Programación 2

Clases y objetos. Diagrama de clase. Implementación. Verificación

En esta clase

- REPASO
- El modelo computacional de la POO
- El caso de Estudio **Cuenta Bancaria**
- El diagrama de una clase
- La estructura de una clase en Python
- Los atributos de clase y de instancia
- Los servicios: constructores, comandos y consultas
- La clase tester

Repaso - proceso de desarrollo

Un **sistema** de software se construye en el marco de un proyecto.

Si el proyecto tiene éxito el sistema desarrollado es una **solución** para el problema que le dio origen.

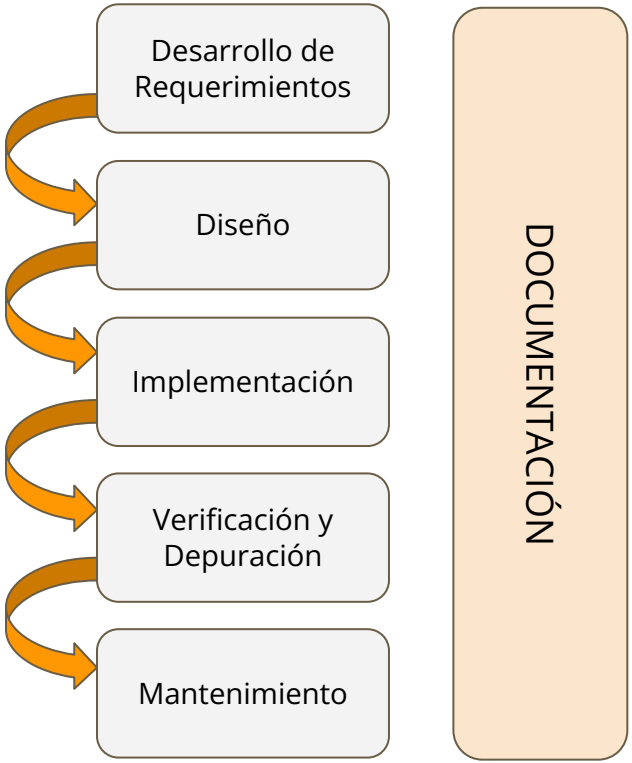
El problema puede ser una necesidad, una oportunidad o una idea de producto.

El **proceso de desarrollo de software** abarca distintas **etapas** que pueden organizarse de diferentes maneras.

Una de las alternativas más simples es la estructura en **cascada**.

Cada etapa es responsabilidad de un profesional o un grupo que ocupa un **rol** específico dentro del **equipo**.

Repaso - ciclo de vida



El proceso de desarrollo requiere creatividad pero también de una metodología que se aplique sistemáticamente.

Para aplicar la metodología necesitamos herramientas, en particular un **lenguaje de modelado** y un **lenguaje de programación**.

En una metodología basada en la programación orientada a objetos el concepto central es el de **objeto**.

Repaso - Concepto de objeto

El término **objeto** se utiliza para referirse a dos conceptos relacionados pero diferentes.

Durante el desarrollo de requerimientos y el diseño del sistema se **identifican** los **objetos del problema**.

Durante la ejecución del sistema que modela el problema se **crean objetos de software**.

Cada objeto del problema está asociado a un objeto de software que lo representa en ejecución.

Repaso - Concepto de clase

Durante el diseño de un sistema una **clase** es un patrón que establece los **atributos** y el **comportamiento** de un conjunto de objetos.

Un **atributo** es una propiedad o cualidad relevante que caracteriza a todos los objetos de una clase.

El **comportamiento** queda determinado por el conjunto de **servicios** que brinda la clase y las **responsabilidades** que asume.

En la implementación de un sistema una **clase** es un **módulo de software** que puede ser desarrollado con alguna independencia del resto de los demás módulos, reflejando lo establecido en el diseño.

Análisis de requerimientos

El ajedrez es un juego de mesa para dos personas. Es uno de los juegos más populares del mundo. Se considera no sólo un juego, sino un arte, una ciencia y un deporte mental.

Cada Jugador posee 16 piezas, con diferentes capacidades de movimiento, que se mueven en un tablero cuadrado de 8×8 casillas, que alternan entre claras y oscuras.

Cada jugador inicia la partida con: un rey, una reina (o dama), dos alfiles, dos caballos, dos torres y ocho peones.

Análisis de requerimientos

En esta etapa se **detallan los requerimientos** y se elabora un **modelo del problema** a partir de la identificación (abstracción) de **objetos** relevantes y de sus **atributos**, lo cual permite agruparlos (clasificarlos) en **clases**.

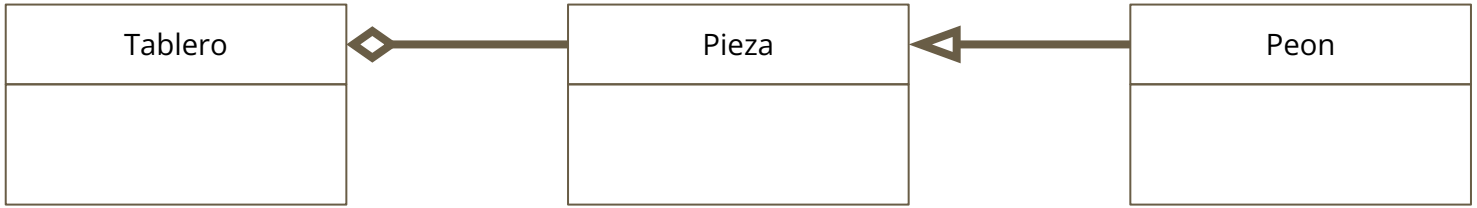
Diseño

Durante el **diseño** se identifica el **comportamiento** de los objetos y se completa la especificación de las clases.

La estrategia que se aplica es Top-Down (dividir el problema en subproblemas) pero los módulos no son subprogramas o funciones y procedimientos, sino clases.

El resultado de esta etapa es un conjunto de diagramas que modelan la solución en un lenguaje de modelado.

Entre ellos el diagrama de clases especifica la colección de clases cómo se relacionan.



Implementación

La implementación de un sistema orientado a objetos consiste en escribir el código en un **lenguaje de programación**.

Cada clase modelada en el diagrama de clases se implementa en un **módulo de software**.

```
class Tablero:
    ...
     ...
```

```
class Pieza:
    ...
     ...
```

```
class Peon(Pieza):
    ...
     ...
```

La implementación de cada servicio puede constituir en sí mismo un problema y demandar el diseño de un **algoritmo** que lo resuelva.

Verificación

La **verificación de una clase** consiste en chequear que cada servicio funciona correctamente para un conjunto de casos de prueba.

Si la clase Tablero **usa** los servicios provistos por la clase Peon, verificaremos cada servicio de la clase Peon y luego los servicios de la clase Tablero.

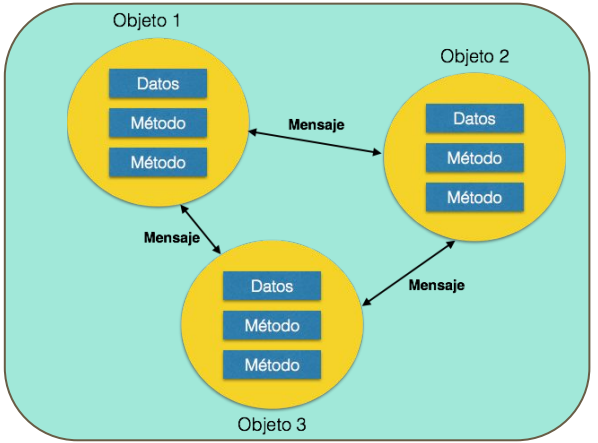
La **verificación del sistema** consiste en chequear la integración de los módulos para un conjunto de casos de prueba.

El modelo de Programación Orientada a Objetos

Objetos del problema

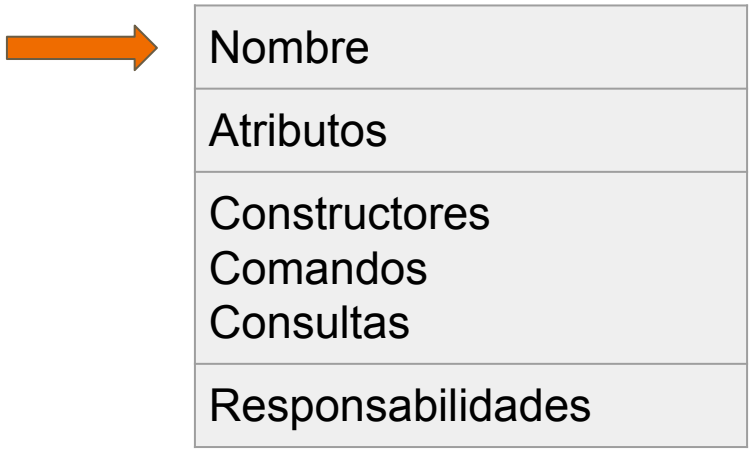
Abstracción

Objetos de software



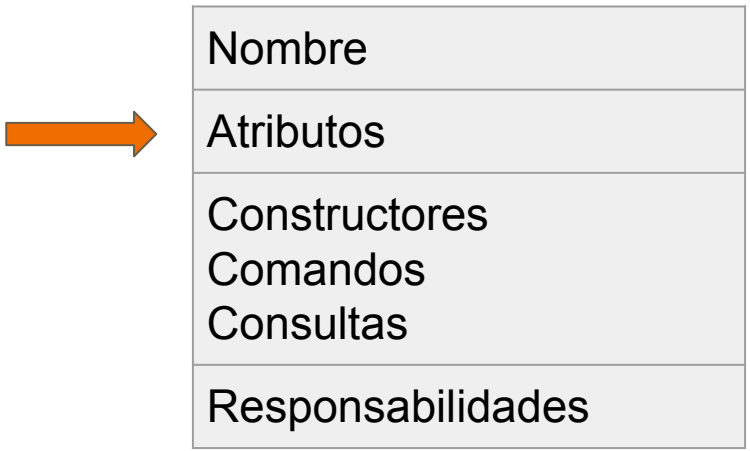
Podemos imaginar el **modelo** computacional de la **POO** como un conjunto de **objetos** comunicándose a través de **mensajes**.
Cada objeto responde a un mensaje de acuerdo al comportamiento determinado por su clase.

Diagrama de una clase



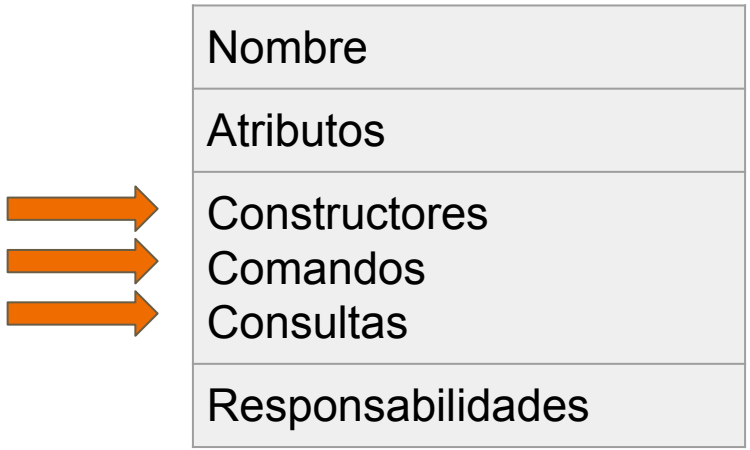
El **nombre** de la clase representa la abstracción del conjunto de instancias

Diagrama de una clase



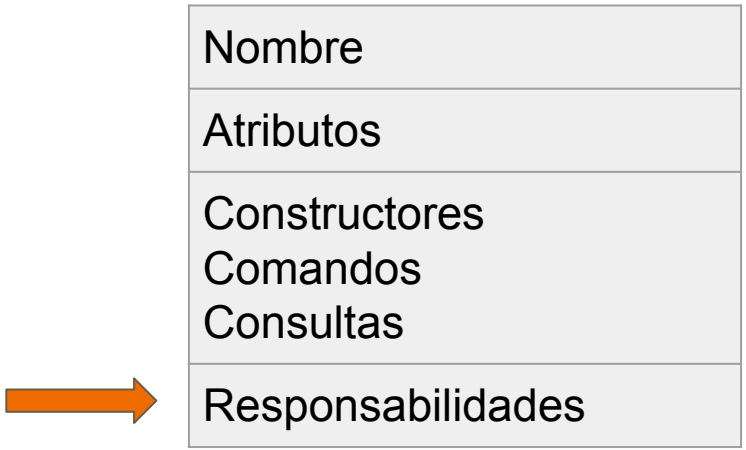
Un **atributo** es una propiedad relevante que caracteriza a cada elemento de la clase

Diagrama de una clase



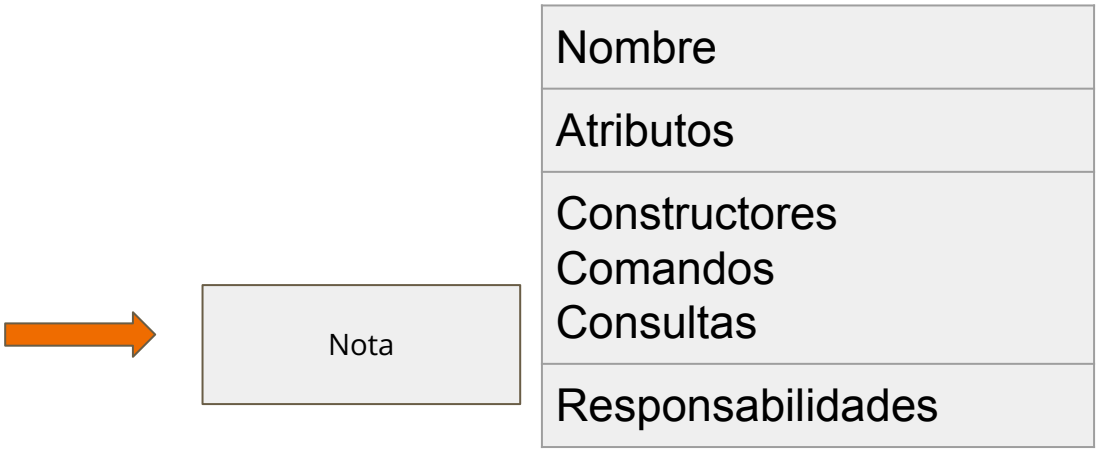
Un **servicio** es una operación que todas las instancias de la clase pueden realizar

Diagrama de una clase



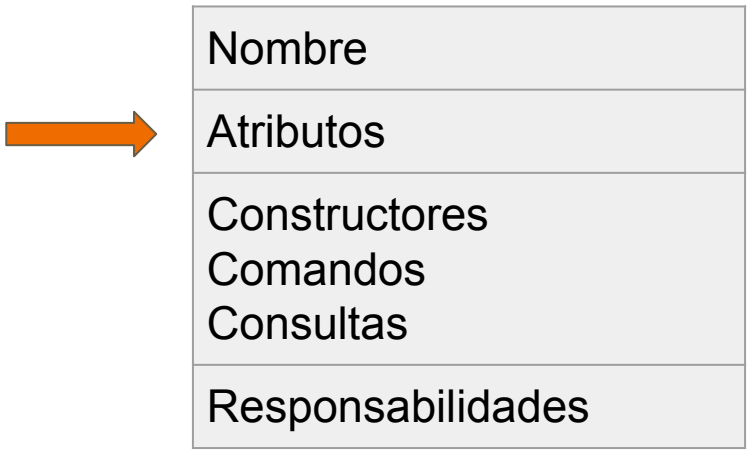
Una **responsabilidad** representa un compromiso para la clase o un requerimiento.

Diagrama de una clase



Las **restricciones** y la **funcionalidad** de los servicios puede especificarse a través de **notas** o **comentarios**.

Diagrama de una clase



Los atributos pueden clasificarse en **atributos de clase** y **atributos de instancia**.

Los atributos de clase comparten un mismo valor para todas las instancias.

Clases y objetos en python

En python todas las variables son objetos.

Cuando creamos un objeto se dice que estamos "*instanciando*" o "*creando una instancia de*" una clase.

Tip: en el código, si queremos verificar que una variable es de determinada clase (o tipo) podemos usar la función integrada **isinstance(variable, clase)**.

```
contador=0
print(type(contador)) # salida: <class 'int'>
print(isinstance(contador, int)) # salida: True
```

Estructura de una clase en python

Cada clase modelada en el diagrama de clases durante la etapa de diseño, va a implementarse como una clase en python.

La estructura de una clase implementada también va a ser similar a la estructura del diagrama.

Los **miembros** de una clase son:

- **Atributos** de instancia y de clase.
- **Servicios**, pueden ser constructores o métodos.

Estructura de una clase en python - constructores

Constructores:

Son un servicio provisto por la clase. En algunos lenguajes se caracteriza porque recibe el mismo nombre que la clase, en python se define con ***__init__(self)***

El constructor **se invoca cuando se crea un objeto** y habitualmente se usa para **inicializar los valores de los atributos de instancia**.

En otros lenguajes las clases pueden brindar **varios constructores**, siempre que tengan diferente número o tipo de parámetros. En Python solo se acepta un constructor, por lo tanto para abarcar los casos de diversos constructores usaremos parámetros opcionales.

Si en una clase no se define explícitamente un constructor, python nos creará uno que no realiza ninguna inicialización específica.

Ej. constructor con parámetros opcionales

```
class Persona:
    __cantidad_personas = 0

    def __init__(self, nombre:str = "", edad:int = 0, ocupacion:str = ""):
        self.__nombre = nombre
        self.__edad = edad
        self.__ocupacion = ocupacion
        Persona.__cantidad_personas += 1
```

Estructura de una clase en python - métodos

Comandos y Consultas:

Los **comandos** son servicios que **modifican los valores de los atributos** de un objeto.

Las **consultas** son servicios que no modifican los valores de los atributos, en general, **devuelven** un resultado que corresponde al **valor de un atributo** o al **cómputo de una expresión**.

Un comando puede retornar también un valor.

Los comandos y consultas conforman los **métodos** de una clase.

Estructura de una clase en python - métodos

Comandos y Consultas:

En general, las clases incluyen métodos que modifican los valores de sus atributos de instancia, y consultas que devuelven el valor de cada atributo de instancia. (son conocidos como **comandos y consultas triviales**). EJ:

Comandos (modifican valores)

- establecerNombre(self, nombre)
- establecerEdad(self, edad)
- establecerOcupacion(self, ocupacion)

Consultas (no modifican valores)

- obtenerNombre(self)
- obtenerEdad(self)
- obtenerOcupacion(self)

En algunos casos, algunos atributos de instancia se inicializan en la creación del objeto y no son modificados.

¿Cómo lo trabajaremos en Python?

- Para declarar una clase, simplemente usa la palabra clave **‘class’** seguida por el nombre de la clase.
- Por convención, los nombres de las clases en Python suelen comenzar con mayúscula, a las variables por lo general le asignamos nombres que comienzan en minúscula.
- Declaramos los atributos como **privados** para que solo sean accesibles dentro de la clase (con doble guion bajo al inicio).
- Para consultar el valor de cada atributo definimos un **método** que retorna el valor del atributo.
- Mantenemos el orden y los comentarios del diagrama en el código para reflejar su estructura.

```
class MiClase:
    pass
```

```
class Persona:
    pass
```

Atributos de clase y de instancia

- Los atributos son variables que pertenecen a la clase.
- Pueden ser variables de instancia (pertenece a una instancia específica de la clase) o variables de clase (compartida por todas las instancias de la clase).
- Cuando utilizamos un atributo, Python busca primero en los atributos de instancia, y luego en los atributos de clase.

```
class Persona:
    __variable_de_clase = "Soy una variable de clase"
    # Constructor con variables de instancia:
    def __init__(self):
        self.__nombre = ""
        self.__edad = 0
        self.__ocupacion = ""
```

Caso de estudio: cuenta bancaria

Un banco ofrece **cuentas corrientes** a sus clientes.

*Los clientes pueden realizar **depósitos**, **extracciones** y **consultar el saldo** de su cuenta corriente.*

En el momento que se crea una cuenta corriente se establece su código y el saldo se inicializa en 0.

También es posible crear una cuenta corriente estableciendo su código y saldo inicial.

El código no se modifica, el saldo cambia con cada depósito o extracción.

Una cuenta bancaria puede tener un saldo negativo hasta un máximo establecido por el banco.

Caso de estudio: cuenta bancaria

Atributos

El diagrama incluye un atributo de clase que establece el monto máximo que cada cliente puede extraer en descubierto.

Este valor es el mismo para todas las cuentas corrientes, de modo que lo modelamos mediante un atributo de clase.

Caso de estudio: cuenta bancaria

Constructores

"En el momento que se crea una cuenta corriente se establece su código y el saldo se inicializa en 0.

También es posible crear una cuenta corriente estableciendo su código y saldo inicial. "

CuentaCorriente
<<Atributos de clase>> limiteDescubierto = 1000 <<Atributos de instancia>> codigo: entero saldo: real
<<Constructores>> **CuentaCorriente(cod:entero)** **CuentaCorriente(cod:entero, saldo:real)** Comandos Consultas
Responsabilidades

Caso de estudio: cuenta bancaria

Comandos

La clase brinda dos comandos, cada comentario establece una restricción y/o la funcionalidad.

Depositar(monto: real)
requiere monto > 0

Extraer(monto: real): boolean
requiere monto>0
Si monto > saldo+limiteDescubierto extraer retorna false y la extracción no se realiza

CuentaCorriente
<<Atributos de clase>> limiteDescubierto = 1000 <<Atributos de instancia>> codigo: entero saldo: real
CuentaCorriente(cod:entero) CuentaCorriente(cod:entero, saldo:real) <<Comandos>> **Depositar(monto: real)** **Extraer(monto: real): boolean** Consultas
Responsabilidades

Caso de estudio: cuenta bancaria

Consultas

La clase brinda dos consultas triviales.

Depositar(monto: real)
requiere monto > 0

Extraer(monto: real): boolean
requiere monto>0
Si monto > saldo+limiteDescubierto
extraer retorna false y la extracción no se realiza

CuentaCorriente
<<Atributos de clase>> limiteDescubierto = 1000 <<Atributos de instancia>> codigo: entero saldo: real
CuentaCorriente(cod:entero) CuentaCorriente(cod:entero, saldo:real) <<Comandos>> Depositar(monto: real) Extraer(monto: real): booleano <<Consultas>> **obtenerCodigo(): entero** **obtenerSaldo(): real**
Responsabilidades

Caso de estudio: cuenta bancaria

Responsabilidades

La clase **CuentaCorriente** es **responsable** de garantizar que el saldo va a ser mayor o igual a una constante.

Depositar(monto: real)
requiere monto > 0

Extraer(monto: real): boolean
requiere monto>0
Si monto > saldo+limiteDescubierto
extraer retorna false y la extracción no se realiza

CuentaCorriente
<<Atributos de clase>> limiteDescubierto = 1000 <<Atributos de instancia>> codigo: entero saldo: real
CuentaCorriente(cod:entero) CuentaCorriente(cod:entero, saldo:real) <<Comandos>> Depositar(monto: real) Extraer(monto: real): booleano <<Consultas>> obtenerCodigo(): entero obtenerSaldo(): real
Asegura codigo > 0 y saldo >= (-1) * limiteDescubierto

Decisiones del diseño

En el diseño se estableció que cada cuenta bancaria se modele con dos atributos de instancia: código y saldo.

El código tiene que ser estrictamente mayor a 0, el saldo debe ser mayor a una constante.

En el momento que se crea una cuenta es indispensable establecer el código, que no va a cambiar.

El saldo puede establecerse explícitamente al crearse una cuenta, en caso que no se especifique quedará inicializado en 0.

Decisiones del diseño

Los comandos que modifican el saldo de la cuenta requieren que el parámetro monto contenga un valor mayor a 0.

La clase CuentaCorriente va a formar parte de una colección de clases.

Según la especificación del diseño, toda clase que use a la clase CuentaCorriente es responsable de garantizar que el parámetro real (monto) de depositar y extraer sea un valor mayor a cero. Si bien esto es **requerido** por la clase CuentaCorriente, ***toda clase debe validar que los datos que recibe sean válidos***.

El comando extraer retorna un valor booleano indicando si la operación pudo realizarse.

Caso de estudio: cuenta bancaria - implementación

CuentaCorriente
<<Atributos de clase>> **limiteDescubierto = 1000** <<Atributos de instancia>> **codigo: entero** **saldo: real**
Constructores Comandos Consultas
Responsabilidades

Caso de estudio: cuenta bancaria - implementación

La clase **CuentaCorriente** define un atributo de clase constante **__LIMITE_DESCUBIERTO**.

Todos los objetos de clase compartirán el mismo valor para el atributo de clase.

En cambio, la clase **CuentaCorriente** define atributos de instancia: __codigo y __saldo.

Cada objeto puede tener diferentes valores en los atributos de instancia.

```
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000

    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        self.__codigo = codigo
        self.__saldo = saldo
```

Caso de estudio: cuenta bancaria - implementación

En python no tenemos modificadores de acceso como en otros lenguajes (C#, C++, Java, etc.), por lo tanto usaremos el doble guión bajo antes del identificador para las variables privadas y métodos privados.

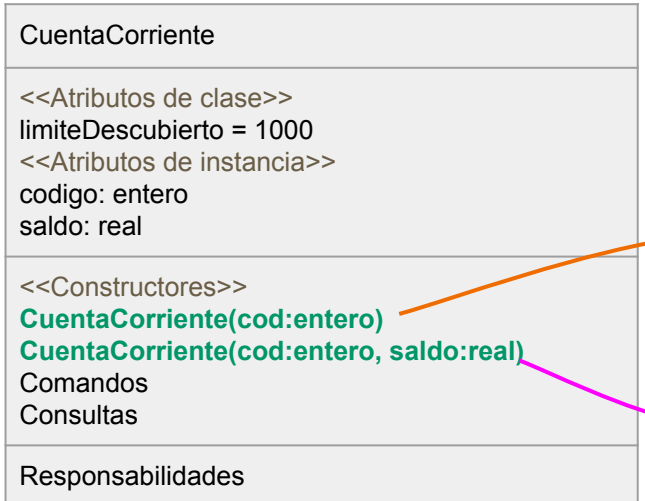
CuentaCorriente
<<Atributos de clase>> **limiteDescubierto = 1000** <<Atributos de instancia>> **codigo: entero** **saldo: real**
Constructores Comandos Consultas
Responsabilidades

```python
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000

    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        self.__codigo = codigo
        self.__saldo = saldo
```

Caso de estudio: cuenta bancaria - implementación

En Python, una clase no puede tener múltiples métodos __init__ (constructores) con diferentes firmas como en otros lenguajes de programación (por ejemplo, Java, C#, C++, etc.). Si tuviéramos varias declaraciones __init__, el último __init__ definido sobrescribirá cualquier definición anterior. Sin embargo, se puede simular el comportamiento de múltiples constructores utilizando valores por defecto para los parámetros y lógica condicional dentro de un único __init__.




```python
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000

    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        self.__codigo = codigo
        self.__saldo = saldo
```

Caso de estudio: cuenta bancaria - implementación

CuentaCorriente
<<Atributos de clase>> limiteDescubierto = 1000 <<Atributos de instancia>> codigo: entero saldo: real
CuentaCorriente(cod:entero) CuentaCorriente(cod:entero, saldo:real) <<Comandos>> **Depositar(monto: real)** **Extraer(monto: real): boolean** Consultas
Responsabilidades

Depositar(monto: real)
requiere monto > 0

Extraer(monto: real): boolean
requiere monto>0
Si monto > saldo+limiteDescubierto
extraer retorna false y la extracción no se realiza

El comando extraer accede a un atributo de clase y uno de instancia, limiteDescubierto y saldo, respectivamente.

```
def depositar(self, monto: float):
    self.__saldo += monto
```

```
def extraer(self, monto: float)->bool:
    puedeExtraer = False
    if self.__saldo + CuentaCorriente.__LIMITE_DESCUBIERTO >= monto:
        self.__saldo -= monto
        puedeExtraer= True
    else:
        puedeExtraer= False
    return puedeExtraer
```

La variable **puedeExtraer** es local al método. Se crea en el momento que se ejecuta la declaración y se destruye al terminar la ejecución de **extraer.**

El tipo que retornamos es compatible con el tipo que establecimos en la declaración del método.

Recordando

El pasaje de parámetros en Python se realiza por referencia para objetos mutables y por valor para objetos inmutables.

Objetos Inmutables: Son aquellos que no pueden ser modificados después de su creación. Ejemplos incluyen números (int, float), cadenas de caracteres (str) y tuplas (tuple).

Objetos Mutables: Son aquellos que pueden ser modificados después de su creación. Ejemplos incluyen listas (list), diccionarios (dict) y conjuntos (set).

La **reasignación** de una variable es diferente de la mutabilidad del objeto al que apunta. Podemos reasignar una variable para que apunte a un nuevo objeto, pero eso no cambia la naturaleza mutable o inmutable del objeto original. Ej:

```
x = 5
x = 10  # Reasignación de la variable x a un nuevo objeto int con valor 10
```

Caso de estudio: cuenta bancaria - implementación

CuentaCorriente
<<Atributos de clase>> limiteDescubierto = 1000 <<Atributos de instancia>> codigo: entero saldo: real
CuentaCorriente(cod:entero) CuentaCorriente(cod:entero, saldo:real) <<Comandos>> Depositar(monto: real) Extraer(monto: real): booleano <<Consultas>> **obtenerCodigo(): entero** **obtenerSaldo(): real**
Responsabilidades

```python
def obtenerSaldo(self)->float:
    return self.__saldo

def obtenerCodigo(self)->int:
    return self.__codigo
```

```python
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000

    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        self.__codigo = codigo
        self.__saldo = saldo

    def depositar(self, monto: float):
        self.__saldo += monto

    def extraer(self, monto: float)->bool:
        puedeExtraer = False
        if self.__saldo + CuentaCorriente.__LIMITE_DESCUBIERTO >= monto:
            self.__saldo -= monto
            puedeExtraer= True
        else:
            puedeExtraer= False
        return puedeExtraer

    def obtenerSaldo(self)->float:
        return self.__saldo

    def obtenerCodigo(self)->int:
        return self.__codigo
```

Verificación de los servicios de una clase

Durante la implementación de un sistema para un Banco, la clase CuentaCorriente puede pensarse como una pieza, un módulo de la colección de módulos que en conjunto va a constituir el sistema.

Antes de que la clase se integre en la colección y pueda ser usada por otras clases, es muy importante **verificar** que actúa de acuerdo a su especificación.

Los casos de prueba puede definirlos el responsable de testing o el diseñador, considerando las **responsabilidades** de la clase y las **restricciones** y **funcionalidades** especificadas en los requerimientos.

Verificación de los servicios de una clase

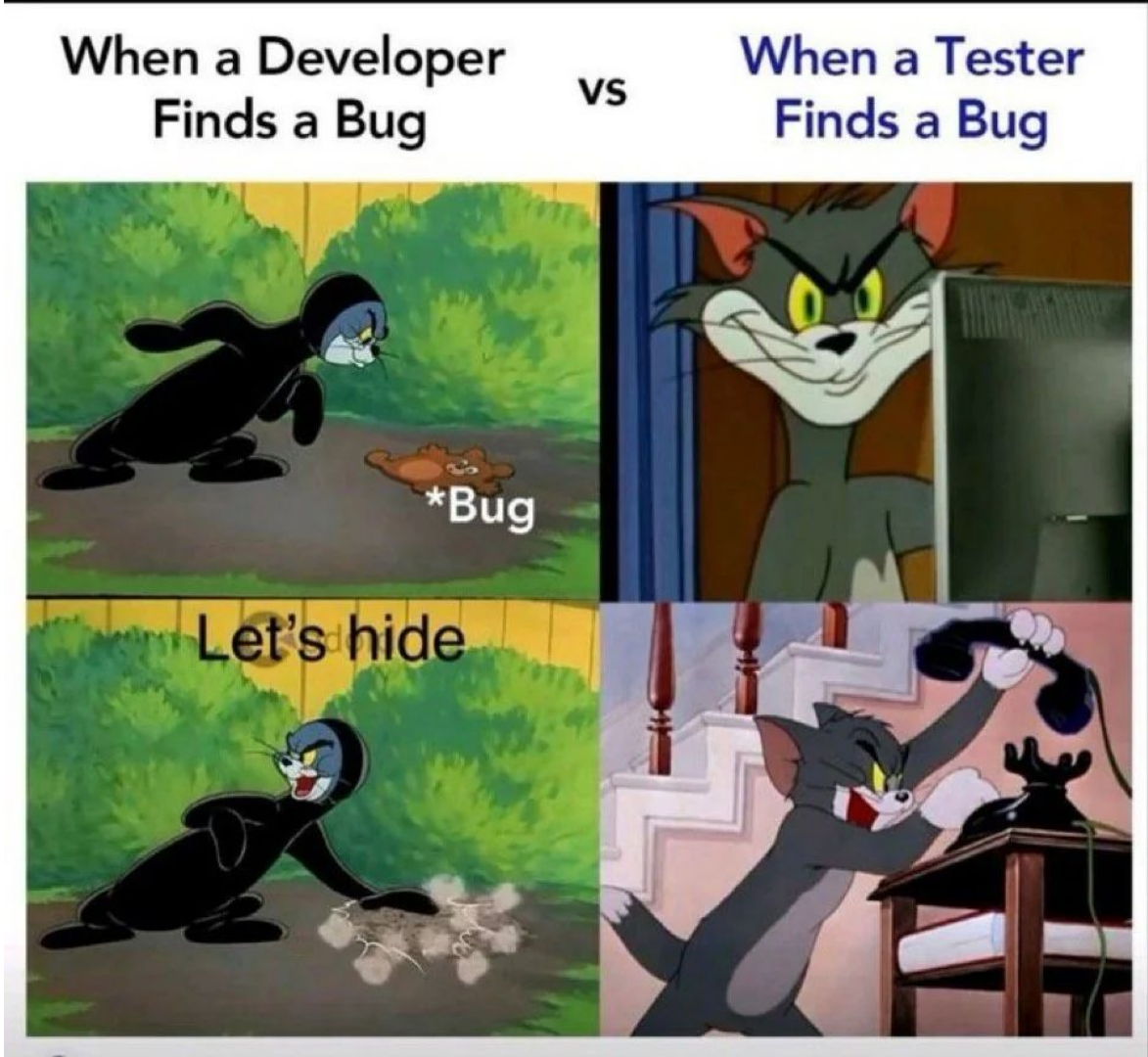
Una alternativa es escribir una clase **testerCuentaCorriente** que verifique los servicios provistos por la clase **CuentaCorriente** para un conjunto de casos de prueba.

La clase testerCuentaCorriente **usa** a la clase CuentaCorriente, e incluye código a ejecutar.

(por las dudas)

Es humor!

Los bugs deben ser **corregidos** (o documentados) cuando se detectan.



```python
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # validamos el funcionamiento simulando el uso de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta_1 = CuentaCorriente(1, 1000)
        cuenta_2 = CuentaCorriente(2)
        cuenta_1.depositar(100)
        cuenta_2.depositar(100)
        print(f"Saldo cuenta 1: {cuenta_1.obtenerSaldo()}") # 1100
        print(f"Saldo cuenta 2: {cuenta_2.obtenerSaldo()}") # 100
        if  cuenta_1.extraer(500):
            print(f"Extracción exitosa. Saldo cuenta 1: {cuenta_1.obtenerSaldo()}")
        else:
            print(f"No se pudo extraer 500 de cuenta_1. Saldo cuenta 1: {cuenta_1.obtenerSaldo()}")
        if cuenta_2.extraer(900):
            print(f"Extracción exitosa. Saldo cuenta 2: {cuenta_2.obtenerSaldo()}")
        else:
            print(f"No se pudo extraer 900 de cuenta_2. Saldo cuenta 2: {cuenta_2.obtenerSaldo()}")
        if  cuenta_1.extraer(300):
            print(f"Extracción exitosa. Saldo cuenta 1: {cuenta_1.obtenerSaldo()}")
        else:
            print(f"No se pudo extraer 300 de cuenta_1. Saldo cuenta 1: {cuenta_1.obtenerSaldo()}")
        if cuenta_2.extraer(300):
            print(f"Extracción exitosa. Saldo cuenta 2: {cuenta_2.obtenerSaldo()}")
        else:
            print(f"No se pudo extraer 300 de cuenta_2. Saldo cuenta 2: {cuenta_2.obtenerSaldo()}")

        cuenta_1.depositar(500)
        cuenta_2.depositar(500)
        print(f"Saldo cuenta 1: {cuenta_1.obtenerSaldo()}") # 800
        print(f"Saldo cuenta 2: {cuenta_2.obtenerSaldo()}") # -300

if __name__ == "__main__":
    TestCuentaCorriente.test()
```

Resultado de la ejecución:

```
Saldo cuenta 1: 1100
Saldo cuenta 2: 100.0
Extracción exitosa. Saldo cuenta 1: 600
Extracción exitosa. Saldo cuenta 2: -800.0
Extracción exitosa. Saldo cuenta 1: 300
No se pudo extraer 300 de cuenta_2. Saldo cuenta 2: -800.0
Saldo cuenta 1: 800
Saldo cuenta 2: -300.0
```

Aclaraciones parámetros opcionales

Para declarar parámetros opcionales simplemente le asignamos al parámetro un valor por defecto.

Reglas para declarar parámetros opcionales:

- Deben declararse después de los parámetros obligatorios. Si no genera un error de sintaxis.
- Documentar claramente los parámetros opcionales y sus valores predeterminados en la cadena de documentación (docstring) de la función.
- Los valores predeterminados deben ser del tipo correcto y el valor predeterminado no debe ser mutable (como listas o diccionarios). Esto se debe a que en Python evalúa los valores predeterminados de los parámetros sólo una vez al definir la función, no cada vez que se la llama.
  Ej:

```
def procedimiento(param1, param2=[]):  # Mala práctica
    pass

def procedimiento(param1, param2=None):  # Mejor práctica
    if param2 is None:
        param2 = []
    pass
```

@staticmethod

El decorador **@staticmethod** se utiliza para definir un método estático dentro de una clase que **no necesita acceder ni a la instancia** (usando self) **ni a la clase** (usando cls). Estos métodos son esencialmente funciones que se colocan dentro del espacio de nombres de la clase para mantener la organización y la lógica relacionada.

@staticmethod - uso

Se usa para:

Funciones independientes: cuando tengas un método que no necesita acceder a los atributos o métodos de la instancia o de la clase, pero conceptualmente tiene sentido que esté agrupado dentro de la clase.

Organización lógica: Es útil para organizar funciones que están relacionadas con la clase, pero no dependen de la instancia o de la clase en sí.

@staticmethod - características

- **No tiene acceso a *self***: A diferencia de los métodos de instancia, los métodos estáticos no tienen acceso al objeto actual (*self*).
- **No tiene acceso a *cls***: Tampoco tienen acceso a la clase (*cls*), como los métodos de clase.
- **Se llama directamente desde la clase**: Para llamar a un método estático, se utiliza el nombre de la clase seguido del nombre del método, no es necesario instanciar la clase:

```
MiClase.mi_metodo_estatico(arg1, arg2)
```

@staticmethod

En resumen, **@staticmethod** se utiliza cuando necesitas una función dentro de una clase que **no requiere acceso a la instancia o a la clase**, pero que tiene sentido que esté agrupada con otras funciones o métodos de la clase por **razones de organización o contexto**.

Convenciones

- Usar **identificadores significativos**.
- Incluir **comentarios** que describan la **estructura** del código, la **funcionalidad** de cada método y las establecidas en el diseño.
- No exagerar con los comentarios oscureciendo la lógica de la resolución.
- No escribir comentarios que expliquen características del lenguaje
- Si un método produce un resultado, incluir una **única instrucción de retorno al final**, excepto si todo el código del método es un if-else con instrucciones simples.
- Los atributos de instancia los declaramos **privados** (doble guión bajo al inicio).
- Los métodos que reciben valores deben **validarlos** antes de realizar operaciones con ellos.

Ejemplo validación (1)

```
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000

    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        """
        Inicializa una nueva cuenta corriente.

        Parámetros:
        - codigo: El código único de la cuenta.
        - saldo: El saldo inicial de la cuenta (default: 0.0).
        """
        if isinstance(codigo, int) and codigo > 0 and isinstance(saldo, (int, float)) :
            self.__codigo = codigo
            self.__saldo = saldo
        else:
            raise ValueError("El código debe ser un entero positivo y el saldo un número.")
```

Genera una excepción si no se cumple algún tipo de dato o rango

Ejemplo validación (2)

```python
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000
    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        """
        Inicializa una nueva cuenta corriente.

        Parámetros:
        - codigo: El código único de la cuenta.
        - saldo: El saldo inicial de la cuenta (default: 0.0).
        """
        if isinstance(codigo, int) and isinstance(saldo, (int, float)) :
            if codigo < 0:
                raise ValueError("El código debe ser mayor a cero.")
            self.__codigo = codigo
            self.__saldo = saldo
        else:
            raise TypeError("El código y el saldo deben ser números válidos.")
```

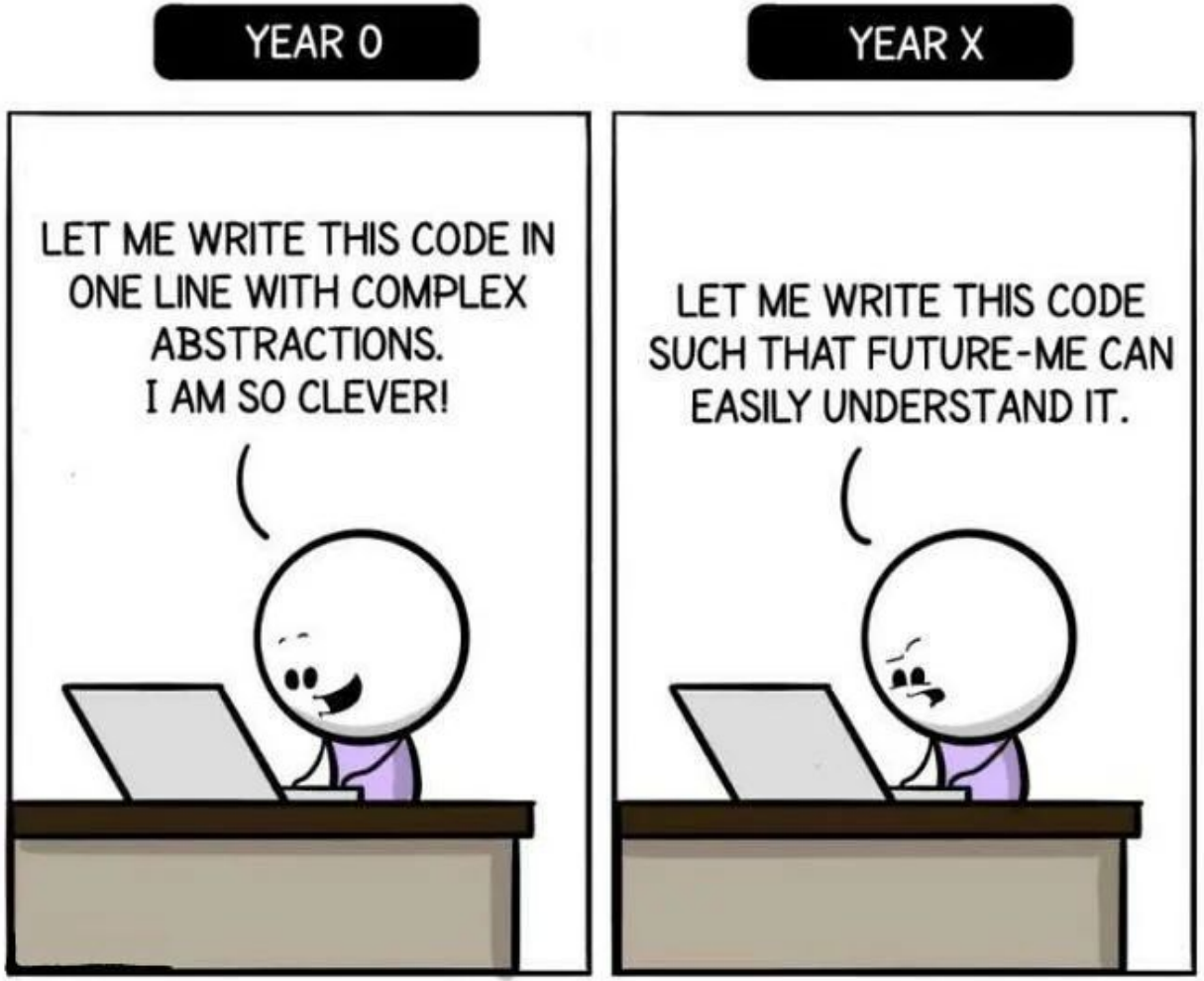
Genera una excepción si no se cumple algún tipo de dato, y otra excepcion si un rango está mal

Ejemplo validación (3)

```python
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000
    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        """
        Inicializa una nueva cuenta corriente.

        Parámetros:
        - codigo: El código único de la cuenta.
        - saldo: El saldo inicial de la cuenta (default: 0.0).
        """
        if not isinstance(codigo, int):
            raise TypeError("El código debe ser un número entero.")
        if not isinstance(saldo, (int, float)):
            raise TypeError("El saldo debe ser un número.")
        if codigo < 0:
            raise ValueError("El código no puede ser negativo.")
        self.__codigo = codigo
        self.__saldo = saldo
```

Genera una excepción para cada posible error. Brinda información detallada de la causa del error.


YEAR 0
YEAR X
LET ME WRITE THIS CODE IN ONE LINE WITH COMPLEX ABSTRACTIONS. I AM SO CLEVER!
LET ME WRITE THIS CODE SUCH THAT FUTURE-ME CAN EASILY UNDERSTAND IT.

Ejercicio en clase: Repartidor de PedidosYa

Un repartidor de PedidosYa desea registrar el dinero recaudado y la cantidad de viajes que realiza en un turno.

Por cada viaje:

- Gana $500 por km recorrido.
- A partir del 3 km, gana un extra de $100 por cada km adicional.

Implemente la clase **Repartidor** y verifique su funcionamiento implementando una **clase Tester**