Projet Foot
Cours 3
2I013

Nicolas Baskiotis
nicolas.baskiotis@lip6.fr

Université Pierre et Marie Curie (UPMC)
Laboratoire d'Informatique de Paris 6 (LIP6)

S2 (2014-2015)

Design Patterns

Someone has already solved your problems

”Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” (C. Alexander)

Pourquoi ?

- Solutions propres, cohérentes et saines
- Langage commun entre programmeurs
- C’est pas seulement un nom, mais une caractérisation du problème, des contraintes,. . .
- Pas du code/solution pratique, mais une solution générique à un problème de design.

Un très bon livre :

Head First Design Patterns, E. Freeman, E. Freeman, K. Sierra, B. Bates, Oreilly

Design Patterns

Quelques Principes

- Identifier les aspects de votre programme qui peuvent varier/évoluer et séparer-les de ce qui reste identique
- Penser de manière générique et non pas en termes d'implémentations
- Composer plutôt qu'hériter (plus flexible) !

En avez-vous déjà vu ?

Design Patterns

Quelques Principes

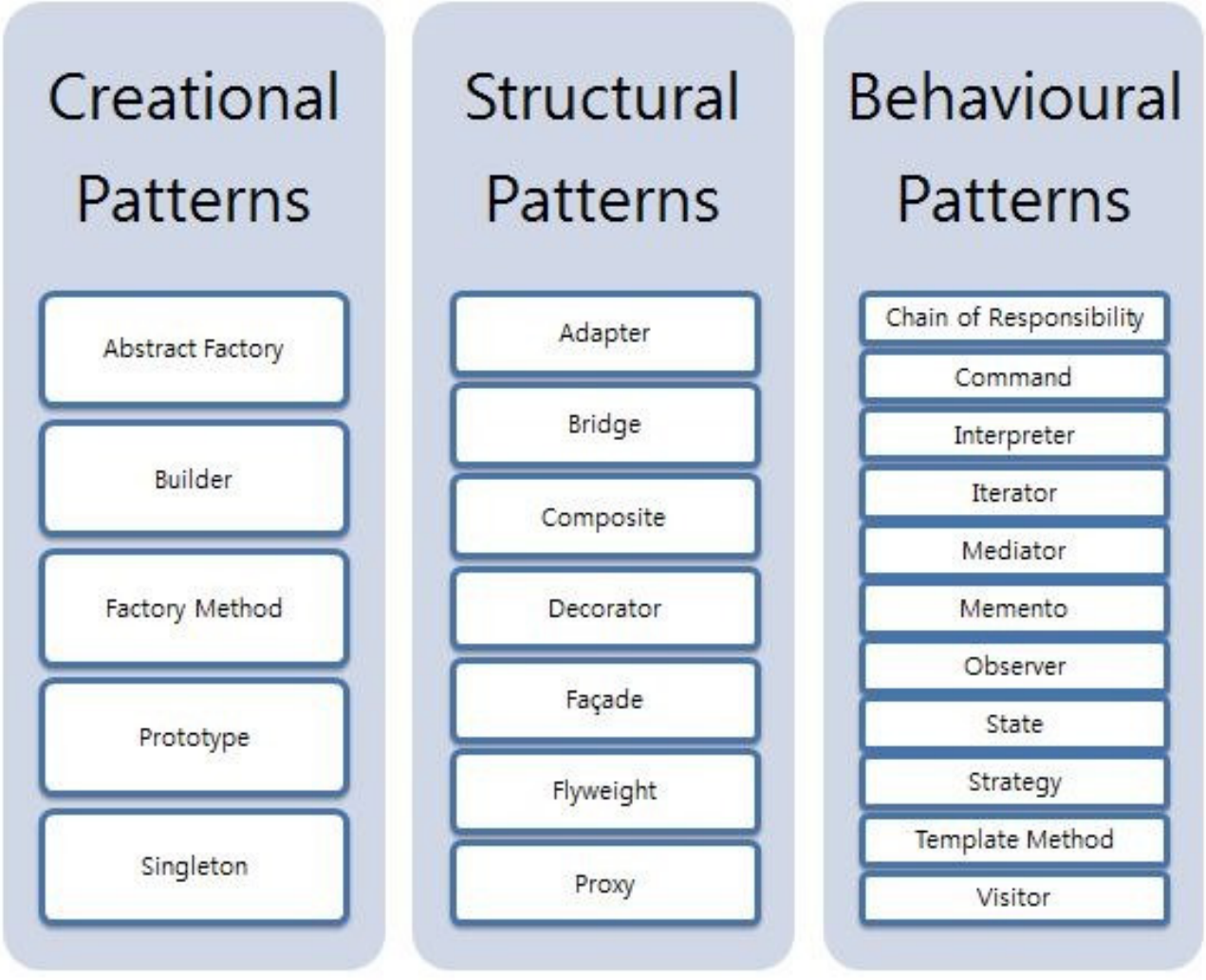
- Identifier les aspects de votre programme qui peuvent varier/évoluer et séparer-les de ce qui reste identique
- Penser de manière générique et non pas en termes d'implémentations
- Composer plutôt qu'hériter (plus flexible) !

En avez-vous déjà vu ?

Ce qu'on va voir

- Decorator
- Strategy
- Composite
- et plus tard Observer, Factory, Visitor, Adaptor, Iterator

Une liste non exhaustive

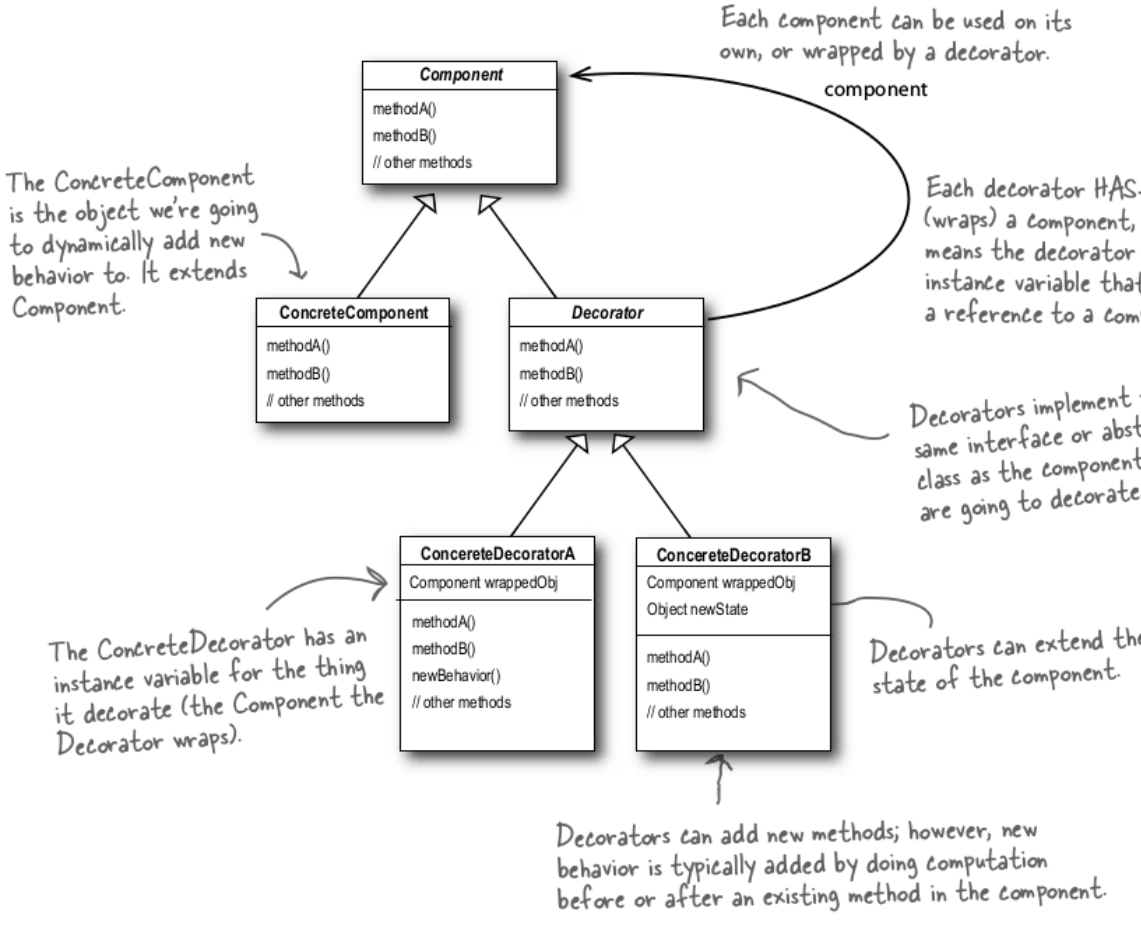


Decorator

Problèmatique

- La classe `SoccerState` contient plein d'informations, mais de bas niveau : position des joueurs adverses, de la balle, de ses partenaires, ...
- Est-ce pratique ? $\Rightarrow$ Non !
- On veut pouvoir ajouter des fonctions sur la balle, sur les joueurs, sur le terrain, ...
- Solution : ajouter des fonctions, mais comment ?
  - Une classe externe $\rightarrow$ oui mais si `SoccerState` change ... et difficile à manier
  - Hériter ? c'est statique et ne permet pas de mélanger plusieurs décorations ...
- Encapsulation + héritage !

Decorator



Pour `SoccerState` (simplifié)

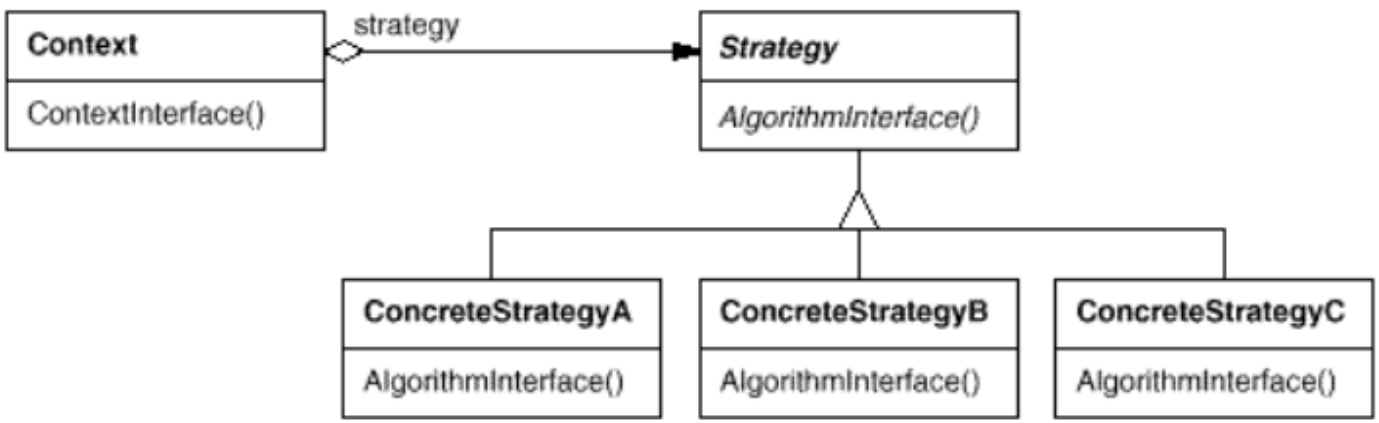
```
class BallStateDecorator(SoccerState):
    def __init__(self,state):
        self.state = state
    def est_proche_de(player): ...
class PlayerStateDecorator(SoccerState):
    def __init__(self,state):
        self.state = state
    def joueur_plus_proche(player): ...
    def joueur_plus_loin(player): ...

state = PlayerStateDecorator(state)
state = BallStateDecorator(state)
# ou
state = PlayerStateDecorator( \
BallStateDecorator(state))
```

Strategy

Utilité

- Le pattern Strategy définie une famille d'algorithmes, les encapsule et les rend interchangeables. Il permet de faire varier l'algorithme de manière dynamique et indépendante.
- Lorsqu'on a besoin de différentes variantes d'un algorithme.
- Lorsqu'on définie beaucoup de comportements à utiliser selon certaines situations



Composite

Problèmatique

- Les stratégies peuvent avoir beaucoup de chose en commun :
  - dégager le ballon
  - aller en défense
  - aller vers la balle
  - aller vers un joueur
- Doit-on toujours tout recoder ? $\rightarrow$ bien sûr que non ...

$\Rightarrow$ Pattern composite : arbre de composition

Exemple simple :

```
class Degager(Strategy) # ne code pas de deplacement
class VersBallon(Strategy) # ne code pas de shoot
class CombineStrategy(Strategy): #abstraite
    def __init__(self,direction,shoot):
        self.direction = direction
        self.shoot = shoot
    def compute_strategy(self,state,player,teamid):
        return SoccerAction(self.direction.compute_strategy(\
            state,teamid,player).direction,\
            self.shoot.compute_strategy(state,teamid,player).shoot)
```

Liste de strategie et sélection

Problème : une stratégie

- peut être complexe
- comprendre une multitude de cas (défense, attaque, gardien)

$\Rightarrow$ Comment coder proprement le choix de la stratégie de manière flexible ?

Encore une autre stratégie : SelectorStrategy

- introduire une classe `SelectorStrategy` (elle-même une stratégie)
- contient une liste de stratégie
- contient une méthode `select_strategy()`
- est-on obliger de tester les cas dans `select_strategy` ?

Schema

```
class SelectorStrategy(Strategy):
    def __init__(self,...):
        self.list_strategy=[]
    def select_strategy(self,...):
        raise NotImplementedError
    def compute_strategy(self,state,player,teamid):
        return select_strategy(self,...).compute_strategy(state,player,teamid)
```