E-JUST
الجامعة المصرية اليابانية للعلوم والتكنولوجيا
エジプト日本科学 技術大学

CSE 323 – Software Engineering

Lab 3 -Introduction to Refactoring

Objectives

The students will be introduced to the concept of code refactoring with application in python.

Task 1: Refactor the code below for better readability

```
def polygon_area(n_sides, side_len):
    """Find the area of a regular polygon

    :param n_sides: number of sides
    :param side_len: length of polygon sides
    :return: area of polygon

    >>> round(polygon_area(4, 5))
    25
    """
    perimeter = n_sides * side_len

    apothem_denominator = 2 * math.tan(math.pi / n_side
    apothem = side_len / apothem_denominator

    return perimeter * apothem / 2
```

Tips:

- Move the logic for calculating the perimeter into polygon_perimeter function.
- Complete the definition of polygon_apothem function, by moving the logic seen in the context
- Utilize the new unit functions to complete the definition of polygon_area.

Task 2:

Study the supplied .py files and find some code in the Pizza class and pizzashop file that needs improvement, and perform refactoring to achieve the following

1. Replace magic numbers and strings with named constants.
2. Use consistent naming - rename symbols.
3. Make code more reusable by moving misplaced code to a better place
4. Replace "switch" (if ... elif ... elif) with object behavior.

Background

Pizza describes a pizza with a size and optional toppings. The price depends on size and number of toppings. For example:

```
pizza = Pizza('large')
pizza.addTopping("mushroom")
pizza.addtopping("pineapple")
print("The price is", pizza.getPrice())
```

There are 2 files to start with:

pizza.py - code for Pizza class
pizzashop.py - create some pizzas and print them.

1. Replace strings with Named Constants

In the Pizza class replace 'small', 'medium', and 'large" with named constants.

1. When you are done, the strings 'small', 'medium', 'large' should only appear **once** in the code.
2. Run the code. Verify the results are the same.

2. Rename Symbols in all files

1. getPrice is not a Python-style name. Use refactoring to rename it to get_price.
2. Rename addTopping to add_topping.
3. Run the code. Verify the code works the same.

3. Extract Method

print_pizza creates a string (description) to describe the pizza. That is poor location for this because:

- same description could be needed elsewhere
- it relies on info about a Pizza that only the pizza has

So, it should be the Pizza's job to describe itself. This is also known as the *Information Expert* principle.

1. Move the code that creates string description (but not the price) into Pizza class.
2. What method should use you? How about __str__?
3. In print_pizza, invoke the method you just created, e.g. str(pizza).
4. Run the pizzashop code. Verify the results are the same.

4. Replace 'switch' with Call to Object Method

The get_price method has a block like this:

```
if self.size == Pizza.SMALL:
   price = ...
elif self.size == Pizza.MEDIUM:
   price = ...
elif self.size == Pizza.LARGE:
   price = ...
```

The pizza has to know pricing rules for each size, which makes the code complex. An O-O approach would be to let the pizza sizes compute their own price. Therefore, define a new datatype for pizza size with a price() method.

Python has an Enum type for this:

```
from enum import Enum

class PizzaSize(Enum):
   # write the sizes and their values
   # one per line
   small = 120
   medium = 200
   large = 280

   def __str__(self):
      return self.name
```

Does this work? Write some short code to try it.

```
def test_pizza_sizes():
   for size in PizzaSize:
      print(size, "pizza price:",size.value)

if __name__ == "__main__":
   test_pizza_sizes()
```

It should price the pizza prices. So we *could* define:

```
   def price(self):
      return self.value    # value of the enum member
```

But what about the price of toppings?

We only need the number of toppings to compute pizza price, so add a parameter to price(). This avoids *coupling* it to the toppings.

```
  def price(self,ntoppings=0):
    return self.value + ???*ntoppings
```

The per-topping price depends on size, so we need separate topping prices for each size.

Here are 2 solutions. They both use the fact that the **value** of an enum member can be **anything**, not just a number or string.

1. Use a dict to specify base-price and topping price:

2. Use a lambda to compute price.

Modify pizzaSize.price(ntopping) to return the pizza price with toppings.
Use whichever solution you find **most easy to read**.

Test the code.

Then modify the Pizza class. Change size to use the PizzaSize enum and **delegate** pricing to it.

```
# in Pizza class
  def get_price(self):
    return self.size.price( len(toppings) )

# In the pizzashop file:
pizza = Pizza( PizzaSize.small )
etc.
```

No "if size=... elif size=..."!

Test the code by running pizzashop. Verify results are same as before.

Bonus:

What if the price for each topping is different? Perform the following refactoring:

1. *Pass whole object instead of values* - instead of calling size.price(len(toppings)), use size.price(toppings).
2. *Delegate to a Strategy* - pricing varies but sizes rarely change, so define a separate class to compute price. (Design principle: "*Separate the parts that vary from the parts that stay the same*")

https://github.com/greatersum/refactoring-exercise-python