Seminario 2: C#

Integrantes

1. Joel David Hernández Cruz
2. Leandro González Montesino
3. Wendy Díaz Ramírez

Tabla de Contenido

Ejercicio:

Para este seminario debemos proponer e implementar un mecanismo de contratos al estilo de `CodeContract` en el lenguaje ***Python***. Con nuestra implementación deberíamos ser capaces de hacer algo como lo siguiente:

```
@Contract(require= lambda x: x > 0, ensure= lambda result: result > 0)
def someFunction(x):
    pass
```

Que es el diseño por Contratos?

La verificación automática de programas ha sido un viejo anhelo de la Ciencia de la Computación, ya desde hace tiempo en lenguajes como `CLU` aparecieron las **aserciones**, que no son mas que mecanismos para especificar requerimientos y garantías sobre nuestro código. No fue hasta la propuesta de ***Bertrand Meyer*** de **Diseño por Contratos *(Design by Contracts)*** en su lenguaje `Eiffel` que el uso de las **aserciones** se integró verdaderamente a un lenguaje de programación.

La idea de ***Meyer***, aunque genial, se basa en una metáfora muy simple. Entre el que diseña una clase y el cliente de la misma se establece un "contrato" que define los deberes y derechos de ambos. Estos deberes y derechos se van a representar, en la terminología de `Eiffel`, en forma de **pre-condiciones**, **pos-condiciones** e **invariantes**, tal y como se muestra en la siguiente tabla.

	Deberes u obligaciones	Derechos o beneficios
Cliente de la clase	**Pre-condiciones**	**Pos-condiciones e invariantes**
Diseñador de la clase	**Pos-condiciones e invariantes**	**Pre-condiciones**

- ***Pre-condición***: Es una cláusula lógica que **debe estar libre de efectos colaterales**, y que debe **cumplirse como requisito para ejecutar el método al que está asociada**. Un uso frecuente de las **pre-condiciones** es validar los parámetros de entrada de los métodos (algo que no puede satisfacerse solo con la declaración y tipado estático de los parámetros).
- ***Pos-condición***: Es una cláusula lógica que **debe cumplirse luego de la ejecución del método**, y funge como garantía de lo que éste hace. Por tanto, las **pos-condiciones son evaluadas justo después de ejecutado el método** y **antes de retornar al código que llama**.
- ***Invariante***: Es una cláusula que **establece las condiciones bajo las cuales el estado de un objeto es "correcto"**. Por tanto, los invariantes **se validan luego de creados los objetos** y **después de ejecutar cualquier método**.

Para terminar este pequeña ***intro*** al **Diseño por Contratos**, un ejemplo de un código sencillo en `Eiffel` que, aunque quizás no se encuentre muy familiarizado con la sintaxis, es fácil captar la esencia del ejemplo.

```
indexing
    description: "Clase Monedero"
class
    MONEDERO
create
    Crea
feature – Se exportan todos
    Saldo: DOUBLE;
    Depositar(cant: DOUBLE) is
        require
            cant >= 0;
        do
            Saldo := Saldo + cant;
        ensure
            Saldo = old Saldo + cant;
        end
    Extraer(cant: DOUBLE) is
        require
            cant >= 0;  cant <= Saldo;
        do
            Saldo := Saldo - cant;
        ensure
            Saldo = old Saldo - cant;
        end
    Crea(saldoInicial: DOUBLE) is
        require
            saldoInicial >= 0;
        do
            Saldo := saldoInicial;
        ensure
            Saldo = saldoInicial;
        end
invariant
    Saldo >= 0;
end
```

Para mas información sobre el Patrón de Diseño por Contratos le compartimos los siguientes enlaces:

Diseño por Contratos: **Aunque principalmente recomendamos el Libro del profesor Miguel Katrib y Jorge Luis de Armas: Diseño por Contratos en .NET 4.0**

- Wikipedia
- es.qwe.wiki
- Koalite

Decoradores en Python

Primero debemos aclarar algunas características del lenguaje `Python` y como funcionan sus funciones, que les serán útiles para entender decoradores.

Funciones como Objetos de Primera Clase:

Esto significa que las funciones en `Python` son objetos y como cualquier otro objeto puede ser pasadas y usadas como argumentos:

```
def say_hello(name):
    return f"Hello {name}"

def be_awesome(name):
    return f"Yo {name}, together we are the awesomest!"

def greet_bob(greeter_func):
    return greeter_func("Bob")
```

En este ejemplo tenemos 3 funciones `say_hello()` y `be_awesome()` son funciones normales que esperan un parámetro `name` de tipo *string*. La función `gree_bob()` sin embargo espera una función como su argumento. Es fácil saber cual sera la salida para el siguiente ejemplo:

```
>>> greet_bob(say_hello)
'Hello Bob'

>>> greet_bob(be_awesome)
'Yo Bob, together we are the awesomest!'
```

Importante notar aquí como las funciones `say_hello()` y `be_awesome()` son llamadas sin los paréntesis, esto significa que solo una referencia a la función es pasada como parámetro.

Funciones anidadas

Es posible definir funciones dentro de otras funciones:

```
def parent():
    print("Printing from the parent() function")

    def first_child():
        print("Printing from the first_child() function")

    def second_child():
        print("Printing from the second_child() function")

    second_child()
    first_child()
```

La salida de llamar a la función `parent()` es justo lo que se espera.

```
>>> parent()
'Printing from the parent() function'
'Printing from the second_child() function'
'Printing from the first_child() function'
```

Funciones que retornan funciones

`Python` también permite usar funciones como valor de retorno. Usando el ejemplo anterior:

```
def parent(num):
    def first_child():
        return "Hi, I am Joel"

    def second_child():
        return "Call me David"

    if num == 1:
        return first_child
    else:
        return second_child
```

Notar que estamos retornado una referencia a las funciones, así que la siguiente salida es lo esperado.

```
>>> first = parent(1)
>>> second = parent(2)

>>> first
<function parent.<locals>.first_child at 0x7f599f1e2e18>

>>> second
<function parent.<locals>.second_child at 0x7f599dad5268>
```

Decoradores Simples

Ahora que ya hemos visto algunas características de las funciones en `Python` veamos que son los decoradores.

my_decorator

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper
```

```
def say_whee():
    print("Whee!")

say_whee = my_decorator(say_whee)
```

Esto no es nada nuevo, es simplemente lo que ya se vio en ejemplos anteriores, así que es fácil saber cual sera la salida de llamar a `say_whee()`.

```
>>> say_whee()
'Something is happening before the function is called.'
'Whee!'
'Something is happening after the function is called.'
```

En este ejemplo el decorador ocurre en la linea `say_whee = my_decorator(say_whee)`, la variable `say_whee` ahora apunta al `wrapper` que se define dentro de `my_decorator`. Pero `wrapper` tiene una referencia a la función original `say_whee` y es la que ejecuta entre los dos `prints`.

En simples términos: ***decorators wrap a function, modifying its behavior.***

Un decorador envuelve una función, agregándole o modificando su funcionalidad.

`Python` te permite un azúcar sintáctico para usar decoradores en el código de una manera mas fácil y legible. **Usando el símbolo @**, el anterior código quedaría entonces:

```
@my_decorator
def say_whee():
    print("Whee!")

# Output: say_whee()
'Something is happening before the function is called.'
'Whee!'
'Something is happening after the function is called.'
```

Que pasaría entonces si quisiéramos decorar una función que reciba un argumento, usando my_decorator

```
@my_decorator
def greet(name):
    print(f"Hello {name}")

# Output
>>> greet("World")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wrapper() takes 0 positional arguments but 1 was given
```

Recordemos que nuestro decorador my_decorator retorna una referencia a `wrapper` y esta es una función que no recibe ningún argumento. Esto se podría arreglar simplemente agregándole un parámetro a `wrapper` pero entonces la primera función `say_whee` dejaría de funcionar, ademas funciones de más de 1 parámetro seguirían fallando. La solución para este caso es usar *args y **kwargs en la función `wrapper` así esta aceptaría un numero arbitrario de parámetros.

Esta azúcar sintáctica que permite `Python` se traduce a lo siguiente.

```
@my_decorator
def func():
    print("func")

@my_decorator
def func_with_args(a, b, c)
    print(f"func_with_args: {a}, {b}, {c}")

# 1. func() => my_decorator(func)()
# 2. func_with_args(1, 2, 3) => my_decorator(func)(1, 2, 3)
```

Por supuesto las declaraciones para decorar una función pueden ser 1 o muchas, veamos un ejemplo de esto:

```
@my_decorator
@my_decorator
def func_with_args(a, b, c)
    pass

# Output
# En este caso al llamar a func_with_args(1, 2, 3) lo que se imprime es

# Something is happening before the function is called.
# Something is happening before the function is called.
# func_with_args: 1, 2, 3
# Something is happening after the function is called.
# Something is happening after the function is called.
```

Decorando Clases

Además de decorar funciones también podemos decorar clases. Las clases pueden ser decoradas de dos maneras. Primero por supuesto puede simplemente decorar los métodos de la clase que quieras, en `Python` los mas comunes que incluso vienen `built-in` son `@classmethod`, `@staticmethod`, `@property`. Aquí les dejamos un ejemplo de uso de estos métodos y documentación sobre ellos.

Más documentación sobre los métodos **built-in**.

- Info sobre @classmethod y @staticmethod
- Info sobre @property

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """Get value of radius"""
        return self._radius

    @radius.setter
    def radius(self, value):
        """Set radius, raise error if negative"""
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be positive")

    @property
    def area(self):
        """Calculate area inside circle"""
        return self.pi() * self.radius**2

    def cylinder_volume(self, height):
        """Calculate volume of cylinder with circle as base"""
        return self.area * height

    @classmethod
    def unit_circle(cls):
```

```
        """Factory method creating a circle with radius 1"""
        return cls(1)

    @staticmethod
    def pi():
        """Value of π, could use math.pi instead though"""
        return 3.1415926535
```

En esta clase:

- `.cylinder_volume()` es un método regular de la clase.
- `.radius` es una propiedad mutable. Como se aprecia en el ejemplo se le define un `setter`, por lo cual su valor puede ser alterado. **Es una propiedad** así que debe ser accedida sin paréntesis.
- `.area` es una propiedad inmutable, al contrario que le ejemplo anterior no tiene un `setter` definido, así que su valor no puede ser cambiado.
- `.unit_circle()` es un método de clase. No esta ligado a una instancia en particular de `Circle`. Estos métodos generalmente son usados como métodos de *fabrica*, que pueden crear instancias especificas de una clase.
- `.pi()` un método estático. No depende de la clase `Circle`, excepto por el hecho de estar en su ***namespace***. Estos pueden ser llamados desde una instancia o desde la clase.

La otra forma de decorar clases, es decorando la clase completa.

```
from decorators import my_decorator

@my_decorator
class Circle:
    pass
```

Decorar una clase, no es muy distinto que decorar una función. Por supuesto la diferencia esta en que el decorador en lugar de recibir una función como argumento, recibe una clase. El decorador que vimos anteriormente my_decorator funcionara igual si se le pasa una clase, lo que es probable que no haga lo que teníamos pensado cuando lo hicimos. Para mas información sobre decoradores les compartimos los siguientes enlaces.

Documentación sobre **Python Decorators**.

- Python-Course Site
- DataCamp Site
- GeeksforGeeks
- TowardsDataScience

Implementación

Librerías usadas:

- inspect
  - signature
  - getfullargspec
- itertools.zip_longest

En la orientación del seminario solo especifican implementar los métodos de pre-condiciones ***(require)*** y pos-condiciones ***(ensure)***. Estos se encuentran en el decorador ***contract***:

```
def contract(require, ensure):

    try:
        if not callable(require):
            raise Exception("require statement must be callable!!")
        if not callable(ensure):
            raise Exception("ensure statement must be callable!!")
    except NameError:
        pass

    req_sig = inspect.signature(require)
    ens_sig = inspect.signature(ensure)

    def func_wrapper(func):
        def args_wrapper(*args, **kwargs):
            req_params = []
            ens_params = []

            # names and values as a dictionary:
            instance_dict = {}
            args_name = inspect.getfullargspec(func)[0]
            args_dict = dict(itertools.zip_longest(args_name, args))

            for attribute, value in args[0].__dict__.items():
                instance_dict[attribute] = value

            for param in req_sig.parameters:
                if str(param) in args_dict.keys():
                    req_params.append(args_dict[str(param)])
                elif str(param) in instance_dict.keys():
                    req_params.append(instance_dict[str(param)])
                else:
                    raise Exception("pre-condition params not defined in this
scorpe")

            for param in ens_sig.parameters:
                if str(param) in args_dict.keys():
                    ens_params.append(args_dict[str(param)])
                elif str(param) in instance_dict.keys():
                    ens_params.append(instance_dict[str(param)])
                else:
                    raise Exception("pos-condition params not defined in this
scope")

            try:
                if require(*req_params):
                    value = func(*args, **kwargs)

                    try:
                        if ensure(*ens_params):
                            return value
                        else:
                            raise Exception()
                    except Exception as e:
```

```
                        exc_type, exc_obj, exc_tb = sys.exc_info()
                        fname =
os.path.split(exc_tb.tb_frame.f_code.co_filename)[1]
                        print(exc_type, fname, exc_tb.tb_lineno)
                        print("pos-conditions fails")
                else:
                    raise Exception()
            except Exception as e:
                exc_type, exc_obj, exc_tb = sys.exc_info()
                fname = os.path.split(exc_tb.tb_frame.f_code.co_filename)[1]
                print(exc_type, fname, exc_tb.tb_lineno)
                print("pre-conditions fails")
        return args_wrapper
    return func_wrapper
```

Primero nos aseguramos que lo que se le paso al decorador son funciones u objetos invocables. Luego haciendo uso de las librerías antes mencionadas, guardamos el nombre de los argumentos que se usan en las funciones ***require*** y ***ensure***, para luego buscarlas en el contexto actual esos valores (**Contexto actual dígase, la instancia de la clase de donde se llamo al método decorado, y los argumentos que se pasaron a este método**).

De las características de diseño por contratos visto anteriormente se implementaron también las invariantes. Estas se hicieron con un decorador a nivel de clase, ya que necesitábamos que la invariante se revisara después del llamado a cualquier método, y además justo después de inicializarse la instancia.

```
def contract_invariant(inva):
    def invariant(Cls):
        class NewCls(object):
            def __init__(self, *args, **kwargs):
                self.oInstance = Cls(*args, **kwargs)

                check_invariant(self, inva)

            def __getattribute__(self, s):
                try:
                    if not callable(inva):
                        raise Exception("invariant statement must be
callable!!")
                except NameError:
                    pass

                try:
                    x = super(NewCls, self).__getattribute__(s)
                except AttributeError:
                    pass
                else:
                    return x
                x = self.oInstance.__getattribute__(s)
                if type(x) == type(self.__init__):  # it is an instance method
                    check_invariant(self, inva)

                    return x
                else:
                    return x
```

```
        return NewCls
    return invariant
```

Ejemplo:

```
@contract_invariant(inva=lambda balance: balance > 0)
class Wallet:

    def __init__(self, balance):
        self.balance = balance

    @contract(require=lambda amount: amount >= 0, ensure=lambda balance: balance
>= 0)
    def deposit(self, amount):
        self.balance += amount

    @contract(require=lambda amount, balance: amount >= 0 and amount <= balance,
 ensure=lambda balance: balance >= 0)
    def withdraw(self, amount):
        self.balance -= amount

    @property
    def check_balance(self):
        print(f"You're current balance is: {self.balance}")
```

Por implementar:

En nuestra solución no se encuentra implementado el ***feature*** de Diseño por Contratos donde se puede tener acceso al valor de una variable antes de la ejecución del método, analicemos primero que para implementar este ***feature*** debemos salvar todas las variables en el ***scope (Dígase en sus parámetros definidos y en la instancia de la cual se llama)*** del método antes de su ejecución, ya que no sabemos cual se usará en el cuerpo de la expresión ***lambda***, esto no es mucho problema, el problema esta en que estas variables nuevas no podemos pasarlas como parámetro a la expresión ***lambda***, así que debemos añadirlas al ***scope*** global para que puedan ser usada y reescribirlas para los próximos usos, además se debe seguir algún tipo de convención de nombres a la hora de definir el contrato, algo como llamar a las variables con un prefijo `old_`, así si se tiene una variable `balance` entonces para referirnos al valor antes de la ejecución del método debería ser como `old_balance`.

Esto aunque quizás no sea muy complicado, creemos que sale fuera de los objetivos de este seminario.