j9


A
B
1
T36
DEVANT
2X TISSU AU PLI
C
D
3
2cm
OUVERTURE

J9: Objectifs

- Design patterns
- Singleton
- Iterator
- Factory
- Decorator
- Façade
- MVC
- Observer
- Strategy
- Gang of Four

Design patterns

- Patrons de conception

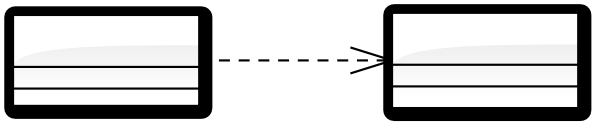
- Ouvrage fondateur
  - "Design Patterns, elements of reusable object-oriented software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994, Addison-Wesley
  - Dénommés la "bande des quatre", "*Gang of Four*", "*GoF*"
  - Liste et classe par famille les *design patterns*

- Concepts indépendants du langage – Par contre, les livres sur le sujet sont généralement fortement teintés d'un langage particulier.

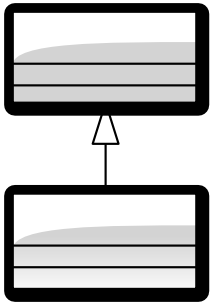
- Notation graphique

Un langage graphique

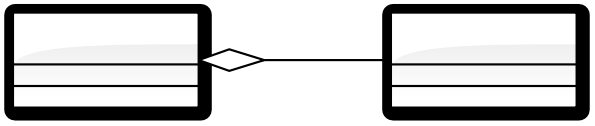
- Une classe en instancie une autre : flèche pointillée



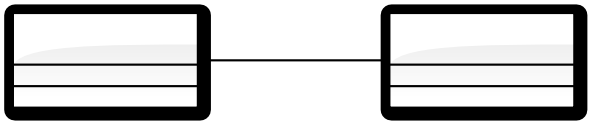
- Une classe hérite d'une autre : flèche vers le haut



- Trait avec un losange : relation a-un (plusieurs)



- Trait sans losange : relation a-un (exactement un)




There can be only one

Besoin

- Instancier une classe
- Retrouver une variable unique
- Stocker et mettre à jour de manière centralisée des informations utilisées dans des endroits très divers d'un même projet
- Exemples d'utilisation :
  - Vérifier que l'utilisateur est authentifié et s'il ne l'est pas, le faire
  - Obtenir la connexion à une base de données si elle existe et sinon s'y connecter
  - Récupérer les préférences de l'utilisateur si on les connait, sinon aller les chercher.
  - Ecrire dans un fichier de log déjà ouvert, sinon l'ouvrir

- Critique :
  - Peut révéler un certain désordre dans la conception (qui fait l'opération initiale? qui utilise ? pourquoi une conception locale ne fixe-t-elle pas définitivement ces responsabilités ?)

Singleton pattern

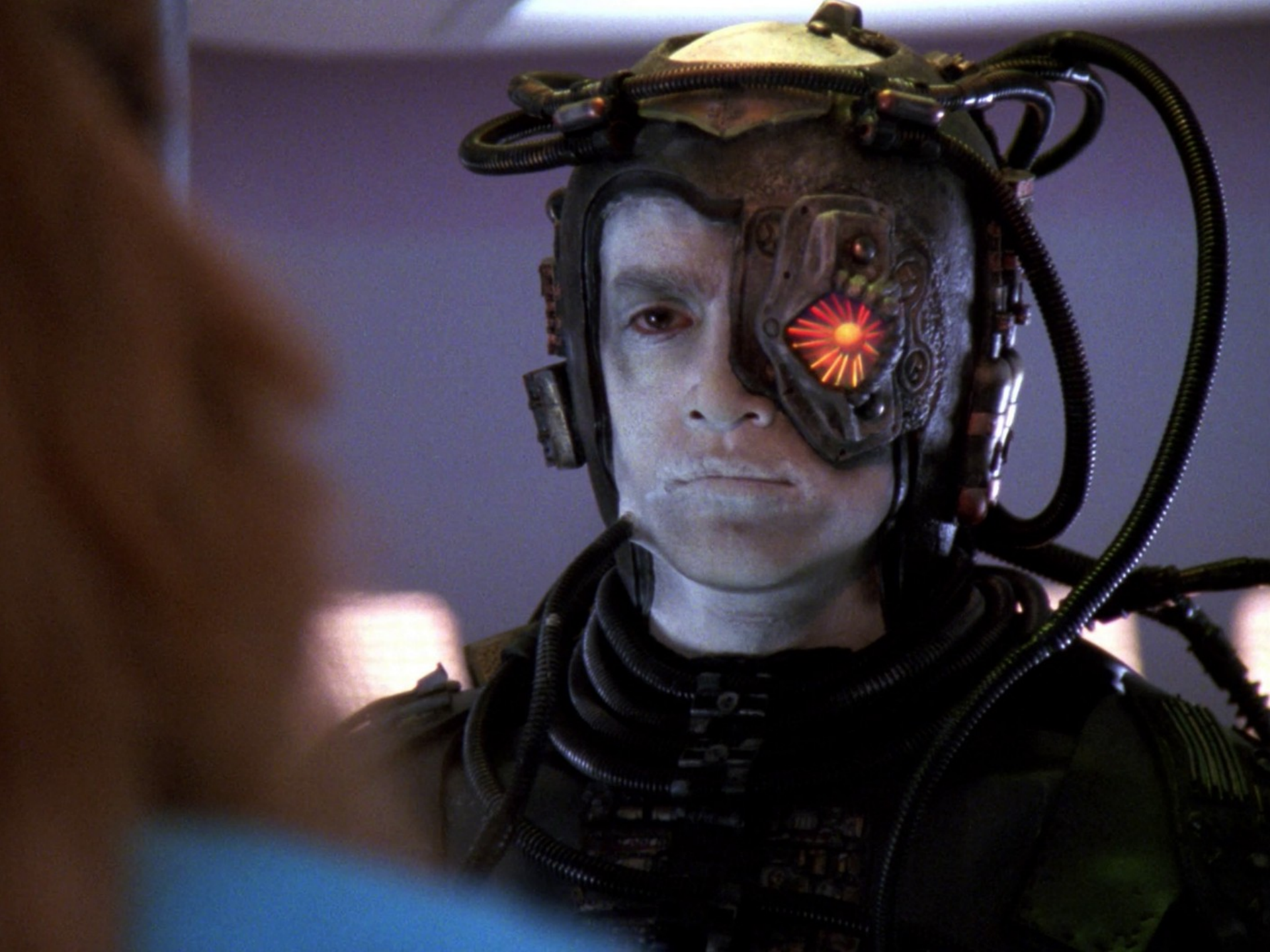
- Une classe qui ne peut être instanciée qu'une fois
- L'instanciation retourne toujours la même instance

"Il ne peut y en avoir qu'un"

Si la méthode new n'est pas surchargée, les instances des sous classes sont l'instance unique de la classe Singleton.

```
class Singleton:
  def __new__(cls):
    if not hasattr(cls,'_instance'):
      cls._instance=super().__new__(cls)
        return cls._instance
```

singleton.py



Borg pattern

- Une classe dont toutes les instances partagent les mêmes attributs avec les mêmes valeurs

```
class Borg:
    _state = {}
    def __init__(self):
        self.__dict__ = Borg._state
```

"Quand l'un le sait, tous le savent"

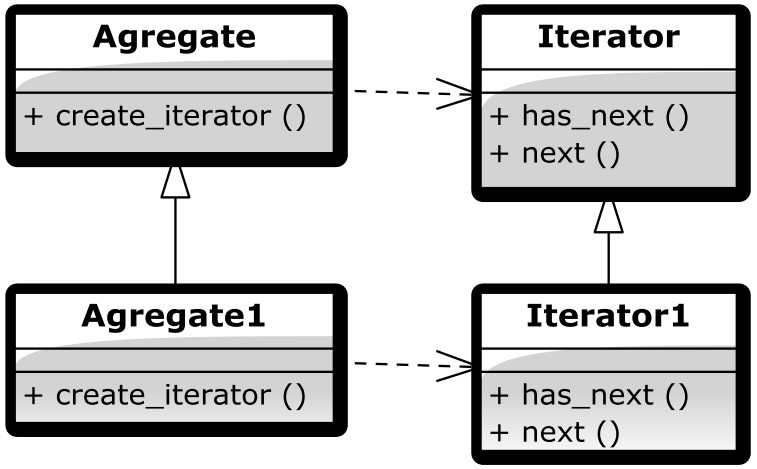
- Les sous-classes partagent le même état
  - à moins d'avoir redéfini l'état

Iterator pattern

- Python implémente ce patron de conception dans le langage
  - tous les objets itérables

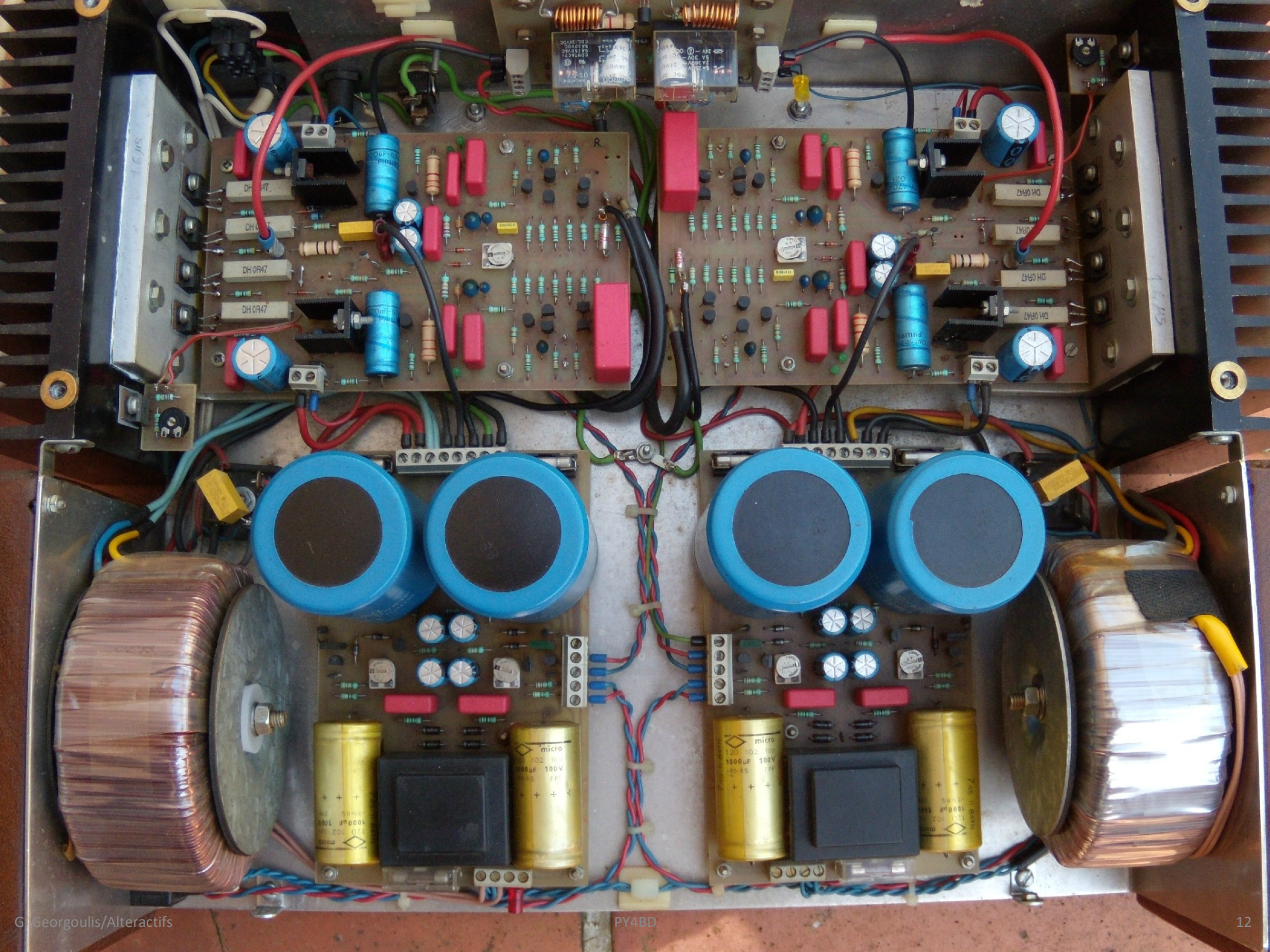
```
__iter__()
__next__()
raise StopIteration
```

- Dans le diagramme général ci-contre, un agrégat abstrait possède une méthode qui instancie son itérateur

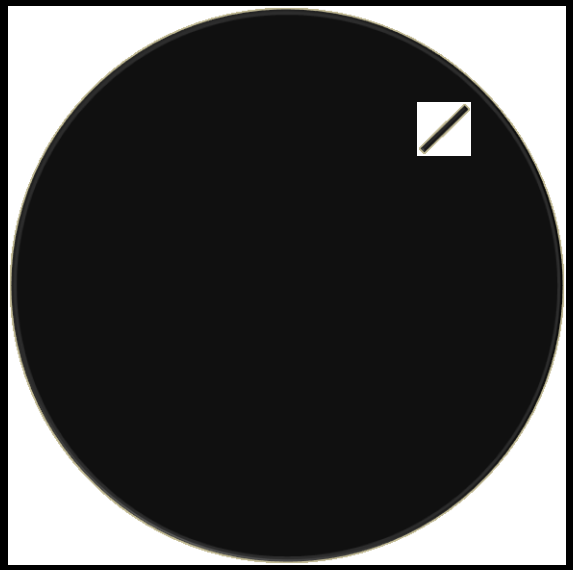


- Dans cette conception, un itérateur offre 2 méthodes qui s'utilisent comme ci contre

```
while it.has_next():
    next()
    …
```

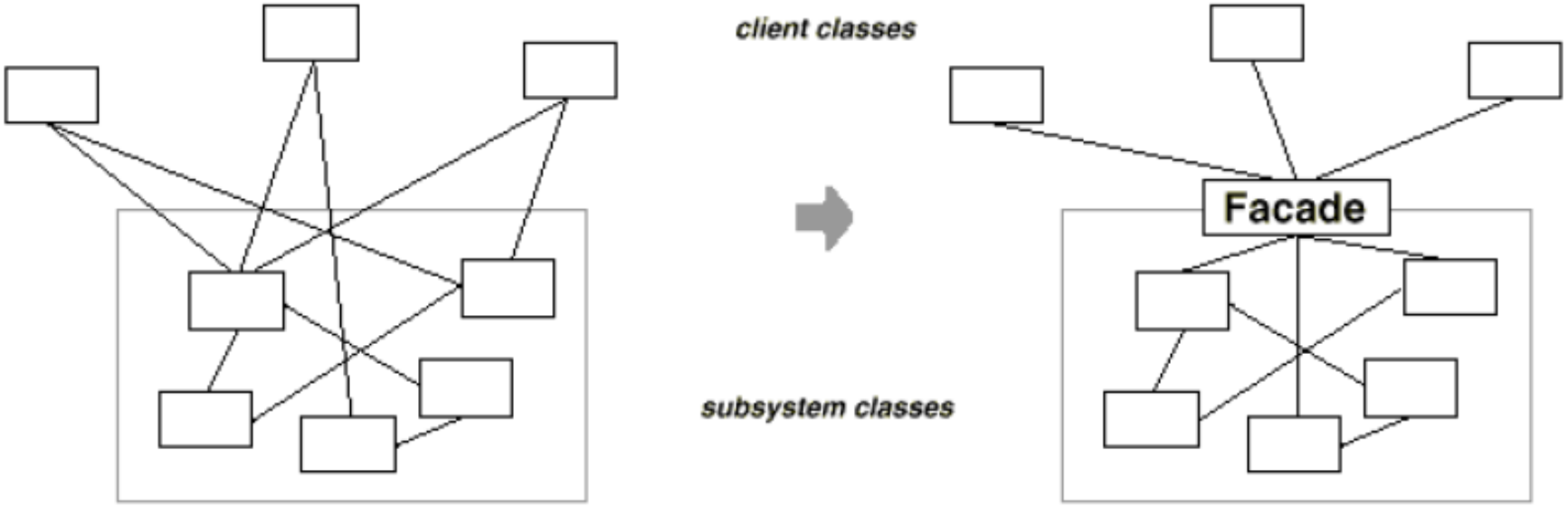

DH 0R47
micro
1000µF 100V

On/Off



Volume

Façade pattern



- Pour occulter la complexité de différents objets qui interagissent, le patron Façade propose un nombre réduit de méthodes, compréhensibles par les clients
- Exemple : mise en place d'un dispositif de fichiers logs complexe
  - rotation des fichiers
  - duplication des messages sur le terminal et dans un fichier

fulllog.py

NOS PIZZAS

TOP VENTES
1. Américaine
2. Mexicaine
3. Chicken
4. 4 Fromages

		JUNIOR	SÉNIOR	MÉGA
Marguerita	Tomate, double fromage, olives	7,00€	8,50€	12,50€
Reine	Tomate, double fromage, jambon*, champignons	8,00€	12,50€	17,50€
Napolitaine	Tomate, double fromage, anchois, câpres, olives			
Tonara	Tomate, double fromage, thon, oignons, olives			
Campione	Tomate, double fromage, viande hachée*, champignons, œuf	8,50€	13,50€	18,50€
Paysanne	Tomate, double fromage, lardons*, champignons, œuf			
Savoyarde	Tomate, double fromage, jambon*, reblochon, pommes de terre, œuf			
Exotique	Tomate, double fromage, jambon*, ananas			
Quattro Gusti	Tomate, double fromage, jambon*, poivrons, aubergines, champignons, artichauts, olives			
Végétarienne	Tomate, double fromage, poivrons, champignons, artichauts, oignons, tomates fraîches			
Orientale	Tomate, double fromage, merguez*, oignons, poivrons, œuf			
4 Fromages	Tomate, double fromage, brie, bleu, chèvre, parmesan			
Charcuterie	Tomate, double fromage, lardons*, jambon*, chorizo*			
Pescatore	Tomate, double fromage, fruits de mer, persillade, ail, citron			
Chicken	Tomate, double fromage, poulet*, champignons, poivrons, tomates fraîches, olives			
Cheese	Tomate, double fromage, chèvre, tomates fraîches, olives			
Boursin	Tomate, double fromage, viande hachée, Boursin, pommes de terre			
Saumon	Tomate, double fromage, saumon fumé, crème fraîche, citron	9,00€	14,00€	19,00€
Mexicaine	Tomate, double fromage, viande hachée*, merguez, oignons, œuf			
Américaine	Tomate, double fromage, jambon*, merguez*, chorizo*, champignons, œuf			
Tikka	Tomate, double fromage, poulet tikka, poivrons, tomates fraîches			
BUFFALO	Sauce barbecue, double fromage, bœuf haché, merguez, pommes de terre, œuf NOUVEAU			
DELICIOSA	Tomate, courgettes, poivrons, aubergines grillées, pointe de soja et poulet mariné NOUVEAU			
PROVENÇALE	Tomate, double fromage, champignons, roquette, tomates cerises, huile d'olives aromatisée au vinaigre balsamic NOUVEAU			

*Toutes nos viandes et charcuteries sont Halal

Base crème fraîche

		JUNIOR	SÉNIOR	MÉGA
Forestière	Crème fraîche, double fromage, lardons*, champignons, œuf	8,50€	13,50€	18,50€
Campagnarde	Crème fraîche, double fromage, lardons*, reblochon, pommes de terre, oignons			
Rimini	Crème fraîche, double fromage, viande hachée*, chèvre, oignons			
Venezia	Crème fraîche, double fromage, poulet*, reblochon, pommes de terre, oignons			
Diva	Crème fraîche, double fromage, brie, chèvre, bleu, parmesan			
Lorraine	Crème fraîche, double fromage, jambon*, oignons, reblochon			
Quickly	Crème fraîche, double fromage, thon, chèvre, oignons			
Fermière	Crème fraîche, double fromage, poulet*, champignons, pommes de terre, olives			
Indienne	Crème fraîche, double fromage, double poulet*, oignons			
Texane	Crème fraîche, double fromage, viande hachée*, merguez, jambon*	9,00€	14,00€	19,00€
Tartiflette	Crème fraîche, double fromage, lardons*, pommes de terre, oignons, chèvre			
Norvégienne	Crème fraîche, double fromage, saumon fumé, œufs de lump, citron			

À EMPORTER 7J/7
1 PIZZA ACHETÉE
= LA 2ÈME PIZZA OFFERTE*
*Offre valable sur Sénior et Méga, sauf Marguerita. La pizza la moins chère sera offerte. Offre non cumulable.

Nos Calzones

Calzone	Tomate, fromage, jambon* ou viande hachée* ou thon ou poulet, œuf	8,50€	13,50€	18,50€
Fromagère	Tomate, fromage, chèvre, brie, bleu			

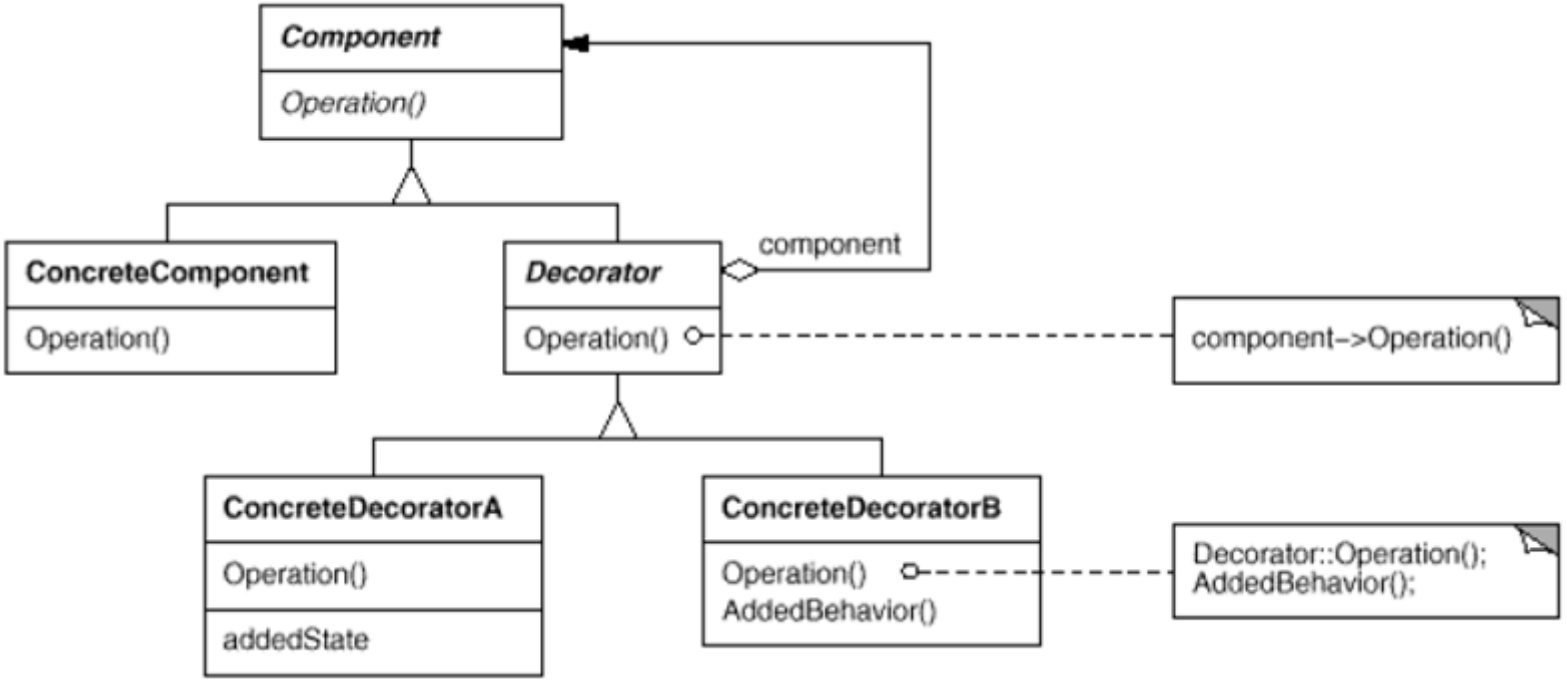
Composez vous-même votre pizza --- à partir de la Marguerita

Ingrédients : fromage, jambon*, champignons, anchois, câpres, olives, thon, oignons, viande hachée*, œuf, lardons*, reblochon, pommes de terre, ananas, poivrons, aubergines, artichauts, tomates fraîches, merguez*, brie, bleu, chèvre, parmesan, chorizo*, fruits de mer, persillade, ail, citron, poulet*, saumon fumé, crème fraîche, moutarde à l'ancienne.

1,00€ | 1,50€ | 2,00€

(Prix par ingrédient supplémentaire)

Decorator pattern



- Les hiérarchies de classes peuvent être inutilement complexes
  - Exemple du sandwich: kebab, avec ou sans oignons, au pain pita, avec un supplément fromage….. Cela marche aussi pour les pizzas…
- Ce patron montre comment réduire l'usage de l'héritage et l'intérêt de la composition

Un très bon article que je vous invite à lire

Le pattern decorator en Python

http://sdz.tdct.org/sdz/le-pattern-decorator-en-python.html

@property

- @property
- @x.setter, getter, deleter

Définir un attribut avec les méthodes pour le modifier, le supprimer, l'obtenir

getter, setter, deleter sont appelés des mutateurs

Ce dispositif est justifié si les mutateurs font quelque chose de plus que = ou del

La propriété est "cachée" par __ et reste manipulable par les mutateurs prévus par le concepteur.

```
class UneClasse:
    def __init__(self,val):
        self.propriete = val

x = uneClasse(12)
print (x.propriete)
```

```
class UneClasse:
    def __init__(self,val):
        self.__propriete = val

    @property
    def propriete(self):
        return self.__propriete

    @propriete.setter
    def propriete(self, val):
        self.__propriete = val

x = uneClasse(12)
print (x.propriete)
x.propriete = 13
```

Retour sur @property

- Un interface avec des mutateurs
- Occulter les champs de la classe
- Utile lorsque les mutateurs ne se réduisent pas à des affectations mais embarquent un peu d'intelligence

- Ce décorateur utilise le *descriptor protocol* basé sur

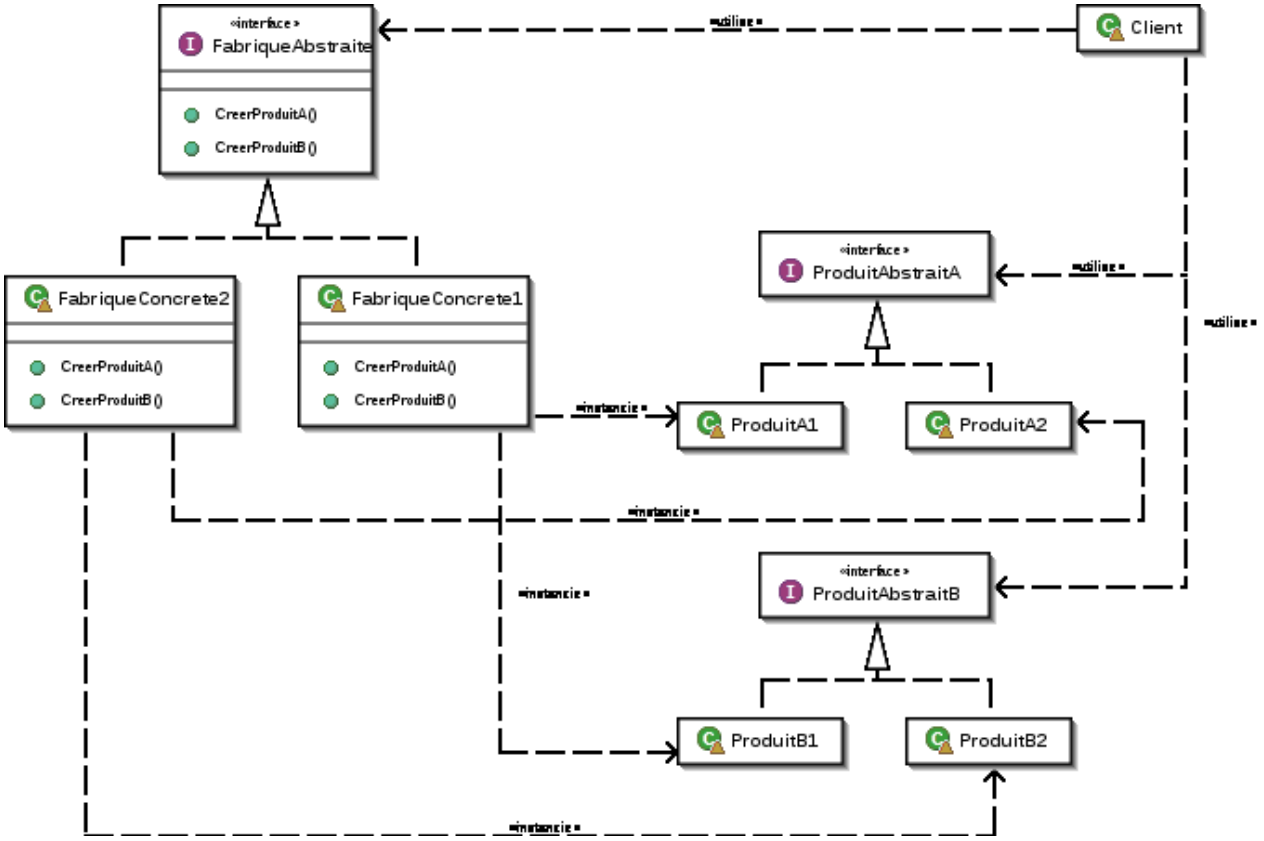
  __set___

  __get___

  __delete__

  __set_name__

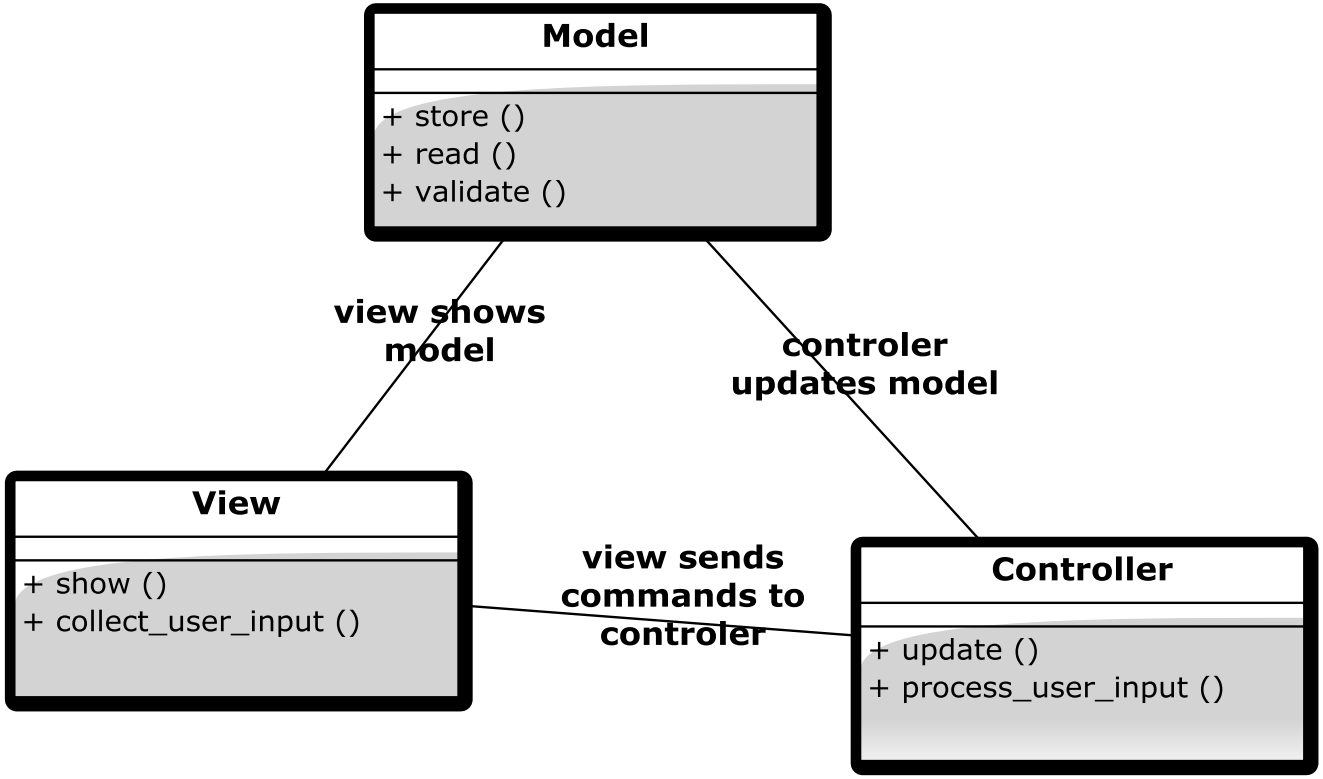
Factory pattern



- La fabrique abstraite est instanciée en une fabrique concrète pour produire les objets qui vont être utilisés dans l'application. La fabrication des objets peut rester ignorée du client qui s'adresse à la fabrique pour cela. Seule, l'utilisation des objets lui est connue.

https://fr.wikipedia.org/wiki/Fabrique_abstraite

MVC pattern



- *Model – View – Controller*
  - Le Modèle contient les données, les règles de gestion. Il est indépendant de la Vue (GUI) et du contrôleur. La Vue dépend du Modèle et le Contrôleur dépend des 2 autres.

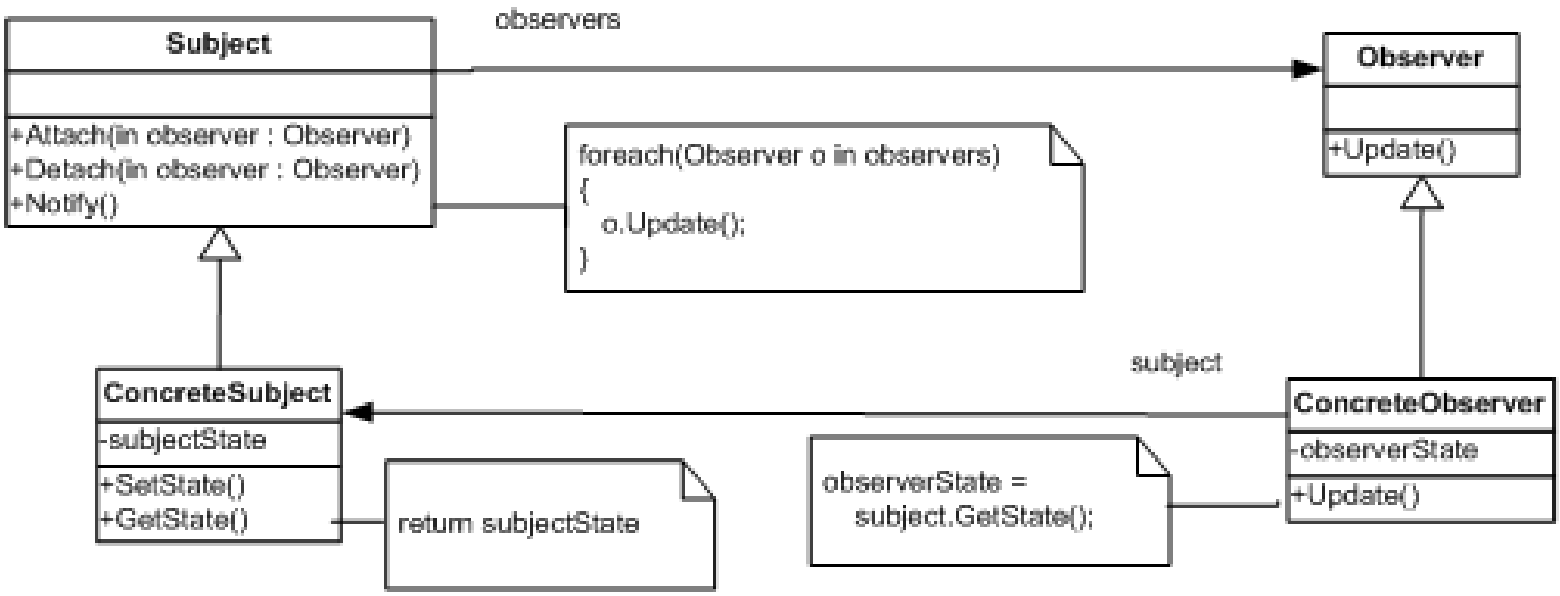
MVC suite

- Lorsque le modèle est mis à jour, qui se donne la peine de mettre à jour la vue ?
  - 1ère solution :
    - le Contrôleur SAIT qu'il a mis à jour le Modèle et il peut dire à la Vue de se régénérer
    - Ceci rend la vue dépendante du Contrôleur.
  - 2ème solution
    - le Modèle SAIT qu'il a été mis à jour et le notifie à la Vue.

- Il y a beaucoup de variantes. On essaye de maintenir toutefois le flux :
  - View → Controller → Model (cad, ne pas contourner le contrôleur)

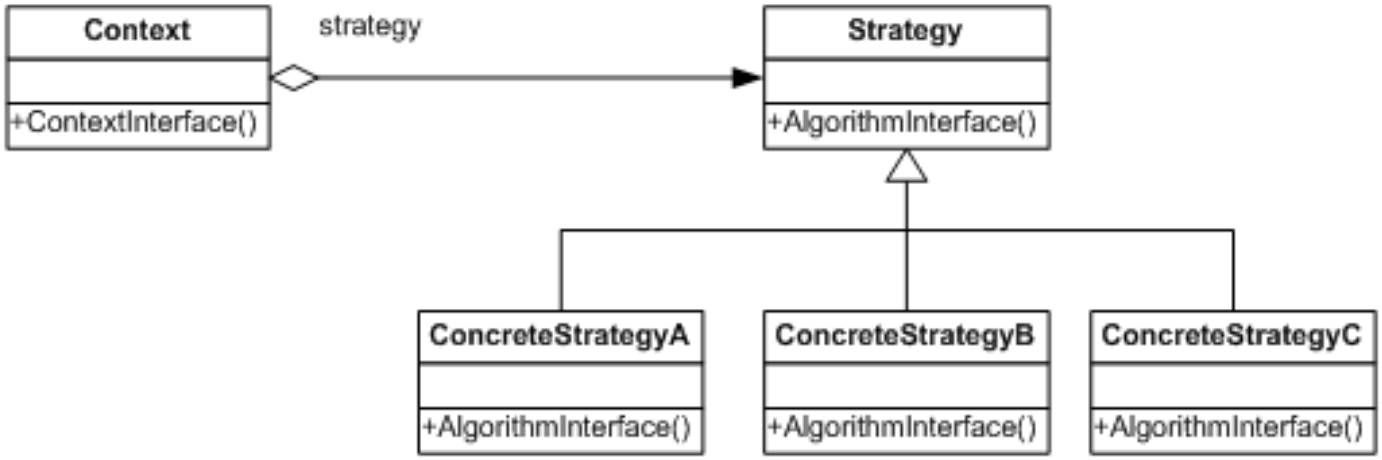
- Pour la mise à jour de la vue, le pattern de l'observateur est utilisé (*observer*)
  - le Modèle est l'Observable
  - La Vue est un Observateur
- Si l'on veut que la vue puisse, selon le contexte, déclencher des comportements différents du contrôleur, le pattern stratégie peut être utilisé (*strategy*)

Observator pattern



- Le pattern associe un Observable (*subject*) et des Observateurs (*observers*).
- Les Observateurs s'abonnent auprès de l'Observable.
- L'Observable notifie alors tout changement aux Observateurs abonnés.
- Les Observateurs peuvent se désabonner.

Strategy pattern



- Selon le contexte, une même tache peut être réalisée différemment
- L'objet contexte référence alors une stratégie différente
- Toutes les stratégies possèdent le même interface (héritent de la même classe)
- Avantage : diminuer la complexité de la tache, éviter d'interroger le contexte (par des **si alors**) dans la tache pour y orienter son exécution, séparer la décision de l'exécution.

MVC suite et fin

- MVC n'est pas répertorié comme un pattern dans l'ouvrage fondateur sur la question du "Gang des 4"
  - Vu plutôt comme une combinaison d'observer et strategy.

- MVC s'applique à toutes les applications interactives
  - même si l'interface est aussi fruste que la ligne de commande
  - mais en particulier lorsque l'on a un interface graphique
    - TKinter et autres
  - Et aussi lorsque l'interface graphique est un interface Web/vue par un navigateur
    - Flask, Django, PyWeb

- L'intérêt de MVC est de bien séparer ce qui est interface homme machine du reste et donc de faciliter des tests unitaires
- La notion de persistance (en anglais : *persistence*) désigne la possibilité de sauver l'état du modèle. Elle se situe au plus près du Modèle.

Exercices J9

Exercice Connexion BdD

- Ecrire une classe Db qui donne accès à la connection à la base de données par
  - Db().get_connection()

- Contrainte : la classe doit veiller à ce qu'au plus une seule connexion soit établie et ré-utilisée partout

- Questions/Suggestions :
  - Avez-vous utilisé le pattern Singleton ou le pattern Borg ?
  - Faites l'autre !
  - Avez vous trouvé un nouveau pattern ? (prévenir le formateur tout de suite)

Exercice Façade

- Voici un code que l'on inclut pour disposer de logs
  - sur le terminal ET
  - dans un fichier sur le disque avec une taille limite pour le fichier
  - puis création d'une nouvelle version
  - avec conservation d'une (1 seule) ancienne version

- Comment camoufler ce code dans une classe facile à utiliser ?

```
import logging
from logging.handlers import RotatingFileHandler

 #setup logging

    logger=logging.getLogger()
    logger.setLevel(logging.DEBUG)
    formatter=logging.Formatter('%(asctime)s \
:: %(filename)s :: %(levelname)s :: %(message)s')
    file_handler = RotatingFileHandler(\
'mon_fichier.log', 'a', 1000000, 1)

    file_handler.setLevel(logging.DEBUG)
    file_handler.setFormatter(formatter)
    logger.addHandler(file_handler)

    steam_handler = logging.StreamHandler()
    steam_handler.setLevel(logging.DEBUG)
    formatter=logging.Formatter('%(asctime)s \
\t %(filename)s \t %(levelname)s \
\t %(message)s', "%H:%M:%S")
    steam_handler.setFormatter(formatter)
    logger.addHandler(steam_handler)
```

Merci !

- Restons en contact :
  - Georges Georgoulis – ggeorgoulis@alteractifs.org – 06 12 68 40 06

  - Alteractifs

    Coopérative d'activité et d'entrepreneurs www.alteractifs.org