POO

Profesor: Alejandro Francisco Peña (al.Pena@bue.edu.ar y alpena@uade.edu.ar)

Ayudante: Adrián Alberto Narducci (adnarducci@uade.edu.ar)

1er parcial : Teórico/Práctico, ***no va a haber código JAVA***, solo diagrama UML. Dado un enunciado explicar conceptos

2do parcial: trabajo modelado

TPO: después tenemos una consigna un poco mas grande donde vamos a tener que modelar y codear el modelado (grupos entre 2 y 3 personas) . Va a ser con interfaz con JAVA Swing

final oral

¿Qué es un paradigma? → Forma de hacer las cosas (no se cuestiona)

▼ TPO N°3 Consultorio

Clases

- [ ] Persona

  #nombre

  #apellido

  #dni

  #edad

  —

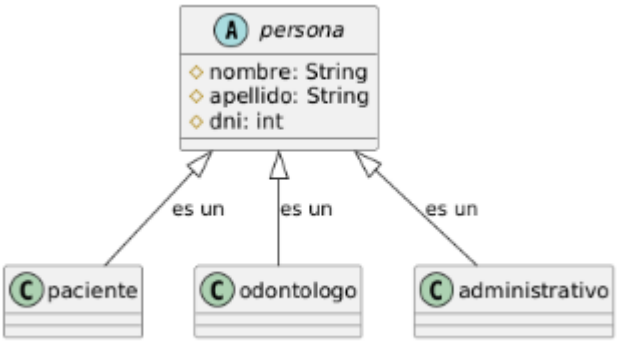
- [ ] Paciente
- [ ] Odontologo

- [ ] Administrativo

- [ ] Consultorio
- [ ] Historial Clinico

Relaciones

Pacientes/Odontologos/Administrativos son **Personas**



▼ Ejercicios

1. Crea una función que tome dos argumentos: el precio original y el porcentaje de descuento como números enteros y devuelva el precio final después del descuento.

```
public int Ejercicio1(int precio , int descuento){
   int    precioFinal = 0;
   precioFinal =
   return precioFinal;
}
```

2. Escriba una función que devuelva verdadero si existe al menos un número que sea mayor o igual a n

▼ Clase 1 - 8/8 - Introducción

La POO → ***"Permite modelar conceptos del mundo real de manera más intuitiva y estructurada"***

¿Qué es POO?

Paradigma de programación que organiza el código en objetos, los cuales encapsulan datos (**atributos** , son adjetivos que describen al objeto en cuestión) y comportamientos (**métodos** , son verbos/acciones). Estos objetos interactúan entre sí para formar más sistemas complejos.

"Los objetos son entes tangibles"

Ejemplo objeto pizarrón

atributos:

- tipo: marcador
- color: blanco
- terminaciones: aluminio

métodos (objeto profesor)

- dicta clase

"Antes de construir necesitamos un plano o croquis que define cómo será (un lenguaje universal que define cómo va a ser)"

Ventaja:

- Disipa las barreras entre el qué y el cómo
- Facilita la reutilización del código y por lo tanto el mantenimiento del mismo

Desventajas:

- Mayor complejidad a la hora de entender el flujo de datos
  - Pérdida de linealidad
- Requiere un lenguaje de modelización de problemas más elaborados:
  - UML
  - Representaciones gráficas más complicadas

¿Qué es una Clase?

Las clases vienen a ser el plano , una plantilla que dice la forma en la que se va a construir.

Se define propiedades (**atributos**) y acciones (**métodos**) que tendrá el objeto

Ejemplo de clase con dos vasos térmicos: ***"Son la misma Clase pero diferentes objetos (tienen diferentes propiedades) . Esto por más que sean idénticos"***

¿Qué es un Objeto?

Lo construido a partir del plano (Clase) es el **objeto**. Cada objeto que se construye a partir de ese mismo plano puede ser distinto (distintos atributos)

"Un objeto es una instancia de una clase" → Ejemplo : *MI CASA* es un **objeto** de la **clase** *CASA* , cada casa tiene características propias (color, tamaño, cantidad de habitaciones)

El dominio es el universo

¿Qué es un Atributo?

Los **Atributos** son las variables dentro de una clase que almacenan la información de cada objeto

¿Qué es un Método?

Convenciones de escritura de objetos:

- Clases → Empiezo con letra **mayúscula y sigo en minúscula** (CamelCase)
- Atributos → Se escriben en **minúscula**. Se listan con " - "
- Métodos → Se listan con " : "

▼ Ejercicio armado de objetos cotidianos

Botella de agua

- Clase: Botella

- Atributos

  -material

  -capacidad

  -color

- Métodos

  : llenar

  : abrir

  : cerrar

Reloj

- Clase: Reloj
  - Atributos

    -tipoDePila

    -marca

    -tipoDeReloj

    -tipografia

  - Métodos

    : darLaHora

    : cronometrar

    : sonarAlarma

Tacho De Basura

- Clase: TachoDeBasura
  - Atributos

    -tipoDeResiduo

    -peso

    -volumen

    -estaLleno

  - Métodos

: almacenarResiduos

: compactarBasura

: compostar

▼ Ejercicio objeto personaje de WoW (Con parámetros en los métodos)

> “Corrección en realidad los métodos si los parámetros son del mismo objeto no hay que pasárselo, hay que pasarle los parámetros externos al objeto”

Personaje

-fuerza

-velocidadDeMovimiento

-mana

-inteligencia

-puntosDeVida

-velocidadDeAtaque

-armadura

-ubicacion

: atacar (

Personaje

)

: atacar (

Bestia

)

: moverPersonaje (

direccion

)

: recibirDaño (

fuerzaDelOtroPersonaje,

velocidadDeAtaqueDelOtroPersonaje

)

Clase 2 - 15/8 -

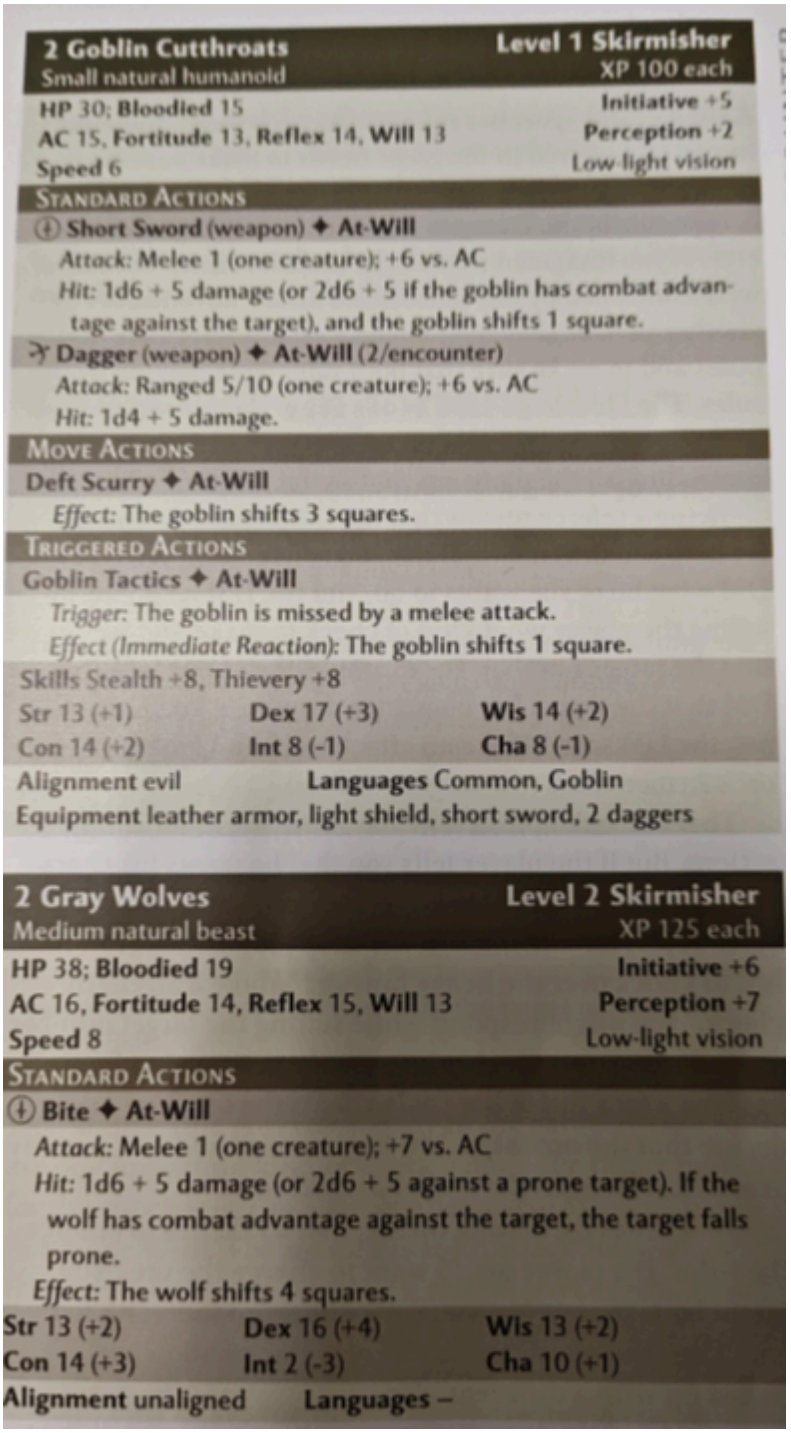
Clase 3 - 22/8 - Herencia

Ejercicio D&D

Sacamos un dominio - modelo del libro de maestro de D&D

Con esto vamos a crear clases , ya que es un documento que vamos a tener que digitalizar

Acá tenemos ejemplos para modelar clases:



2 Goblin Cutthroats — **Level 1 Skirmisher**
Small natural humanoid — XP 100 each

HP 30; **Bloodied** 15 — **Initiative** +5
AC 15, **Fortitude** 13, **Reflex** 14, **Will** 13 — **Perception** +2
Speed 6 — Low-light vision

STANDARD ACTIONS

⊕ **Short Sword** (weapon) ✦ **At-Will**
Attack: Melee 1 (one creature); +6 vs. AC
Hit: 1d6 + 5 damage (or 2d6 + 5 if the goblin has combat advantage against the target), and the goblin shifts 1 square.

Dagger (weapon) ✦ **At-Will** (2/encounter)
Attack: Ranged 5/10 (one creature); +6 vs. AC
Hit: 1d4 + 5 damage.

MOVE ACTIONS

Deft Scurry ✦ **At-Will**
Effect: The goblin shifts 3 squares.

TRIGGERED ACTIONS

Goblin Tactics ✦ **At-Will**
Trigger: The goblin is missed by a melee attack.
Effect (Immediate Reaction): The goblin shifts 1 square.

Skills Stealth +8, Thievery +8

Str 13 (+1)	Dex 17 (+3)	Wis 14 (+2)
Con 14 (+2)	Int 8 (-1)	Cha 8 (-1)

Alignment evil — **Languages** Common, Goblin
Equipment leather armor, light shield, short sword, 2 daggers

2 Gray Wolves — **Level 2 Skirmisher**
Medium natural beast — XP 125 each

HP 38; **Bloodied** 19 — **Initiative** +6
AC 16, **Fortitude** 14, **Reflex** 15, **Will** 13 — **Perception** +7
Speed 8 — Low-light vision

STANDARD ACTIONS

⊕ **Bite** ✦ **At-Will**
Attack: Melee 1 (one creature); +7 vs. AC
Hit: 1d6 + 5 damage (or 2d6 + 5 against a prone target). If the wolf has combat advantage against the target, the target falls prone.
Effect: The wolf shifts 4 squares.

Str 13 (+2)	**Dex** 16 (+4)	**Wis** 13 (+2)
Con 14 (+3)	**Int** 2 (-3)	**Cha** 10 (+1)

Alignment unaligned — **Languages** –

Clase: Monster

Atributos (Recordemos que deben ser minusculas , y camel case):

- hp - Este atributo lo hacemos público y un integer
- ac - Integer
- position - (int, int)

- speed - Integer
- reflex - Integer
- will - Integer
- fortitude - Integer
- lowLightVision - Boolean
- xpGiven - Int
- level - Int
- name - String
- initiative - Int
- perception - Int
- description - string
- weapons - weapon (Este caso sería un array capaz, y weapon es un **OBJETO**)

Métodos:

attack(

weapon;

creature;

diceResultAC;

diceResultHit;

)

move(

positionToReach;

)

Clase : Goblin

Atributos:

- Hereda del tipo mosnter
- wearsSkirt : Boolean (Atributo que le agregamos a la clase goblin)

Metodos:

- Hereda del tipo monster

Clase: Cutthroat

Atributos:

- Hereda de Goblin

Metodo

deftScurry()

En Herencias podemos sobrescribir métodos. Si lo Sobrescribimos esto es Polimorfismo (métodos dentro de la clase padre que puedo heredar y modificar)

Polimorfismo puede hacer los overrides en las clases hijos (Tener mismo nombre pero que haga otras cosas)

Override ≠ Polimorfismo (Pero son parecidos)

Paradigma

Un paradigma de programación indica método para realizar cómputo y manera en que se debe estructurar y organizar las tareas que llevara acabo un programa

Se asocian a un estilo determinado de programación

Hay varios paradigmas , uno de ellos es el orientado a objetos

Herencia

Es una relación básica que tiene la POO

Expresa tanto la especialización como la generalización

Permite evitar REPETIR las características comunes a varias clases . Una de las clases comparte la estructura y/o comportamiento de otras clases

También se denomina relación: ***“Is a” / “Es un”*** → Ejemplo: Un Cuthroat ***ES UN*** goblin ***ES UN*** mounstro

Relación de Herencia

Es el mecanismo que permite a un objeto heredar propiedades de otra clase de objetos.

Permite a un objeto contener sus propios procedimientos o funciones y heredar los mismo de otros objetos.

Es un mecanismo potente que no se encuentra en sistemas procedimentales

Atributos (Se pueden compartir en varias clases) ≠ Atributos Propios (Son únicos para una clase)

Vocabulario De herencia

Clase base o superclase → clase de la cual se hereda

Clase derivada o subclase → clase que hereda

Herencia simple→ Hereda de una sola clase

Herencia múltiple → JAVA no acepta múltiple herencia

Clase Abstracta → La que no lleva, ni puede llevar, ningún objeto asociado

Polimorfismo → Posibilidad de usar indistintamente todos los objetos de una clase derivada

Diagramas UML se leen de abajo a arriba o derecha a izquierda

Código JAVA Herencias

Vamos a hacer la clase "**animal**" , van a haber 2 subclases "**perro**" y "**gato**" con atributos y métodos

1. Hacemos un .java que va a ser la entrada de la aplicación , en este caso la aplicación se llamará Herencia1"

2. Creamos otra clase animal separada. Y la inicializamos como abstract

```
public abstract class animal {

}
```

3. Definimos atributos para la clase animal

```
public abstract class animal {
    private String name;
    private int age;
}
```

4. Ahora constructores

▼ Clase 4 - 29/8 - Encapsulamientos, Herencia, Abstracción, Polimorfismo e Interfaces

JAVA - puntos clave

nombre clase = nombre archivo

main → es el punto de entrada del programa y siempre tiene la misma firma: `public static void main(String[] args)`

Encapsulamiento

Restringe acceso directo a los datos de una clase . Se accede a ellos a través de métodos públicos (getters y setters)

Los modificadores dan seguridad de alcance

Modificadores de acceso:

- public → accesible desde todos lados, incluyendo otros proyectos/clases. Incluso el usuario puede acceder
- protected → Accedido dentro del paquete y por subclases. Es de la parte de herencia. Son declarados en una superclase y solo puedan ser accedidos por las subclases

- default → Es casi un `public` pero no funciona con clases que no estén en ese mismo paquete
- private → accesible dentro de la clase , pero no fuera de ella. Hay que utilizarlo por no exponer métodos . Vamos a trabajar más con esto

Herencia y Jerarquía

Sintaxis → Se usa palabra clave `extends` . ***Una clase solo puede heredar de una superclase, no permite herencia múltiple.***

```
//Terrestre hereda de transporte
Terrestre extends Transporte{}
```

Jerarquía de clases → Java tiene una jerarquía de clases integrada, donde `Object` es la superclase del resto de clases

`@Override` → Anotación opcional pero recomendada para indicar que un método sobrescribe uno de la superclase . Indica al compilador que se sobreescribe el método

```
class Animal{
    public void hacerSonido()
}

class Perro extends Animal{
    @Override
    public void hacerSOnido(){}
}
```

`super` → palabra clave para acceder a los miembros de una superclase incluido su constructor.

```
@Override
memoria(){
    super
}
```

Abstracción

La caja negra (Ocultan la complejidad y mostrar solo la funcionalidad). Se logra con clases e interfaces abstractas

clases abstractas → se definen con la palabra clave `abstract`

- No se pueden instanciar , solo las clases finales “las worker”
- Pueden contener métodos abstractos (sin cuerpo) y métodos correctos
- La manera correcta sería que la **superclase** sea una `abtract class`

Interfaces

- se definen con `interface`
- Varias clases no relacionadas
- No tiene implementación de métodos
- se definen con contrato: una lista de métodos que una clase de be implementar
- Una clase puede implementar múltiples interfaces
- Ejemplo: Una interfaz “Dibujable” con el método dibujar(), que puede ser implementada por circulo, cuadrado, etc

```
interface Volador{void volar();}
```

Reduce la necesidad de IF else

Polimorfismo

Muchas formas para un mismo método (Un objeto puede tomar muchas formas)

Un objeto puede tomar muchas formas. Permite tratar a objetos de diferentes subclases como objetos de su superclase

Ejemplo un control remoto universal. Un solo botón (El método) puede encender TV, DVD, aire Acondicionado (Los objetos) y cada uno responde de forma diferente

Tipos de polimorfismo:

- polimorfismo por subtipo o inclusión:
  - Cuando una subclase es utilizada en lugar de su superclase
  - se basa en la herencia y es el tipo más común
- Polimorfismo paramétrico o Genérico:
  -
- sobrecarga de métodos o Overloading :
  - `"Otro metodo con el mismo nombre pero con distinta firma = sobrecarga`"
  - múltiples métodos con el mismo nombre pero diferentes parámetros (numeros, tipos, o ambos)
  - Compilador elige el método
  - No se considera polimorfismo en el sentido estricto pero es asociado a la POO

```
Animal miAnimal = new Perro(); //Polimorfismo
miAnimal.hacerSonido(); //LLama al método de Perro
```

Relaciones

Existen 5 tipos de relaciones

- cardinalidad
- asociación → bidireccional
- herencia → relación “Es un/a”
- agregación
- instanciación

▼ Práctica/Ejercitación Batalla Pokémon

1. Pokémon va a ser una clase abstracta ya que no la vamos a instanciar (NO ES UN OBJETO)

Clase Pokémon

Atributos:

- public nombre
- public HP
- public tipo
- public nivel
- public vidaMaxima

Métodos:

- public CrearPokemon()
- atacar(Pokémon)

Clase Pikachu (NO ES UN OBJETO)

Atributos:

-

Metodos:

CrearPokermon(Electrico)

CargarHP()

atacar(Pokémon)

Clase Charmander

Atributos:

-

Metodos:

CrearPokemon(Fuego)

CargarHP()

atacar(Pokémon)

---

2. Entrenador

Clase Entrenador

Atributos:

Métodos:

yoTeElijo(Pokémon)

▼ Clase 5 - 5/9 - UML y Métodos Wrapper

Test Unitarios → test que caen en un apartado de una función

LAS CLASES ABSTRACTAS NO SE INICIAN , SOLAMENTE SE HEREDAN

enum → es una lista de enumeración de estados

```
enum tipo{
    FUEGO, AGUA, PLANTA
}
```

interfaz → No pueden ser instanciadas

UML

UML → Unified Modeling Language

modelar sistemas como flujos de trabajo . El diagrama ofrece una vista del sistema a modelar

¿Qué es?

El UML es un lenguaje basado en diagramas para la:

- Especificación
- Visualización
- Construcción
- Documentación

De cualquier sistema complejo que nos permite visualizarlo

Nos sirve para :

- Modelar
- Mostrar el comportamiento de un sistema
- Facilitar la creación de código
- Crear las especificaciones de un sistema
- Describir la arquitectura de un sistema

Características

NO ES UNA METODOLOGÍAS sino que Es un lenguaje visual que puede utilizar diferentes metodologías , se utiliza para modelar la realidad

Nos permite modelar:

- Procesos
- Sistemas
- Software

Es escalable , flexible y extensible

Modelo

Patrón sobre lo que se producirá algo

Es una forma visual que describe las reglas de un negocio

- Modelar ayuda a comunicar diseño de manera más efectiva
- Clarifica los problemas complejos
- Nos permite ahorrar tiempo al comprender el negocio y los procesos
- Ayuda a que los diseños sean cercanos a la realidad
- Permite trabajar de manera más efectiva y eficiente
- Comprendemos el dominio del negocio

Diagrama

Forma visual de representar diferentes elementos de modelado descriptos en UML

Cada diagrama posee un propósito especifico

Posee símbolos propios y especiales para lograr su cometido

Representa un proceso o sistema en su totalidad

Notación

Son elementos que trabajan en conjunto dentro del diagrama

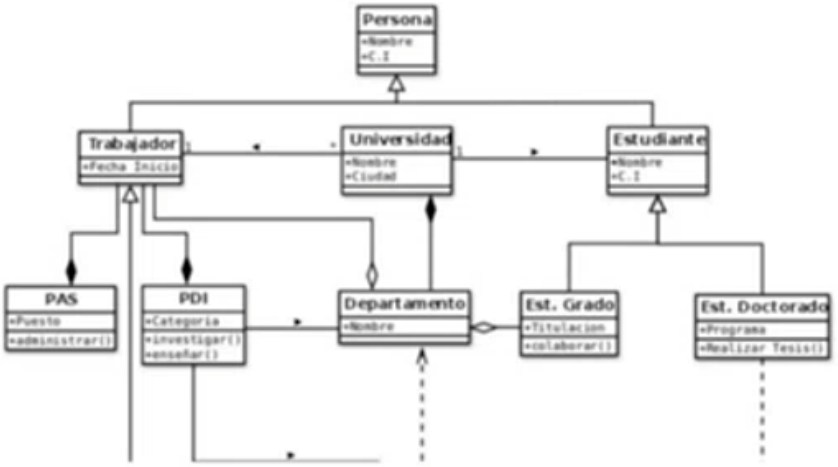
Poseen conectores y símbolos que le son propios

Tipos de diagramas

De estructura

Muestran la parte estática de los elementos del sistema

- De clases

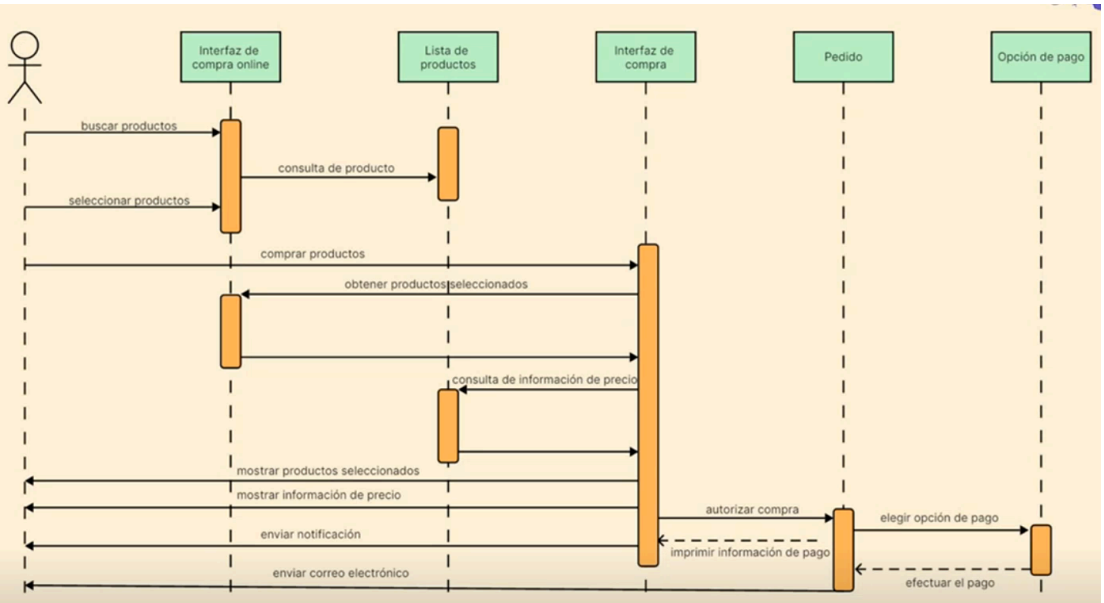


- De objeto
- De implementación
- De paquetes

De comportamiento

Proveen de manera visual el comportamiento del sistema

- De casos de uso
- De estado
- De colaboración
- De Secuencia (importante , son muy usados)



Orientación de objetos bajo UML

Clase

Agrupamiento, categoría dentro del cual podemos agrupar objetos con características similares. Poseen:

- Atributos - propiedades, “el exterior”
- Operaciones - Funcionalidades, métodos

Objeto

Instancia particular de una clase . Será quien realice la operación en si misma:

Existe dentro del contexto del sistema y posee una entidad en particular

Herencia

Permite extender características de una clase padre. Los hijos adquieren características de la clase principal o padre

Asociación

Dos clases relacionadas o conectadas de alguna forma

Relaciones de asociación:

- Linked: La información de una clase esta ligada a los datos de otra clase
- Colaboración: Dos clases trabajan en conjunto para llevar Adelante un objetivo
- Acts: Una clase actúa sobre la otra. “Comer quita Hambre”

Agregación y Composición . Todo-Parte

Agregación → Un objeto tiene otros objetos, pero pueden existir de forma independiente

> Ejemplo: Una oficina posee empleados , pero los empleados existe aunque la oficina cierre (débil)

Composición → Un objeto posee otros objetos y estos no pueden existir sin el. Hay dependencia (Fuerte)

> Ejemplo: Un libro compuesto por páginas

Diferencias:

Característica	Agregación	Composición
Tipo de relación	“Tiene un / una”	“Es dueño de”
Existe por fuera	Si	No
Tipo	Debil	Fuerte
Rep. UML	◇	◆
Ejemplo	Universidad-Estudiante	Libro-Página

Carinalidad

Indica la cantidad de objetos que participan en una relación:

- 1 a 1
- 1 a varios
- Varios a Varios

Una instancia de clase ‘k’ puede estar relacionada con 0 o + instancias de clase ‘g’

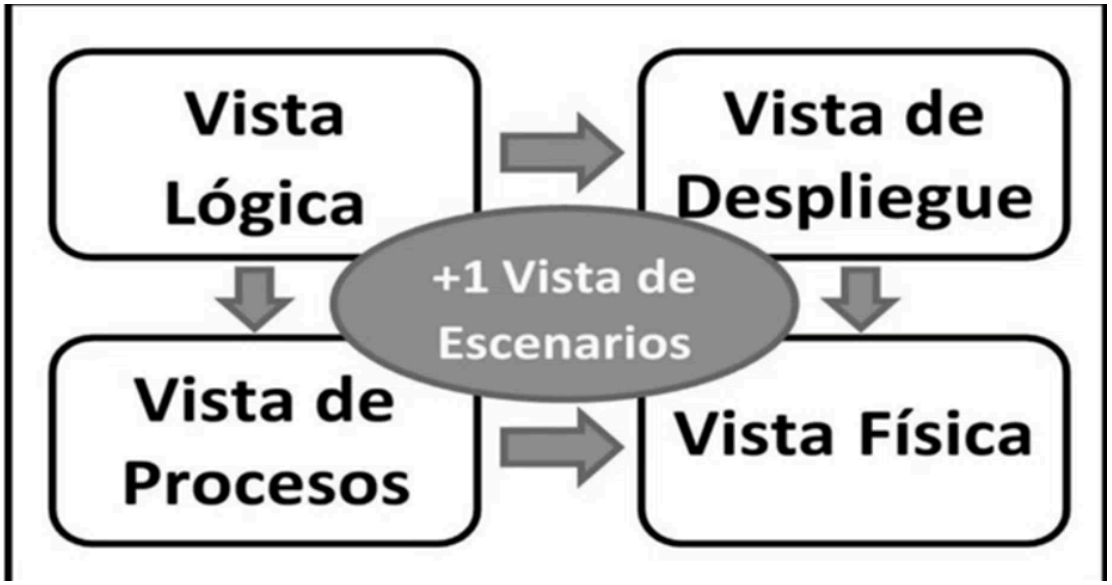
Polimorfismo

Capacidad de usar el mismo nombre para realizar diferentes acciones

Las operaciones se llevan a cabo de diferentes maneras dependiendo quien las implemente

Animal → Habla → Gato: MAULLA - Perro: LADRA

▼ Modelo 4 + 1



Vista lógica

Énfasis en las clases y objetos

Representa abstracciones

Diagramas:

- Clases
- Estado
- Objetos
- Secuencia
- Comunicación

Veremos las partes de las clases y objetos , que componentes tienen y cómo interaccionan.

Se ven los puntos más importantes de un sistema

Vista de procesos

Muestra comunicación entre procesos

Útil cuando se trabaja con hilos o threads

Diagramas:

- Actividad

Vista física

Modela ambiente de ejecución

Sobre qué HW se estará ejecutando nuestro sistema

Mapea el SW con el HW que lo ejecutará

Se usa el diagrama de implementación

Vista de desarrollo

Módulos o componentes que tiene el sistema

Muestra que elementos constituyen el sistema en sí

Diagramas:

- Componente
- Paquetes

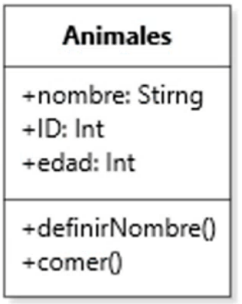
Casos de Uso

Permite modelar sistemas

Facilita la comunicación, actualizaciones y modificaciones

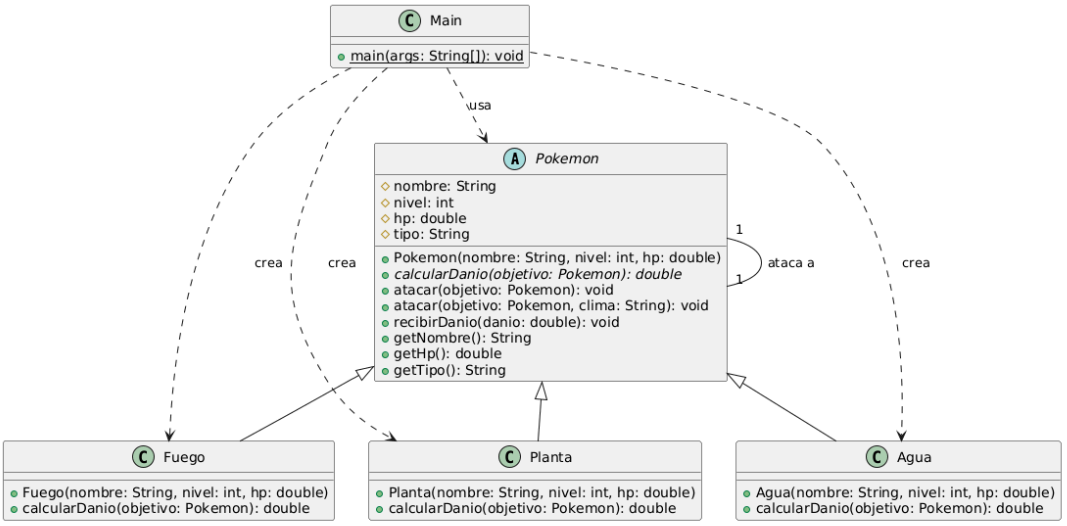
Diagrama de Clases UML

El “+” quiere decir que es `public`



Usamos PlantUML para crear los diagramas UML

▼ Diagrama UML de ejercicio Pokémon (Versión profesor)




```
@startuml
abstract class Pokemon {
# nombre: String
# nivel: int
# hp: double
# tipo: String

    + Pokemon(nombre: String, nivel: int, hp: double)
    + {abstract} calcularDanio(objetivo: Pokemon): double
    + atacar(objetivo: Pokemon): void
    + atacar(objetivo: Pokemon, clima: String): void
    + recibirDanio(danio: double): void
    + getNombre(): String
    + getHp(): double
```

```
    + getTipo(): String
}

class Fuego {
+ Fuego(nombre: String, nivel: int, hp: double)
+ calcularDanio(objetivo: Pokemon): double
}

class Agua {
+ Agua(nombre: String, nivel: int, hp: double)
+ calcularDanio(objetivo: Pokemon): double
}

class Planta {
+ Planta(nombre: String, nivel: int, hp: double)
+ calcularDanio(objetivo: Pokemon): double
}

class Main {
+ {static} main(args: String[]): void
}

Pokemon <|-- Fuego
Pokemon <|-- Agua
Pokemon <|-- Planta

Main ..> Pokemon : usa
Main ..> Fuego : crea
Main ..> Agua : crea
Main ..> Planta : crea

Pokemon "1" -- "1" Pokemon : ataca a

@enduml
```

Métodos Wrapper

Métodos que permiten trabajar con tipos de datos primitivos (int, double, boolean, etc.) como si fueran objetos

Estos métodos se encuentran en las clases wrapper de Java como Integer , Double , Character , entre otras

El objetivo es proveer una manera de manipular tipos primitivos como objetos y ofrecer funcionalidades adicionales , como la conversión entre tipos y operaciones matemáticas proporcionando métodos

Clases Wrapper:

- Integer → para int
- Double → para double
- Boolean → para boolean
- Character → para char

parseInt() , parseDouble(): Convierte un String a un tipo primitivo

> Ejemplo: Integer.parseInt(”123”) → Convierte el string “123” a un valor entero

valueOf(): Convierte un tipo primitivo a su clase wrapper

> Ejemplo: Integer.valueOf(123) → Convierte el entero 123 a un objeto Integer

toString(): Convierte un objeto wrapper a su representación en cadena

> Ejemplo: Integer.toString(123) → Convierte el entero 123 a la cadena “123”

Relaciones entre Clases

Permiten representar cómo se vinculan las entidades dentro de un sistema. Se pueden clasificar en 3 grupos: Generalización , Asociación y Dependencia

Relaciones de Generalización

Relación jerárquica entre tipos. Una subclase hereda de una superclase. También se aplica cuando una clase implementa una interfaz. Estas

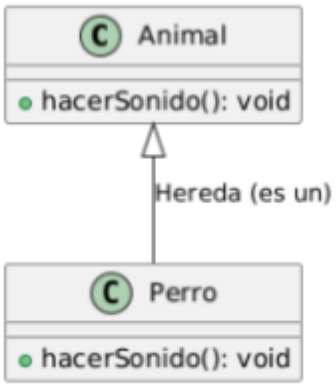
relaciones:

- No presentan cardinalidad especifica
- Se aplica entre tipos y no objetos

Relación de Herencia o Extensión

Es la relación más común de todas

Una subclase hereda atributos y comportamientos de una superclase. Se encuentra representada por la relación de tipo “es un”.



Relación de Implementación

Cuando una clase se compromete a implementar una interfaz, la interfaz define el qué debe implementar la clase pero no especifica el cómo. Esto deja abierta la posibilidad de aplicar polimorfismo sobre los métodos definidos.

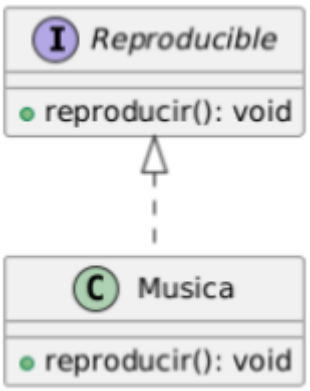
Las interfaces en la POO establecen un contrato: define las operaciones disponibles (el qué), peor no su comportamiento (el cómo). Por lo tanto:

1. La clase concreta define el cómo implementar esos métodos , de la forma que mejor se adapte a su lógica.
2. El código cliente puede interactuar con la interfaz sin conocer, ni depender directamente, de la clase concreta que la implementa

Así una interfaz resulta más flexible que una clase abstracta o una superclase

1. Soportas múltiples implementaciones . Una clase puede implementar varias interfaces
2. No impone estructura interna , como si lo hace la superclase

3. La clase solo implementa métodos públicos sin implementación
4. Es útil cuando es necesaria flexibilidad y bajo acoplamiento



Relaciones de Asociación

indica que una clase conoce o interactúa con otra

Las clases se refieren entre sí como parte de su definición

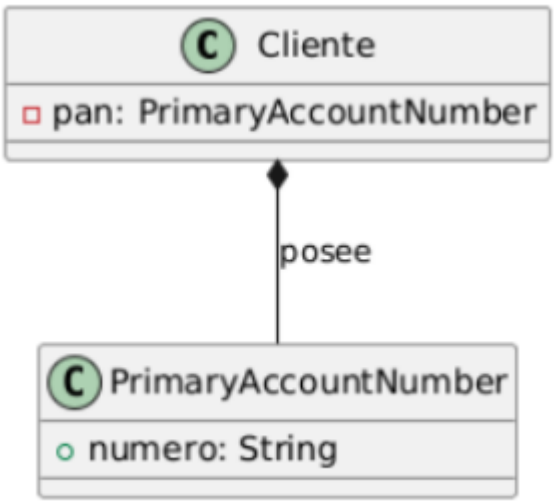
1. Puede existir cardinalidad
2. Puede ser unidireccional o bidireccional
3. La clase solo implementa métodos públicos sin implementación
4. Es útil cuando es necesaria flexibilidad y bajo acoplamiento

Relación de Composición

Cuando una clase contiene a otra forma esencial e inseparable. Una no tiene razón de ser sin la otra.

Ejemplo:

- Una página no existe sin su libro
- Un sueldo no puede existir sin empleado (Siempre debe ser abonado a un empleado por lo que siempre estará asociado a un empleado)
- Una tarjeta de crédito/debito su número (PAN - Primary Account Number) siempre debe estar asociado a un titular. Si por alguna razón este da de baja su relación con la entidad emisora, el PAN se daría de baja de forma automática junto al cliente. El ciclo de vida del PAN depende totalmente del ciclo de vida del todo



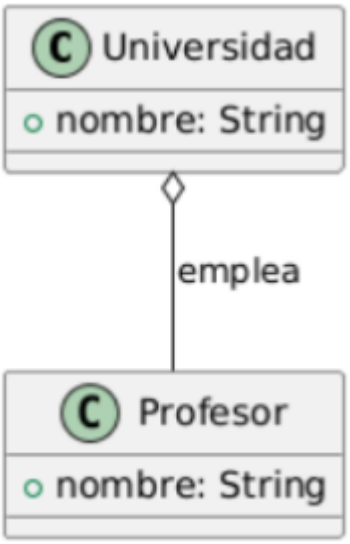
Relación de Agregación (relación de tipo debil)

Una clase contiene y agrupa a otras sin embargo estas pueden existir de forma independiente.

Ejemplo: Universidad y profesores. La universidad agrega profesores para su funcionamiento pero no los crea ni destruye. El ciclo de vida de cada profesor es independiente al de la universidad. La universidad “emplea” un profesor

El todo utiliza las partes y estas pueden existir de forma independiente

1. un estudiante existe por fuera de la Universidad (el sueldo y el PAN no)
2. Puede estar relacionado con varias universidades a la vez o carreras
3. Si la universidad o curso desaparece, el estudiante o profesor continua con su vida



Relación de Asociación Simple

Representa que una clase conoce a otra y puede interactuar con ella, no implica propiedad, composición ni agrupación lógica. Solo indica que hay un vinculo estructural o de colaboración entre ambas clases

1. Pueden ser bidireccionales y tener cardinalidad
2. No existe dependencia existencial . Una clase puede existir sin la otra

Por ejemplo: Alumno se inscribe en un curso, tanto el alumno como el curso pueden seguir existiendo y ni el alumno es dueño del curso ni viceversa. Sin embargo se conocen.

Se utiliza cuando una clase necesita conocer a otra pero sin implicar pertenencia , exclusividad ni dependencia del ciclo de vida

Relaciones de Dependencia

Se trata de un vínculo temporal y débil . No existe una relación permanente entre clases, representa que una clase requiere de otra para funcionar

1. No son parte del objeto
2. Existe acoplamiento funcional, la clase requiere a la otra para realizar su función
3. No llevan cardinalidad
4. Tienen variaciones de fuerza

Usos más comunes:

1. Llamadas a servicios externos
2. Inyección de dependencias
3. Librerías o componentes temporales
4. Configuraciones opcionales
5. Plugins

Es un vínculo de acoplamiento funcional, no estructural. No implica propiedad ni composición

Dependencia Fuerte

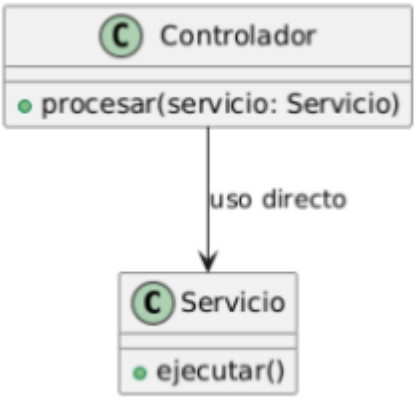
Cuando una clase usa otra de forma directa , concreta y obligatoria sin margen de flexibilidad o reemplazo

Se manifiesta cuando:

1. La clase se pasa como parámetro obligatorio
2. Se invoca sin validación previa
3. El sistema falla si esta no existe

Esto implica alto acoplamiento y dependencia funcional fuerte. La clase queda vinculada de forma permanente, entonces los cambios en la clase proveedora afectan a la consumidora

Se debe evitar siempre que se pueda este tipo de dependencias

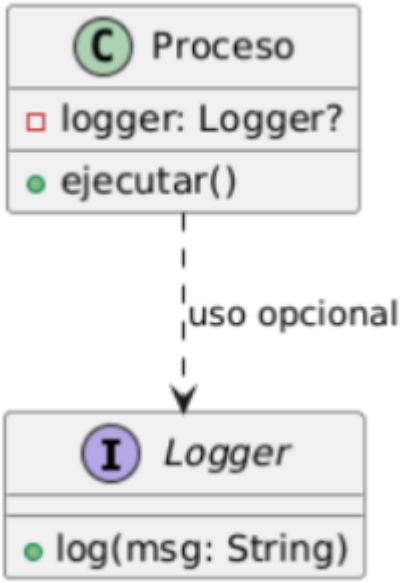


Dependencia Débil

Una clase puede usar otra, pero no la requiere para funcionar correctamente

Suele aparecer cuando existe verificación previa al uso. Se usan interfaces en lugar de clases , se aplican mecanismos de inyección de dependencias

La clase se ocupa de su lógica sin recurris a otra para realizar su función. Favorece la flexibilidad, extensibilidad y sostenibilidad del sistema.

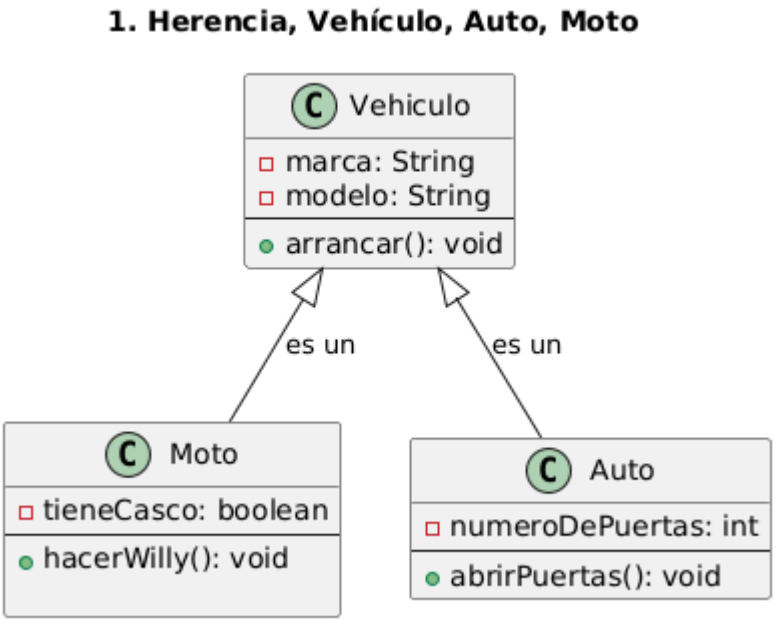


▼ Ejercicios modelado

Ejercicio 1-Relación Herencia

Ejercicio 1 Trabajo practico N°6
Modelar una relación de herencia entre Vehículo, Auto y Moto.
Dibujar el diagrama UML correspondiente, indicando la relación de g
eneralización entre las clases.

```
@startuml

title 1. Herencia, Vehículo, Auto, Moto

class Vehiculo{
   -marca: String
   -modelo: String
   --
   + arrancar(): void
}

class Moto{
   -tieneCasco: boolean
   --
   +hacerWilly(): void

}


class Auto{
   -numeroDePuertas: int
   --
   +abrirPuertas(): void
}

'La flecha con triangulo blanco indica generalización
Vehiculo <|-- Auto: es un
Vehiculo <|-- Moto: es un
@enduml
```



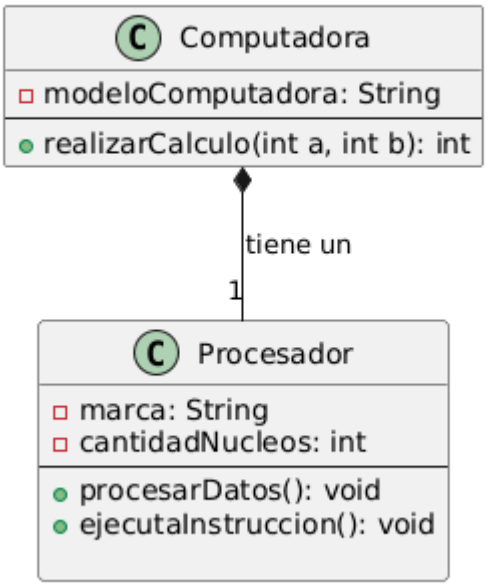
Ejercicio 2-Relación de Composición

```
    'Ejercicio 3 Trabajo practico N°6
'Modelar una relación de composición entre Computadora y Procesa
dor.
'Dibujar el diagrama UML correspondiente, indicando que el Procesa
dor no existe sin la Computadora.

'Es una relación fuerte

@startuml

title 2. Relación entre Computadora y Procesador

'Clase contenedora
class Computadora{
    -modeloComputadora: String
    --
    + realizarCalculo(int a, int b): int
}

'Clase Contenida
class Procesador{
    -marca: String
```

```
    -cantidadNucleos: int
    --
    +procesarDatos(): void
    +ejecutaInstruccion(): void

}

'Relaciones
'El rombo negro , indica una composición fuerte (dependencia de exi
stencia)
Computadora *-- "1" Procesador: tiene un

@enduml


'Tambien puede ser una relación a varios

@startuml

title 2. Relación entre Computadora y Procesador

'Clase contenedora
class Computadora2{
    -modeloComputadora: String
    --
    + realizarCalculo(int a, int b): int
}

'Clase Contenida
class Procesador2{
    -marca: String
    -cantidadNucleos: int
    --
    +procesarDatos(): void
    +ejecutaInstruccion(): void

}
```

```
class Memoria{
    -marca: String
    -capacidad: int
    --
    +guardarInformacion(): void

}


'Relaciones
'El rombo negro , indica una composición fuerte (dependencia de exi
stencia)
Computadora2 *-- "*" Procesador2: tiene un
Computadora2 *-- "*" Memoria: tiene un

@enduml
```

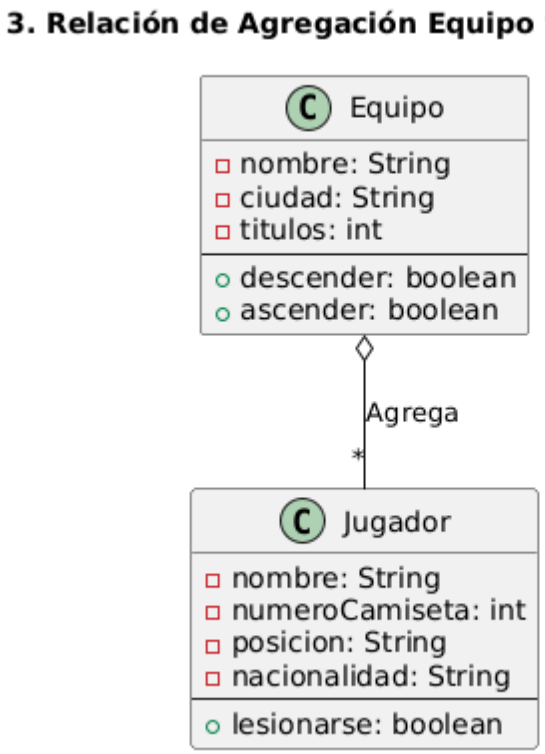
2. Relación entre Computadora y Procesador



Ejercicio 3 - Relación de Agregación

```
'Ejercicio 3 Trabajo practico N°6
'Modelar una relación de composición entre Equipo y Jugador.
'Dibujar el diagrama UML correspondiente, teniendo en cuenta que l
```

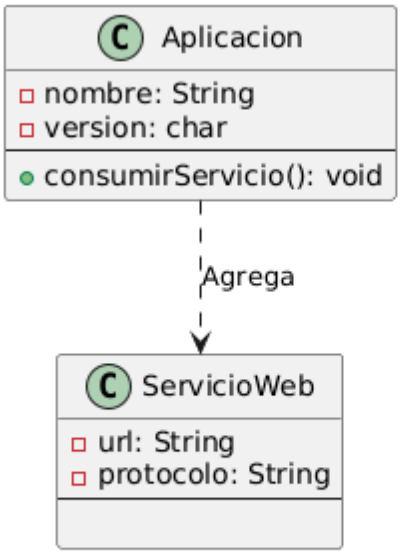
```
os Jugadores pueden existir independientemente del Equipo.


@startuml

title 3. Relación de Agregación Equipo y Jugador

class Equipo{
   -nombre: String
   -ciudad: String
   -titulos: int
   --
   +descender: boolean
   +ascender: boolean
}


class Jugador{
   -nombre: String
   -numeroCamiseta: int
   -posicion: String
   -nacionalidad: String
   --
   +lesionarse: boolean
}

Equipo o-- "*" Jugador: Agrega

@enduml
```



Ejercicio 4 - Relación de dependencia

```
'Ejercicio 4 Trabajo practico N°6
'Modelar una relación de dependencia entre Aplicación y ServicioWe
b.
'Dibujar el diagrama UML correspondiente, representando que la Apli
cacion utiliza el ServicioWeb de forma opcional o temporal.

@startuml

title 4. Relación de Dependencia Aplicacion y Servicio Web

class Aplicacion{
   -nombre: String
   -version: char
   --
   +consumirServicio(): void
}


class ServicioWeb{
```

```
    -url: String
    -protocolo: String
    --

}

Aplicacion ..> ServicioWeb: Agrega

@enduml
```



Ejercicio 5 - Relación de Herencia

▼ Clase 6 - 12/9 - Practica UML/JAVA/Relaciones

Consultorio Odontológico

Primero leemos todo para entender el contexto:

- Personas
- Registro de Atención

- Prácticas Odontológicas
- Gestiones Administrativas (Ejemplo sacar turno)

Tips PlantUML

Ver la documentación de PlantUML (plantuml.com/class-diagram)

- `top to bottom direction` → Ordena desde arriba hacia abajo

Clase 7 - 19/9 - Interfaces

Agenda

- Interfaces en JAVA - polimorfismo, abstracción, desacoplamiento
- Clases abstractas vs Interfaces (“Es un” vs “Utiliza”)
- Problema del diamante → Resolución en JAVA, C++ , C# y Python
- UML
- Practica Pre-Parcial

Interfaces

Contratos para las clases, ponemos metodos que tienen que ser implementados en una clase → ***“Implemento una interfaz”***

Es abstracta → O sea todos los metodos son públicos por defecto

No se pueden instanciar

Palabras reservadas → `interface` y `extends` (Palabra reservada que uso en la clase)

Una clase puede implementar una o mas interfaces, permitiendo si la herencia múltiple

Implementa el **QUE** pero no el **COMO**

¿Qué ventaja le doy a la clase?

- Otorgan abstracción: Oculta detalles de implementación al ojo ajeno
- Habilitan polimorfismo: Permiten que diferentes clases respondan al mismo método de forma única. Ejemplo el mismo metodo hacerSonido()

lo pueden hacer el Perro, Gato , Pájaro todos hacen sonido pero un sonido diferente

- Facilitan el desacoplamiento: reducen dependencia directa entre clases logrando un código mas fácil de mantener y testear (No esta bueno que una clase dependa de otra clase, evitar en lo posible)

```
//Archivo: Figura.java
public interface Figura{
    double calcularArea();
}

//Implementamos la interfaz en una clase concreta Circulo.java
public class Circulo implements Figura{
    private double radio;

    public Circulo(double radio){
        this.radio = radio;
    }

    @Override
    public double calcularArea(){
        return Math.PI * radio * radio;
    }
}

//Archivo Cuadrado.java
public class Cuadrado implements Figura{
    private double lado;

    public Cuadrado(double lado){
        this.lado =lado;
    }

    @Override
    public double calcularArea(){
        return lado * lado;
```

```
    }
}
```

```
//Interfaz vehiculo
public interface Vehiculo{
    void acelerar
}

public class Coche implements Vehiculo{

}

pblic class Bicicleta implements Vehiculo{

}

public class Main{

    Vehiculo miCoche = new Coche();
}
```

Resumen

Son contratos que definen un conjunto de reglas que las clases deben seguir . para que clases muy diferentes puedan aplicar algo mismo

Código flexible y extensible

Permiten que un objeto tome muchas formas → polimorfismo

Simula la herencia múltiple → Una clase puede heredar comportamientos de múltiples interfaces

Una clase puede implementar múltiples interfaces

Es una relación “Puede hacer” → define capacidad o comportamiento que pueden tener clases relacionadas o no

Ejemplo clase avion, clase Superman ,clase pajaro → interfaz Volar

Tiene el **QUE** , no el **COMO**

No tienen atributo, tienen comportamientos

Clase Abstracta vs Interfaz

- Las clases abstractas pueden tener implementación o no de sus métodos, por lo que son de tipo `abstract`
- Una clase abstracta es el molde de los moldes ya que es una base para crear otras clases
- La clase abstracta extiende una única clase abstracta . No permite herencia múltiple
- Se usa para compartir código común entre subclases
- Clase abstracta es una relación “Es un”

Problema del Diamante

El problema surge cuando una clase hereda de dos o más clases que a la vez heredan de una misma clase en común

Ejemplo imaginemos que tenemos una clase base llamada personas, dos clases estudiante y empleado que heredan de personas . Finalmente una clase pasante hereda tanto de estudiante como de empleado. En términos conceptuales es correcto pero el compilador y los lenguajes es donde esta el tema

> En JAVA esto lo podemos arreglas con una clase persona base, una clase pasante que hereda de persona y don interfaces estudiante y empleado

- C++ → Permite la herencia múltiple
- JAVA y C# → No permiten la herencia múltiple de clases, una clase solo puede heredar de una superclase. Ofrecen flexibilidad de la herencia múltiple permitiendo que una clase implemente múltiples interfaces
- Python → Permite herencia múltiple de clases , resuelve este problema con un algoritmo MRO (Method Resolution Order)
- Kotlin → Se asemeja a JAVA

UML - Pokemon

Usamos 3 pokemones de tipo planta

Interfaces

interface TipoPlanta

+latigoCepa()

+hojaNavaja()

+rayoSolar()

interface TipoVeneno

+envenenar()

+polvoVenenoso()

+picar()

Clase Abstracta

abstract class Pokemon

vida

tipo

nombre

atacar()

yoTeElijo()

regresar()

Clases

Bulbasaur extends Pokemon implements TipoPlanta , TipoVeneno

```
new Bulbasaur(){
   this.nombre = "Carlos";
```

```
    this.tipo = "Planta";
}
```

TP ENERGÍAS RENOVABLES

```
@startuml


enum Turno{
    MANIANA
    TARDE
    NOCHE
}
'----- Generacores de energía -------
abstract class UnidadDeGeneracion{


}


class PanelSolar{

}
class TurbinaHidraulica{

}

class TurbinaEolica{

}

class TurbinaKaplan{

}

class TurbinaFrancis{
```

```
}

'-----Unidades de almacenamiento -------

class BancoDeBaterias{
    -id: String
    -eficiencia: double
    -temperatura: double
    -perdidaMW: double
    - modulos: List<ModuloDeBaterias>
    + almacenarEnergia(mw: double):void
    + suministrarEnergia(mw: double): dpuble
    + getNivelCargaActual(): double
    + balancearCargas(): void

}


class ModuloDeBaterias{
    - codigo: String
    - cargaAtualKwh: double
    - capacidadMaximaDeCargaKwh: double
    - temperaturaDelModulo: double
    - voltajeActual: double
    - ciclosDeCarga: int
    + conectarModulo(): void
    + desconectarModulo(): void
    + actualizarCarga(deltaKwh: double): void
    + estadoDeSaludo(): double
}

'----- Personal y Contratistas -----

abstract class Empleados{
    # nombre: String
    # legajo: int
    # turno: Turno
    # dni: String
```

```
    # email: String
    +getEmail():String
    +getLegajo():String
    +cambiarTurno(nuevoTurno: Turno): void
}

class Ingeniero{
    - especializacion: String
    - herramientas: ???
    + ejecutarPlabDeMantenimiento(unidad: UnidadDeGeneracion): void
    + planificarMantenimiento(unidad: UnidadDeGeneracion, fecha: Strin
g): void
    + validarSeguridad(unidad: UnidadDeGeneracion): boolean
}

class Operador{
    - unidadesAsignadasASupervision: List<UnidadDeGeneracion>
    + monitorear(unidadGeneracion: UnidadDeGeneracion): void
    + iniciarTurno(): void
    + finalizarTurno(): void
    + generarAlerta(unidadGeneracion: UnidadDeGeneracion, mensaje: S
tring): void
}

class Proveedor <<provider>> {
    - empresaContratista: String
    - nombre: String
    - CUIT: String
    - telefono: String
    - responsableStaff: Empleado
    + setResponsable(empleado: Empleado): void
    + realizarMantenimiento(unidad: UnidadDeGeneracion): void

}

'----Herramientas de Diagnostico ----
interface HerramientaDeDiagnostico{
    +diagnosticar(unidad: UnidadDeGeneracion): void
```

```
    +calibrar(fecha: String): void
    +version(): String
}

class HerramientaParaEnergiaSolar{
    +diagnosticar(unidad: UnidadDeGeneracion): String
    +calibrar(fecha: String): void

    +version():String
    +medirIrradancia():double

    -modelo: String
    +medirRadiacion(): double
}

class HerramientaParaEnergiaEolica{
    +diagnosticar(unidad: UnidadDeGeneracion): String
    +calibrar(fecha: String): void

    +version():String
    +medirVelocidadViento():double

    -modelo: String

}

class HerramientaParaEnergiaHidraulica{
    +diagnosticar(unidad: UnidadDeGeneracion): String
    +calibrar(fecha: String): void

    +medirCaudal():double
}


'-----Relaciones -----
UnidadDeGeneracion <|-- TurbinaHidraulica: es una
UnidadDeGeneracion <|-- TurbinaEolica: es una
UnidadDeGeneracion <|-- PanelSolar: es una
```

```
TurbinaHidraulica <|-- TurbinaKaplan: es una
TurbinaHidraulica <|-- TurbinaFrancis: es una

@enduml
```

▼ TP Lockers Inteligentes

- Lockers
  - Poseen multiples casilleros
  - **Atributos:**
    - códigoUnico
    - dirección
    - estadoOperativo (activo , fuera de servicio)
- Casilleros → No existen sin su locker *guiño*
  - Tienen tamaños
  - algunos con control de temperatura
  - **Atributos:**
    - idInterno
    - esRefrigerado
    - estado (libre, pcupado, bloqueado, manetnimiento)
- Usuarios (va a ser la clase abstracta)
  - CLIENTE
  - TRANSPORTISTA
  - TECNICO
- Paquetes y Ordenes
  - paquete posee:
    - idUnico
    - dimensiones
    - peso

        - tipo (existen tres variantes (estandar, fragil y refrigerado))
    - orden de entrega
        - esta relacionada con un locker/casillero, cliente y transportista
        - posee:
            - fechaRetiro
            - HoraRetiro
            - ventanadetiemporetiro
            - QRRetiro
            - estado (creada, depositada, lista para retiro, vencida, retirada , reubicada)

Flujo

Transportirsa deposita un paquete en un casillero , el ciente lo retira dentro de una venata de tiempo. El sistema gesitona notificaciones, venciminetos y fallas

📝 Apuntes Parcial