Projet Foot
2I013

Nicolas Baskiotis
nicolas.baskiotis@lip6.fr
http://webia.lip6.fr/~baskiotisn
http://github.com/baskiotisn/SoccerSimulator-2017

Sorbonne Universités - Université Pierre et Marie Curie (UPMC)
équipe MLIA, Laboratoire d'Informatique de Paris 6 (LIP6)

S2 (2017-2018)

Plan

Comment organiser vos fichiers ?

Garder votre code séparé du module `soccersimulator` !!!

Exemple d'organisation

```
2I013/
      SoccerSimulator-2017/                    # git du prof
                  .git
                  setup.py
                  soccersimulator/
                  ...
      MonGit/                                  # votre git
             mes fichiers
```

Pourquoi ?

Comment organiser vos fichiers ?

Garder votre code séparé du module `soccersimulator` !!!

Exemple d'organisation

```
2I013/
      SoccerSimulator-2017/                    # git du prof
                    .git
                    setup.py
                    soccersimulator/
                    ...
      MonGit/                                  # votre git
             mes fichiers
```

Pourquoi ?

- Vous ne devez pas changer le code du module !
- Votre code doit pouvoir être distribué !
- Votre code est indépendant du module.
- Le module change (souvent), ne pas mélanger les versions.

Les importations en python

Deux manières de programmer en python :

- Script : pour du développement rapide, pour tester des fonctionnalités, pour utiliser principalement du code déjà existant, pour prototyper, ...
  commande : `python monscript.py`
- Package/Module : pour du *vrai* développement, pour partager/diffuser son code, pour coder proprement ...
  commande : `python -m module/script.py` ou dans un fichier script `import module`

$\Rightarrow$ La seule grande différence : la gestion des `import`

Fonctionnement des `import` en python

- Un fichier `.py` est importable, un répertoire contenant un `__init__.py` est importable (et plus depuis python 3).
- Mais doivent se trouver dans le chemin défini par `PYTHONPATH` ...
- Et ce n'est pas le même en fonction de l'exécution script ou de l'importation du module ...

Les importations en python

Deux manières de programmer en python :

- Script : pour du développement rapide, pour tester des fonctionnalités, pour utiliser principalement du code déjà existant, pour prototyper, ...
  commande : `python monscript.py`
- Package/Module : pour du *vrai* développement, pour partager/diffuser son code, pour coder proprement ...
  commande : `python -m module/script.py` ou dans un fichier script `import module`

⇒ La seule grande différence : la gestion des `import`

Utilisation de `import`

`import module`	`from module \` `import myf,myvar`	`from module import\` `myf as f,myv as v`	`import module` `as m`
`module.myf()` `module.myvar`	`myf()` `myvar`	`f()` `v`	`m.myf()` `m.myvar`

Organisation des fichiers pour du script

Exemple de répertoire

```
mon_projet/
            [__init__.py]
            strategies.py
            tools.py
            team.py
            test.py
            data/
                donnees.pkl
```

Exemple de contenu

- `strategies.py`

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

- `team.py`

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

Organisation des fichiers pour du script

Exemple de répertoire

```
mon_projet/
          [__init__.py]
           strategies.py
           tools.py
           team.py
           test.py
           data/
               donnees.pkl
```

Exemple de contenu

- `test.py`

```
from soccersimulator import show_simu,Simulat
from team import team1, team2, team4
if __name__ == __main__:
  show_simu(Simulation(team1,team1))
```

- `__init__.py`

```
from team import team1,team2,team4
def get_team(x):
    if x==1:
      return team1 ...
```

Attention

- Utiliser `reload` pour recharger un module.
- Dans tout fichier importé, le fichier est exécuté en totalité sauf la partie `if __name__==__main__`

⇒ Jamais de `show_simu()` (ou de code exécutable) dans un fichier importé par `__init__.py`

Organisation des fichiers pour un module

Exemple de répertoire Python 2

```
projet/
      module/
            __init__.py
            mafonction.py
            script.py
      test.py
```

```
#Imports relatifs
__init__.py :
    from mafonction import fonction
mafonction.py :
    def fonction(): return "fonction"
script.py :
    from mafonction import fonction
    print(fonction())
# Import absolu
test.py : from module import fonction
          print(fonction())
```

Différence entre Python 2 et 3

- Import chemin absolu : la même chose
- Import chemin relatif : différent !
  - Python 2 : import relatif *par défaut*
  - Python 3 : import relatif doit être explicite

  $\Rightarrow$ `from .mafonction import fonction`
  - ne fonctionne que pour l'import de package, pas le script . . .

Exemple casse-tête

```
# dans chaque fichier : def fonction() : ...
module/mafonction.py
module/autrefonction.py
module/sousmodule/autrefonction.py
module/autresousmodule/autresousfonction.py
```

Comment savoir ce que l'on exporte ? Ce que l'on importe ?

Import/export

```
module/__init__.py                          Chemin relatif obligatoire !
module/mafonction.py
module/autrefonction.py
module/sousmodule/__init__.py          -> from .autrefonction
                                             import fonction
module/sousmodule/autrefonction.py
module/autresousmodule/__init__.py     -> from .autresousfonction
                                             import fonction
module/autresousmodule/autresousfonction.py
```

- Rend accessible au module de base les fonctions des sous-modules (import absolu) :
  `from module.sousmodule import fonction`
- Autre possibilité dans `module/mafonction.py` (import relatif) :
  `from .sousmodule.autrefonction import fonction`
- Et depuis `sousmodule` :
  `from ..autresousmodule.autresousfonction import fonction`

Renommé les imports

```
module/__init__.py                          Chemin relatif obligatoire !
module/mafonction.py
module/autrefonction.py
module/sousmodule/__init__.py          -> from .autrefonction
                                             import fonction
module/sousmodule/autrefonction.py
module/autresousmodule/__init__.py     -> from .autresousfonction
                                             import fonction
module/autresousmodule/autresousfonction.py
```

Dans `module/mafonction` :

```
from module.autrefonction import fonction as autref
from module.sousmodule import fonction as sousmodulef
from module.autresousmodule import fonction as autresousmodf

print(autref(),sousmodulef(),autresousmodf())
```

Problème : les fichiers scripts ...

```
module/__init__.py                          Chemin relatif obligatoire !
module/script.py
module/mafonction.py
module/autrefonction.py
module/sousmodule/__init__.py               -> from .autrefonction
                                                import fonction
module/sousmodule/autrefonction.py
module/autresousmodule/__init__.py          -> from .autresousfonction
                                                import fonction
module/autresousmodule/autresousfonction.py
```

Dans script.py :

```
# Marche en execution module : python -m module/script.py
from module.mafonction import fonction
#ou
from .mafonction import fonction
# Mais ne marche pas en execution script ! python module/script.py
```

Structure répertoire recommandé

```
projet/
  module/__init__.py     : get_team, get_challenge, ...
  module/team.py
  module/sousmodules/ ...
  script_essai.py    # from module import get_team
  script2_essai.py   # from module import get_team ...
```

Compatibilité Python 2

Rajouter : `from __future__ import absolute_import` au début de tous les fichiers.

Pour importer d'autres joueurs :

- copier le répertoire du module dans votre répertoire
- `import binome; binome.get_team(1)`

Plan

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

- Surtout pour les langages fortement typés, structurés (**Java** par exemple)
- Identifier les aspects de votre programme qui peuvent varier/évoluer et les séparer 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

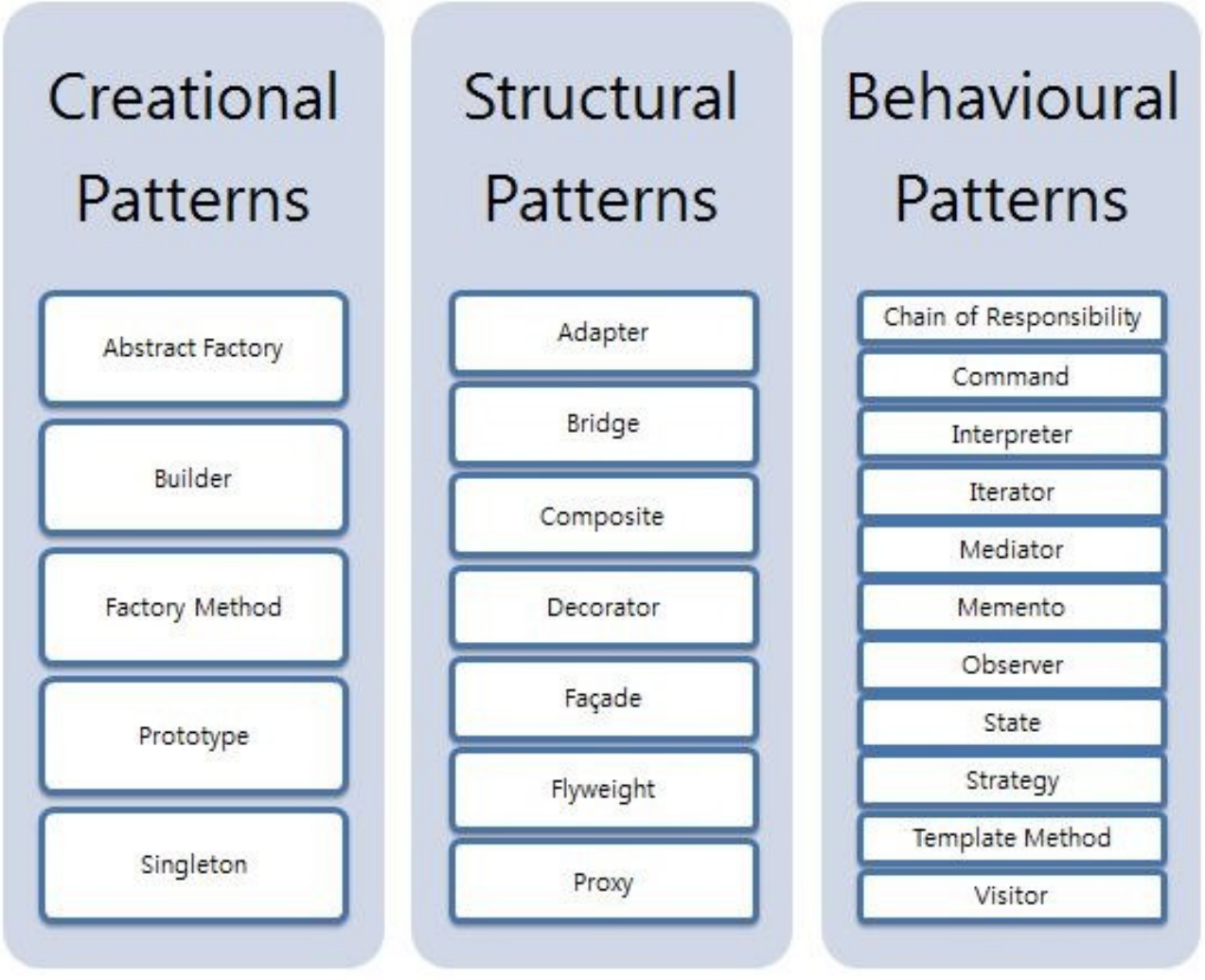
- Surtout pour les langages fortement typés, structurés (**Java** par exemple)
- Identifier les aspects de votre programme qui peuvent varier/évoluer et les séparer de ce qui reste identique
- Penser de manière générique et non pas en termes d'implémentations
- Composer plutôt qu'hériter (plus flexible) !

En avez-vous déjà vu ?

3 grandes classes

- *Creational :* Comment créer des objets
- *Structural :* Comment interconnecter des objets
- *Behavioral :* Comment faire une opération donnée

Une liste non exhaustive



Creational patterns

En python, il n'y en a pas vraiment (sauf le singleton). Pour créer un objet d'une certaine manière, il suffit de faire une fonction.

```
 def get_random_vec(x,y):
     return Vector2D.create_random(x,y)
def from_polar(x,y):
     return Vector2D.from_polar(0,2)
def from_cartesien(x,y):
     return Vector2D(x,y)
def get_null():
     return Vector2D()
   ...
```

Quelques caractéristiques de Python

Dans un objet :

- `def __init__(self,*args,**kwargs)`
  - `args` : arguments non nommés (`args[0]`)
  - `kwargs` : arguments nommés (`kwargs[''nom'']`)
- `__getattr__(self,name)` : appelé quand `name` n'est pas trouvé dans l'objet
- `__getattribute__(self,name)` : appelé pour toute rercherche de `name`

- Propriété : pour interroger de manière dynamique

```
class MyClass:
  @property
  def name(self): return ...
...
a = MyClass()
a.name # plutot que a.name()
```

En python, pas d'erreur de typage, uniquement à l'exécution !

Python : Duck Typing

If it looks like a duck and quacks like a duck, it's a duck!

Typage dynamique

- La sémantique de l'objet (son type) est déterminée par l'ensemble de ses méthodes et attributs, dans un contexte donné
- Contrairement au typage nominatif où la sémantique est définie explicitement.

Concrétement

```
Class  Duck:
  def quack(self):
     print("Quack")
Class Personne:
  def parler(self):
    print("Je␣parle")
donald  = Duck()
moi = Personne()
autre = "un␣canard"
try:
  donald.duck()
  moi.duck()
  autre.duck()
except AttributeError:
    print("c'est␣pas␣un␣canard")
```

Adapteur : et si je veux que ce soit un canard ?

- Il suffit d'y ajouter une méthode qui le fait se comporter comme un canard.
- Toutes les autres méthodes doivent être disponibles !

```
class PersonneAdapter:
    def __init__(self,obj):
        self._obj = obj
    def __getattr__(self,attr):
        if attr == "duck":
            return self.parler()
        return attr(self._obj,attr)

moi = DuckAdapter(Personne())
moi.duck()
```

Iterator

Pouvoir parcourir une liste d'éléments sans connaître l'organisation interne des éléments

Un itérateur est un objet qui dispose

- d'une méthode `__iter__(self)` qui renvoie l'itérateur
- d'une méthode `next(self)` qui renvoie la prochaine valeur ou lève une exception `StopIteration`

Un itérateur peut être renvoyé par une fonction grâce à `yield`.

```
class Counter:
    def __init__(self,low,high):
        self.current = low
        self.high = high
    def __iter__(self):
        return self
    def next(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current+=1
            return self.current-1
```

```
def counter(low,high):
    current = low
    while current <= high:
        yield current
        current += 1

for c in counter(3,8):
    print(c)
```

Chain of responsability

Chaque bout de code ne doit faire qu'une et une seule chose

Quand beaucoup d'actions complexes doivent être appliquer, il vaut mieux multiplier des petites fonctions en charge de chaque action que faire une unique grosse fonction.

```
class ContentFilter(object):
  def __init__(self, filters=None):
        self._filters = list()
        if filters is not None:
            self._filters += filters

  def filter(self, content):
        for filter in self._filters:
            content = filter(content)
        return content
filter = ContentFilter([offensive_filter, ads_filter, video_filter])
filtered_content = filter.filter(content)
```

State (ou Proxy dans la version simple)

Changer le comportement d'une fonction en fonction de l'état interne du système.
Proxy quand il n'y a pas d'état interne.

```
class Implem1:
 def f(self):
   print("Je suis f")
 def g(self):
   print("Je suis g")
 def h(self):
   print("Je suis h")

   class Implem2:
 def f(self):
   print("Je suis toujours f.")
 def g(self):
   print("Je suis toujours g.")
 def h(self):
   print("Je suis toujours h.")
```

```
class State_d:
 def __init__(self, imp):
   self._implem = imp
 def changeImp(self, newImp):
   self._implem = newImp
 def __getattr__(self, name):
   return getattr(self._implem, name)


def run(b):
  b.f()
  b.g()
  b.h()
b = State_d(Implem1())
run(b)
b.changeImp(Implem2())
run(b)
```

Decorator : très similaire à Proxy et Adaptor

Comment ajouter des fonctionnalités de manière dynamique à un objet

Exemple : tirer au but

```
class Decorator:
     def __init__(self,state):
        self.state = state
     def __getattr__(self,attr):
        return getattr(self.state,attr)
class Shoot(Decorator):
     def __init__(self,state):
         Decorator.__init__(self,state)
     def shoot(self,p):
        return SoccerAction(Vector2D(...))
class Passe(Decorator):
     def __init__(self,state):
         Decorator.__init__(self,state)
     def passe(self,p):
        return SoccerAction(Vector2D(...))

mystate = Shoot(Passe(state))
```

Decorator : peut changer le comportement d'une fonction

Exemple : modifier la passe

```
  class MeilleurPasse(Decorator):
     def petite_passe(self,p):
         return SoccerAction(...)
     def passe(self,p):
        if (condition):
           return self.petite_passe(p)
        return self.state.passe(p)
mystate = MeilleurPasse(Passe(state))
```

Strategy

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

```
class StrategyExample:
      def __init__(self,func):
         self.compute_strategy = func
       @property
       def name(self):
            if hasattr(self.func,"name"):
                return self.func.name
            return self.func.__name__
def passe(state,id_team,id_player):
    return fait_une_passe()
def cours(state,id_team,id_player):
    return cours_versr()

stratCours = StrategyExample(cours)
stratCours.name = "cours"
stratPasse = StrategyExample(passe)
stratPasse.name = "passe"
```

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
- Extraire de l'information des états
- Faire des stratégies génériques
- Réagir en fonction de situations

Quelques conseils

- Ne mélangez pas les outils, les actions et les stratégies !
  - Une classe (ou plusieurs) pour enrichir vos objets
  - Des classes (ou fonctions) pour agir
  - Des classes (ou fonctions) pour chaque stratégie
- Tout doit être générique ! Si vous décidez à un moment de changer votre façon de courir, il ne faut rien toucher à la description ou aux stratégies !
- Que faire pour les symétries ?
- N'oubliez pas d'identifier de façon unique chaque petite stratégie (par son nom).
- Vous pouvez additionner deux `SoccerAction`

Observer : un design pattern de plus

`SoccerEvents`

- Une simulation possède une liste d'observeurs (`listeners`)
- A chaque évènement marquant, tous les observeurs sont avertis par l'appel d'une fonction
- il est possible d'ajouter à la volée ou de supprimer des observeurs de la simulation.

Actions déclenchées lors d'une simulation

- `begin_match(team1,team2,state)` : au début de la simulation
- `end_match(team1,team2,state)` : à la fin
- `begin_round(team1,team2,state)` : au début de chaque engagement
- `end_round(team1,team2,state)` : à chaque but marqué
- `update_round(team1,team2,state)` : à chaque fin de tour

Comment utiliser l'observeur ?

Pour simuler !

```
class Observer(object):
        MAX_STEP=40
    def __init__(self,simu):
         self.simu = simu
         self.simu.listeners+=self #ajout de l'observer
    def begin_match(self,team1,team2,state):
         #initialisation des parametres ...
         self.last, self.cpt, self.cpt_tot = 0, 0, 0
    def begin_round(self,team1,team2,state):
         self.simu.state.states[(1,0)].position = ...
         #ou self.simu.set_state(state)
         self.strat.shoot = ...
         self.last = self.simu.step
     def update_round(self,team1,team2,state):
         if state.step>self.last+self.MAX_STEP: self.simu.end_round()
     def end_round(self,team1,team2,state):
         if state.goal>0: self.cpt+=1
         self.cpt_tot+=1
         self.res[...]= self.cpt*1./self.cpt_tot.
         if ... : #fin de la simu
             self.simu.end_match()
```

Ou hériter de `Simulation`

```
class ChallengeFonceurButeur(Simulation):
    def __init__(self,team1,max_but=20,max_steps=5*settings.MAX_GAME_S
        super(ChallengeFonceurButeur,self).__init__(team1,None,max_ste
        ...
    def begin_round(self):
        self._last_step = self.state.step
        super(ChallengeFonceurButeur,self).begin_round()
    def update_round(self):
        super(ChallengeFonceurButeur,self).update_round()
    def end_round(self):
        if self.state.goal==1: self.resultats.append(self.state.step-s
        super(ChallengeFonceurButeur,self).end_round()
    def stop(self):
        return super(ChallengeFonceurButeur,self).stop() or self.state
    def end_match(self):
        if len(self.resultats)>0: self.stats_score = sum(self.resultat
        super(ChallengeFonceurButeur,self).end_match()
    def get_initial_state(self):
        state = SoccerState()
        state.ball = Ball(Vector2D(0,0))
        state.states[(1,0)] = PlayerState(position = ...,vitesse = ...
        state.states[(2,0)] = PlayerState(position = ..., vitesse = ..
        ...
        return state
```