Tecnologías de Programación

Resumen 2025

Paradigmas de Programación

Un **paradigma** es un modelo que determina una manera especial de entender problemas y sus soluciones.

Un **paradigma de programación** es un enfoque particular para la construcción de software. Cada paradigma tiene ventajas y desventajas y, en teoría, cualquier problema debería poder ser resuelto por cualquier paradigma *pero existen situaciones donde un paradigma es más apropiado que otro*. Por esto, para resolver un determinado problema debemos conocer cuál paradigma se adapta mejor a su resolución

Programación

Un **programa** es un conjunto de código que describen, definen o caracterizan las acciones que puede ejecutar una computadora. Como la estructura final de un programa es determinada por el paradigma que se utilizó para su diseño existen diferentes conceptos de "programa"para cada paradigma.

El proceso de **programación** incluye diseñar, codificar, depurar y mantener el código.

Criterios de la buena programación:

- Modelos cercanos a la *realidad*
- Implementaciones que puedan ser *extendidas* y *modificadas*
- Flexibilidad para la *reutilización*
- *Código claro* y compacto
- Soluciones *genéricas*
- *Focalización* de las funciones

Paradigmas fundamentales

Los primeros lenguajes de programación eran **imperativos** o **procedurales**, se centran en operaciones que modifican datos en memoria y el estado del programa a través de algoritmos que detallan *el paso a paso* con estructuras de control. Se suele usar este paradigma cuando hay una secuencia clara y queremos tener mayor control sobre el algoritmo. (**Paradigma estructurado**)

Los paradigmas **declarativos** surgieron como alternativa, enfocándose en el *qué* de la solución en vez del *cómo*. Utilizan declaraciones (condiciones, proposiciones, restricciones, etc) para describir el problema, dejando los detalles de la implementación a mecanismos internos. Se suele usar este paradigma cuando se expresan características del problema y queremos tener menos control sobre el algoritmo. (**Paradigmas funcional y lógico**)

El paradigma **orientado a objeto** agregan el concepto de objeto, que encapsula datos y comportamientos.

Paradigma Estructurado

ALGORITMO + ESTRUCTURA DE DATOS = PROGRAMA

Se define por el estado del programa y las instrucciones que lo modifican.

Lenguajes asociados: Fortran, C, Pascal

Ventajas	Desventajas
• Código directo y de rápida ejecución • De fácil corrección y mantenimiento (*De estar bien modularizado*)	

Paradigma Funcional

FUNCIONES + CONTROL = PROGRAMA

Se enfoca en la aplicación y composición de funciones, permitiendo soluciones más elegantes.

Lenguajes asociados: Lisp, Haskell, Scheme, Scala.

Aplicaciones: IA y aprendizaje automático, sistemas de control tolerantes a fallos de software (ej. tráfico aéreo, mensajería instantánea, etc), matemáticas y científicas, procesamientos de datos, telecomunicaciones, interfaces de usuario reactivas.

Ventajas	Desventajas
• Altos niveles de abstracción • Cortos y sencillos de entender (*Por el nivel de abstracción*) • Rapidez en la codificación • Administración automática de memoria. • Evita gastos computacionales innecesarios (*Por la evaluación perezosa*)	• Portabilidad y herramientas limitadas. • Baja eficiencia de ejecución. (*Por estar alejado al modelo de la máquina de von Neumann*)

Paradigma Logico

LÓGICA + CONTROL = PROGRAMA

Basado en la definición de reglas lógicas para responder, mediante un motor de interferencias lógicas, preguntas planteadas al sistema.

Lenguajes asociados: Prolog, Datalog.

Aplicaciones: Sistemas expertos, sistemas de soporte de decisiones (DSS), sistemas de reglas de negocio (Drools), sistemas de conocimiento y aprendizaje, robótica.

Ventajas	Desventajas
• Sencillez y potencia en la búsqueda de soluciones. • Sencillez en la implementación de estructuras complejas.	• Poco utilizado en gestión. • Pocas herramientas de depuración. • Pocos eficientes por lentitud de ejecución.

Paradigma Orientado a Objetos

OBJETOS + MENSAJES = PROGRAMA

Se basa en la idea de *encapsular* estado y operaciones en objetos los cuales no están aislados entre sí, sino que colaboran para lograr un único objetivo.
La colaboración se realiza a través de *mensajes*.
Lenguajes asociados: Smalltalk, Java, C++, Python.
Aplicaciones: Tecnología web, modelado y simulación, bases de datos, programación en paralelo, sistemas que requieren administración de estructuras complejas.

Ventajas	Desventajas
• Reusabilidad. • Mantenibilidad (*Fácil abstracción*) • Fiabilidad (*Fácil aislación de problemas*)	• Algunos de los mecanismos (*Como la herencia*) provocan una ejecución más lenta. • Curva de aprendizaje más grande.

Ejemplo

Calcular promedio. Mismo problema, diferentes paradigmas.

Imperativo

```python
numeros = [10, 20, 30, 40, 50]
suma = 0

for numero in numeros:
    suma += numero

promedio = suma / len(numeros)
print("promedio:", promedio)
```

Python.
Define explícitamente cómo sumar los números y cómo calcular el promedio mediante iteraciones y asignación de variables.

Orientado a objetos

```python
class CalculadoraPromedio:
    def __init__(self, numeros):
        self.numeros = numeros
    def calcular_promedio(self):
        suma = sum(self.numeros)
        return suma / len(self.numeros)

numeros = [10, 20, 30, 40, 50]

calculadora = CalculadoraPromedio(numeros)
print("Promedio:",
calculadora.calcular_promedio())
```

Python.
Modela objetos y sus interacciones.

Funcional	
(define (promedio numeros) (define suma(apply + numeros)) (/suma (length numeros)) (define numeros '(10 20 30 40 50)') (display ln (promedio numeros))	Racket. La función promedio no depende de ningún estado ni modifica variables globales-
Lógico	
promedio(Numeros, Promedio):- suma(Numeros, Suma) length(Numeros, Len) Promedio is Suma / Len. suma ([],0). suma ([H|T], Suma) :- suma (T, ST), Suma is H + ST	Prolog. Define las relaciones lógicas entre los elementos en lugar de dar instrucciones explícitas.

Lenguajes de programación

Un **lenguaje de programación** es una notación para escribir programas y darle órdenes para la realización de un proceso al Hardware.

Lenguajes Puros

Se apegan rigurosamente a las normas de un paradigma. Tienen una mayor potencia expresiva.

Lenguajes: Smalltalk, Eiffel, HyperTalk y Acto.

Lenguajes Hibridos

Integran aspectos distintivos de diferentes paradigmas. Son menos dogmáticos.

Lenguajes: Objetive C, Objetive Pascal, Java, C#.

Paradigma Orientado a Objetos

Está basado en la encapsulación del estado, con **atributos**, y el comportamiento, **métodos**. Procura favorecer un buen diseño modular, conformado por pocas interfaces pequeñas y explícitas.

Ventajas	Desventajas
• Reusabilidad. • Fácil mantenimiento y modificación. • Estructura modular clara. • Buen marco para GUIs. • Acopla bien a bases de datos.	• Limitaciones del programador. • Múltiples formas de resolver problemas. • Requiere amplia documentación.

Para el paradigma,

*Un programa es un conjunto de **objetos** que colaboran entre sí enviándole mensajes.*

Características de la POO

Abstracción: Seleccionar las características esenciales de un objeto a modelar e identificar comportamientos comunes.

Modularidad: Subdividir una aplicación en partes más pequeñas (módulos) que deben ser lo más independientes posibles de la aplicación en sí y las partes restantes. Se pueden compilar por separado.

Encapsulamiento: Reunir todos los elementos pertenecientes a una misma entidad al mismo nivel de abstracción.

Principio de ocultación: Aislar cada objeto del exterior y exponer una interfaz (protocolo) a otros objetos que específica cómo interactuar con él. Protege las propiedad de un objeto de modificaciones que no sean por métodos propios, eliminando interacciones inesperadas.

Cohesión y acoplamiento: Ofrecer alta cohesión (Cada objeto se responsabiliza por *una sola* cosa) y bajo acoplamiento (poca o nula interdependencia)

Herencia: Formar una jerarquía de clasificación. Los objetos heredan propiedades y comportamientos de todas las clases a las que pertenecen.

Polimorfismo: Implementar el mismo mensaje en formas diferentes en objetos diferentes.

Recolección de basura: Técnica de destrucción automática de objetos.

Objetos

Los **objetos** son una unidad que encapsula estado y comportamiento. Son instancias de una clase.
Los objetos deben tener:
Abstracción: Representación computacional de entes (con sus características y comportamientos) de la realidad.

- **Esencial:** Hacen lo que el ente sea lo que es.
- **Accidental:** si el ente no las tiene o se cambia no deja de ser lo que es.

Identidad: Existencia única de un mismo objeto.
Comportamiento: Conjunto de mensajes que puede responder (Protocolo) y mensajes que puede entender (Vocabulario)
Inspección: Visualizar en todo momento a sus colaboradores y poder cambiarlos.

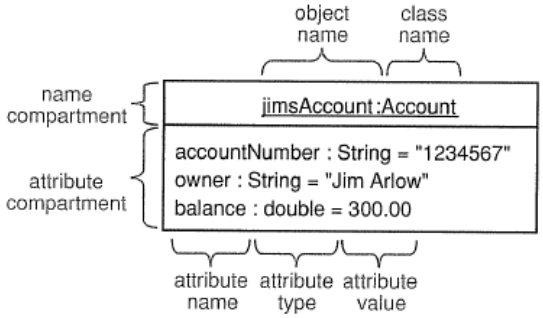
Colaboraciones

Los objetos se mandan mensajes para ejecutar las funciones del sistemas. Esto se llama **colaboración**.

Los **métodos** son el conjunto de colaboraciones que un objeto tendrá con otros. Dictan qué hacer con cada mensaje. Implementan el comportamiento del objeto.
El protocolo o interfaz es público, la definición de métodos y colaboradores es privada.

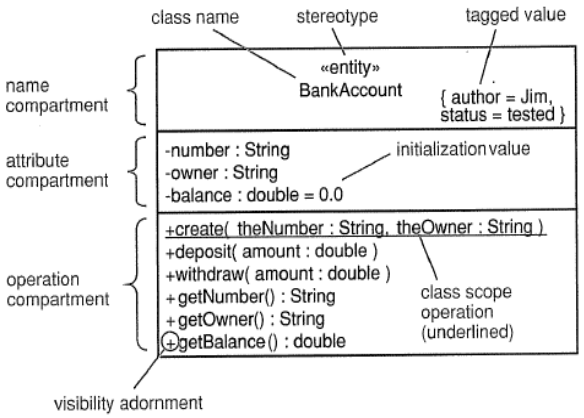
Cuando se envía un mensaje hay un **emisor**, un **receptor** y un **nombre** que identifica el mensaje (puede tener, además, *parámetros*)
Puede ocurrir que un objeto reciba un mensaje que no entienda, esto puede ocurrir por dos razones: (1) se manda al objeto correcto un mensaje equivocado o (2) se manda un mensaje correcto al objeto equivocado.

Hay dos tipos de colaboradores: **habituales** (internos) que son el conjunto de variables que va a tener un objeto y **eventuales** (externos) que son el conjunto de métodos públicos.
En UML;



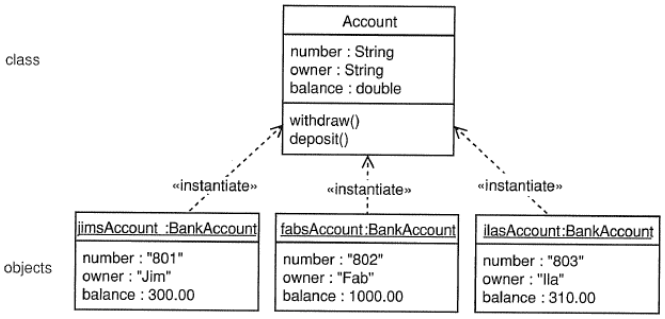
Clases

Las **clases** son plantillas que describen la estructura de los objetos.
En UML;



Instancias

Las relación entre clases y objetos es de <<instantiate>>. La **instanciación** es la creación de nuevas instancias a partir de un elemento modelo.
En UML,

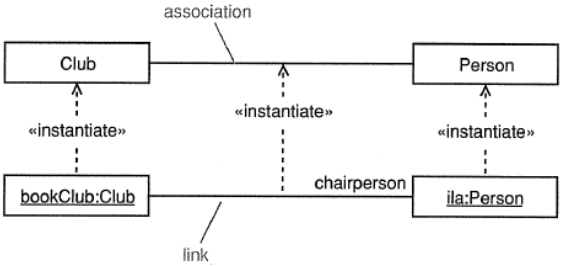


Relaciones entre clases

Una **relación** es una conexión semántica significativa entre elementos modelo.

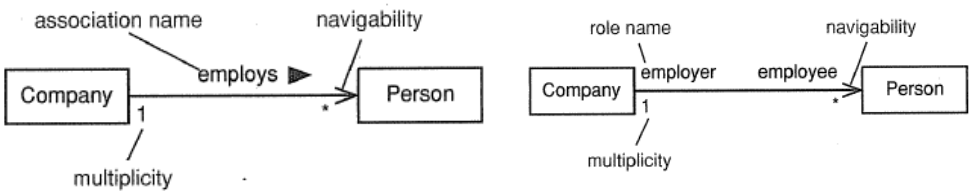
Asociación

Una clase *usa* a otra clase.
Los objetos se mandan mensajes mediante conexiones llamadas **enlaces** (links). Los enlaces son instancias de una asociación.

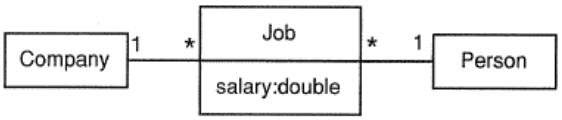


No es una relación fuerte. El tiempo de vida de un objeto no depende del otro.

Las asociaciones tienen: un nombre de asociación o roles, multiplicidad (número de objetos de una clase que puede estar involucrado en una relación) y navegabilidad (hacia donde los mensajes pueden ser mandados).



Clase asociación: Clase intermedia que representa una relación muchos-muchos mediante entradas que tienen relación 1 a muchos. No solo conecta dos clases sino que también define un conjunto de características propias de la relación.



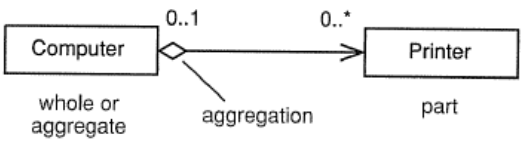
Agregación

Una clase *es parte* de otra clase.

La agregación es un tipo de relación *todo-parte*, donde un objeto (el todo) usa servicios de otro objeto (la parte)

El todo puede a veces existir independientemente de sus partes pero las partes pueden siempre existir independientemente del todo y es posible que varios objetos todos compartan un mismo objeto parte.

La agregación es **transitiva** (Si un objeto C es parte de B y B es parte de A, C es también parte de A) y **asimétrica** (Un objeto jamás puede ser parte de sí mismo)

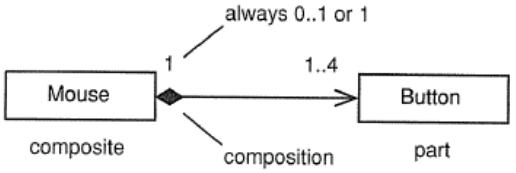


Composición

Más fuerte que agregación.

En este caso las partes no tienen independencia del todo y cada parte le puede pertenecer a máximo un solo todo.

El todo tiene la responsabilidad de manejar el ciclo de vida de las partes y si es destruido, las partes tienen que ser eliminadas (o su responsabilidad pasada a otro objeto)



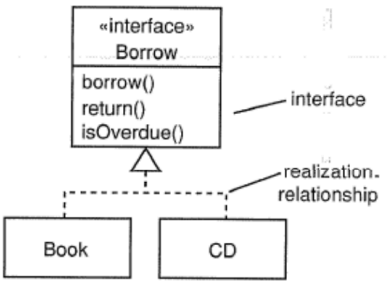
Otros conceptos

Polimorfismo

Invocar métodos distintos con el mismo nombre desde distintos objetos

Interfaces

Las **interfaces** especifican un conjunto de operaciones. El propósito de las interfaces es separar la *especificación* de la funcionalidad de la *implementación*.



En C + + se aplica utilizando clases abstractas.

Principios SOLID

Principio de Responsabilidad Única

Los módulos/clases deben tener una sola responsabilidad y una única razón para cambiar. El SRP está relacionado con la **cohesión**. No aplicarlo puede provocar dependencias innecesarias.

Una clase con muchas responsabilidades puede ser replanteada en varias clases y distribuir sus responsabilidades.

Principio de Abierto/Cerrado

Los módulos deben estar *abiertos* para la extensión (adaptarse a nuevos requerimientos o cambiar el dominio del problema) pero *cerrados* a la modificación (alterar la lógica del código actual). No aplicar el OCP puede provocar que un solo cambio afecte a toda la lógica del programa.

Este principio está relacionado con el uso efectivo del **polimorfismo**.

Principio de Sustitución de Liskov

Las clases hijas deben poder ser sustituidas por la clase base sin alterar el funcionamiento del programa.

LSP está relacionado con el **polimorfismo** y la **herencia**.

Si existen métodos en la clase base que las clases hijas no pueden usar correctamente, se pueden reescribir para que si puedan.

Principio de Segregación de Interfaces

Las interfaces deben ser pequeñas. En Python las interfaces se definen utilizando las clases abstractas (a partir del módulo abc).
El ISP está relacionado con la **cohesión**.
Si una interfaz provee muchos métodos, es mejor dividirla en múltiples interfaces que contengan menos métodos.

Principio de Inversión de Dependencia

Las abstracciones tienen que ser pensadas de tal forma que la implementación concreta no dependa de la abstracción.

Otros conceptos

Ley de Demeter

“Habla solo con tus amigos”
Un objeto solo puede invocar métodos suyos y de los objetos miembros de su clase.

Tell don’t ask

En lugar de preguntarle a un objeto su estado y tomar decisiones basadas en su respuesta, se debe simplemente "decirle" al objeto qué hacer y permitirle que gestione internamente su estado.

Único punto de salida

Un programa debe tener un punto de entrada principal y bien definido donde inicia la ejecución e, idealmente, solo un punto de salida donde la ejecución del programa termina.

Patrones de diseño

Base para la búsqueda de soluciones a problemas comunes en el desarrollo de software.
Su uso no es obligatorio pero sí aconsejable. Forzar el uso de los patrones puede ser un error.

Ventajas:

- **Evitar** la reiteración en la búsqueda de soluciones a problemas conocidos y solucionados.
- **Formalizar** un vocabulario común entre diseñadores
- **Estandarizar** el modo que se realiza el diseño.
- **Facilitar** el aprendizaje condensando conocimiento ya existente.

Componentes

- **Nombre:** Describe en pocas palabras un problema.
- **Problema:** Indica cuándo aplicar el patrón (Requerimientos a cumplir, restricciones a considerar, propiedades deseables de la solución)
- **Solución:** Brinda una descripción abstracta del problema y cómo resolverlo.
- **Consecuencias:** Resultados en términos de ventajas e inconvenientes.

Otros tipos

Según el nivel de abstracción se distinguen tres patrones de diseño.

- **Patrones arquitectónicos:** Definen una estructura fundamental sobre la organización del sistema.
- **Patrones de diseño:** Describen una estructura recurrente y común de componentes comunicantes.
- **Patrones de codificación:** Implementan aspectos particulares del diseño en un lenguaje de programación específico.

Cátalogos

Los patrones definen subsistemas o componentes de software. Abstraen el proceso de instanciación, así el sistema es independiente de cómo se crean, componen y representan los objetos.

Patrones Creacionales

Estos patrones se encargan de manejar la instanciación de objetos.
Características:

- **Instanciación genérica:** Permite que los objetos sean creados sin especificar clases concretas.
- **Simplicidad:** Facilitan la creación de objetos evitando código complejo.
- **Restricciones creacionales**: Ayudan a establecer restricciones sobre la creación.

Patrón Singleton

Asegura que una determinada clase sea instanciada *una sóla vez*, proporcionando un único punto de acceso global a ella.

Problemas:

(1) Garantizar única instancia: Pensado para controlar el acceso de algún recurso compartido (Imposible con un constructor normal)

(2) Único punto de acceso: Pensado para evitar que otro código reescriba el estado de la instancia.

Solución:

(1) Hacer privado al constructor: Evita el uso del operador new.

(2) Crear un método de creación estático: Invoca al constructor privado para crear un objeto y lo guarda en un campo estático.

Ventajas	Desventajas
• Certeza de única instancia y punto de acceso • Inicialización cuando se requiera	• Vulnera el Principio de Responsabilidad Única. • Puede enmascarar mal diseño • Dificil de simular

Ejemplo en Python

```python
class SingletonClass(object):
  def __new__(cls):
    if not hasattr(cls, 'instance'):
      cls.instance = super(SingletonClass, cls).__new__(cls)
    return cls.instance

singleton = SingletonClass()
new_singleton = SingletonClass()

print(singleton is new_singleton) // “true”
```

Crea una instancia solo si no existe ya una instancia. De lo contrario, devuelve la instancia ya creada.

Patrones Estructurales

Estos patrones explican cómo combinar objetos y clases en estructuras más grandes.

Composite

Compone objetos en estructuras de árboles para representar jerarquías parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los complejos.

Problema:

(1) El modelo central puede ser pensado como un árbol.

Solución:

(1) Trabajar desde una interfaz común.

Estructura:

(1) Interfaz componente: Describe operaciones comunes entre elementos simples y complejos del árbol.

(2) Hoja: Elemento sin suplementos. Terminan haciendo la mayoría del trabajo, ya que no se lo delegan a nadie más.

(3) Contenedor: Elemento con subelementos. Delega su trabajo a los subelementos, procesa los resultados intermedios y devuelve el resultado final.

Ventajas	Desventajas
• Certeza de única instancia y punto de acceso • Inicialización cuando se requiera	• Vulnera el Principio de Responsabilidad Única. • Puede enmascarar mal diseño • Dificil de simular

Patrones de Comportamiento

Observer