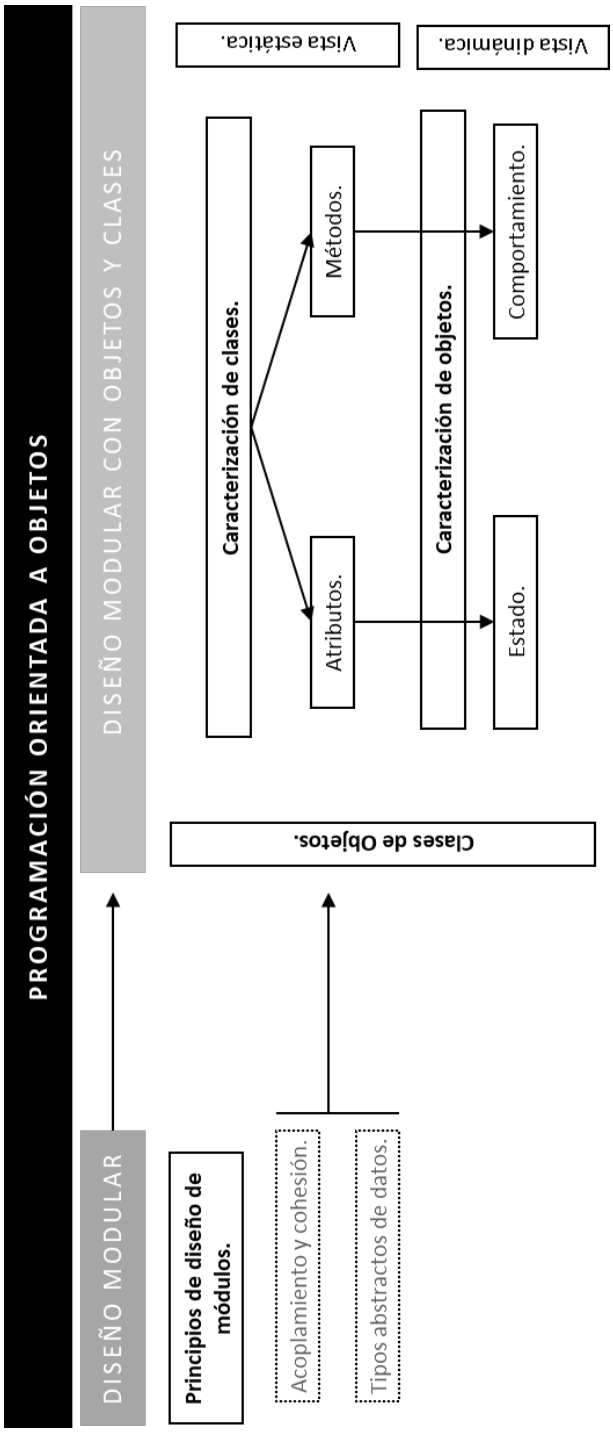
Programación Avanzada

Tema 1. Introducción a la programación orientada a objetos

Índice

Esquema



1.1. Introducción y objetivos

La programación orientada a objetos es una forma de implementar la lógica de un sistema *software* en la que los programas se organizan como colecciones de objetos que intercambian mensajes para realizar algún tipo de proceso y donde cada objeto es una instancia de una clase que abstrae algún concepto relevante en el dominio del problema.

Comprender bien el concepto de clase y objeto es el objetivo principal de este tema. Pero antes de abordar estos conceptos, se presentará la programación orientada a objetos en el contexto de la evolución de los lenguajes de programación y el diseño modular. El estudio de esta evolución permitirá comprender dos criterios de diseño muy importantes como son la cohesión y el acoplamiento y como la programación orientada a objetos permite implementarlos de manera eficiente.

Los **objetivos generales** de este tema son los siguientes:

- Comprender la importancia del diseño modular y qué influencia ha tenido en la programación orientada a objetos.
- Comprender como abstraer conceptos del dominio de un problema a través del concepto de clases y objeto.
- Aprender a identificar y diseñar de clases.
- Entender el papel de los métodos dentro de la definición de una clase y como definen el comportamiento de los objetos.

1.2. Diseño modular de programas

Piense en un gran sistema software compuesto de miles y miles de líneas de código sin algún tipo de organización que dé forma y organice todo ese código, este sistema sería realmente difícil de mantener e incluso de comprender. Tendríamos un bloque monolítico, propenso a errores colaterales al modificar cualquier parte. Piense ahora en ese mismo sistema compuesto por un conjunto de módulos interconectados, donde cada módulo lleva a cabo una función específica y conocida. Ahora veríamos un **sistema formado por partes independientes interconectadas en lugar de un bloque monolítico sin estructura,** lo que facilitaría la comprensión general del sistema.

Un sistema diseñado de forma modular permite ser comprendido como la unión de un conjunto de partes (módulos) interconectadas. Cada parte (módulo) idealmente realizará un tarea bien definida e independiente de las demás.

En su más amplia concepción, un **módulo** podría ser **cualquier elemento funcional dentro del sistema, identificable y relevante.** Por ejemplo: subrutinas, librerías, subsistemas o clases.

Otra de las ventajas del diseño modular está en que el cumplimiento de muchos de los factores de calidad (como la mantenibilidad o capacidad de pruebas) dependen —en gran medida— del modularidad presente en las estructuras de representación, tanto de código como de datos.

Ideas clave

El modularidad como propiedad ofrece la posibilidad de subdividir una aplicación en piezas más pequeñas (denominadas módulos), donde cada una debe ser tan independiente como sea posible. Los diferentes lenguajes ofrecen diferentes aproximaciones al concepto de módulo. **La programación orientada a objetos en particular lo hará a través del concepto de clase.**

El modularidad puede definirse como: «la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados» (Booch, 2002, p.17).

Veamos a continuación los criterios de diseño que permiten alcanzar la tan ansiado modularidad.

1.3. Principios de diseño de módulos

Aunque el diseño modular persigue la división de un sistema grande en módulos más pequeños y manejables, no siempre esta división es garantía de un sistema bien organizado. Muchos aspectos de la modularización pueden ser comprendidos solo si se examinan los módulos en relación con otros.

Para construir un software con un diseño modular efectivo, hay un factor importante como es el de **independencia funcional.** El significado de independencia funcional hace referencia a que **la función del módulo debe ser de naturaleza atómica, de modo que realice una sola tarea sin la menor interacción con otros módulos.** La independencia funcional se considera un signo de aumento en el grado de modularidad, es decir, la presencia de una mayor independencia funcional da como resultado un sistema de software más modular.

Los **módulos deben diseñarse bajo los criterios de acoplamiento y cohesión.** El criterio del acoplamiento busca la menor dependencia posible entre módulos y el criterio de cohesión busca que todo lo agrupado o encerrado bajo el módulo obedezca al mismo propósito.

El acoplamiento entre módulos es el grado de interdependencia entre dos módulos. La cohesión es el grado de interacción y relación dentro de un módulo.

Podemos determinar el objetivo global del diseño modular con las siguientes máximas:

Mínimo acoplamiento entre módulos. Máxima cohesión interna en el módulo

Veamos, a continuación, en más detalle cada uno de estos criterios.

Acoplamiento

El acoplamiento es una **medida de interdependencia entre módulos.** Se dice que existe un alto acoplamiento cuando el cambio de un módulo fuerza al cambio de los que dependen de él. El objetivo es hacer que esta interdependencia sea mínima.

De manera general, **podemos pensar que en un sistema grande siempre habrá algún tipo de acoplamiento entre módulos** porque siempre habrá algún tipo de interconexión entre ellos. Lo que buscamos es que este acoplamiento sea mínimo.

En la siguiente figura el módulo MA tienen una relación con el MB. En la medida que los cambios del módulo MB afecten a MA, se dice que MA sufre más o menos acoplamiento de MB.

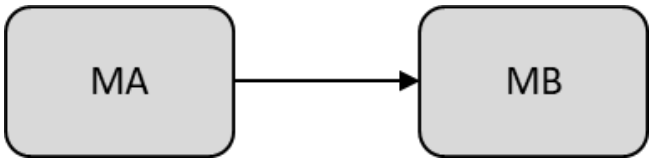


Figura 1. Interdependencia entre módulos. Fuente: elaboración propia.

Como el acoplamiento se puede producir solo en las relaciones entre módulos, cuantas más relaciones tenga un módulo, mayor probabilidad habrá de que se produzcan acoplamientos. Por lo tanto, podemos extraer las siguientes conclusiones o reflexiones:

- Cuantas menos conexiones existan entre módulos, menos oportunidad habrá para que se produzca el efecto onda, es decir, que un defecto en un módulo pueda afectar a otro.
- Cuanto menos acoplados estén los módulos menos implicaciones o efectos colaterales podrán tener los cambios realizados.

Ideas clave

Un acoplamiento bajo indica un sistema bien particionado y puede obtenerse de dos maneras:

- **Reduciendo el número de relaciones:** cuantas menos conexiones existan entre módulos, menor será la posibilidad del efecto en cadena (un error en un módulo provoca otro colateral en otro modulo).
- **Debilitando el nivel de dependencia en las relaciones necesarias:** ningún módulo debería depender de los detalles internos de implementación de cualquier otro. Lo único que tiene que conocer un módulo de otro es como invocarlo y como interpretar el resultado obtenido.

Una forma de debilitar el acoplamiento es haciendo que un módulo oculte o encapsule su implementación interna y no la haga visible al exterior.

Pero si un módulo no hace visible su implementación, ¿cómo lo utilizamos? La respuesta a esta pregunta es que el módulo solo expondrá al exterior su interfaz.

La interfaz del módulo será la única vía desde el exterior para usar el módulo.

Las ventajas de un sistema débilmente acoplado son muchas. Tal como define Booch, un sistema modular débilmente acoplado facilita:

- **La sustitución de un módulo por otro,** de modo que los módulos afectados por un cambio serán menos.
- **El seguimiento de un error y el aislamiento del módulo defectuoso** que produce ese error.

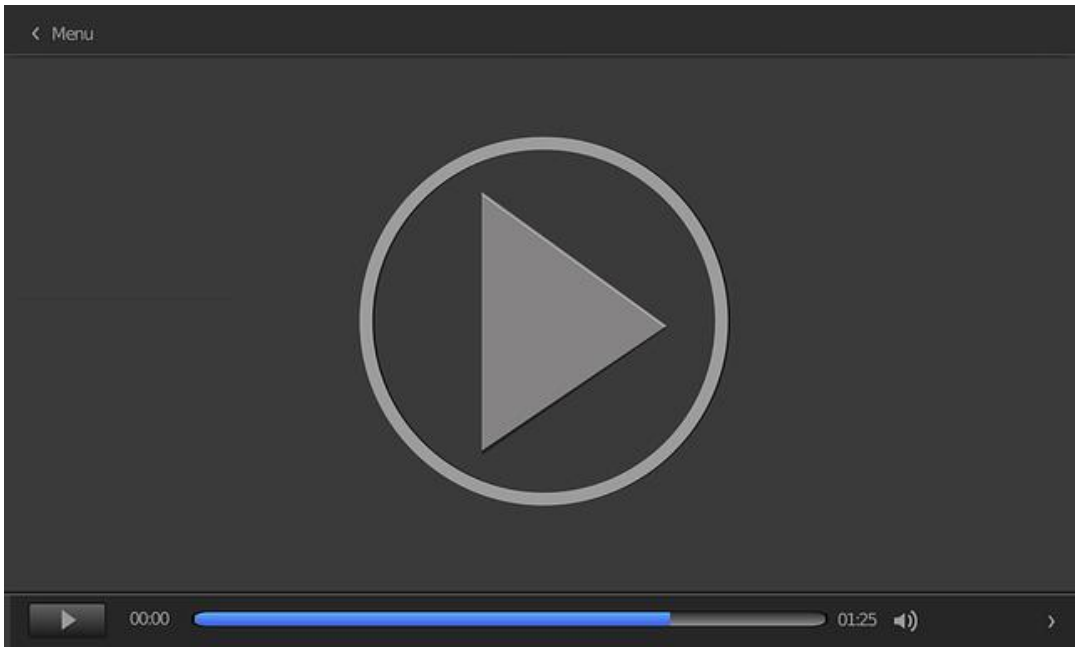
Cohesión

La cohesión tiene que ver con el **grado de relación que tienen los elementos internos dentro del módulo,** o más generalmente, la forma en la que agrupamos elementos en una unidad conceptual de mayor nivel como es el módulo. Por ejemplo, la forma en la que agrupamos métodos en una clase o la forma en la que agrupamos clases en una librería, etc. Su objetivo es **organizar los elementos** de tal manera que los que tengan más relación a la hora de realizar una tarea pertenezcan al mismo módulo y los elementos no relacionados figuren en módulos separados.

La cohesión también se puede definir corno **la medida de la relación funcional de los elementos existentes dentro de un módulo;** se entiende por elementos tanto la sentencia o grupo de sentencias que lo componen como las definiciones de datos o las llamadas a otros módulos. Idealmente, un módulo coherente y diseñado de forma cohesiva entre sus elementos internos solo debe hacer una única cosa.

El objetivo es diseñar módulos con una alta cohesión, cuyos elementos estén fuerte y genuinamente relacionados unos con otros.

A continuación, te dejamos el vídeo *Acoplamiento y cohesión:*



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=5673c768-33aa-402b-8847-aff600bed3c8

1.4. El paradigma de orientación de objetos

Hay diversas formas de enfocar un problema utilizando una solución basada en software. Pero, sin lugar a duda, contar con una aproximación modular se muestra como la más conveniente a lo largo de la historia del desarrollo software.

El modelado, diseño e implementación que usa objetos **es una manera de estructurar los componentes de un sistema software de forma modular.** El objeto se presenta aquí como el **módulo que organiza el código.** Desde una perspectiva más dinámica, los objetos posteriormente serán manipulados mediante una colección de funciones llamadas métodos y se comunicarán entre sí mediante un protocolo de intercambio de mensajes.

La programación orientada a objetos es una forma de implementar programas organizando el código como un conjunto de objetos que cooperan entre sí, donde la definición de cada objeto se encuentra definida en la clase a la que pertenece. A partir de aquí, hay distinguir tres cuestiones entre la **programación estructurada** (funciones y datos) y la **programación orientada a objetos:**

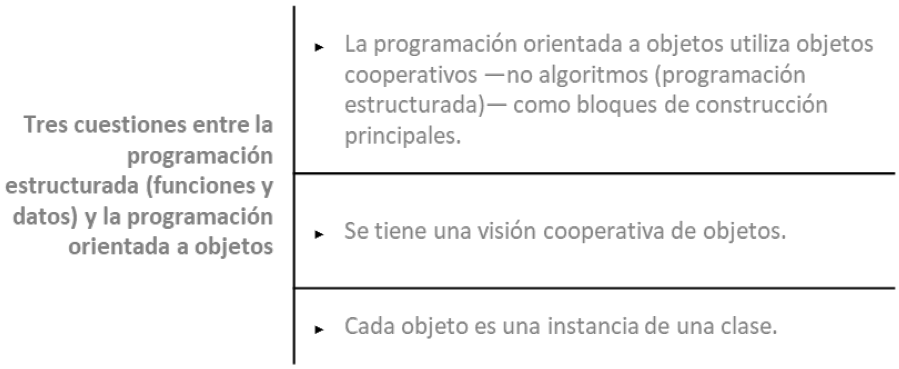


Figura 2. Cuestiones entre la programación estructura y la orientada a objetos. Fuente: elaboración propia.

Ideas clave

El concepto fundamental, como vemos, es el **objeto,** entendido como una **combinación estructural de datos y comportamientos agrupados en una única entidad.** De esta forma, podemos definir un objeto como:

Un objeto es un módulo autocontenido de datos y operaciones que representa una entidad tangible o intangible en el dominio del problema.

De manera general, la programación orientada a objetos divide el proceso de construcción de software en dos partes:

- Definición de clases (tipos de objetos) y sus relaciones.
- Puesta en funcionamiento de los objetos mediante la instanciación y el intercambio de mensajes entre ellos para desarrollar la lógica del programa software.

Clases de objetos

Unido al concepto de objeto está íntimamente relacionado el concepto de clase. La clase en los lenguajes orientados a objetos es la **unidad mínima de descomposición funcional del software** (módulo).

La clase unifica los principios del diseño modular y la definición del tipo de datos.

Desde el punto de vista de los tipos de datos., una clase es un tipo abstracto de datos compuesto por datos y operaciones. Desde el punto de vista del diseño y la programación modular, una clase es un módulo. Desde un punto de vista cognitivo/conceptual, una clase representa una abstracción de alguna entidad presente en el dominio del problema, caracterizada por un conjunto de atributos y por el conjunto de operaciones que admite.

Ideas clave

Una clase es una definición abstracta de un conjunto de objetos. Una definición de clase especifica un comportamiento invariable y atributos comunes a un conjunto de objetos.

A veces, es difícil diferenciar los conceptos de clase y objeto. Un objeto es la particularización o la instanciación de algo general e ideal representado por la clase. Por ejemplo, la clase Cuenta Corriente representa a nivel de abstracción lo que se entiende por cuenta corriente. Cuando hablamos de la cuenta corriente con código 6773-0100-89-2223873, con un saldo 10 000 euros, nos referimos a una cuenta concreta, es decir, a un objeto concreto. En este caso, esta cuenta corriente se debe entender como un objeto particular o, más concretamente en programación, como una instancia de la clase Cuenta Corriente

La clase describe un grupo de objetos que comparten un mismo conjunto de características, mientras que el objeto es un representante particular de ese grupo.

La relación entre clases y objetos es una relación de pertenencia. Así, un objeto se dice que pertenece a una clase. Desde un punto de vista más técnico, **un objeto instancia a una clase.**

La siguiente Figura 3 muestra la relación de instancia de los objetos *jimsAccount, fabsAccount* y *astridsAccount* respecto a la clase Account de forma gráfica usando UML.

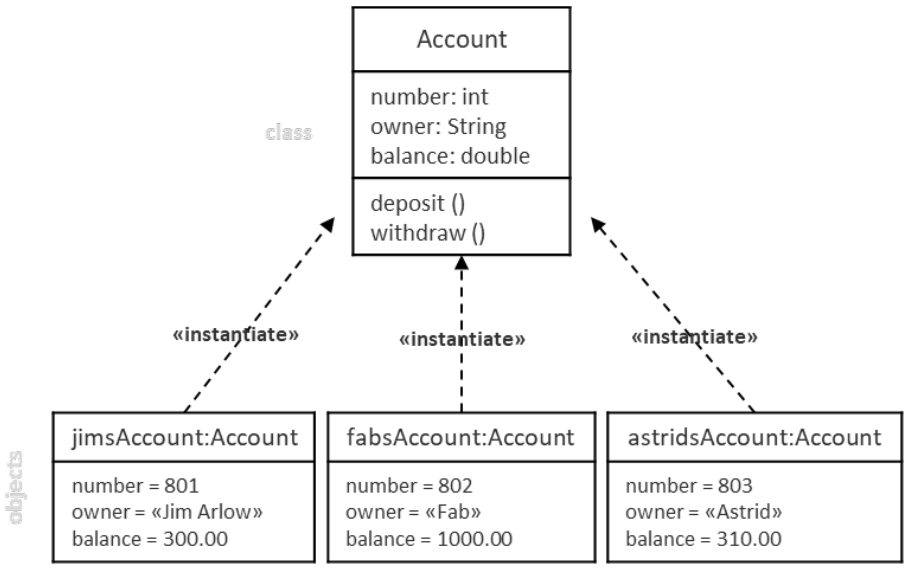


Figura 3. Relación de pertenencia/instancia clase-objeto. Fuente: elaboración propia.

A partir de esta representación, podemos enunciar las siguientes características:

- Todos los objetos del tipo Account comparten una misma caracterización. Es decir, se caracterizan por tener un valor de: *number, owner* y *balance*
- Todos los objetos de tipo Account se comportan de la misma forma. Entendiendo por comportamiento la posibilidad de responder ante la llamada a las operaciones *deposit()* y *withdraw().*

Objetos

El manual de referencia de UML (The UML Reference Manual) define un objeto como: «Una entidad discreta con limites perfectamente definidos, que encapsula un estado y un comportamiento instancia de una clase».

En este mismo manual de referencia, se describe un objeto como: «un objeto es un conjunto cohesivo de datos y funciones».

Cuando hablamos de **objetos,** hacemos referencia a un **conjunto de datos concretos que pueden cambiar;** mientras que cuando hablamos de **clase,** hacemos referencia a **como es la estructura que representa a esos objetos y que se usa como plantilla para crearlos.** En este sentido, la clase es un concepto estático, mientras que el objeto es un concepto más dinámico.

Mientras existe, un objeto tiene un estado y un comportamiento. El estado se expresa mediante atributos y el comportamiento se expresa mediante los métodos asociados con el objeto. Los objetos individuales tienen identidad. Para poder utilizar cualquier objeto, se requiere el uso de su identidad. En Java, por ejemplo, se usan referencias para realizar un seguimiento de los objetos individuales. Las referencias de Java son variables que se declaran utilizando el nombre de la clase o el tipo de objeto. **Es posible tener más de una referencia que se refiera al mismo objeto.** Los mensajes se envían a un objeto utilizando su referencia con el nombre de método apropiado.

Una vez que un objeto ya no es necesario se puede destruir. La destrucción de un objeto puede ser gestionada por el entorno de ejecución (como en el caso de Java y su recolector) o, explícitamente, haciendo una llamada al método destructor del objeto.

Propiedades de una clase

Conceptualmente y de forma general, **una clase queda caracterizada por un nombre de clase, un conjunto de atributos y un conjunto de métodos.** La siguiente Figura 4 muestra una representación gráfica de una clase en notación UML.

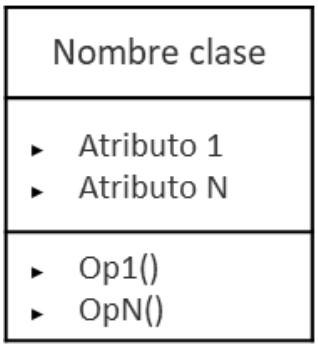


Figura 4. Representación UML de una clase. Fuente: elaboración propia.

En esta representación podemos ver claramente dos secciones diferenciadas:

- La sección de **definición de atributos:** representa las características que definen a la clase de objetos y, en conjunción, hace que se puedan diferenciar de los demás tipos de objetos. Por ejemplo, una clase Producto podría estar caracterizada por un identificador, una descripción, un modelo y un precio.

- La sección de **definición de métodos:** representa el comportamiento de los objetos ante la invocación de determinados mensajes. Por ejemplo, para la clase Producto podríamos querer obtener su precio o su descripción.

A nivel de lenguaje de programación, la clase debe representarse obviamente con los elementos de sintaxis propios que defina el lenguaje. En concreto, para el lenguaje Java se representa de la siguiente forma:

```
public class Nombre_Clase {

    //atributos de la clase

    [visibilidad] Tipo_atributo nombre_atributo1;

    [visibilidad] Tipo_atributo nombre_atributoN;

   //métodos de la clase
```

```
    [visibilidad] Tipo_retorno nombre_método1 (parametros){}

    [visibilidad] Tipo_retorno nombre_métodoN (parametros){}

}
```

Donde [visibilidad] indica la restricción de acceso permitido y puede ser por regla general. Los valores posibles son: **privado** *(private)* o **público** *(public)*. Existe otro modificador denominado **visibilidad protegida** *(protected),* que se usa para las relaciones de herencia. **Por defecto, si no se indica nada, se toma como acceso privado.**

Por ejemplo, una clase que representa un ordenador como tipo de objetos en Java podría representarse de la siguiente forma:

```
public class Ordenador {

    //atributos de la clase

    String marca;

    String modelo;

    String procesador;

    int memoria;

    float frecuencia;

    boolean encendido;

    //métodos

    public Ordenador(){}

    public Ordenador (String mar,String mod,String micro,int men,float fre){
```

```
        marca=mar;
        modelo=mod;
        procesador=micro;
        memoria=men;
        frecuencia=fre;
        }
        public void encender(){
            if (!encendido)
                encendido=true;
        }
}
```

Atributos de una clase

Cualquier objeto o entidad existente se puede describir a partir de una serie de atributos, como hemos visto. Una clase puede quedar definida por un número arbitrario de atributos que definirán sus características más relevantes dentro del dominio del problema y que identificarán unívocamente el tipo de objetos que se está representado.

Un atributo es una característica o propiedad de una entidad conceptual representada como una clase.

Un atributo siempre toma valor en un dominio determinado; el cuál define el tipo de valores válidos en la interpretación de ese atributo. Por ejemplo, suponga el atributo color de una clase denominada Coche, los valores válidos del domino en el atributo color serían, por ejemplo: **blanco, negro, plata, gris, azul, rojo, amarillo, verde.** Existen dominios predefinidos por los lenguajes, como los enteros, reales o cadenas alfanuméricas y otros que se pueden definir específicamente.

Estado de un objeto

En un instante de tiempo determinado en la ejecución del programa software, un objeto creado tendrá una serie de valores en todos y cada uno de sus atributos. Al conjunto de valores que toman los atributos de un objeto en un instante de tiempo dado se denomina **estado del objeto.**

El estado de un objeto es el valor de sus atributos observados en un instante de tiempo determinado.

Como es evidente, el estado de un objeto puede variar a lo largo del ciclo de vida del objeto en el programa. El cambio de estado se realizará a partir de la invocación de los métodos permitidos y definidos en el comportamiento del objeto.

El concepto de estado **define los valores transitorios y estables por los que puede pasar un objeto,** de forma que se puede determinar la corrección de un sistema a partir del conjunto de estados alcanzado por los objetos del sistema.

Mensajes, operaciones y métodos

Cuando se diseña una clase, la estructura interna y los detalles de implementación se ocultan (encapsulan), con la posibilidad de intercambiar mensajes como la única posibilidad de conexión con el objeto de la clase.

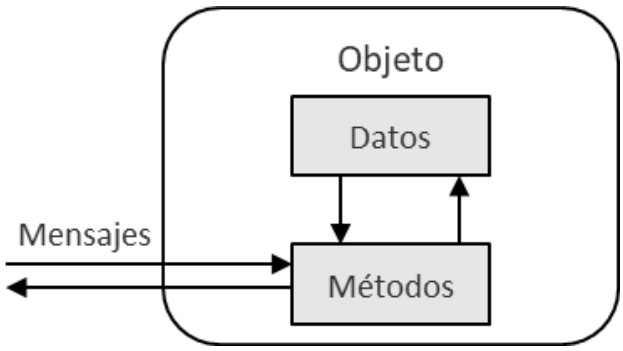


Figura 5. Mensajes, operaciones y métodos. Fuente: elaboración propia.

Un método es la especificación e implementación de un servicio u operación que puede ser requerido a cualquier objeto de la clase.

Recordemos que la ocultación de la representación interna al exterior obedece a criterios de diseño y, en concreto, el objetivo es bajar el acoplamiento.

Cada método está caracterizado por un nombre y un cuerpo de implementación que lleva a cabo algún tipo de acción o comportamiento asociado.

Cuando un objeto recibe algún mensaje, siempre inicia algún tipo de procesamiento o acción. **Cada una de las operaciones que define la clase proporciona una representación del comportamiento de los objetos a los que representa.** Por eso, el conjunto de operaciones que admite un objeto o clase se denomina comportamiento del objeto, y está íntimamente relaciona con la funcionalidad que puede realizar.

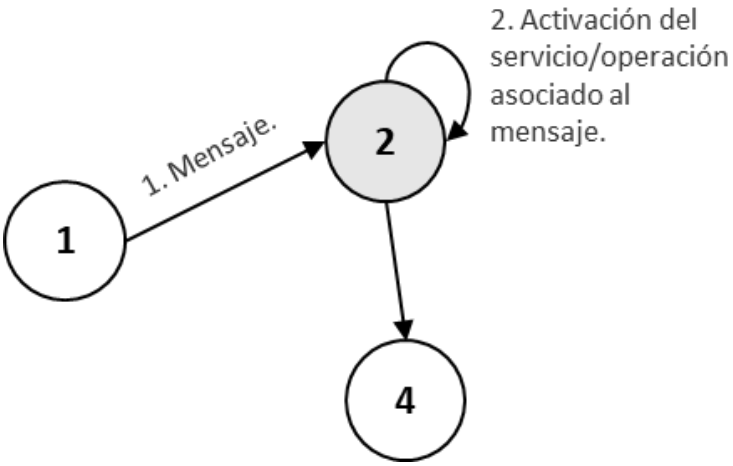


Figura 6. Comportamiento del objeto. Fuente: elaboración propia.

En un modelo de clases y sus objetos, los métodos (operaciones o servicios) asociados a estas clases describen el comportamiento asociado a un objeto.

Supongamos una clase llamada Punto, que representa puntos en un espacio bidimensional y que podemos representar gráficamente en notación UML de la siguiente forma:

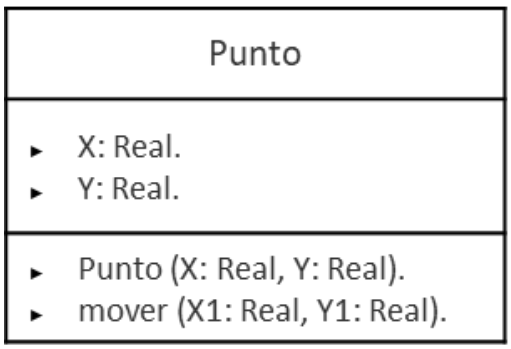


Figura 7. Clase Punto. Fuente: elaboración propia.

La consecuencia de la existencia del método **mover (X1: Real, Y1: Real)** es que la clase Punto ha sido diseñada para recibir un estímulo, llamaremos a este estímulo **mensaje.** Cada vez que un objeto recibe un mensaje/estimulo inicia un comportamiento. El comportamiento especificado para un objeto instancia de esta clase en particular es moverse a un desplazamiento determinado. **El método o procedimiento para moverse,** en este caso particular, vendrá implementado en el cuerpo del método **mover (X1: Real, Y1: Real).**

Un método se implementa en una clase de objetos e indica cómo debe actuar el objeto cuando recibe el mensaje vinculado a ese método. Por ejemplo, el método para mover un punto es incrementar el valor de los atributos X e Y en tantas unidades como se le indique. Debemos considerar, a este respecto, que **la ejecución de un método puede conducir a un cambio de estado del objeto.**

Definición de un método

Un método se define por un nombre, por los argumentos o parámetros de entrada que necesita y por el valor de retorno que se obtiene al ejecutar el comportamiento que implementa. Estos elementos (valor de retorno, nombre y parámetros) de un método se conocen como prototipo del método o de una función miembro de la clase.

De manera general, el esquema de implementación sería el siguiente:

NombreMetodo (parametro1 Tipoparam1…, parametroN TipoparamN) → Tipo de retorno.

La implementación del método define un procedimiento o algún servicio que se llevará a cabo como consecuencia de la invocación del método.

De manera informal, se pueden clasificar los tipos de métodos dentro de una clase en los siguientes:

- Métodos constructores y destructores de objetos.
- Métodos de acceso a propiedades: métodos set (establecer) y get (obtener).
- Métodos de servicio (en general).

La primera categoría representa los métodos dedicados a crear y eliminar objetos. Veremos, más adelante, estos dos métodos en profundidad. La segunda categoría corresponde a unos tipos de métodos muy comunes en cualquier clase de objetos, los métodos *set* y get están diseñados para obtener y establecer el valor de los atributos de un objeto. La tercera, son métodos generales que ofrecerán algún servicio o comportamiento específico del objeto en el dominio del problema.

Métodos constructores y destructores

El **objetivo del constructor es el de inicializar un objeto dando valor a sus atributos** cuando este es creado. Cuando se crea un objeto (instanciación), se pueden pasar los valores de los parámetros al constructor utilizando una sintaxis similar a una llamada a un método. El siguiente ejemplo en Java muestra la creación de un objeto Punto en las coordenadas (1,3):

```
Punto p= new Punto (1,3);
```

Cuando se ejecuta la instrucción anterior, el compilador reservará memoria para albergar un objeto instancia de la clase Punto y llamará al código asociado el constructor de esta clase.

Un constructor es el mecanismo del que disponen las clases para inicializar los objetos.

Un constructor se ejecuta cuando se crea una instancia de una clase. El constructor tiene como propósito principal la inicialización de los datos miembro o atributos del objeto de la clase, lo que deja al objeto en su estado inicial.

Podemos ver un constructor como un procedimiento en general con las siguientes características:

Características de un constructor
Es un **método obligatorio** de la clase.
Tiene el **mismo nombre que la clase.**
Puede tener cualquier número de argumentos (incluso ninguno).
Puede haber más de un constructor, es decir, distintas maneras de crear un objeto.

Figura 8. Características de un constructor. Fuente: elaboración propia.

1.5. Tarjetas CRC (Class, Responsability, Collaboration)

Las tarjetas CRC constituyen una técnica para diseñar clases a través de la identificación de las colaboraciones y responsabilidades presentes en una clase de objetos. Fueron introducidas en la mayor conferencia de diseño orientado a objetos, OOPSLA89, por Kent Beck y Ward Cunningham.

Las tarjetas CRC son útiles, sobre todo, en etapas tempranas de desarrollo, en las que se está identificando clases en el dominio del problema

La tarjeta CRC se compone de las siguientes partes:

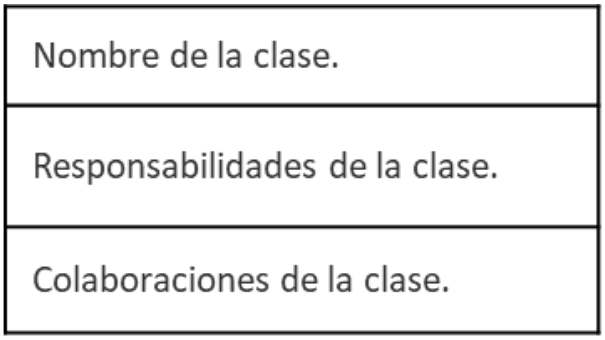


Figura 9. Partes de la tarjeta CRC. Fuente: elaboración propia.

A continuación, se describen cada una de las secciones:

- **Nombre de la clase:** nombre descriptivo de la entidad a la que representa la clase. Por ejemplo: Libro, Socio, Préstamo.

- **Responsabilidades de la clase:** describen a alto nivel el propósito de la existencia de la clase y están relacionadas estrechamente con las operaciones que proporciona la clase. Por ejemplo, para una clase Libro, la responsabilidad mantener los datos del libro es una responsabilidad válida. Se recomienda que una clase no tenga más de tres o cuatro responsabilidades, y muchas de ellas tendrán solo una o dos. Demasiadas responsabilidades están relacionadas con una mala cohesión en el diseño de la clase. Cada una de las operaciones que pueda ejecutar un objeto debería formar parte de las responsabilidades de la clase.

- **Colaboraciones de la clase:** normalmente, aparecerá como una relación semántica dentro del dominio del problema. Si un objeto de una clase colabora con otro, lo hará enviándole algún mensaje. Si se identifican muchas colaboraciones, se tendrá un acoplamiento alto.

Las siguientes son ejemplos tarjetas CRC:

Médico	
Responsabilidad	**Colaboradores**
▸ Atender pacientes. ▸ Diagnosticar a pacientes.	▸ Paciente. ▸ Diagnostico.
Paciente	
Responsabilidad	**Colaboradores**
▸ Mantener los datos sobre un Paciente. ▸ Mantener el diagnóstico realizado por el médico.	▸ Diagnóstico.
Diagnóstico	
Responsabilidades	**Colaboradores**
▸ Mantener los datos sobre un diagnóstico. ▸ Mantener los datos de un Tratamiento.	▸ Tratamiento.

Figura 10. Ejemplos de tarjetas CRC. Fuente: elaboración propia.

1.6. Modelado con objetos

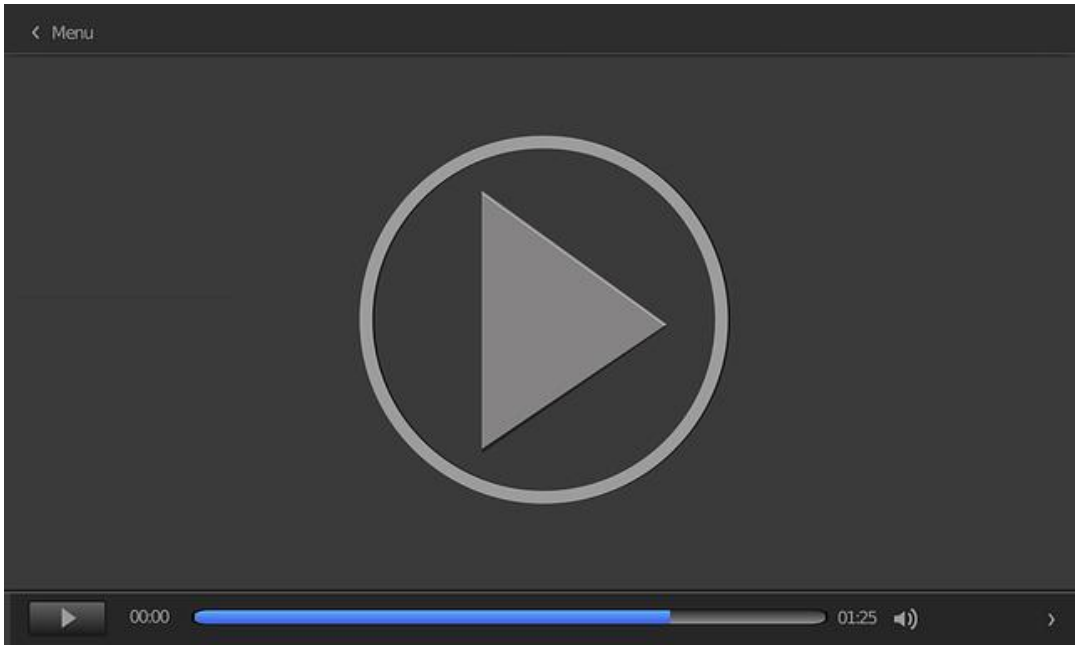
En general, el modelado con objetos exige un análisis de la semántica del dominio del problema donde se identifiquen entidades conceptuales relevantes para después buscar su caracterización, responsabilidades y colaboraciones o asociaciones

De manera informal, podríamos definir el proceso en los siguientes pasos:

- **Identificar las clases de objetos relevantes** en el domino del problema junto con los atributos que los caracterizan: utiliza nombres significativos, por ejemplo, Libro, Préstamo, Socio, Punto, Recta.
- **Asignar responsabilidades a cada clase de objeto:** una responsabilidad es algo que una clase conoce o hace. Ejemplo de esto sería la clase Estudiante, que conoce su nombre, su DNI, su domicilio. Un Estudiante se matricula en asignaturas, se presenta a exámenes, etc.
- **Buscar colaboraciones y asociaciones entre objetos:** la lista de colaboradores de una clase debe incluir todas las clases que necesita conocer para hacer algo. Por ejemplo, las que suministran servicios que la clase actual necesita para llevar a cabo alguna de sus responsabilidades

Todos los objetos establecen relaciones con otros objetos. Es difícil encontrar un objeto que no requiera de una colaboración. Llamamos **objetos colaboradores** a aquellos que **recibirán mensajes o enviarán mensajes al objeto actual, con el objetivo de poder llevar a cabo una responsabilidad.**

A continuación, puedes ver el vídeo *Entorno de desarrollo eclipse,* el video muestra las partes más importantes de la interfaz para que puedan desarrollar sus proyectos de programación en este entorno.



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=e1ec7567-384a-4306-83fe-aff600c517c4

1.7. Cuaderno de ejercicios

1. Defina una clase que represente una matriz de números enteros de dimensiones N x M. Defina e implemente el **método getValor (x,y)** que devuelve el entero en la posición (x,y) de la matriz.

Solución

```
public class Matriz {

    int [][] valores;



    public Matriz (int n, int m) {

        valores = new int [n][m];

    }

    public int getValor(int x, int y) {

        return valores[x][y];

    }

}
```

2. La sucesión de Fibonacci es una sucesión definida por recurrencia. Los números de Fibonacci quedan definidos por las siguientes ecuaciones para n>=0:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

Figura 11. Ejercicio 2. Fuente: elaboración propia.

Implemente una clase que represente un generador de términos de la serie de Fibonacci. El diseño de la clase se muestra en el siguiente diagrama UML:

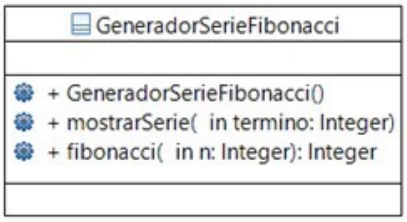


Figura 12. Ejercicio 2 parte 2. Fuente: elaboración propia.

Donde:

- El método **mostrarSerie (int termino)** imprime por pantalla la serie de Fibonacci hasta el término que indica el parámetro.
- El método **fibonacci(n)** devuelve el término n-ésimo de la sucesión.

Una posible solución sería

```
public class GeneradorSerieFibonacci {



public GeneradorSerieFibonacci() {
```

```
// constructor vacio
    }

public int fibonacci(int n)
{
    if (n>1){
        return fibonacci(n-1) + fibonacci(n-2); //Llamada recursiva
    }
    else if (n==1) { // caso n=1
        return 1;
    }
    else if (n==0){ // caso n=0
        return 0;
    }
    else{ //error
        System.out.println(" N debe ser => 0");
        return -1;
     }
  }
```

```
    public void mostrarSerie(int termino){



        for (int i = 0; i < termino; i++) {

            System.out.print(fibonacci(i)+" ");

        }

        System.out.println();

    }

}
```

3. Responda a las siguientes preguntas con respecto al ejercicio anterior:

- ¿Qué ocurriría si el método fibonacci(n) definido por la clase GeneradorSerieFibonacci se hiciera privado? ¿Se podría usar desde fuera de la clase?
- ¿Afecta este cambio al método mostrarSerie(int termino)?

Solución

- El métodod fibonacci(n) no podría ser invocado directamente mediante la instanciación de un objeto.
- No, respecto al método mostrarSerie(int termino), no tendría efecto puesto que es un método de la clase y tiene acceso al método tanto si se define privado o se define protegido.

4. Dada la siguiente codificación de la clase BufferMuestras, represéntela usando UML.

```
import java.util.ArrayList;

public class BufferMuestras {

    ArrayList <Muestra> lista;

    public BufferMuestras(int n) {

        lista = new ArrayList<Muestra>(tam); // lista con tamaño n

    }

    private void cleanBuffer() {

        lista.clear();

    }

    public Muestra getItem(int pos) {

        return lista.get(pos);

    }

    public void setItem(int pos, Muestra m) {

        lista.add(pos, m);

    }

}
```

Solución

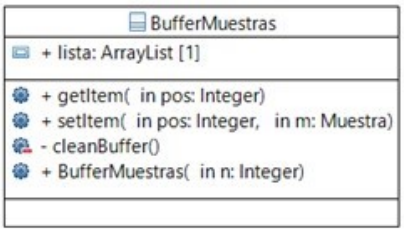


Figura 13. Solución ejercicio 4. Fuente: elaboración propia.

5. Tomando la definición de la clase del ejercicio anterior, indique cuál sería su interfaz pública y cuál sería su interfaz privada.

Solución

La **interfaz publica** la forman los métodos que se han definido con le modificador public y son:

BufferMuestras (int n).

getItem.

setItem.

La **interfaz privada** la forman los métodos declarados con la restricción de acceso private y son:

cleanBuffer.

6. Realice un análisis utilizando la técnica de tarjetas CRC sobre la siguiente definición textual de una clase:

> «Una Cuenta Bancaria mantiene un saldo activo, un titular de cuenta, una fecha de apertura y una serie de comisiones asociadas. Además, la cuenta corriente mantiene una serie de domiciliaciones, que, llegada la fecha en la que están establecidas decrementan el saldo activo de la cuenta».

Cuenta Corriente	
Responsabilidad	Colaboradores

Figura 14. Ejercicio 6. Fuente: elaboración propia.

Posible solución

Cuenta Corriente	
Responsabilidad	**Colaboradores**
▸ Mantener información de saldo activo, titular, fecha de apertura. ▸ Decrementar periódicamente el saldo ante la fecha efectiva de una domiciliación.	▸ Titular, Comisión, Domiciliación (pueden ser clases).

Figura 15. Solución ejercicio 6. Fuente: elaboración propia.

Dada la siguiente definición de clase:

```
public class Circle {

    // atributos de clase

    private double radius;

    private String color;

// Constructor

/** Construye una instancia círculo con los valores de color y radio por defecto */

public Circle() { // constructor por defecto

    radius = 1.0;

    color = "red";

}

    /** Construye una instancia círculo con el valor de radio dado por parámetro */

    public Circle(double r) { // constructor

        radius = r;

        color = "red";

    }
```

```
    /** Devuelve el valor de radio */
     public double getRadius() {
          return radius;
     }
     /** Devuelve el valor del área */
     public double getArea() {
         return radius*radius*Math.PI;
    }
}
```

7. Defina las instrucciones necesarias para llevar a cabo las siguientes acciones:

- Crear un círculo de radio cinco.
- Obtener el área del círculo anterior e imprimirla por pantalla.
- Sobrecargar el constructor para que admita como parámetro un valor de radio y un color.
- Añadir un método para que sea posible obtener el color de un círculo.

Solución

Crear un círculo de radio cinco:

Circle c= new Circle (5)

Obtener el área del círculo anterior e imprimirla por pantalla:

System.out.println(c.getArea());

Modificar el constructor para que admita como parámetro un color distinto:

```
public Circle(double r, string c) {
    radius = r;
    color = c;
}
```

8. Añadir un método para que sea posible obtener el color de un círculo:

```
public String getColor() {
    return color;
}
```

9. Dada la siguiente clase UML, implemente el método **incrementarSalario (int inc),** donde inc representa el incremento porcentual sobre el salario. Nota: el atributo salario representa el salario anual.

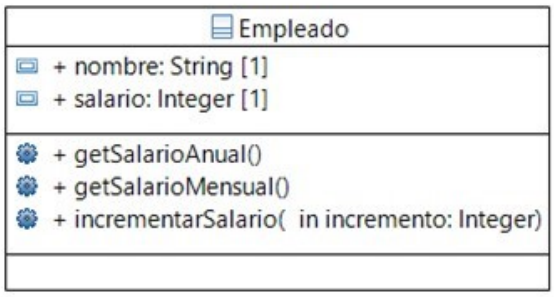


Figura 16. Ejercicio 8. Fuente: elaboración propia.

Solución

```
public void incrementarSalario (int inc) (int inc){

//asignamos al salario actual el incremento porcentual de la subida

This.salario= this.salario*(inc/100);

}
```

10. Complete la definición de la clase anterior con los métodos **set** que considere que faltan.

Solución

Los métodos de acceso a atributos que faltarían sería los siguientes:

getNombre():String.

getApellidos():String.

1.8. Referencias Bibliográficas

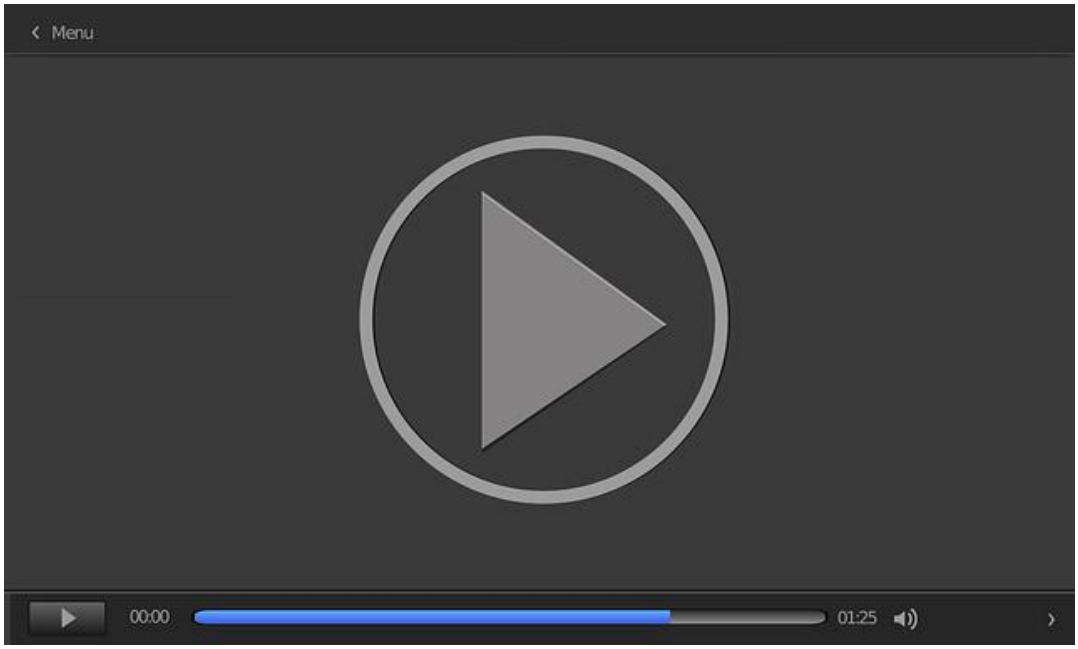
Stevens, P. R. (2002). *Utilización de UML en Ingeniería del Software con Objetos y Componentes*. Addison Wesley.

Rumbaugh, J., Jacobson, I. y Booch, G. (2006). *El lenguaje unificado de modelado.* Addison Wesley. https://ingenieriasoftware2011.files.wordpress.com/2011/07/el-lenguaje-unificado-de-modelado-manual-de-referencia.pdf

S.O.L.I.D. Principles of Object-Oriented Design — A Tutorial on Object-Oriented Design

Grace Hopper Academy. (2016, octubre 25). *S.O.L.I.D. Principles of Object-Oriented Design - A Tutorial on Object-Oriented Design.* [Vídeo]. YouTube. https://www.youtube.com/watch?v=GtZtQ2VFweA

En este tutorial se da una descripción general de los principios de diseño orientados a objetos denominados SOLID, un concepto introducido por Robert C. Martin en 1995. Estas reglas están diseñadas para ayudar al programador a desarrollar un software que sea fácil de mantener y ampliar.



Accede al vídeo:

https://www.youtube.com/embed/GtZtQ2VFweA

Mesa redonda sobre programación orientada a objetos

JDGARCÍA. (2014, diciembre 3). *Mesa redonda: ¿La programación orientada a objetos debe morir?.* Arcos. https://www.arcos.inf.uc3m.es/jdgarcia/2014/12/03/round-table-oop-must-die-spanish/

Interesante mesa redonda donde el profesor José Daniel García debate con otros profesores sobre las ventajas e inconvenientes de la programación orientada a objetos.

Diseñar y programar, todo es empezar. Vélez, J. (2011)

Vélez Serrano, J. F., Peña Abril, A., Sánchez Calle, A. y Gortazar Bellas, P. (2011). *Diseñar y programar, todo es empezar: una introducción a la programación orientada a objetos usando UML y Java.* Universidad Rey Juan Carlos. Disponible en la Biblioteca Virtual de UNIR.

Una introducción a la programación orientada a objetos usando UML y Java. Dentro de este libro podrás encontrar una descripción de cómo desarrollar software orientado a objetos, desde el diseño hasta la implementación, aplicando técnicas de ingeniería del software.

Guía sencilla para la representación de UML

Microsoft 365 Team. (2019, septiembre 24). *La guía sencilla para la representación de UML mediante diagramas y la creación de modelos de base de datos.* https://www.microsoft.com/es-es/microsoft-365/business-insights-ideas/resources/guide-to-uml-diagramming-and-database-modeling

UML es el lenguaje estándar para representar los distintos modelos que se usan en desarrollo e ingeniería software. En esta página encontrarás una sencilla guía que te ayudara a comprender mejor este importante lenguaje relacionado estrechamente con la programación orientada objetos.

Test

1. Respecto al método constructor de una clase, señale la incorrecta:

A. El constructor por defecto asigna valores por defecto a cada atributo de la clase.

B. Una clase puede definir más de un constructor.

C. El método constructor puede estar sobrecargado.

D. El método constructor debe llevar parámetros obligatoriamente.

2. ¿Cuál no es una propiedad de la programación orientada a objetos?

A. Encapsulación.

B. Módulos cohesivos.

C. Módulos acoplados.

D. Visión de objetos cooperantes.

3. Sobre los métodos públicos definidos para una clase de objetos, señale la incorrecta:

A. Definen el comportamiento de una clase de objetos.

B. Solo son accesibles por objetos del mismo tipo de clase.

C. Pueden ser invocados por otros objetos.

D. Constituyen la única vía para comunicarse con el objeto.

4. Respecto a los métodos de una clase, indique la afirmación correcta:

A. Los métodos de una clase identifican el comportamiento de la clase.

B. Los métodos de una clase siempre tienen que ser públicos y accesibles desde fuera.

C. Los métodos de una clase no definen comportamiento.

D. Los métodos de una clase cambian no cambian el estado del objeto.

5. ¿Qué es instanciar una clase?

A. Eliminar una clase.

B. Copiar una clase.

C. Implementar una clase

D. Crear un objeto.

6. Sobre el concepto de objeto indique la afirmación incorrecta:

A. Todo objeto pertenece a una clase.

B. Todo objeto tiene un estado.

C. Toda clase es instancia de un objeto

D. Distintos objetos de la misma clase pueden existir al mismo tiempo.

7. ¿Cuál de las siguientes no es una propiedad característica de un sistema software orientado a objetos?

A. Clases encapsuladas.

B. Comunicación entre objetos vía mensajes.

C. Baja cohesión en el diseño de clases.

D. Alto grado de modularidad.

8. Señale la respuesta correcta:

A. Todos los métodos de una clase deben ser privados.

B. Todos los métodos de una clase deben ser públicos.

C. Los métodos de una clase pueden ser privados o públicos.

D. El constructor de una clase debe ser privado.

9. Los métodos denominados *getters* se utilizan para:

A. Obtener el valor del estado inicial del objeto.

B. Asignar el valor a los atributos de una clase.

C. Obtener el valor de los atributos de una clase.

D. No existen dentro de la definición de una clase.

10. Respecto a la encapsulación en el diseño modular, señale la incorrecta:

A. Es el proceso de ocultar los detalles de implementación internos.

B. Es la responsable de que el acoplamiento se incremente.

C. Se fuerza en los lenguajes de programación mediante modificadores de acceso.

D. Permite que el estado de un objeto solo pueda ser cambiado mediante métodos específicos.