Lab 5: Design Patterns

Factory Method Pattern

Read about the Factory Method Pattern:

- Factory Method on Refactoring Guru
- Factory Method on Wikipedia - includes example code in Python

Github Assignment: https://classroom.github.com/a/0CD2z89z

Problem 1: Answer the questions in the README file of the starter code

Write your answers in README.md after you read about the Factory Method pattern.

Problem 2: Implement & Improve the MalayMoneyFactory

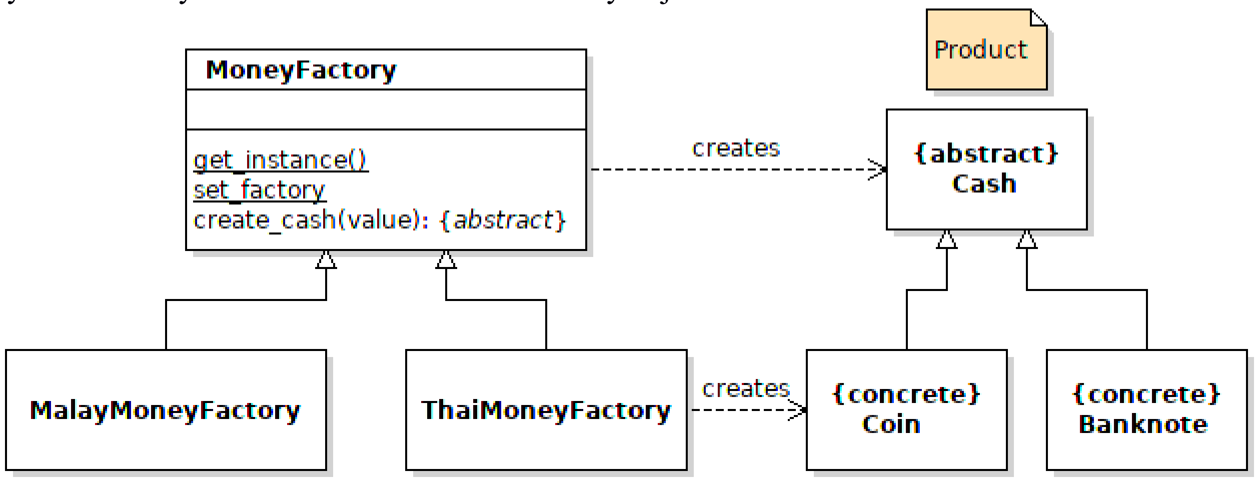
2.1 Clone the assignment code from Github
2.2 Copy your cash.py and money.py from the Wallet assignment into this project.
Modify Banknote to allow any multiple of 1 (no decimal values). The MoneyFactory will be responsible for validating the values.
2.3 Improve the code for create_cash in MalayMoneyFactory. It violates the guideline "*No magic numbers in code*". Define those arrays of "magic numbers" in the constructor.

Malaysian money has coins for 1, 5, 10, 20, and 50 *sen* which equals 1/100th of a *Ringgit*. That is, the coins have values 0.01, 0.05, 0.1, 0.2, and 0.5 *Ringgit*. Since 2012, Malaysian banknotes with values 1, 5, 10, 50, and 100 *Ringgit*. Those are the only allowed values.

Problem 3: Create a ThaiMoneyFactory

3.1 In money_factory.py create a class for Thai Money. It should create only valid Thai currency.

3.2 In main.py, call the "set_factory" method of MoneyFactory to make ThaiMoneyFactory be the concrete factory. Then verify it creates the correct Thai money objects.



Problem 4: Decorator to Show Function Calls & Returns (*The Decorator Pattern*)

Write a decorator to show each time a function is called. We started this in class but your decorator should do more. Name this function **`tracer`** in file **`decorators.py`**. Include it in the `patterns-lab` repo.

Example 1: apply decorator by "wrapping" an existing function

```
>>> from decorators import tracer
>>> mymax = tracer(max)
>>> mymax(3, 8, 4)
max(3, 8, 4)              <---  tracer prints this before calling the wrapped function (max)
8                         <---  print this after calling the wrapped function
8                         <---  value returned by wrapped function (max)
```

Example 2: apply decorator using @-notation.

```
from decorators import tracer

@tracer
def factorial(n: int):
     # recursively compute factorials
     if n <= 1: return 1
     return n*factorial(n-1)

>>> print("3! is ", factorial(3))
factorial(3)             <--- blue lines are printed by tracer
factorial(2)
factorial(1)
1
2
6
3! is 6
```

For recursive function calls, it would be easier to see what's happening if the tracer output is indented 2 spaces for each level of recursion. Can you figure out a way to do that? Section 2.8 of *Composing Programs* shows you how to store a value as part of the decorator (the **`count`** decorator).

```
>>> print("3! is ", factorial(3))
factorial(3)             <--- blue lines are printed by tracer
  factorial(2)           <--- indent the output for recursive calls
    factorial(1)
    1
  2
6
3! is 6
```

References

Section 2.8 of *Composing Programs* (Efficiency)
Python Decorators on `pythontutorial.net`

Singleton Pattern

In `MoneyFactory.get_instance` there is code like:

```
def get_instance(cls):
    if not cls.instance:
        # create the single factory instance
        cls.instance = (some concrete MoneyFactory)
    # always return the same instance
    return cls.instance
```

The purpose of this is so that there is only one instance of `MoneyFactory` in the application. Many applications involve a shared resource, such as a database connection or network connection, that you want to access from different parts of the application. You don't want the parts to each create their own instance of this resource (e.g., it's expensive or needs to be synchronized across all uses).

The Singleton Pattern is a way to ensure that only one instance of a class is created. The get_instance method (along with a constructor that is either private or raises an exception) is the typical way of doing this.

You can verify it using a Python interactive interpreter:

```
>>> factory1 = MoneyFactory.get_instance()
>>> factory2 = MoneyFactory.get_instance()
>>> factory1 is factory2
True
```

Python has a better way of defining Singleton classes using a **metaclass**. However, it only works for Python and requires knowing how to use a metaclass.