Série TP n°7 Chapitre 3: Design Patterns	Module	MTI Méthd. et tech d'implement.	
	Filière	Master GSI	1ère Année

Modèle design patterns

Sans Design pattern.

1- exécuter le code suivant

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#http://python-3-patterns-idioms-test.readthedocs.io/en/latest/Factory.html
# A simple static factory method.
import sys
class Circle(Shape):
    def draw(self): print("Circle.draw")
    def erase(self): print("Circle.erase")

class Square(Shape):
    def draw(self): print("Square.draw")
    def erase(self): print("Square.erase")
if __name__ == "__main__":
  for type in ("Circle", "Square" ,"Circle", "Square"):
    if type == "Circle":
        shape = Circle()
    elif type == "Square":
        shape = Square()
    else:
        print "Bad shape creation: " + type
        sys.exit()
    shape.draw()
    shape.erase()
```

2- Modifier le code pour ajouter une forme triangle et rectangle.

Simple Factory method

3- on veut encapsuler l'operation d'instanciation des objets, en utilsant une classe ShapeFactory

```

class ShapeFactory:
    @staticmethod
    def createShape(type):
        if   type == "Circle":  return Circle()
        elif type == "Square":  return Square()
        else:
            print "Bad shape creation: " + type
            sys.exit()
if __name__ == "__main__":
  for type in ("Circle", "Square" ,"Circle", "Square"):
    shape = ShapeFactory.createShape(type)
    shape.draw()
    shape.erase()
```

2- Modifier le code pour ajouter les formes Triangle et rectangle.

Factory method

3- On veut créer une Factory (ShapeFactory_SCT) spécialisée qui ne fabrique que les carrés, les cercles, et les triangles. Une autre Factory (ShapeFactory_SCR) qui fabrique que les carrés et les cercles et les rectangles.

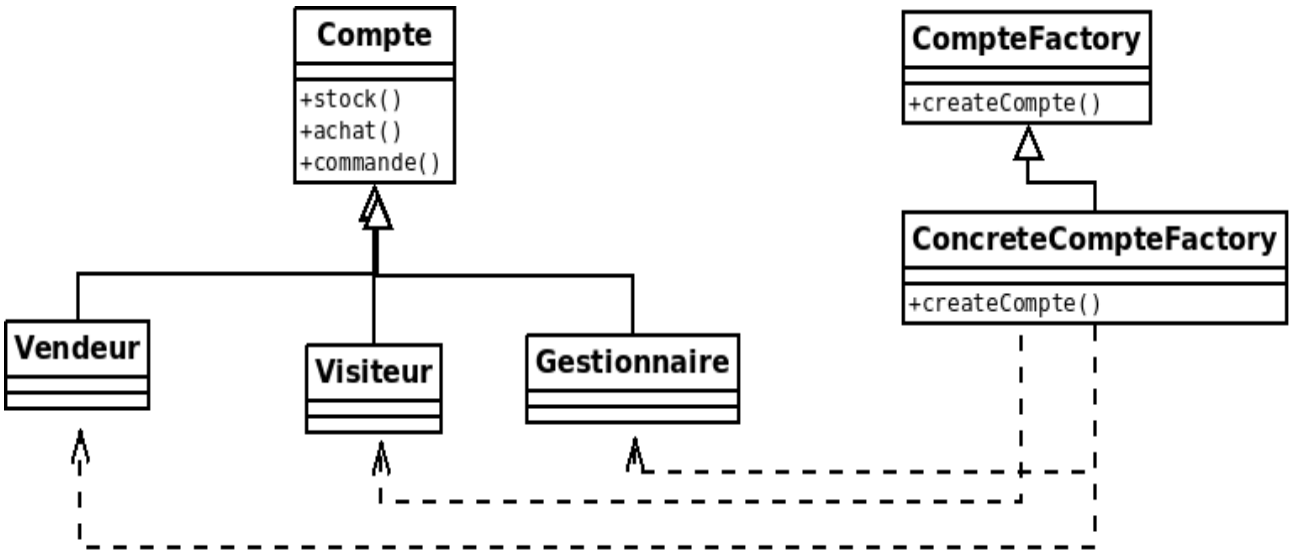
Application

On veut une application de gestion de stock, notre client exige d'avoir trois types de comptes :

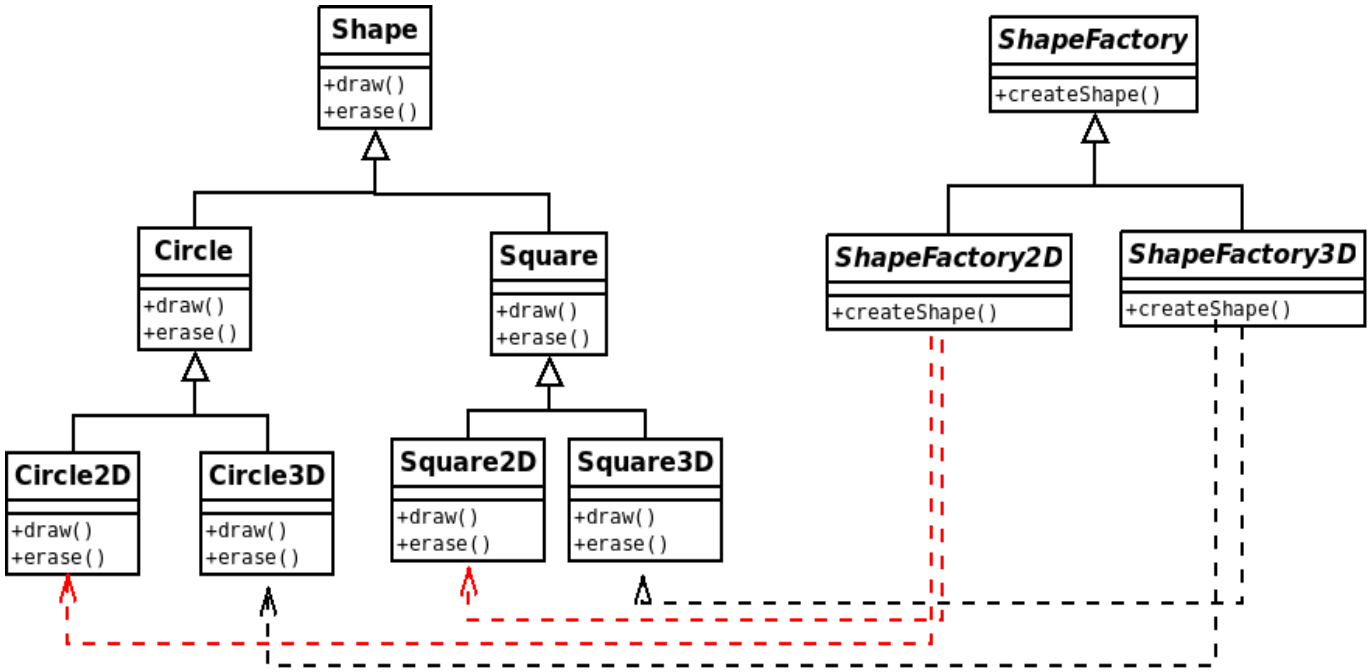
- un compte gestionnaire, pour gérer le stock, faire des achats et des commandes, voir la recette.
- un compte vendeur, pour permettre aux vendeurs de vendre, faire des achats et des commandes.
- Un compte visiteur pour permettre aux clients de consulter le stock, faire des achats et des commandes.

Proposer un design pattern, en inspirant du diagramme de classe.

Implémenter les classes de base et la classe de création.



Partie 2
Abstract Factory
On reprends les classes de la première partie.
On veut Créer des formes 2D et des formes 3D, selon le diagramme suivant.
1- créer les sous classes square2D, Square3D, circle2D, circle3D.
2- Créer l'Usine (Factory) abstraite, ShapeFactory
3- Créer les usines concrètes ShapeFactory2D, et ShapeFactory3D



Le code suivant illustre juste des exemples.

```
class ShapeFactory:
    @staticmethod
    def createShape(type):
        pass

class ShapeFactory2D(ShapeFactory):
    @staticmethod
    def createShape(type):
        if   type == "Circle":  return Circle2D()
        elif type == "Square":  return Square2D()
        else:  return None

#Exemple
class Circle2D(Circle):
    def draw(self) : print("Circle.draw2D")
    def erase(self): print("Circle.erase2D")


# creation des objets 2 D
for type in ("Circle", "Square" ,"Circle", "Square"):
    shape = ShapeFactory2D.createShape(type)
    shape.draw()
    shape.erase()
```

Partie 3
Builder Pattern

```
"""
Separate the construction of a complex object from its representation so
that the same construction process can create different representations.
"""
class Director:
    """
    Construct an object using the Builder interface.
    """
    def __init__(self):
        self._builder = None
    def construct(self, builder):
        self._builder = builder
    def basic_product(self,)
        self._builder._build_part_a()
    def average_product(self,)
        self._builder._build_part_a()
        self._builder._build_part_b()

    def full_product(self,)
        self._builder._build_part_a()
        self._builder._build_part_b()
        self._builder._build_part_c()

class Builder():
    """
    Specify an abstract interface for creating parts of a Product
    object.
    """
    def __init__(self):
        self.product = Product()

    def _build_part_a(self):
        pass

    def _build_part_b(self):
        pass

    def _build_part_c(self):
        pass


class ConcreteBuilder(Builder):
    """
    Construct and assemble parts of the product by implementing the
    Builder interface.
    Define and keep track of the representation it creates.
    Provide an interface for retrieving the product.
    """
```

```
    def _build_part_a(self):
        pass

    def _build_part_b(self):
        pass

    def _build_part_c(self):
        pass


class Product:
    """
    Represent the complex object under construction.
    """
    pass


def main():
    concrete_builder = ConcreteBuilder()
    director = Director()
    director.construct(concrete_builder)
    product = concrete_builder.product


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

Questions :

1- Soit l'objet Sandwich qui composé d'un variétés ingrédients, on veut donner des noms aux différentes configurations (choix) des sandwich afin de faciliter la tâche de création des sandwich :

- Sandwich Shawarma
- Complet viande haché
- Frite omlette.
- Etc…

Implémenter la création des produits à l'aide de Builder.

2- Une voiture neuf est vendu en plusieurs options (la base, la toute, avec ou sans quelques options.

Implémenter la création des objets voitures selon les commandes.

Partie 3
Observer Pattern

```
"""
Define a one-to-many dependency between objects so that when one object
changes state, all its dependents are notified and updatedautomatically.
"""
class Subject:
    """
    Know its observers. Any number of Observer objects may observe a
    subject.
    Send a notification to its observers when its state changes.
    """
    def __init__(self):
        self._observers = set()
        self._subject_state = None

    def attach(self, observer):
        observer._subject = self
        self._observers.add(observer)

    def detach(self, observer):
        observer._subject = None
        self._observers.discard(observer)

    def _notify(self):
        for observer in self._observers:
            observer.update(self._subject_state)

    def subject_state(self):
        return self._subject_state

    def subject_state(self, arg):
        self._subject_state = arg
        self._notify()


class Observer():
    """
    Define an updating interface for objects that should be notified of
    changes in a subject.
    """

    def __init__(self):
        self._subject = None
        self._observer_state = None

    def update(self, arg):
        pass


class ConcreteObserver(Observer):
```

```
    """
    Implement the Observer updating interface to keep its state
    consistent with the subject's.
    Store state that should stay consistent with the subject's.
    """

    def update(self, arg):
        self._observer_state = arg
        # ...


def main():
    subject = Subject()
    concrete_observer = ConcreteObserver()
    subject.attach(concrete_observer)
    subject.subject_state = 123


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

Questions :

1- On veut implémenter une classe de notification à partir de site de département, où l'étudiant peut s'inscrire pour recevoir les notifications des nouvelles postes.

2- On veut implémenter un calendrier ou les utilisateurs peuventt s'inscrire pour recevoir les rappels.

Travail à Domicile : *Choisir un des deux projets*

Projet n° 1

1- Factory méthode :

On veut implémenter L ‘exemple de Pizza vue en Cours.

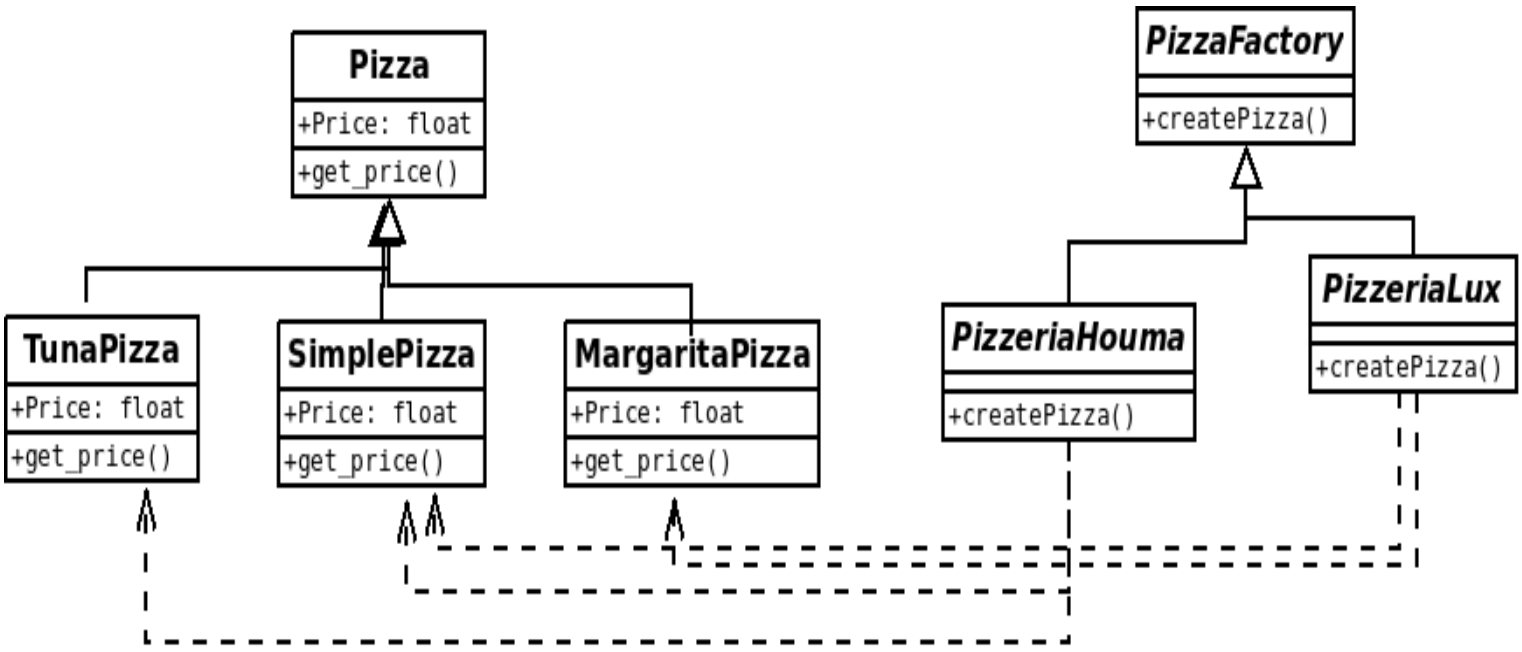
Une Pizza peut avoir plusieurs variantes (pizza simple, pizza thon, pizza couverte, pizza marguerite,…)

Une pizzeria est considérée comme une usine de fabrication des produits pizza. Une pizzeria n’offre que quelques types de produits.

Exemple : Pizzeria Simple offre ; pizza thon, pizza simple.

Pizzeria Lux offre d’autres types : pizza thon, pizza marguerite.

En utilisant le patron de conception « Factory method » implementer ce système.



2- Abstract Factory method :

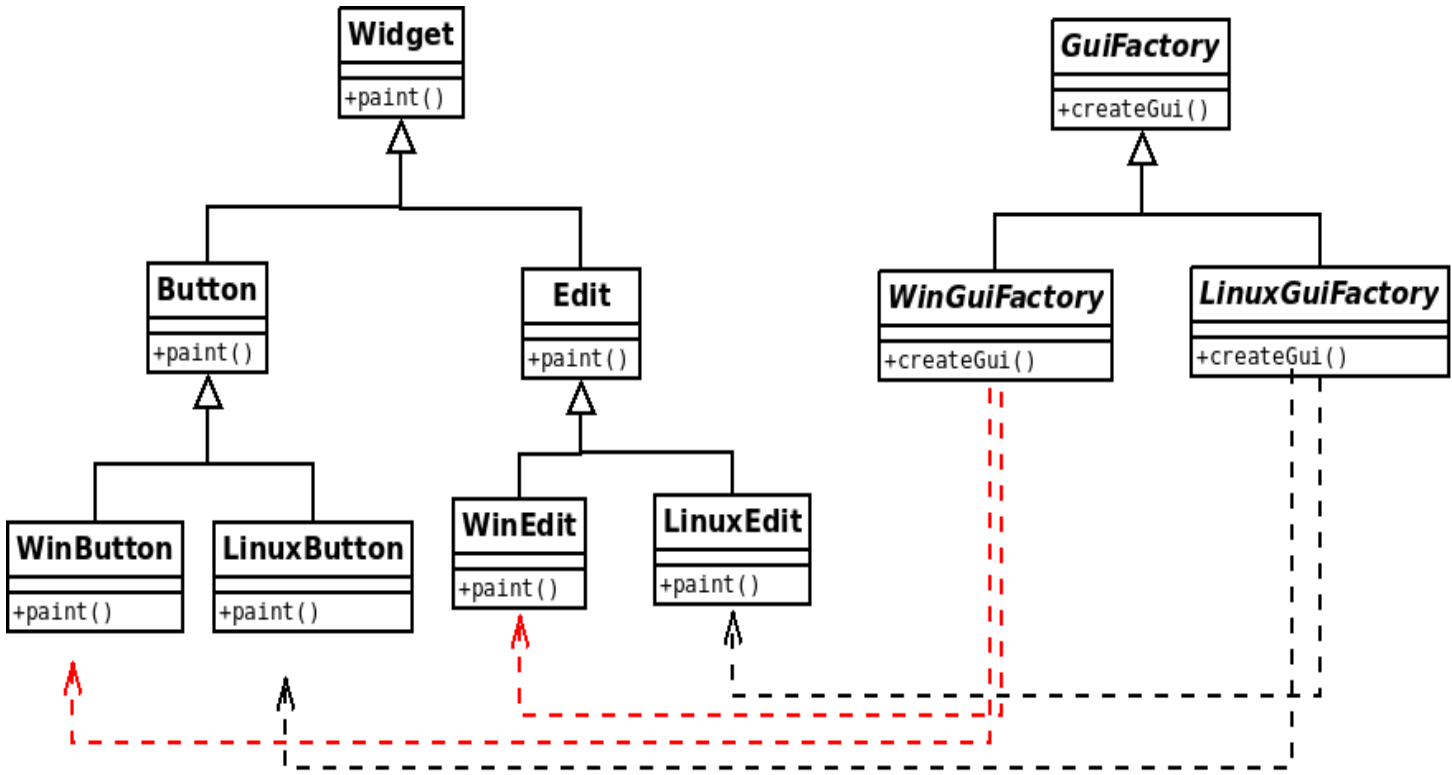
On veut implémenter L ‘exemple de l’interface graphique qui crée des widgets (objets graphiques) en fonction du système d’exploitation.

L’interface GUI (Graphical User Interface), propose des buttons et des zones de texte.

En utilisant le patron « Abstract Factory » implémenter le système.

Simplification : simuler l'interface graphique avec des affichage avec « print ».

Astuce : utiliser la bibliothèque python « platform » et la fonction platform.system() pour obtenir le nom de système d'exploitation, puis vous pouvez la tester sous Linux et windows.



3- Refaire les deux parties avec le langage Java.

2 ème projet (Optionnel):

Pour la gestion d'une agence touristique, on veut développer :

- Un façade pour les services sous-traités : transport, hébergement et restauration.
- Plusieurs stratégies pour chaque service, afin de satisfaire les différents budgets des clients.
- Des Adaptateurs pour le payement en dinars des services à l'étranger.
- Un contrôle d'accès aux ressources afin d'éviter les chevauchement des réservations (utiliser le singleton patterns).
- Possibilité d'annuler les réservations (utiliser le memento patterns)

Questions :

- Tracer les diagrammes des classes.

- Expliquer l’utilisation des patterns.
- Simuler l’implémentation des classes.