se crearon dos con el mismo nombre, pero con diferentes parámetros; el mismo caso pasa con el método descontarProducto, que está creado dos veces, pero con diferentes parámetros.

6. Comunicación entre clases

Las clases, al igual que los objetos, no existen de modo aislado. La programación orientada a objetos (POO) intenta modelar aplicaciones del mundo real tan fielmente como sea posible y por lo tanto debe reflejar estas relaciones entre clases y objetos. Según Rumbaugh, J., Jacobson, I. & Booch, G. (2004), existen tres clases básicas de relaciones entre los objetos: Asociación, Agregación/Composición, Generalización/Especialización. Herencia.

6.1. Implementación de asociaciones

La implementación de asociación en Java se realiza mediante el uso de un campo de instancia. La relación puede ser bidireccional y cada clase tiene una referencia a la otra. La agregación y la composición son tipos de relaciones de asociación. A continuación, se explicitan los tipos de relaciones:

Asociación bidireccional. Un cliente tiene una cuenta corriente y una cuenta pertenece a un dueño. Para realizar la implantación en Java de esta asociación bidireccional lo que se hace es:

En la clase **Cliente** crear una instancia de la clase **Cuenta** y en la clase **Cuenta** crear una instancia de la clase **Cliente** así:

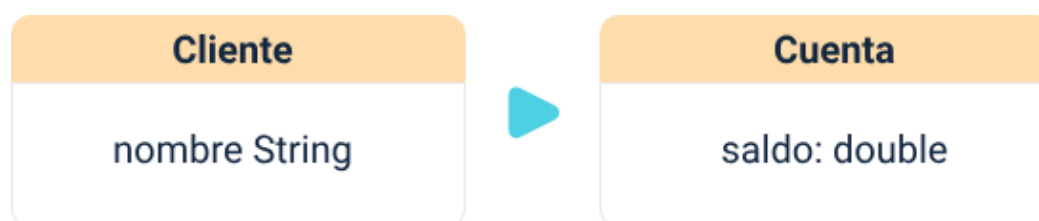


Figura 19. Ejemplo de asociación bidireccional en Java

```

/**
 * @author z
 */
public class Cliente {

    private String nombre;
    public Cuenta cuenta; // asociacion con la clase Cuenta

}

```

```

/**
 * @author z
 */
public class Cuenta {

    private double saldo;
    public Cliente cliente; // Asociacion Cuenta con Cliente

}

```

Asociación unidireccional. Un usuario tiene una clave. En Java esta relación unidireccional se implementa como un atributo dentro de la clase Usuario así:

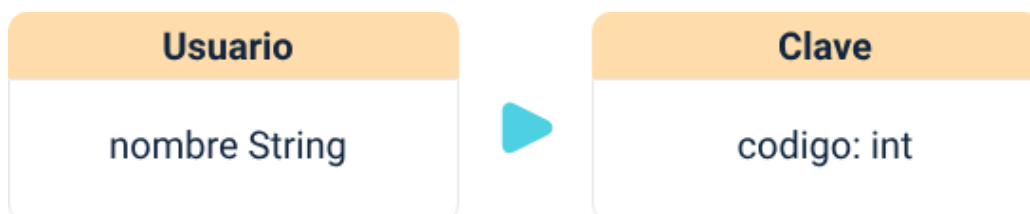


Figura 20. Ejemplo asociación unidireccional en Java

```

*
* @author z
*/
public class Usuario {
    private String nombre;
    public Clave clave;
}

```

6.2. Implementación de composición/agregación

La agregación es un tipo de relación que comprende a las clases y sus diferentes componentes, constituyendo un todo, a su vez, la composición en cambio es un tipo de agregación, donde cada componente solo puede pertenecer a un todo.

La composición en Java consiste en crear una clase nueva agrupando objetos de clases que ya existen, los objetos contenidos se declaran con visibilidad “private” y se inicializan en el constructor de la clase.



El anterior ejemplo implementado en Java quedaría de la siguiente manera:

Figura 21. Ejemplo implementación de composición en Java

```
* @author z
*/
public class Torre {
    public DiscoDuro disco; // se crea como atributo de la clase torre

    public Torre() {
        disco = new DiscoDuro();
        // En el constructor se crea una instancia de tipos DiscoDuro
    }
}
```

La imagen muestra la creación de la clase Torre con un atributo de tipo DiscoDuro, con un constructor que instancia un objeto disco, si destruimos la instancia de tipo Torre se destruye la instancia de tipo DiscoDuro.

La agregación en Java es la relación que existe entre dos clases donde la clase agregada contiene una referencia de otra clase.



Torre



Parlantes

El siguiente código muestra cómo se hace la implementación del ejemplo anterior en Java:

Figura 22. Ejemplo agregación en UML

```

* @author z
*/
public class Torre {
    public Parlante parlante; // se crea como atributo de la clase Parlante
}
public void ensamblar(Parlante par) {
    parlante = par;
    // Se crea un método sin retorno que recibe un objeto de tipo parlante
}
}

```

La imagen muestra la creación de un atributo de tipo Parlante en la clase Torre, para vincular el objeto se crea un método ensamblar que recibe un parámetro de tipo parlante. En este caso si se destruye la instancia de tipo Torre no se destruye la instancia de tipo parlante.

6.3. Implementación de herencia

La herencia es el mecanismo por el que se crean nuevos objetos definidos en términos de objetos ya existentes. Por ejemplo, si se tiene la clase Persona, se puede crear la subclase Empleado, que es una extensión de Persona.

```

class Empleado extends Persona {

    int ventas;

}

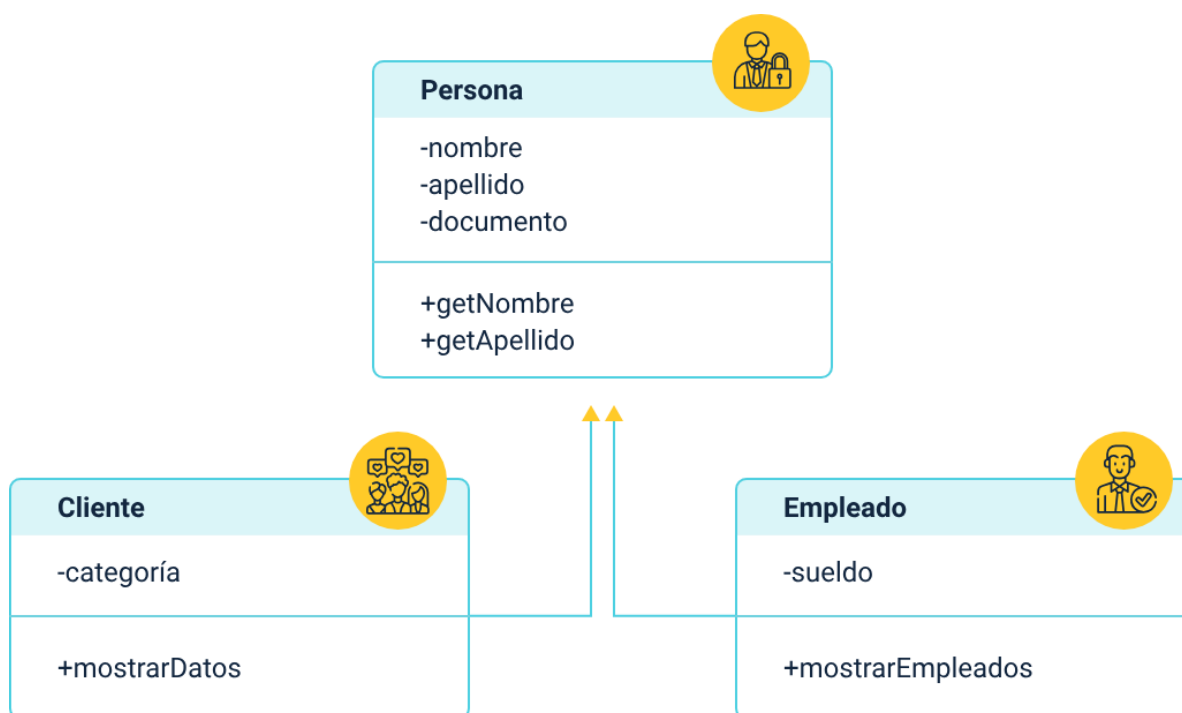
```

La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

En Java esto es posible mediante el uso de la palabra clave extends. Por lo tanto, la subclase se añade (se extiende) a la superclase.

En el ejemplo anterior, la clase Empleado extiende de la clase principal Persona heredando todos sus métodos y atributos.

Figura 23. Ejemplo de herencia en UML



Descripción de la imagen: Ejemplo de herencia en UML

Superclase: Persona

Atributos:

nombre

apellido

documento

Operaciones o métodos:

+getNombre

+getApellido

Clase: Cliente

Atributo: -categoría

Operaciones o métodos: +mostrarDatos

Clase: Empleado

Atributo: -sueldo

Operaciones o métodos: +mostrarEmpleados

La **superClase Persona** tiene los atributos de nombre, apellido, documento, que también los necesita la clase Cliente y la clase Empleado. Una de las ventajas de la herencia es la reutilización de código, en este caso las dos clases hijas no necesitan volver a colocar los atributos porque ya los hereda de la clase persona.

Implementado la relación de herencia del ejemplo Persona, Cliente y Empleado en Java sería:

- a. Se crea la clase padre o superClase Persona, con sus atributos, constructores, métodos “get” y “set”.

Figura 24. Ejemplo de implementación de herencia en Java

```
1  package herenciaejemplo;
2  //author z
3
4  public class Persona {
5      private String nombre;
6      private String apellido;
7      private String documento;
8
9      public Persona(String nombre, String apellido, String documento) {
10         this.nombre = nombre;
11         this.apellido = apellido;
12         this.documento = documento;
13     }
14
15     public String getNombre() {
16         return nombre;
17     }
18
19     public void setNombre(String nombre) {
20         this.nombre = nombre;
21     }
22
23     public String getApellido() {
24         return apellido;
25     }
26
27     public void setApellido(String apellido) {
28         this.apellido = apellido;
29     }
30
31     public String getDocumento() {
32         return documento;
33     }
34
35     public void setDocumento(String documento) {
36         this.documento = documento;
37     }
38 }
39
40
```

- b. Se crea la clase Cliente con su propio atributo categoría y su método mostrar datos, para heredar los atributos de la superClase Persona utilizamos la palabra reservada extends así:

Figura 25. Ejemplo de implementación de extends en Java

```

6  package herenciaejemplo;
7
8  /**
9   *
10  * @author z
11  */
12  public class Cliente extends Persona{ // para heredar de Persona utilizamos la palabra extends
13
14      private String categoria; // creamos el atributo de la clas Cliente
15
16  }
17

```

Cuando se extiende de la clase Persona hay que implementar el constructor que trae los atributos de nombre, apellido y documento de la clase persona, al hacer clic en el bombillo que señala la flecha azul o presionar Alt + Enter se crea el constructor

Figura 26. Ejemplo de implementación **constructor** en herencia Java

```

package herenciaejemplo;

/**
 *
 * @author z
 */
public class Cliente extends Persona{ // para heredar de Persona utilizamos la palabra extends

    private String categoria; // creamos el atributo de la clas Cliente

    public Cliente(String nombre, String apellido, String documento) {
        super(nombre, apellido, documento);
    }

}

```

La imagen muestra la herencia de la clase persona con la creación del constructor.

Figura 27. Ejemplo de implementación de herencia clase **Cliente** en Java

```

7  package herenciaejemplo;
8
9  // @author z
10
11  public class Cliente extends Persona{ // para heredar de Persona utilizamos la palabra extends
12
13      private String categoria; // creamos el atributo de la clas Cliente
14
15      public Cliente(String nombre, String apellido, String documento, String categoria) {
16          super(nombre, apellido, documento);
17          this.categoria = categoria;
18      }
19
20      public String mostrarDatos(){
21
22          return getNombre()+ "Apellido/n" +getApellido()+ "Docuemnto/n" +getDocumento()+ "Categoria" +categoria;
23          // en el método mostrar datos se implementas los métodos que vienen de la clase persona getNombre,
24          // getApellido, getDocumento
25      }
26
27  }
28

```

La imagen muestra la clase Cliente completa reutilizando atributos y métodos de la superClase Persona después de realizar la herencia.

Figura 28. Ejemplo de implementación de la clase **Empleado** en Java

```

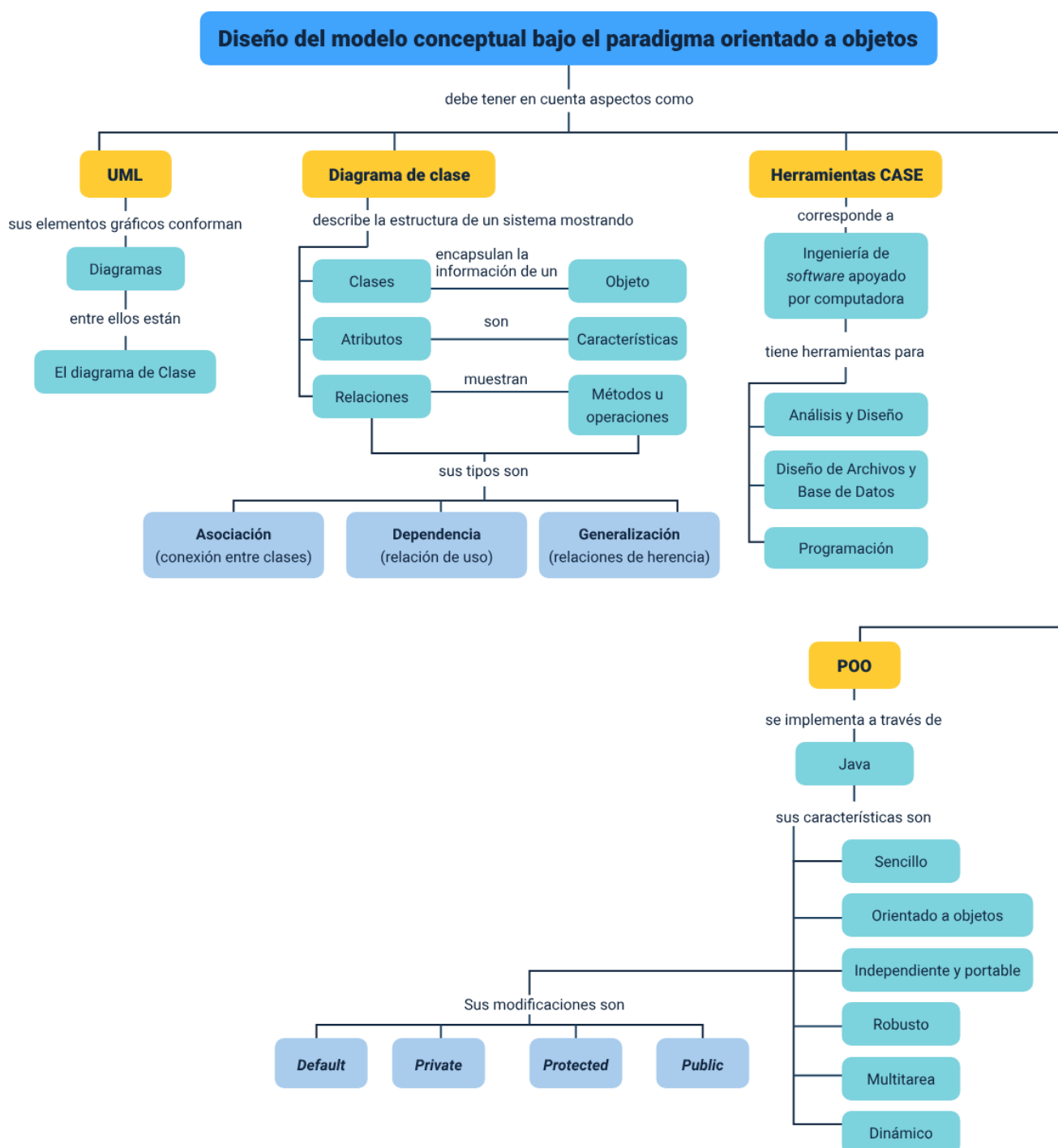
1  * @author z
2  */
3  public class Empleado extends Persona{
4
5      private double sueldo;
6
7      public Empleado(String nombre, String apellido, String documento, double sueldo) {
8          super(nombre, apellido, documento);
9          this.sueldo = sueldo;
10     }
11
12     public String mostrarEmpleado(){
13
14         return getNombre()+ "Apellido/n" +getApellido()+ "Docuemnto/n" +getDocumento()+ "Sueldo" +sueldo;
15         // en el método mostrar Empleado se implementan los métodos que vienen de la clase persona getNombre,
16         // getApellido, getDocumento
17     }
18
19 }

```

La imagen muestra la implementación de la herencia de la clase Empleado.

Síntesis

A continuación, se presenta a modo de diagrama las temáticas desarrolladas en este componente formativo.



Material complementario

| Tema | Referencia | Tipo de material | Enlace del recurso |
|---|---|------------------|---|
| 1. Introducción a UML | Lucid Software Español. (2019). Tutorial - Diagrama de Clases UML. [Archivo de video]. Youtube. | Video | https://www.youtube.com/watch?v=Z0yLerU0g-Q |
| 1. Introducción a UML | Todo Calls. (2017). Curso UML Diagramas de Clases. [Archivo de video]. Youtube | Video | https://www.youtube.com/watch?v=7WRYH2ei5Rw&t=106s |
| 2. Diagrama de Clase | SENA. (2021). Diagrama de clases - Ejercicios resueltos. | Documento | https://ecored-sena.github.io/APLICACION_SERVICIOS_NUBE_M4_C_F2//downloads/ANEXO_DIAGRAMA_CLASES_EJERCICIOS.pdf |
| 2. Diagrama de Clase | Robles, S. (2016). HERRAMIENTAS Y ENTORNOS DE PROGRAMACIÓN. [Pdf]. UCLM. | Página web | https://silo.tips/download/herramientas-y-entornos-de-programacion |
| 3. Herramientas CASE | Herramienta CASE. (s.f.) EcuRED. | Blog | https://www.ecured.cu/Herramienta_CASE#Ejemplos_de_Herramientas_Case_m.C3.A1s_utilizadas |
| 3. Herramientas CASE | StarUML. (s.f.). StarUML. | Página web | https://staruml.io/ |
| 4. Características del lenguaje de programación orientada a objetos | Universidad Autónoma del Estado de Hidalgo. (2014). 1.1 Lenguajes de Programación Orientados a Objetos. | Página web | http://200.57.56.254/lcc/mapa/PROYECTO/libro15/11_lenguajes_de_programacin_orientados_a_objetos.html |

| Tema | Referencia | Tipo de material | Enlace del recurso |
|---------------------------------------|---|------------------|---|
| 5. Implementación de clases y objetos | ARIMETRICS. (2020). Qué es Entorno de desarrollo. | Página web | https://www.arimetrics.com/glosario-digital/entorno-de-desarrollo |
| 5. Implementación de clases y objetos | Codigofacilito. (2012). Clases, Objetos y Métodos en Java. | Video | https://www.youtube.com/watch?v=AEXLtATMkZM |
| 6. Comunicación entre clases | Oracle Java Technologies Oracle. (2021). Java Is the Language of Possibilities | Página web | https://www.oracle.com/java/technologies/ |
| 6. Comunicación entre clases | Systemoff. (2020). Cómo implementar relaciones de asociación de los diagramas de clases en Java. [Archivo de video]. YouTube. | Video | https://www.youtube.com/watch?v=BGStNP0qRk |

Glosario

camelCase: es una convención de Java que consiste en escribir palabras o frases compuestas de modo que cada palabra o abreviatura comienza con una letra mayúscula o una primera palabra con una letra mayúscula y el resto con minúscula.

CASE: ingeniería de “software” asistida por computación.

Herramienta CASE: una herramienta “software” que automatiza una parte de las fases del ciclo del desarrollo de sw.

IDE: entorno de desarrollo integrado que se utiliza para el desarrollo de un código fuente o programa.

JVM: máquina virtual de Java.

Tecnología CASE: conjunto de instrumentos y técnicas “software” para automatizar una disciplina de la ingeniería, incluyendo metodologías estructuradas y herramientas.

Referencias bibliográficas

Andaluza, U. & Garcilaso, I. (2012). Programación en Java Francisco Javier Cruz Vílchez.

Cuevas Agustín, G. (1991). Ingeniería del Software. Práctica de la programación. Editorial RA-MA.

Deitel, H. M. & Deitel, P. J. (2004). Como programar en C/C++ y Java. (4ta Ed.)

Flores Cueto, J. J., & Bertolotti Zuñiga, C. (2013). Diagrama de clases en uml. 6.

Holt, R. (2009). Introduction to UML: Unified Modeling Language. U Waterloo <https://cupdf.com/document/introduction-to-uml-unified-modeling-language-ric-holt-u-waterloo-march-2009.html>

Pressman, R. (1998). Ingeniería del Software. Un enfoque práctico. (4ta ed.). Editorial Mc. Graw-Hill

Rumbaugh, J., Jacobson, I., & Booch, G. (2004). El Lenguaje Unificado de Modelado. Elements, 30. <http://portal.acm.org/citation.cfm?id=993859&dl=>

Schmuller, J. (2001). Aprendiendo UML en 24 horas. Pearson Educación.

Créditos

| Nombre | Cargo | Regional y Centro de Formación |
|--------------------------------|------------------------------------|--|
| Claudia Patricia Aristizábal | Responsable del Ecosistema | Dirección General |
| Rafael Neftalí Lizcano Reyes | Responsable de Línea de Producción | Regional Santander - Centro Industrial del Diseño y la Manufactura |
| Jonathan Guerrero Astaiza | Experto Temático | Centro de teleinformática y producción industrial - Regional Cauca |
| Zulema León Escobar | Experto Temático | Centro de teleinformática y producción industrial - Regional Cauca |
| Paula Andrea Taborda Ortiz | Diseñadora Instruccional | Centro de diseño y metrología - Regional distrito capital |
| Andrés Felipe Velandia Espitia | Revisor metodológico y pedagógico | Centro de Diseño y Metrología Regional Cauca |
| Alix Cecilia Chinchilla Rueda | Experto Temático | Centro de Gestión Industrial - Regional Distrito Capital |
| Jhana Johanna Bustillo Ardila | Revisión de estilo | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Miroslava González Hernández | Diseñadora Instruccional | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Juan Carlos Tapias Rueda | Diseñado web | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Zuleidy Maria Ruiz Torres | Producción audiovisual | Centro de Comercio y Servicios - Regional Tolima |
| Wilson Andrés Arenales Cáceres | Producción audiovisual | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Lina Marcela Pérez Manchego | Producción audiovisual | Centro Industrial del Diseño y la Manufactura - Regional Santander |

| Nombre | Cargo | Regional y Centro de Formación |
|---------------------------------|---|--|
| Carlos Eduardo Garavito Parada | Producción audiovisual | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Ludwyng Corzo García | Producción audiovisual | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Andrea Paola Botello De la Rosa | Desarrollo front-end | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Veimar Celis Melendez | Validación de diseño y contenido | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Zuleidy María Ruiz Torres | Validador de Recursos Educativos Digitales | Regional Santander - Centro Industrial del Diseño y la Manufactura |
| Luis Gabriel Urueta Alvarez | Validador de Recursos Educativos Digitales | Regional Santander - Centro Industrial del Diseño y la Manufactura |
| Daniel Ricardo Mutis Gómez | Evaluador para Contenidos Inclusivos y Accesibles | Regional Santander - Centro Industrial del Diseño y la Manufactura |