LU2IN013 Groupe 3

Sérialisation, Threading, Controleur et Design Pattern

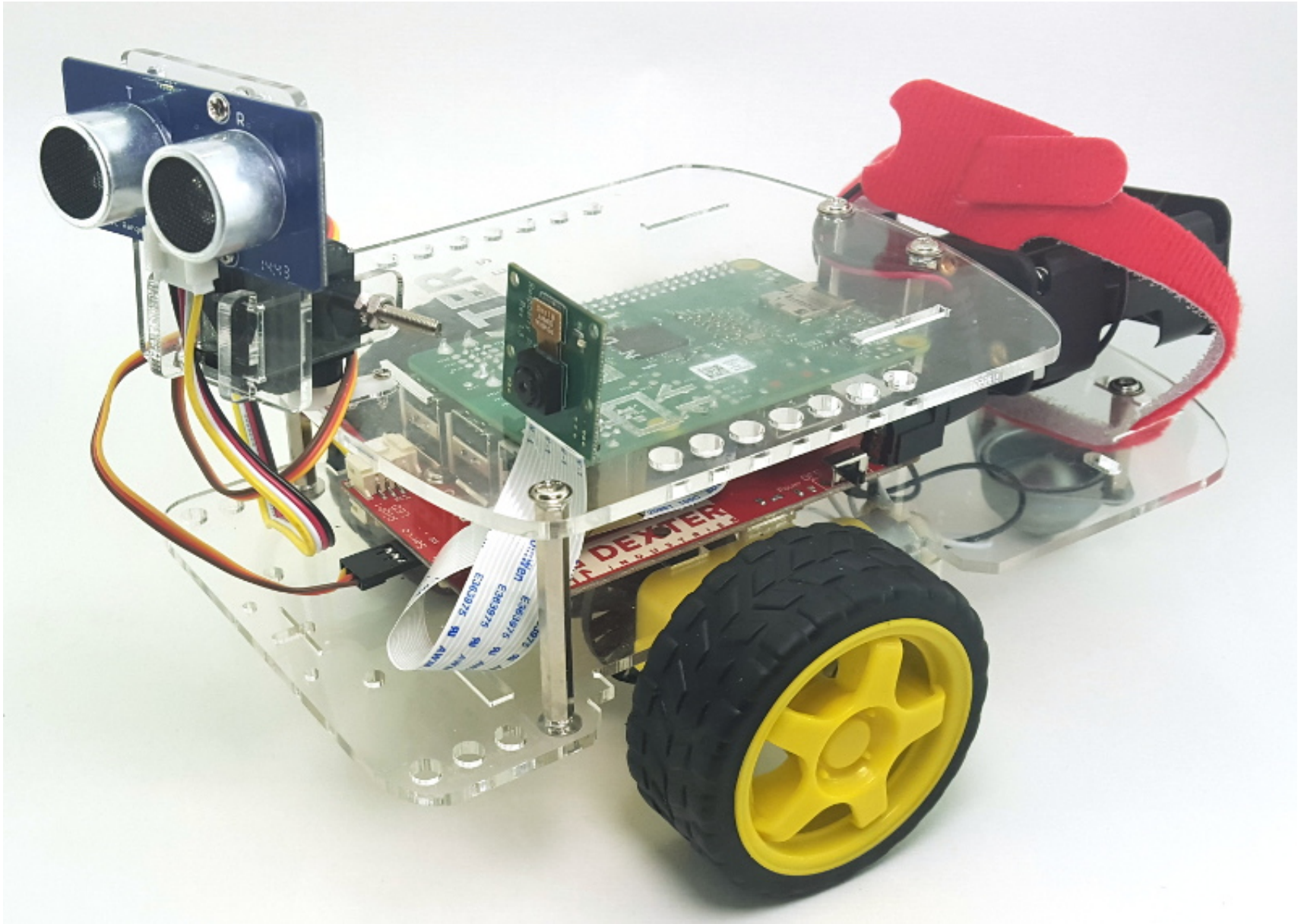
Nicolas Baskiotis
nicolas.baskiotis@sorbonne-universite.fr

équipe MLIA, Institut des Systèmes Intelligents et de Robotique
Sorbonne Université

S2 (2023-2024)

Plan

Le robot



Package `robot2I013`

Contenu :

Une classe `Robot2I013` qui contient :

- des constantes qui décrivent les propriétés physiques du robot :
  - `WHEEL_BASE_WIDTH` : écartement des roues
  - `WHEEL_DIAMETER` : diamètre des roues
- des constantes pour le contrôle du robot :
  - `MOTOR_LEFT`, `MOTOR_RIGHT` : moteurs gauche et droit
  - `LED_[LEFT|RIGHT]_[EYE|BLINKER]` : les différentes LEDs du robot
- des méthodes pour récupérer l'état du robot :
  - `get_voltage()` : état des batteries
  - `get_distance()` : distance à l'objet le plus proche
  - `get_image()` : image de la caméra
  - `get_motor_position()` : position des roues
- des méthodes pour contrôler le robot :
  - `set_led(led,r,g,b)` : donner la couleur (r,g,b) à une led
  - `set_motor_dps(port,dps)` : fixer la vitesse d'une roue
  - `offset_motor_encoder(port,offset)` : fixer l'offset d'un moteur
  - `servo_rotate(port,angle)` : tourner la tête du robot

Contrôle du mouvement

Moteurs à encodeur

Il est possible de :

- donner une vitesse de rotation au moteur
- de connaître sa position relative par rapport à une origine.

$\Rightarrow$ permet de connaître exactement le mouvement effectué par la roue.

Exemple d'utilisation

```
robot = Robot2I013()
#lit la position des moteurs
l_pos, r_pos = robot.get_motor_position()
# remet à 0 le moteur gauche puis le droit
robot.offset_motor_encoder(robot.MOTOR_LEFT,l_pos)
robot.offset_motor_encoder(robot.MOTOR_RIGHT,r_pos)
# fixe la vitesse à 50
robot.set_motor_dps(robot.MOTOR_LEFT + robot.MOTOR_RIGHT, 50)
sleep(5)
# affiche l'angle relatif de chacun des moteurs depuis la remise à 0
print(robot.get_motor_position())
```

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

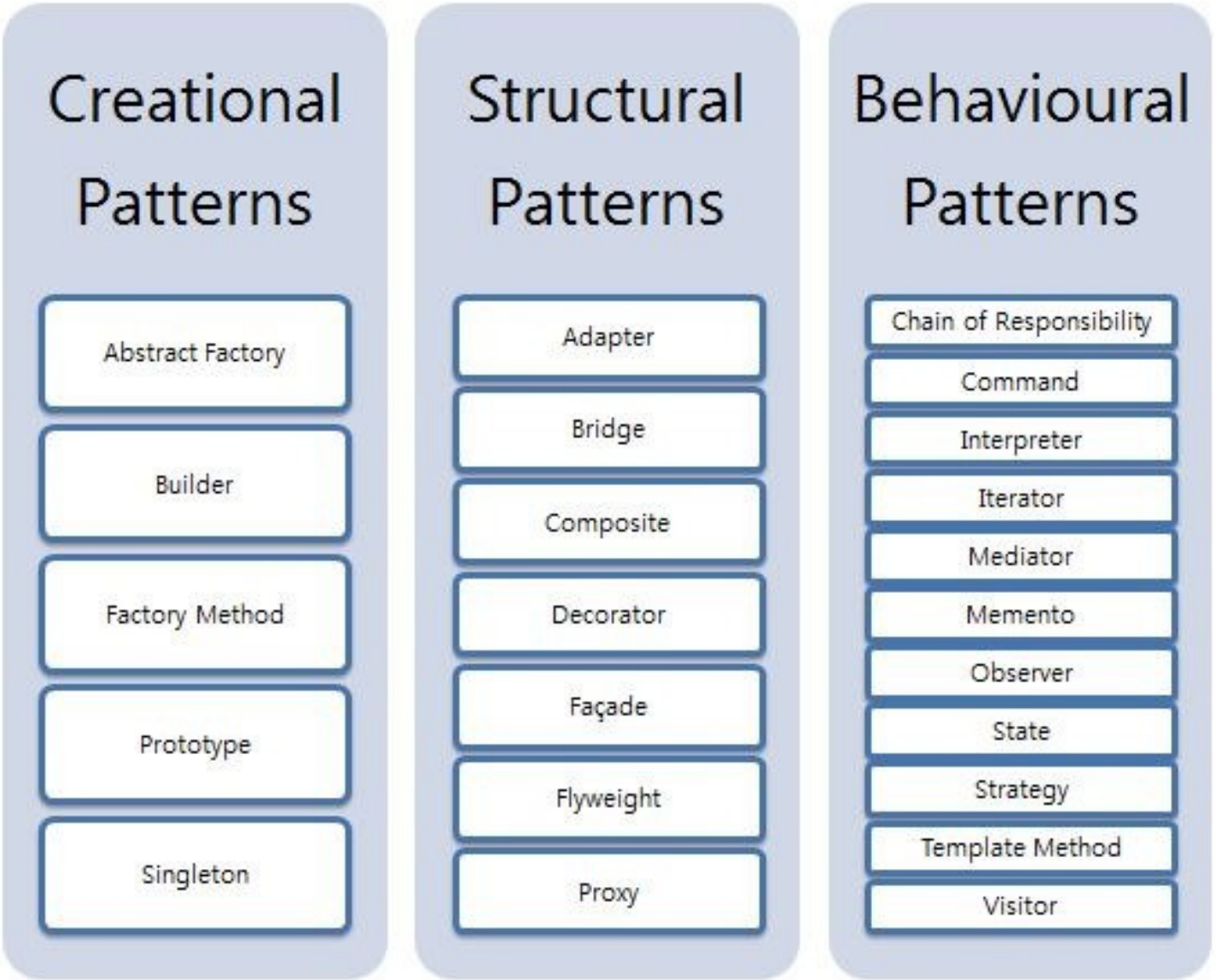
- Surtout pour les langages fortement typés, structurés (**Java** par exemple)
- Identifier les aspects de votre programme qui peuvent varier/évoluer et les séparer de ce qui reste identique
- Penser de manière générique et non pas en termes d'implémentations
- Composer plutôt qu'hériter (plus flexible)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

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 self._name
  @name.setter
  def name(self,value): self._name = value
...
a = MyClass()
print(a.name) # plutot que a.name()
a.name="toto" #plutot que a.set_name("toto")
```

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.quack()
  moi.quack()
  autre.quack()
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 self._obj.__getattribute__(attr)

moi = PersonneAdapter(Personne())
moi.quack()
```

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

```
class Decorator:
     def __init__(self,robot):
        self.robot = robot
     def __getattr__(self,attr):
        return getattr(self.robot,attr)
class Avance(Decorator):
     def __init__(self,state):
         Decorator.__init__(self,robot)
     def avance(self):
        return ...
class Tourne(Decorator):
     def __init__(self,state):
         Decorator.__init__(self,robot)
     def tourne(self):
        return ...
robot = Tourne(Avance(robot)) # tout dans robot accessible
# donne acces a robot.tourne() et robot.avance()
```

Decorator : peut changer le comportement d'une fonction

Exemple : modifier la manière d'avancer

```
class AvancerAuPas(Decorator):
      def aupas(self):
          return ...
      def avancer(self):
         if (condition):
            return self.aupas()
         return self.avancer()
robot = AvancerAuPas(Avancer(robot))
```

Strategy

Le pattern Strategy 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.update = func
  @property
  def name(self):
      if hasattr(self.func,"name"):
          return self.func.name
      return self.func.__name__
def avanceVite(state):
    return ...
def avanceLentement(state):
    return ...
stratVite = StrategyExample(avanceVite)
stratVite.name = "vite"
stratLent = StrategyExample(avanceLentement)
stratLent.name = "lent"
```

```
class Robot:
    def __init__(self,strat):
      self.strat = strat
      self.state = ...
    def update(self):
      return self.strat.update(
          self.state)

def strat_complexe(state):
    if ...:
        return stratLent(state)
    return stratVite(state)
```

Plan

Retour sur les Design patterns

Problèmes :

- vous voulez parler ”simplement” à votre robot (*avance*, *tourne*)
- votre robot ne comprend que `set_motor_dps`
- Capteurs bruités : `get_distance` pas précis

Solutions

- Adapter : quand vous avez deux APIs qui n’ont pas les mêmes noms de méthode mais font plus ou moins la même chose
- Decorator : ajouter des fonctionalités à un objet
- Facade : rendre plus simple les appels à un/plusieurs sous-sytèmes
- Bridge : proche de l’adapter, mais rend abstrait l’implémentation de la ”traduction”; permet de jongler entre plusieurs implémentations.

Solution Adapter

```
class Adapter:
  def __init__(self,robot):
    self.robot = robot
  def forward(self,speed):
    self.robot.set_motor_dps(self.robot.MOTOR_LEFT+\
            self.robot.MOTOR_RIGHT,speed)
  def turnRight(self,speed):
    self.set_motor_dps(self.robot.MOTOR_LEFT,speed)
  def get_distance(self):
    return self.robot.get_distance()
....
```

Problèmes :

Solution Adapter

```
class Adapter:
  def __init__(self,robot):
    self.robot = robot
  def forward(self,speed):
    self.robot.set_motor_dps(self.robot.MOTOR_LEFT+\
            self.robot.MOTOR_RIGHT,speed)
  def turnRight(self,speed):
    self.set_motor_dps(self.robot.MOTOR_LEFT,speed)
  def get_distance(self):
    return self.robot.get_distance()
....
```

Problèmes :

- Pas flexible
- Pas possible de tester diverses implémentations
- Toujours même façon d'avancer quelque soit l'action

Décorateur

Vous avez envie de rajouter des fonctionalités :

- Pouvoir logger les actions du robot
- Lisser les résultats de `get_distance()`
- Avoir une mémoire

Solution possible : décorateur

```
class Lisser:
  def __init__(self,obj,size=5):
    self._obj = obj
    self.hist =[0]*size
    self.cpt = 0
  def __getattr__(self,name):
    self.hist[self.cpt]=self._robot.get_distance()]
    self.cpt= (self.cpt+1) %len(self.hist)
    return getattr(self._robot,name)
  def get_distance(self):
      return sum(self.hist)/len(self.hist)
```

Décorateur

Vous avez envie de rajouter des fonctionalités :

- Pouvoir logger les actions du robot
- Lisser les résultats de `get_distance()`
- Avoir une mémoire

Solution possible : décorateur

```
class LogAction:
    def __init__(self,obj):
      self._obj = obj
      self.log =[]
    def __getattr__(self,name):
      if name in ["get_distance","set_motor_dps"]:
        self.log.append(name)
      return getattr(self._robot,name)
```

Avantage : on peut mixer les décorateurs :

```
monobjet = LogAction(Lisser(objet))
```

Stratégie : mauvaise solution

Stratégie synchrone

```
class TracerCarre:
  def __init__(self,distance,...):
    self.distance = distance
  def run(self):
    parcouru = 0
    while parcouru<self.distance:
      self.robot.set_motor_dps(...)
    # Tourner ...
    # ....

strategie = TracerCarre()
strategie.run()
```

Problèmes

- Commandes non ré-utilisables
- Stratégies non flexibles (et si on veut faire un triangle ? un hexagone ?)
- mais surtout, non réactive, très dangereuse !
  si un mur ou un obstacle et sur le chemin ?

$\Rightarrow$ Possibilité d'intégrer des conditions dedans, mais le code devient horrible.

Stratégie : bonne solution

Stratégie asynchrone

```
class AvancerDroit:
  def __init__(self,distance,...):
    self.distance = distance
  def start(self):
    self.parcouru = 0
  def step(self):
    self.parcouru += ...
    if self.stop(): return
    self.avancer()
  def stop(self):
    return self.parcouru>self.distance
strategie = AvancerDroit()
while not strategie.stop():
    strategie.step()
    sleep(1./update_time)
```

Avantages :

- Flexibilité
- Ré-utilisation
- Complexification en mixant des stratégies élémentaires

Stratégie

Principe: découper les actions de bases en petits blocs

- Aller tout droit sur une certaine distance
- Tourner d'un certain angle
- Approcher un point

Puis imbriquer les stratégies entre elles : une meta-stratégie

Nécessite :

- initialiser une stratégie
- savoir quand elle est finie
- savoir où en est (asynchrone !!!)

Mixer les stratégies

Stratégie conditonnelle

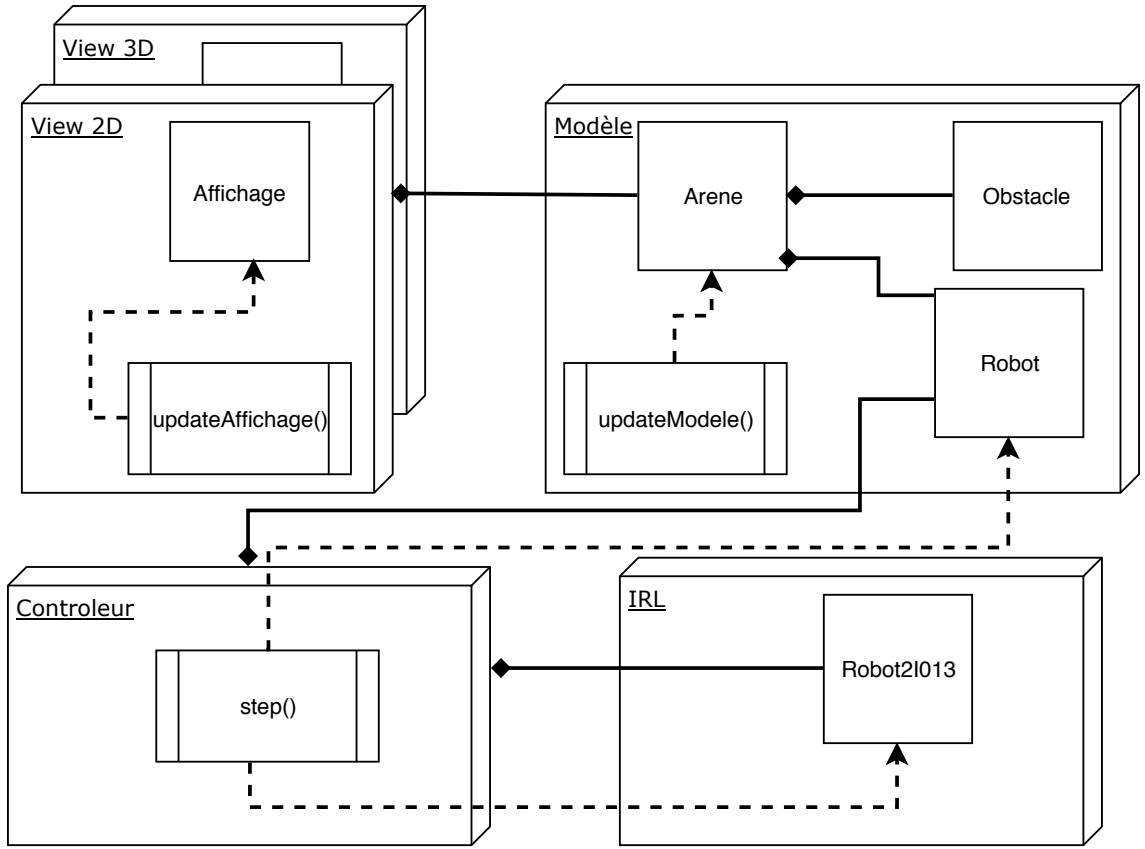
```
    def step(self):
      if loin: self.avanceVite.step()
      else: self.avanceLentement.step()
```

Stratégie séquentielle

```
def __init__(self):
  self.strats = [stratDroit, stratTourne,stratDroit,stratTourne]
  self.cur= -1
def start(self):
  self.cur = -1
def step(self):
  if self.stop():return
  if self.cur <0 or self.strats[self.cur].stop():
    self.cur+=1
    self.strats[self.cur].start()
  self.strats[self.cur].step()
def stop(self):
  return self.cur== len(self.strats)-1\
      and self.strats[self.cur].stop()
```

Plan

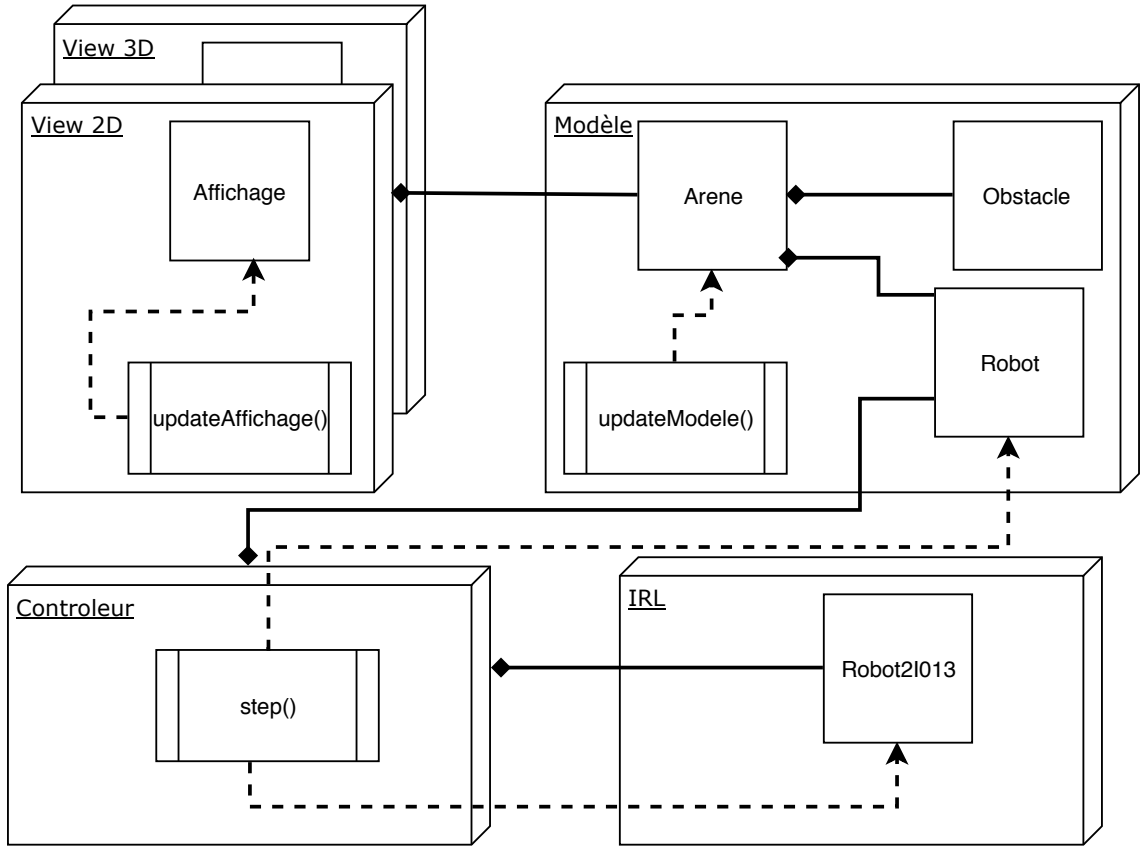
Etat des lieux



Modules indépendants

- Vue : un `update` pour maj de l'affichage
- Modèle : un `update` pour la maj du monde virtuel
- Contrôleur : un `step` pour les ordres au robot
- IRL : rien . . .

Etat des lieux



Contraintes

- `updateModele` (au moins) plus souvent que `step`
- `updateAffichage` (au moins) plus souvent que `step`
- un script (presque) commun à l'IRL et à la simulation

Organisation code (exemple)

```
arene = None
affichage = None
robot = None
from monprojet import Controleur
try:
  from robot2I013 import Robot2I013 as Robot
  robot = Robot()
except ImportError:
  from monprojet import MonRobot as Robot
  from monprojet import Arene,affichage
  robot = Robot()
  arene = Arene(robot)
  affichage = Affichage(arene)

....
ctrl = Controleur(...)
```

Solution naïve

```
def runCtrl(ctrl,fps=100):
    while True:
      ctrl.step()
      if arene is not None:
        arene.update()
      if affichage is not None:
        affichage.update()
      time.sleep(1./fps)
```

Mais . . .

- les `update` ne sont pas indépendants
- mélange dans la boucle de IRL et simulé, pas idéal

Solution : Thread

Thread

- un thread = bout de code qui s'exécute en parallèle
- Attention : délicat dans le cas général (programmation concurrente) :
  - Cas de variables partagées entre thread
  - Cas des ressources partagées (fichiers, connections)
- pour faire un thread : soit appel de la classe `Thread` avec en paramètre `target` la fonction à exécuter; soit héritage de la classe `Thread` et redéfinition de `run`.

```
from threading import Thread

class Affichage:
  ....
  def boucle(self,fps):
    while True:
      self.update()
      time.sleep(1./fps)
affichage = Affichage(...)
threadAff = Thread(target=affichage.boucle,args=(fps,))
thread.start()
```

Solution : Thread

Thread

- un thread = bout de code qui s'exécute en parallèle
- Attention : délicat dans le cas général (programmation concurrente) :
  - Cas de variables partagées entre thread
  - Cas des ressources partagées (fichiers, connections)
- pour faire un thread : soit appel de la classe `Thread` avec en paramètre `target` la fonction à exécuter; soit héritage de la classe `Thread` et redéfinition de `run`.

```
from threading import Thread

class Affichage(Thread):
    def __init__(self,...):
      super(Affichage,self).__init__()
    def run(self,fps):
      while True:
        self.update()
        time.sleep(1./fps)
affichage = Affichage(...)
affichage.start()
```

Quelques autres objets pour les threads

- la méthode `join` d'un thread permet d'attendre la fin du thread
- la méthode `setDaemon(true)` permet de rendre le thread indépendant de la fin du programme principal
- l'objet `RLock` permet de synchroniser les threads :

```
lock = RLock()
...
#dans affichage
def updateAffichage():
  with lock: #bloque execution des autres threads qui testent lock
    dessine(robot)

#dans modele
def updateModele():
  with lock:  #est bloque qd updateAffichage s'execute
      updatePosition(robot)
```