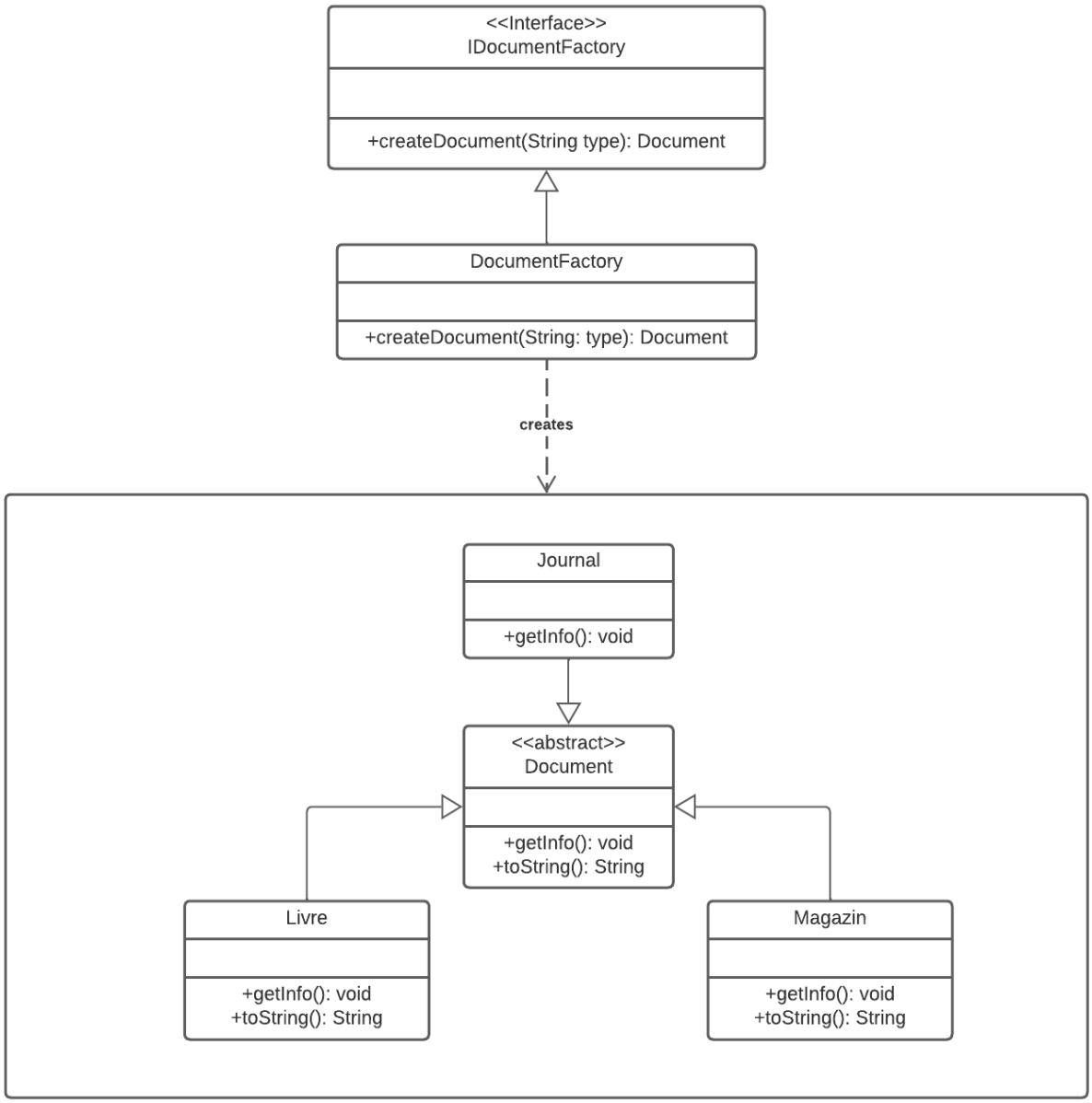
TP-PROJET DESIGN PATTERNS

IQAIDER Younes

Exercice 1:

Le Design pattern qu'on doit utiliser doit être de type ***Creational*** car le problème est à propos de la création des objets.
Le design pattern qui semble approprié à la problématique de l'exercice est le " Factory design pattern " . Les designs patterns ne sont que des méthodes générales de conception, alors il y aura plusieurs manières afin d'implémenter chacun d'eux. Une approche qui nous semble suffisante pour cet exercice est celle là :



On peut aussi procéder en créant des factory concrets pour chaque type concret de la classe document, mais ceci ne sera mieu que lorsque les algorithmes responsable de la créations de livre, magasin, et journal sont très différentes.

Exercice 2:

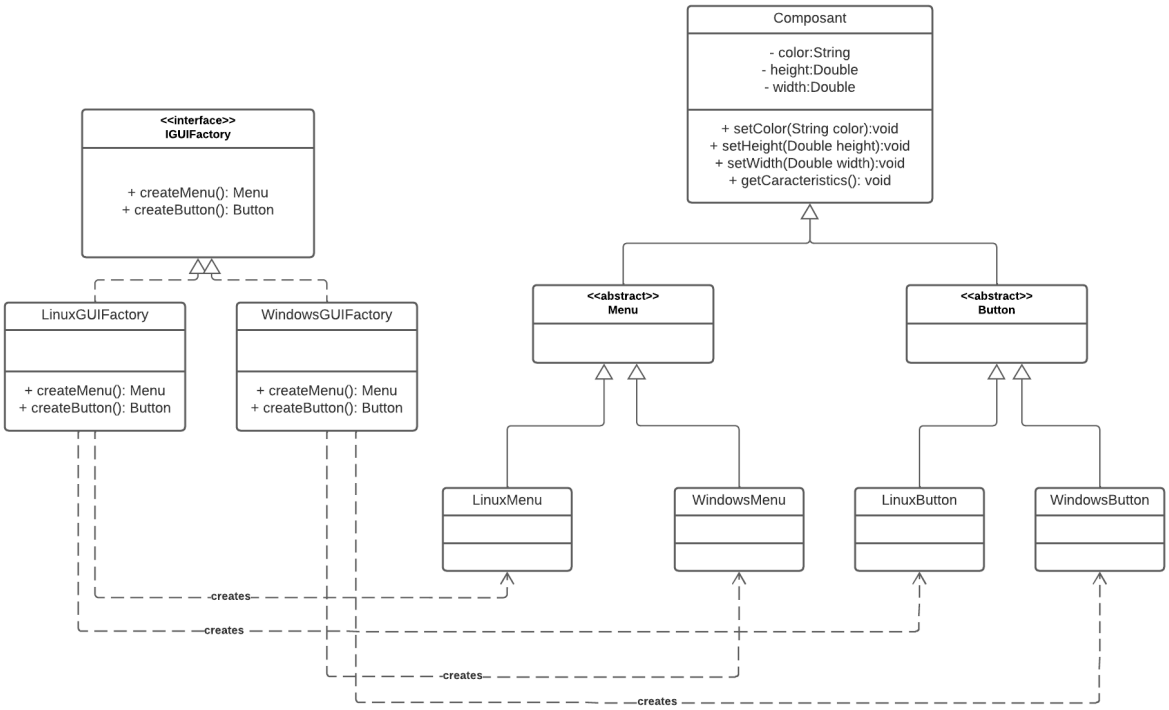
Aussi pour cette exercice on a à un problème de création donc, on aura aussi besoin d'un patron de conception ***Creational*** .

D'après le livre **Head first Design patterns :**

" Abstract Factory - Provide an interface for creating families of related or dependent objects without specifying their concrete classes."

Ce qui est exactement le cas pour cet exercice .

Un Diagramme UML qui me semble solution à ce problème :



Remarque: On peut procéder un peu différemment, car il peut paraître bizarre qu'une classe abstraite étende une classe concrète (par exemple: abstract menu extends concret Composant), mais ça devient utile surtout car tous les composant (menus et boutons) ont des méthodes et des attributs qui se ressemble et qu'on veut leurs donné des implémentations par défaut.

Test d'un exemple:

```java
public static void main(String args[]) {

    // afficher l'output pour le system d'exploitation windows

    IGUIFactory monEcran = new WindowsGUIFactory();

    Button btn = monEcran.createButton();
    btn.setColor(color: "#dddddd");
    Menu menu = monEcran.createMenu();
    menu.setColor(color: "#f3f3f3");


    System.out.println(btn);
    btn.getCaracteristics();

    System.out.println(menu);
    menu.getCaracteristics();

    // afficher l'output pour le system d'exploitation Linux

    IGUIFactory monEcran2 = new LinuxGUIFactory();

    Button btn2 = monEcran2.createButton();
    Menu menu2 = monEcran2.createMenu();

    System.out.println(btn2);
    System.out.println(menu2);


}
```

Execution:

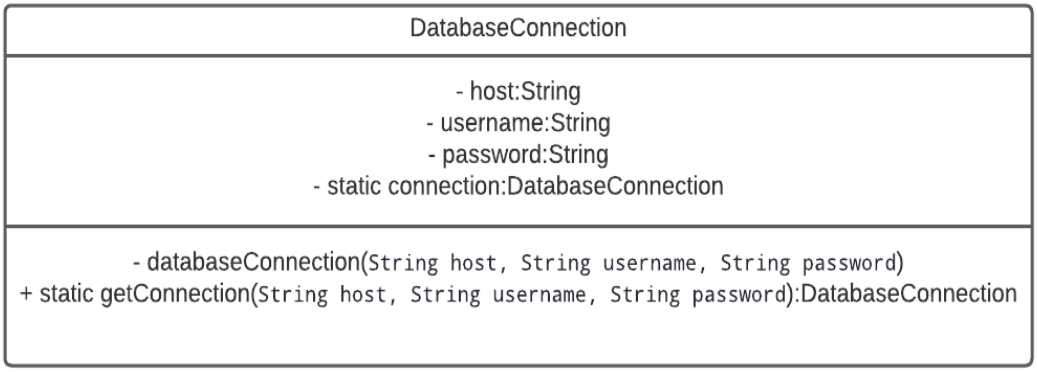
```
I'm a Windows Button
color: #dddddd
height: null
width: null
I'm a Windows Menu
color: #f3f3f3
height: null
width: null
I'm a Linux Button
I'm a Linux Menu
```

Exercice 3:

Pour ce problème, je vois que le patron de conception dont on a besoin est le Singleton Pattern.
D'après le livre **Head first Design patterns :**

“ The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it. “
Ce qui est le cas pour cet exercice, car on aura besoin d’une seule instance représentant la connexion à la base de données, et on a besoin d’y accéder dans toutes les parties du programme.

Un diagramme UML pour ce cas est simplement la classe représentant la connexion à la base de données (la classe équivalente à singleton).



On essai le programme qui simule la création d’une connexion à une base de donnée , on remarque que malgré la création d’une autre instance, il n y a pas de changement, ce qui signifie qu’il l’instance se crée une seule fois.

```
Run | Debug
public static void main(String args[]) {


    DatabaseConnection con = DatabaseConnection.getConnection(host: "host test", username: "username",
    password: "password");

    System.out.println(con);

    // on essai ici de crée une autre instance

    DatabaseConnection con2 = DatabaseConnection.getConnection(host: "host test 2", username: "username2",
    password: "password2");

    System.out.println(con2);



}
```

Execution :

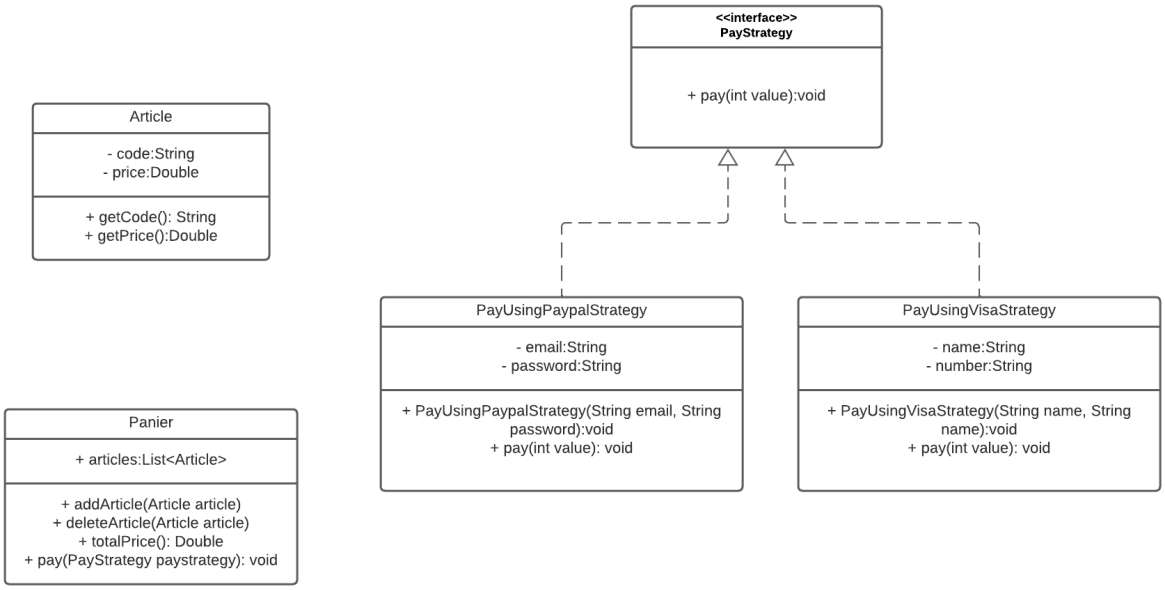
```
connected successfully, username: host test, username: username
connected successfully, username: host test, username: username
```

Remarque: l'implémentation (le code) écrit pour cet exercice n’est pas très efficace, car parfois si des threads multiples appellent le code responsable de l'instanciation de classe supposée singleton, plusieurs instance peuvent être créés due à l’effet synchrone du programme, c’est pour cela il y a une version qui est plus sécurisé, le **double-checked locking pattern.**

Exercice 4:

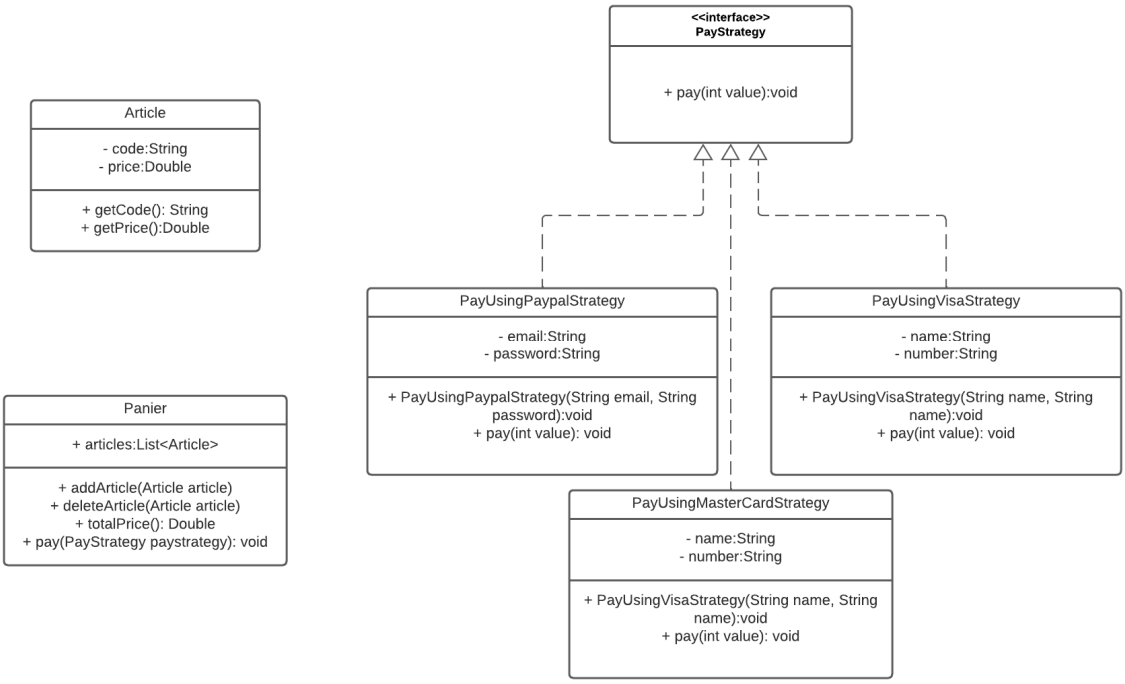
Le patron adapté à cette situation est le patron Stratégie.

Un diagram UML qui semble solution de ce problème:



Maintenant si l'on veut ajouter n'importe quelle autre méthode de paiement on n'aura besoin que d'ajouter une autre classe concrète qui implémente l'interface **PayStrategy** et qui aura sa propre méthode pay.
Et comme ça notre programme respecte le principe d'OCP (open closed principle), qui dit que notre code doit être ouvert pour l'extension et fermé pour la modification ce qui un avantage fort.

L'ajout de la stratégie de paiement MasterCard:



Un autre avantage qu'on peut citer dans ce cas (le strategy pattern) c'est qu'on a favorisé la composition au lieu de l'héritage (composition over inheritance principle), ceci se voit un petit peut dans la classe panier qui est composé de la fonction pay qui accepte une stratégie de paiement comme paramètre.
On peut donc injecter les méthodes de paiement dans n'importe quelle place dans notre code,
et c'est pour cette raison que ce patron de conception est très fort.

D'après le livre head first design patterns:

"The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it"

fin