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)

Comment organiser vos fichiers ?

Exemple de répertoire

```
mon_projet/
           __init__.py
           strategies.py
           tools.py
           team.py
           test.py
           ...
```

Exemple de contenu

- `strategies.py`

```
from soccersimulator import BaseStrategy
class MaStrategy(BaseStrategy):
....
```

- `team.py`

```
from strategies import MaStrategy
# Attention execute le fichier !!
...
team1 = SoccerTeam(..)
team2 = ...
team4 = ...
```

Comment organiser vos fichiers ?

Exemple de répertoire

```
mon_projet/
            __init__.py
            strategies.py
            tools.py
            team.py
            test.py
            ...
```

Exemple de contenu

- `test.py`

```
from soccersimulator import show
from soccersimulator import SoccerTeam,Player
from team import team1, team2, team4
if __name__ == __main__:
  show(SoccerMatch(team1,team1))
```

- `__init__.py`

```
from team import team1,team2,team4
```

Comment organiser vos fichiers ?

Exemple de répertoire

```
mon_projet/
           __init__.py
           strategies.py
           tools.py
           team.py
           test.py
           ...
```

Exemple de contenu

- `test.py`

```
from soccersimulator import show
from soccersimulator import SoccerTeam,Player
from team import team1, team2, team4
if __name__ == __main__:
  show(SoccerMatch(team1,team1))
```

- `__init__.py`

```
from team import team1,team2,team4
```

Surtout pas :

- `show(match)` dans un fichier importé par `__init__.py`

Pour importer d'autres joueurs :

- copier le répertoire du module dans votre répertoire
- `from autre_joueur import team1,...`
- possible d'importer tout ce qui est déclaré dans le fichier `__init__.py` (stratégies, joueurs, ...)

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

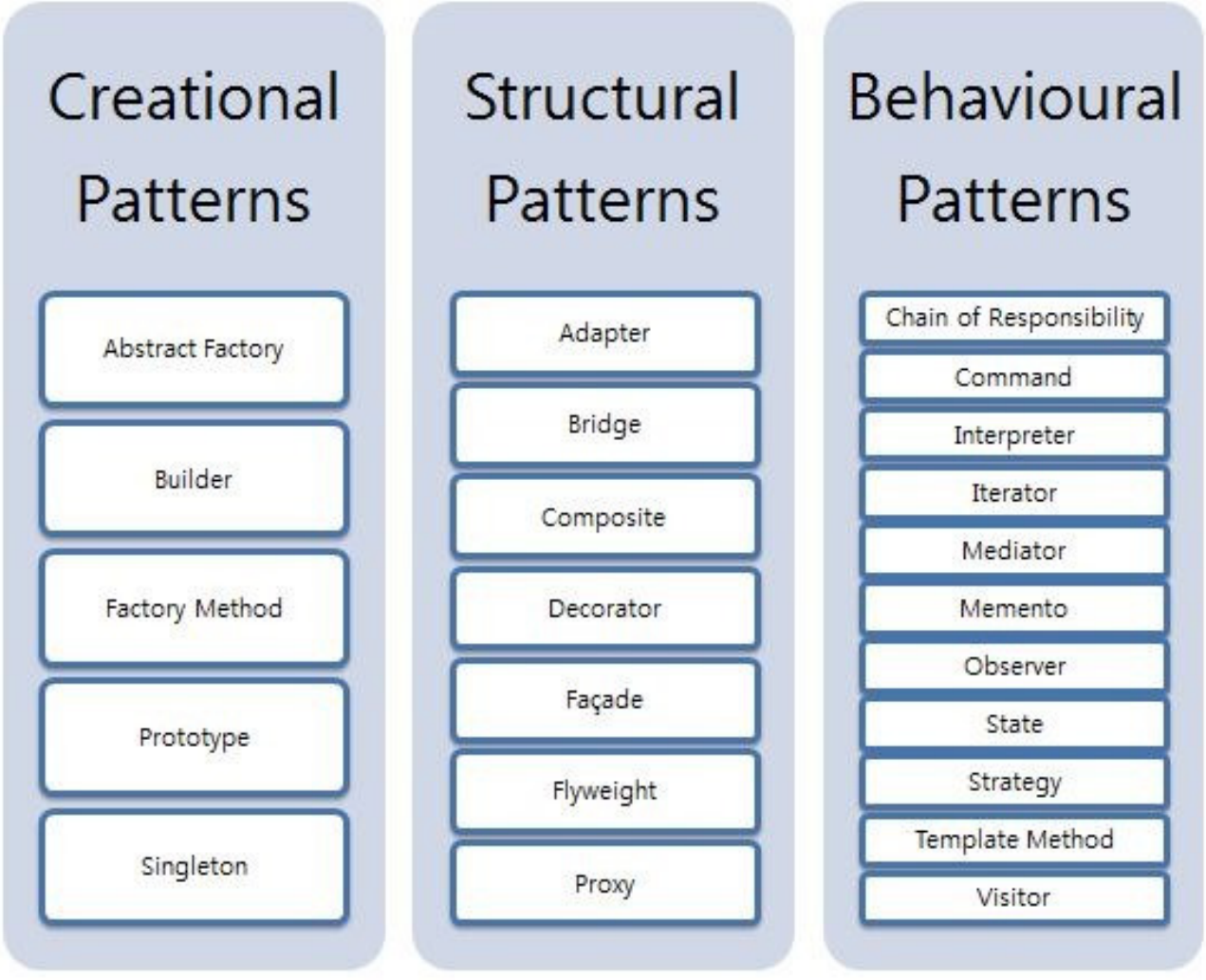
- Identifier les aspects de votre programme qui peuvent varier/évoluer et séparer-les de ce qui reste identique
- Penser de manière générique et non pas en termes d'implémentations
- Composer plutôt qu'hériter (plus flexible) !

En avez-vous déjà vu ?

Ce qu'on va voir

- Decorator
- Strategy
- Composite

Une liste non exhaustive



Vos difficultés pour l'instant

Vos difficultés pour l'instant

- Décomposer et préciser vos stratégies
- Extraire de l'information des états
- Faire des stratégies génériques
- Réagir en fonction de situations

Vos difficultés pour l'instant

- Décomposer et préciser vos stratégies
  $\Rightarrow$ Se forcer à coder des stratégies simples
- Extraire de l'information des états
  $\Rightarrow$ Design pattern decorator
- Faire des stratégies génériques
  $\Rightarrow$ Design pattern proxy
- Réagir en fonction de situations
  $\Rightarrow$ Design pattern strategy

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 ?

Decorator

Problèmatique

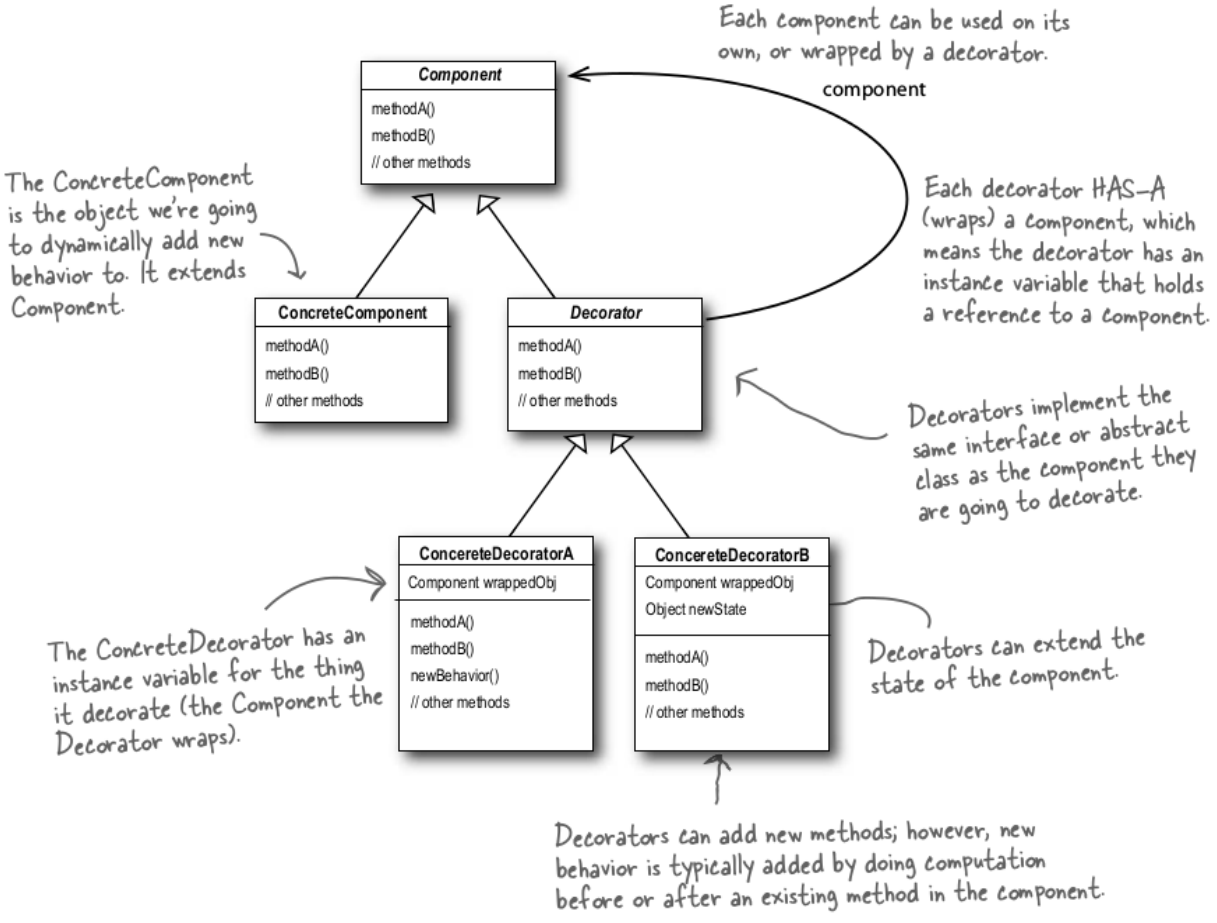
- La classe `SoccerState` contient plein d'informations, mais de bas niveau : position des joueurs adverses, de la balle, de ses partenaires, ...
- Est-ce pratique ?
  $\Rightarrow$ Non !
- On veut pouvoir ajouter des fonctions sur les joueurs, sur le terrain, ... : `ma_position, plus_proche_adv, ball_distance, ...`
- Solution : ajouter des fonctions, mais comment ?

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 les joueurs, sur le terrain, ... : `ma_position`, `plus_proche_adv`, `ball_distance`, ...
- 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


Each component can be used on its own, or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

State Decorator (1)

Par exemple un décorateur pour un joueur :

```
class PlayerStateDecorator:
   def __init__(self,state,idteam,idplayer):
     self.state = state
     self.idteam = idteam
     self.idlpayer = idplayer
   def position(self):
       return self.state.player_state(idteam,idplayer).position
   def distance_ball(self):
     return self.state.ball_position.distance(self.position())
     ...
```

Et pour la stratégie :

```
class MaStrategie(BaseStrategy):
      def __init__(self):
         BaseStrategy.__init__(self,"ma␣strategie")
      def compute_strategy(self,state,idteam,idplayer):
         mystate = PlayerStateDecorator(state)
         if mystate.distance_ball()<10
    ....
```

Utiliser des property, moins lourd

Par exemple un décorateur pour un joueur :

```
class PlayerStateDecorator:
   def __init__(self,state,idteam,idplayer):
     self.state = state
     self.idteam = idteam
     self.idlpayer = idplayer
   @property
   def position(self):
       return self.state.player_state(idteam,idplayer).position
   @property
   def distance_ball(self):
     return self.state.ball_position.distance(self.position)
```

Et pour la stratégie :

```
class MaStrategie(BaseStrategy):
      def __init__(self):
         BaseStrategy.__init__(self,"ma_strategie")
      def compute_strategy(self,state,idteam,idplayer):
         mystate = PlayerStateDecorator(state)
         if mystate.distance_ball<10
    ....
```

Faire “hériter” la composition

Fonction `__getattr__`

Elle est appelée si aucun attribut n’est trouvé, idéal pour déléguer.

```
class PlayerStateDecorator:
   def __init__(self,state,idteam,idplayer):
      ...
   def __getattr__(self,name):
      return getattr(self.state,name)
```

De cette manière, possibilité de composer les compositions !

Séparation du code, plus clair :

```
class PlayerStateDecorator:
  def __init__(self,state,idteam,idplayer):
      self.state,self.idteam,self.idplayer = state,idteam,idplayer
class BallDecorator:
  def __init__(self,state,idteam,idplayer):
      self.state,self.idteam,self.idplayer = state,idteam,idplayer
class MyState:
 def __init__(self,state,idteam,idplayer)
  self.state = BallDecorator(PlayerStateDecorator(state,idteam,idplaye
  self.idteam, self.idplayer = idteam, idplayer
```

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 . . .

$\Rightarrow$ Décomposer en action minimale : `degager()`,`shooter_but()`,`foncer()`,... qui retournent des `SoccerAction` puis les additionner.

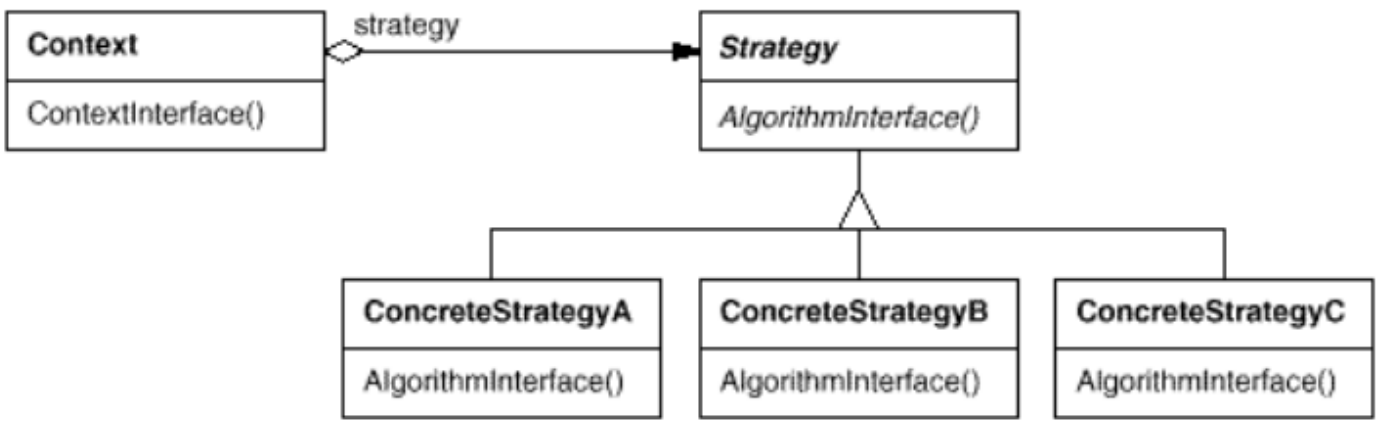
```
def degager(mystate): # ne code pas de deplacement
def versBallon(mystate): # ne code pas de shoot
def shooter(mystate): #shooter

class CombineStrategy(Strategy): #abstraite
      def compute_strategy(self,state,idteam,player):
  mystate = MyState(state,idteam,player)
          return degager(mystate)+versBallon(mystate)
```

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



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 ?

Exemple de sélection

```
from collections import namedtuple
def defenseur(mystate): #code un defenseur
def goal(mystate) : # code un guardien
def attaquant(mystate): # code un attaquant

def Polyvalent(mystate):
    if mystate.distance_ball < 5: return goal(mystate)
    if mystate.distance_plus_proche : return defenseur(mystate)
    return attaquant(mystate)
```

Et le problème de la symétrie des équipes ?