TEMA: CLASE DERIVADA

NOMBRE DEL ALUMNO: Woolfolk Cerecer Brian

SEMESTRE: 8vo

NOMBRE DE LA MATERIA: Seminario de programación

CLAVE DE LA MATERIA: COM35C1

INTRODUCCIÓN

En el presente documento, se expone un programa que ejemplifica el uso de clases derivadas, gracias al concepto de herencia de la programación orientada a objetos.

La actividad consiste en crear tres clases diferentes: la primera guardará información básica de un empleado, como su nombre y edad únicamente. El propósito de esta clase no es utilizarse para instanciar objetos, sino que servirá como prototipo para crear otras clases derivadas (que en programación orientada a objetos se conoce como clases padre y clases hijos).

La clase de empleado también contiene un método especial llamado `calcular_salario` que de momento simplemente marcar un error en consola, pues es una función que se utilizará en las clases hijos. Dichas clases hijos son para los empleados de tiempo completo, y los empleados por hora.

La primera clase contendrá las propiedades exclusivas de `salario_base` y `bono`, que serán las propiedades que utilice entonces la función para calcular el salario. La segunda clase contendrá las propiedades exclusivas de `salario_por_hora` y `horas_trabajadas`, para poder ser utilizadas nuevamente en la función de calcular el salario.

La función `calcular_salario` también demuestra una característica de las clases hijos, que es la posibilidad de sobrescribir métodos existentes en la clase padre (como será en este caso). Otra forma de relacionar las clases hijos con las clases padres, es a través del constructor y la función `super()`, que nos permite comunicarnos con la clase original para acceder a su constructor e inicializarlo.

Estos conceptos pueden ser muy abstractos si no se tiene experiencia trabajando con la programación orientada a objetos; y es el propósito principal de esta actividad el demostrar su función y utilización en el lenguaje de programación de Python.

Cabe mencionar que, con el fin de reutilizar código previo y buscar la mejora continua del mismo, se utilizarán funciones útiles de actividades anteriores, como las funciones de `leer_string` y `leer_entero`, que nos garantizan la integridad de los datos, tipado y manejo efectivo de errores. Sin embargo, esta actividad requiere trabajar con números flotantes a causa de los salarios de los empleados, por lo cual se creó una función similar a `entero_valido` llamada `flotante_valido`, cuya funcionalidad es completamente igual, a excepción de trabajar con variables de tipo `float` en lugar de `int`.

DESARROLLO

Código

```
# ------------------------------- CLASE EMPLEADO
# QUE GUARDE INFORMACION DE SU NOMBRE Y EDAD, PUES LA INFORMACION
# DEL SALARIO DEPENDE DE LAS SUBCLASES
class Empleado:
    # -------------------------------- METODO INICIALIZADOR
    # MARCAMOS LOS PARAMETROS COMO OPCIONALES PARA CONSIDERAR DEFAULT
    def __init__(
            self,
            nombre: str | None = None,
            edad: int | None = None
    ):
        # DECLARAMOS VALORES POR DEFECTO DEL EMPLEADO,
        # ESTO GARANTIZA LA INTEGRIDAD DE LA INFORMACION
        self._nombre = "Nuevo empleado"
        self._edad = 1

        # EN CASO QUE EXISTAN PARAMETROS, LOS USAREMOS
        # RECORDEMOS QUE LAS FUNCIONES MANEJAN LOS ERRORES
        if nombre is not None:
            self.nombre = nombre

        if edad is not None:
            self.set_edad = edad

    # -------------------------------- METODO PARA MOSTRAR COMO CADENA
    def __str__(self):
        return f"Nombre: {self.nombre} ({self.edad} Anio(s))"

    # -------------------------------- PROPIEDADES (GETTER)
    # SIMPLEMENTE REGRESAN EL VALOR INDICADO
    @property
    def nombre(self):
        return self._nombre

    @property
    def edad(self):
        return self._edad

    # -------------------------------- PROPIEDADES (SETTER)
    # AYUDAN A VERIFICAR LOS TIPOS DE DATOS Y RESTRICCIONES
    @nombre.setter
    def nombre(self, valor: str):
        if string_valido(valor):
            self._nombre = valor

    @edad.setter
```

```
    def edad(self, valor: int):
        if entero_valido(valor, True, True) is not None:
            self._edad = valor

    # ------------------------------- CALCULAR SALARIO (MARCA ERROR)
    def calcular_salario(self):
        print("Esta operacion solo es valida con "
              "empleados de tiempo completo y "
              "empleados por hora!")
        return 0


# ------------------------------- CLASE EMPLEADO TIEMPO COMPLETO
# QUE GUARDE INFORMACION DEL EMPLEADO, PERO AHORA CON SALARIO Y BONOS
class EmpleadoTiempoCompleto(Empleado):
    # ------------------------------- METODO INICIALIZADOR
    # MARCAMOS LOS PARAMETROS COMO OPCIONALES PARA CONSIDERAR DEFAULT
    def __init__(
            self,
            nombre: str | None = None,
            edad: int | None = None,
            salario_base: float | None = None,
            bono: float | None = None
    ):
        super().__init__(nombre, edad)
        # DECLARAMOS VALORES POR DEFECTO DEL EMPLEADO,
        # ESTO GARANTIZA LA INTEGRIDAD DE LA INFORMACION
        self._salario_base = 1.0
        self._bono = 0.0

        # EN CASO QUE EXISTAN PARAMETROS, LOS USAREMOS
        # RECORDEMOS QUE LAS FUNCIONES MANEJAN LOS ERRORES
        if salario_base is not None:
            self.salario_base = salario_base

        if bono is not None:
            self.bono = bono

    # ------------------------------- METODO PARA MOSTRAR COMO CADENA
    def __str__(self):
        return f"Nombre: {self.nombre} ({self.edad} Anio(s)) - Salario:
{self.calcular_salario()}"

    # ------------------------------- PROPIEDADES (GETTER)
    # SIMPLEMENTE REGRESAN EL VALOR INDICADO
    @property
    def nombre(self):
        return self._nombre

    @property
```

```
    def edad(self):
        return self._edad

    @property
    def salario_base(self):
        return self._salario_base

    @property
    def bono(self):
        return self._bono

    # ------------------------------- PROPIEDADES (SETTER)
    # AYUDAN A VERIFICAR LOS TIPOS DE DATOS Y RESTRICCIONES
    @nombre.setter
    def nombre(self, valor: str):
        if string_valido(valor):
            self._nombre = valor

    @edad.setter
    def edad(self, valor: int):
        if entero_valido(valor, True, True) is not None:
            self._edad = valor

    @salario_base.setter
    def salario_base(self, valor: float):
        # EL SALARIO NO PUEDE SER 0
        if flotante_valido(valor, True, True) is not None:
            self._salario_base = valor

    @bono.setter
    def bono(self, valor: float):
        # EL BONO SI PUEDE SER 0
        if flotante_valido(valor, True, False) is not None:
            self._bono = valor

    # ------------------------------- CALCULAR SALARIO
    def calcular_salario(self):
        # SUMAR EL SALARIO BASE CON EL BONO
        return self.salario_base + self.bono


# ------------------------------- CLASE EMPLEADO POR HORA
# QUE GUARDE INFORMACION DEL EMPLEADO, PERO AHORA CON SALARIO/HORA Y
HORAS TRABAJADAS
class EmpleadoPorHora(Empleado):
    # ------------------------------- METODO INICIALIZADOR
    # MARCAMOS LOS PARAMETROS COMO OPCIONALES PARA CONSIDERAR DEFAULT
    def __init__(
            self,
            nombre: str | None = None,
```

```
            edad: int | None = None,
            salario_por_hora: float | None = None,
            horas_trabajadas: float | None = None
    ):
        super().__init__(nombre, edad)
        # DECLARAMOS VALORES POR DEFECTO DEL EMPLEADO,
        # ESTO GARANTIZA LA INTEGRIDAD DE LA INFORMACION
        self._salario_por_hora = 1.0
        self._horas_trabajadas = 0.0

        # EN CASO QUE EXISTAN PARAMETROS, LOS USAREMOS
        # RECORDEMOS QUE LAS FUNCIONES MANEJAN LOS ERRORES
        if salario_por_hora is not None:
            self._salario_por_hora = salario_por_hora

        if horas_trabajadas is not None:
            self._horas_trabajadas = horas_trabajadas

    # -------------------------------- METODO PARA MOSTRAR COMO CADENA
    def __str__(self):
        return f"Nombre: {self.nombre} ({self.edad} Anio(s)) - Salario:
{self.calcular_salario()}"

    # -------------------------------- PROPIEDADES (GETTER)
    # SIMPLEMENTE REGRESAN EL VALOR INDICADO
    @property
    def nombre(self):
        return self._nombre

    @property
    def edad(self):
        return self._edad

    @property
    def salario_por_hora(self):
        return self._salario_por_hora

    @property
    def horas_trabajadas(self):
        return self._horas_trabajadas

    # -------------------------------- PROPIEDADES (SETTER)
    # AYUDAN A VERIFICAR LOS TIPOS DE DATOS Y RESTRICCIONES
    @nombre.setter
    def nombre(self, valor: str):
        if string_valido(valor):
            self._nombre = valor

    @edad.setter
    def edad(self, valor: int):
```

```
        if entero_valido(valor, True, True) is not None:
            self._edad = valor

    @salario_por_hora.setter
    def salario_por_hora(self, valor: float):
        # EL SALARIO NO PUEDE SER 0
        if flotante_valido(valor, True, True) is not None:
            self._salario_por_hora = valor

    @horas_trabajadas.setter
    def horas_trabajadas(self, valor: float):
        # LAS HORAS TRABAJADAS SI PUEDEN SER 0
        if flotante_valido(valor, True, False) is not None:
            self._horas_trabajadas = valor

    # ------------------------------- CALCULAR SALARIO
    def calcular_salario(self):
        # MULTIPLICAR EL SALARIO POR HORA, CON LAS HORAS
        return self.salario_por_hora * self.horas_trabajadas


# ------------------------------- FUNCIONES DE COMPROBACION DE TIPOS
# REVISA SI EL STRING ES VALIDO Y LO REGRESA, SINO REGRESA None
def string_valido(valor) -> str | None:
    if not isinstance(valor, str) or valor == "":
        print("--- Ingrese una cadena de texto valida!\n")
        return None
    return valor  # ESTE VALOR ES VALIDO


# REVISA SI EL ENTERO ES VALIDO Y LO REGRESA, SINO REGRESA None,
# TAMBIEN NOS PERMITE INDICAR SI QUEREMOS QUE SEA POSITIVO O NATURAL
def entero_valido(valor, es_positivo: bool, es_natural: bool) -> int |
None:
    if not isinstance(valor, int):
        print("--- Ingrese un numero entero!\n")
    elif es_positivo and valor < 0:
        print("--- Ingrese un numero entero positivo!\n")
    elif es_natural and valor <= 0:
        print("--- Ingrese un numero entero mayor a 0!\n")
    else:
        return valor  # ESTE VALOR ES VALIDO
    return None  # FALLO LAS VERIFICACIONES


# REVISA SI EL NUMERO ES VALIDO Y LO REGRESA, SINO REGRESA None,
# TAMBIEN NOS PERMITE INDICAR SI QUEREMOS QUE SEA POSITIVO O NATURAL
# EXACTAMENTE IGUAL QUE entero_valido PERO CONSIDERANDO NUMEROS
FLOTANTES
```

```
def flotante_valido(valor, es_positivo: bool, es_natural: bool) ->
float | None:
    if not isinstance(valor, float):
        print("--- Ingrese un numero valido!\n")
    elif es_positivo and valor < 0:
        print("--- Ingrese un numero positivo!\n")
    elif es_natural and valor <= 0:
        print("--- Ingrese un numero mayor a 0!\n")
    else:
        return valor  # ESTE VALOR ES VALIDO
    return None  # FALLO LAS VERIFICACIONES


# ------------------------------- FUNCIONES DE LECTURA RAPIDA
# FUNCION PARA OBTENER UNA CADENA DE TEXTO VALIDA 100%
def leer_string(mensaje: str) -> str:
    # CREAMOS UN CICLO 'INFINITO' PARA INSISTIR CON LA CAPTURA
    while True:
        try:
            # SOLICITAMOS EL VALOR Y LO COMPROBAMOS
            valor = string_valido(input(mensaje))

            # SI LA COMPROBACION FUNCIONA, REGRESAMOS EL VALOR
            if valor is not None:
                return valor  # CIERRA EL CICLO CON UN VALOR CORRECTO

            # NO ES NECESARIO MOSTRAR ERRORES GRACIAS A string_valido
            # continue
        except ValueError:
            # 'ATRAPAMOS' CUALQUIER ERROR INESPERADO Y REINICIAMOS EL
CICLO
            print("--- Ingrese una cadena de texto valida!\n")


# FUNCION PARA OBTENER UN NUMERO ENTERO VALIDO 100%,
# PUDIENDO DECIR SI QUEREMOS QUE SEA POSITIVO O NO
def leer_entero(mensaje: str, es_positivo: bool, es_natural: bool) ->
int:
    # CREAMOS UN CICLO 'INFINITO' PARA INSISTIR CON LA CAPTURA
    while True:
        try:
            # SOLICITAMOS EL VALOR, LO CONVERTIMOS A int Y LO
COMPROBAMOS
            valor = entero_valido(int(input(mensaje)), es_positivo,
es_natural)

            # SI LA COMPROBACION FUNCIONA, REGRESAMOS EL VALOR
            if valor is not None:
                return valor  # CIERRA EL CICLO CON UN VALOR CORRECTO
```

```
            # NO ES NECESARIO MOSTRAR ERRORES GRACIAS A entero_valido
            # continue
        except ValueError:
            # 'ATRAPAMOS' CUALQUIER ERROR INESPERADO Y REINICIAMOS EL
CICLO
            print("--- Ingrese un numero valido!\n")


# FUNCION PARA OBTENER UN NUMERO VALIDO 100%,
# PUDIENDO DECIR SI QUEREMOS QUE SEA POSITIVO O NO
def leer_flotante(mensaje: str, es_positivo: bool, es_natural: bool) ->
float:
    # CREAMOS UN CICLO 'INFINITO' PARA INSISTIR CON LA CAPTURA
    while True:
        try:
            # SOLICITAMOS EL VALOR, LO CONVERTIMOS A float Y LO
COMPROBAMOS
            valor = flotante_valido(float(input(mensaje)), es_positivo,
es_natural)

            # SI LA COMPROBACION FUNCIONA, REGRESAMOS EL VALOR
            if valor is not None:
                return valor  # CIERRA EL CICLO CON UN VALOR CORRECTO

            # NO ES NECESARIO MOSTRAR ERRORES GRACIAS A entero_valido
            # continue
        except ValueError:
            # 'ATRAPAMOS' CUALQUIER ERROR INESPERADO Y REINICIAMOS EL
CICLO
            print("--- Ingrese un numero valido!\n")


# ------------------------------- COMENZAR EL PROGRAMA
print("=== ACTIVIDAD 6 - CLASE DERIVADA ===\n")

# CREAR UN EMPLEADO DE TIEMPO COMPLETO, SOLICITAREMOS AL USUARIO LOS
CAMPOS
print("Nuevo empleado de tiempo completo: ")
empleado_completo = EmpleadoTiempoCompleto(
    nombre=leer_string("> Nombre del empleado: "),
    edad=leer_entero("> Edad del empleado: ", True, True),
    salario_base=leer_flotante("> Salario base del empleado: ", True,
True),
    bono=leer_flotante("> Bono del empleado: ", True, False)
)

# CREAR OTRO EMPLEADO PERO POR HORAS, SOLICITANDO AL USUARIO LOS CAMPOS
print("\nNuevo empleado por horas: ")
empleado_horas = EmpleadoPorHora(
    nombre=leer_string("> Nombre del empleado: "),
```

```
    edad=leer_entero("> Edad del empleado: ", True, True),
    salario_por_hora=leer_flotante("> Salario por hora del empleado: ",
True, True),
    horas_trabajadas=leer_flotante("> Horas trabajadas del empleado: ",
True, False)
)

# MOSTRAR LA INFORMACIÓN CAPTURADA
print("\n============")
print("Salario de los empleados: ")
print(f"El salario de {empleado_completo.nombre} es
${empleado_completo.calcular_salario():.2f}")
print(f"El salario de {empleado_horas.nombre} es
${empleado_horas.calcular_salario():.2f}")
```

Visualización de los resultados del código

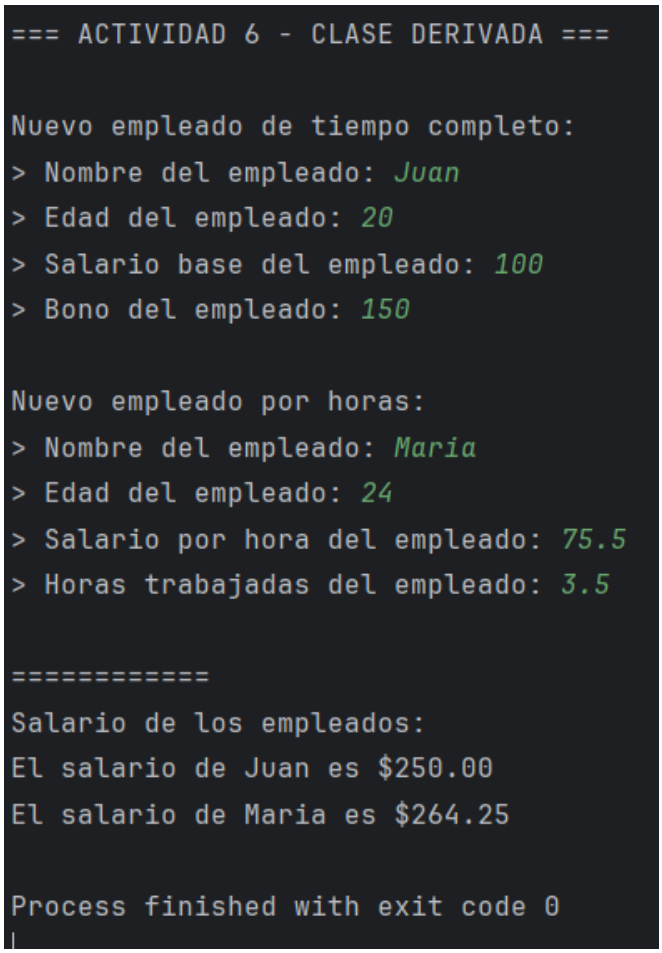

Imagen 1. Inicio del programa, se solicitan los datos al usuario sobre el empleado de tiempo completo y del empleado por horas. Después de la captura, se imprime el salario de los empleados.

CONCLUSIÓN

Nuevamente, esta actividad se trata de una aplicación relativamente sencilla y simple, sin mayor grado de complejidad que requiera de estructuras de datos complejas, menú de opciones u otras funcionalidades.

Es fácil darse cuenta de lo útiles que pueden ser las clases heredadas, tanto desde la perspectiva de las clases padres, como de las clases hijos, ya que podemos comenzar a crear clases a modo de prototipos, que contengan propiedades y métodos globales o generales que puedan ser utilizados bajo otro contexto más adelante, nuevamente utilizando el concepto de herencia de la programación orientada a objetos.

Esta actividad de nueva cuenta refuerza el uso de una "biblioteca de funciones" personales, que nos ayuda a seguir implementando métodos generales para lectura y comprobación de valores, sin tener que detenernos a generarlas de nuevo.

Así como la actividad pasada de Encapsulamiento, existen múltiples mejoras que se pueden realizar en el código, pero todas ellas resultan innecesarias para demostrar el uso de la herencia. Sin embargo, esta actividad ya permite crear otras funciones más complejas en futuras y previas actividades, como es el caso de la Actividad 5 con el sistema de gestión de los libros de una biblioteca. Utilizando la herencia, podemos crear diferentes "tipos" de libros, como revistas, novelas, artículos científicos, entre otros.

Finalmente, el uso de la herencia depende grandemente en el problema que queramos atacar, pues es una herramienta que considero como "a largo plazo", ya que en situaciones reducidas y controladas es más rápido generar clases separadas, que heredadas; pero en sistemas más grandes (también considerado una regla en programas que utilicen bases de datos) permite el manejo de objetos con características similares, organizando los datos y preservando ante todo, la integridad de la información, sin la necesidad de reescribir código en cada ocasión.

REFERENCIAS BIBLIOGRÁFICAS

Python Software Foundation. (s.f.). *Classes*. Python documentation. https://docs.python.org/3/tutorial/classes.html