JSON & MVC

Vincent Guigue
UPMC - LIP6

Sérialisation

Principe

- Pouvoir stocker/transférer un objet ...
- ...et pouvoir le reconstruire possiblement dans un autre environnement (autre système d'exploitation, autres versions, ...)
- L'objet reconstruit doit être un **clone** sémantique de l'objet initial

$\Rightarrow$ Transformer un objet en une séquence de bits (sérialisation) et pouvoir reconstruire l'objet à partir de cette séquence de bits (désérialisation).

En python : Module `Pickle`

- Méthode native de python
- Adapté pour des objets complexes (composés d'autres objets, références récursives, . . .)
- différents *protocoles* :
  - 0 : format *human-readable*
  - 2 : binaire, par défaut en python 2
  - 3 : binaire compressé, par défaut en python 3, non rétro-compatible
- Avantages : simple à utiliser, sérialise beaucoup d'objets (structures de base mais aussi fonctions, classes)
- Inconvénients : parfois lourd, propre à python.

Exemple :

```
1  import pickle
2  with open('data.pkl','wb') as f:
3      pickle.dump(monObjet,f)
4  with open('data.pkl','rb') as f:
5      monObjet = pickle.load(f)
```

JSON : JavaScript Object Notation

Format JSON

- Format de fichiers ouvert, en texte clair, standard, très répandu
- Encodage par le biais de dictionnaires clé-valeur qui peuvent contenir les types natifs suivants :
  - `Nombre` : entier ou réel
  - `String` : séquence de caractère unicode
  - `Boolean` : `true` ou `false`
  - `Array` : une séquence ordonnée de valeurs, les types peuvent être mixés
  - `Object` (ou dictionnaire) : ensemble non ordonné de couples clé/valeur

```
1  { "type" : "Arene",
2    "dimension" : [100, 200],
3    "objets" : {
4      "premier": { "type" : "Cube", "position" : [[0, 0],[0, 1]]},
5      "second": { "type" : "Robot", "position" : [ 0.5, 0.5 ] }
6    }
7  }
```

JSON et Python

Module json natif mais n'encode que les types de base

Types : `dict`, `list`, `string`, `int`, `long`, `boolean`
ne permet pas d'encoder nativement un objet !!

```
>>>import json
#json.dumps -> string, json.dump -> fichier
>>>json.dumps(['foo',{'bar':('baz', None, 1.0, 2)}])
'["foo",{"bar":["baz",null,1.0,2]}]'
>>>json.loads('["foo",{"bar":["baz",null,1.0,2]}]')
[u'foo', {u'bar': [u'baz', None, 1.0, 2]}]
```

Pour un objet Python

- Tous les attributs de l'objet sont dans la variable `__dict__`
- la classe d'un objet est dans la variable `__class__.__name__`

```
class A(object) :
    def __init__(self):
        self.a=1; self.b = "c'est moi";self.c =[1,True,"dix"]
print(A().__dict__, A().__class__.__name__)
# -> {'a': 1, 'b': "c'est moi", 'c': [1, True, 'dix']}, 'A'
```

Solution simple (mais incomplète)

```python
import json
class A(object):
    def __init__(self,a=1,b="moi",c=[1,True,"dix"]):
        self.a,self.b,self.c = a,b,c
a = A()
aserial = json.dumps(a.__dict__)
# -> '{"b": "moi", "c": [1, true, "dix"], "a": 1}'
## **kwargs permet de passer le dictionnaire kwargs comme argument
newa = A(**json.loads(aserial))

def myencoder(obj):
    dic = dict(obj.__dict__)
    dic.update({"__class":obj.__class__.__name__})
    return json.dumps(dic)
def mydecoder(s):
    dic=json.loads(s)
    cls = dic.pop("__class")
    return eval(cls)(**dic)
mydecoder(myencoder(a))
# -> <__main__.A at 0x7f8ec872f668>
```

⇒ **Toutes les caractéristiques de l'objets doivent être données en argument du constructeur !**

Problème : objet composé d'autres objets ...

```
1 class B(object):
2     def __init__(self,autre):
3         self.a = A()
4         self.autre = autre
5 b=B(12)
6 myencoder(b)
7 # -> TypeError: <__main__.A object at 0x7f8ec86c4b70>
8 #       is not JSON serializable
```

Solution : paramètres `default`/`object_hook` (ou hériter de `JSONEncoder` et `JSONDecoder`)

- `default(obj)` : méthode qui encode un objet; si l'objet n'est pas natif, cette méthode est appelée, elle doit sérialiser son dictionnaire et ajouter le nom de la classe.
- `object_hook(s)` : méthode qui est appelée avec chaque dictionnaire désérialisé avant le retour.

Solution complète (1)

En cas de problème, `dumps` appelle la fonction: `default`
Cette fonction: `obj` $\rightarrow$ `dict`

```python
import json

class A(object):
    def __init__(self,a=1,b="moi",c=[1,True,"dix"],d={1:2,"a":True}):
        self.a,self.b,self.c = a,b,c

class B(object):
    def __init__(self,autre):
        self.autre = autre

def my_enc(obj):
    dic = dict(obj.__dict__)
    dic.update({"__class":obj.__class__.__name__})
    return dic

# Usage
b = B(A())
bserial = json.dumps(b,default=my_enc)
# -> '{"__class": "B",
#     "autre": {"b": "moi", "a": 1, "c": [1, true, "dix"], "__class": "A"}}'
```

Solution complète (2)

Chargement:

```
## pour rappel:
import json

class A(object):
    def __init__(self,a=1,b="moi",c=[1,True,"dix"],d={1:2,"a":True}):
      self.a,self.b,self.c = a,b,c

class B(object):
    def __init__(self,autre):
      self.autre = autre

## Chargement

def my_hook(dic):
  if "__class" in dic:
    cls = dic.pop("__class")
    return eval(cls)(**dic)
  return dic

# Usage
b = B(A())
bserial = json.dumps(b,default=my_enc)
b = json.loads(bserial,object_hook=my_hook)
```

Dernier problème

Les références circulaires:

- `A` est composé de `B`...
- ... et `B` référence `A`
  $\Rightarrow$ Récurrence infinie
- Ca peut arriver avec 3 objets (plus dur à voir)
- Dans la vraie vie:
  - les cubes qui composent l'arène référencent parfois l'arène
  - le robot a une tête, qui garde une référence vers le robot (pour se localiser)
  - ...

Autres usages

Le JSON est un standard du web... Si vous voulez un programme qui

- récupère des résultats sur google, google map, twitter, vélib...
  $\Rightarrow$ Gestion du JSON

Un exemple de récupération de récupération d'un historique météo sur Roissy-Charles de Gaulle:

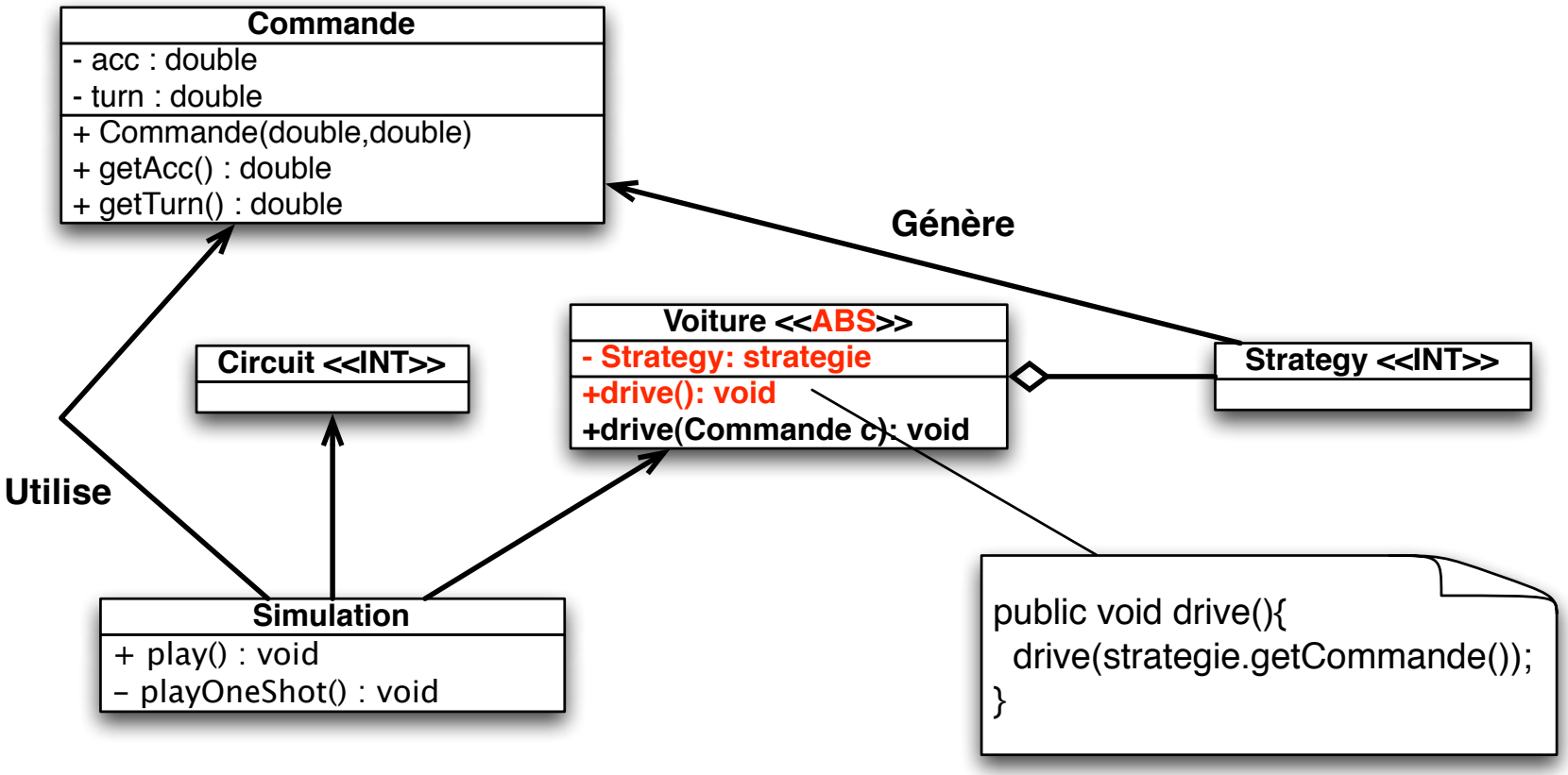
```
1  import urllib2
2  import json
3
4  # cible:
5  #http://api.wunderground.com/api/XXX/history_20081031/q/CDG.json
6  key = "XXXX" # a récupérer sur le site
7  date = "history_20081031"
8  place = "CDG"
9
10 f = urllib2.urlopen("http://api.wunderground.com/api/"+key+"/"+date+"/q/"+p
11 json_string = f.read() # récupération au format JSON
12
13 parsed_json = json.loads(json_string)
14 obs = parsed_json['history']['observations'] # manip en dico
15
16 ...
17 f.close()
```

JSON

MVC

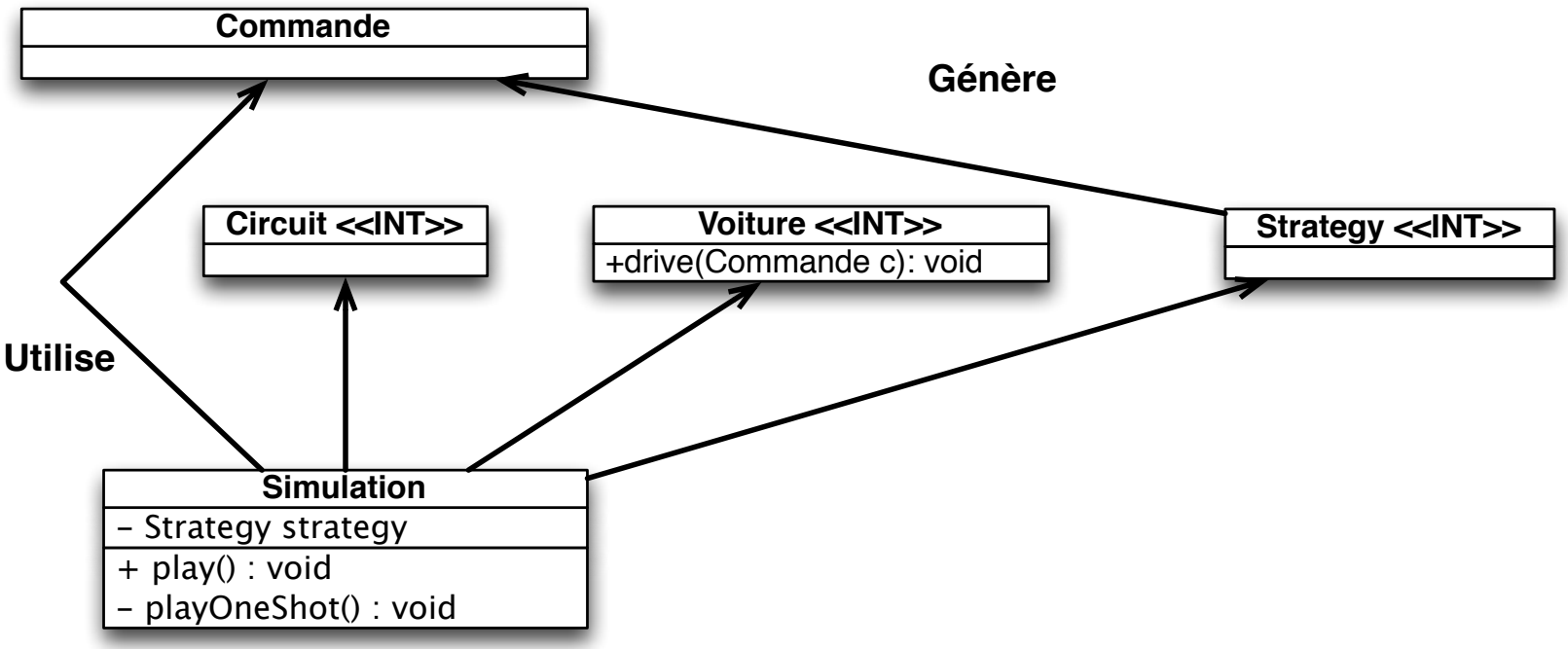
Rendre le robot intelligent

2 possibilités: Voiture ou Simulation



Rendre le robot intelligent

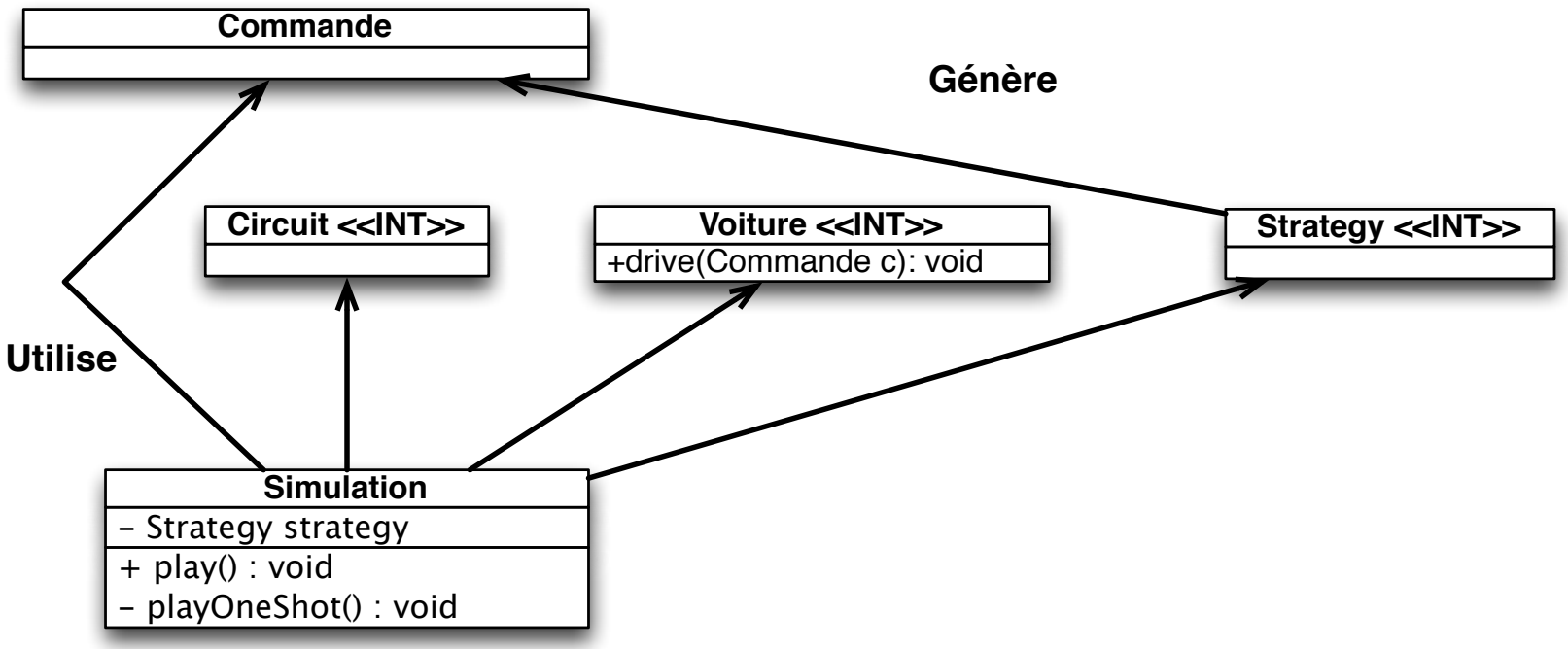
2 possibilités: Voiture ou Simulation



Comment gérer l'affichage?

- Le main gère les paramètres...
- sauvegardes / chargements

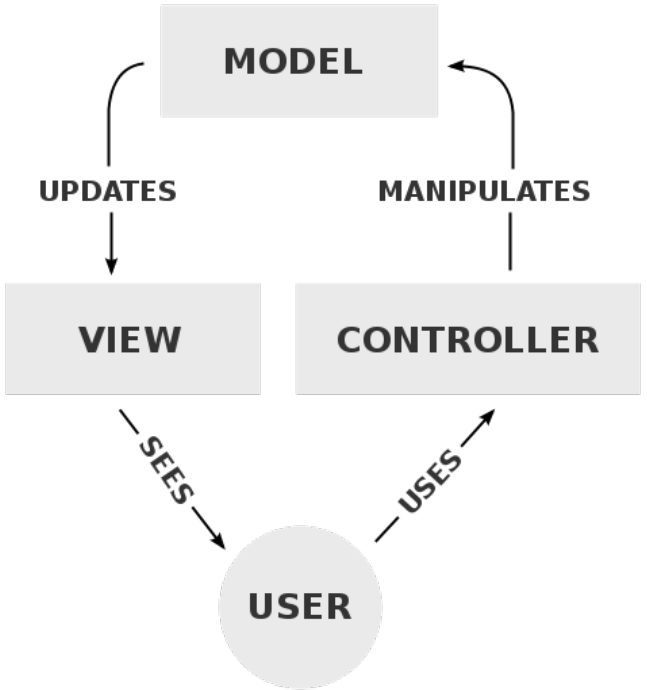
- Les informations utiles pour l'affichage sont présentes dans la simulation...



Vers un système indépendant

- On veut un système indépendant
  - Pouvoir le brancher ou le débrancher facilement, sans intervenir dans le code de la simulation
  - Pouvoir changer de système de visualisation
- Il existe un modèle standard (assez lourd) pour gérer cette situation: MVC Model View Controller

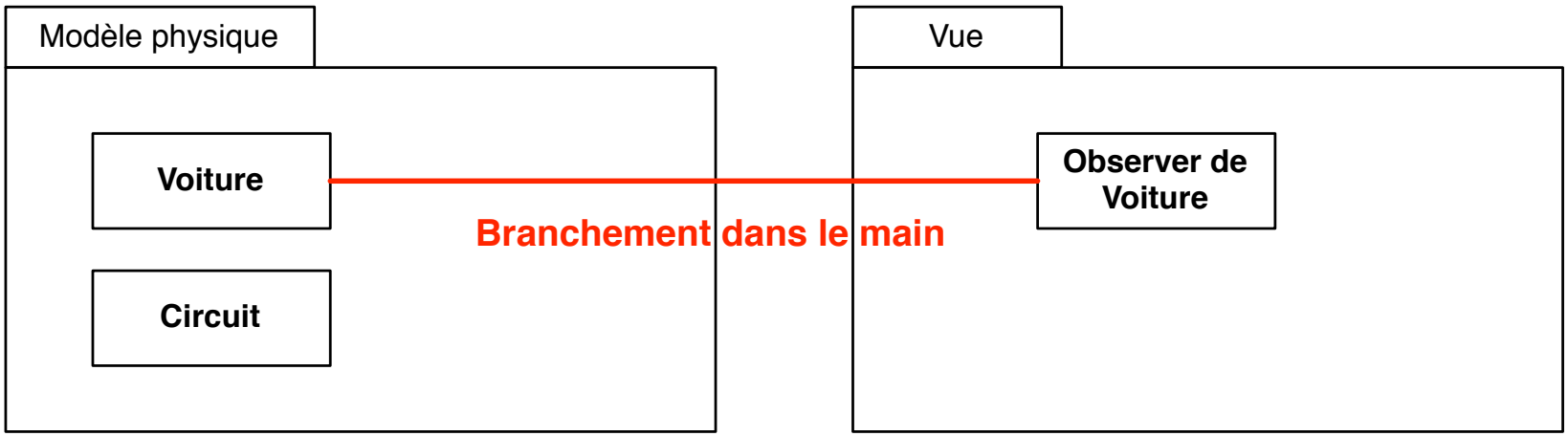
- Pour chaque élément (voiture, circuit,...), un objet vue est créé.
- Une vue générique est élément capable de gérer la vue d'un objet (cf slide suivant)
- Lorsque le modèle change il informe le Controller (évènement)
- Le Controller met à jour les informations d'affichage



credit: wikipedia

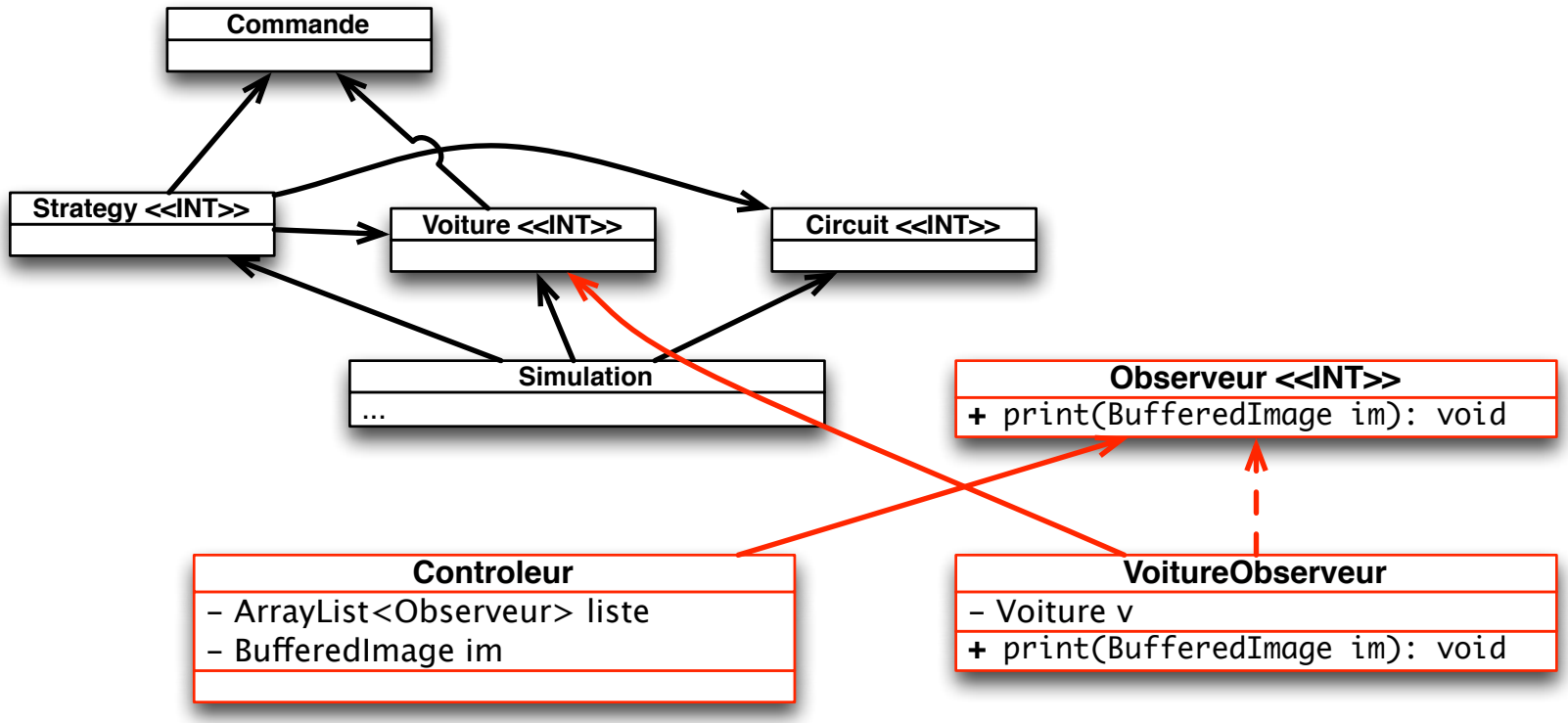
Proposition de base

- Information toujours stockées dans une image
- On travaille sur un système de branchement universel depuis le main
  - On pourra brancher un autre système plus tard
- Modèle physique:
  - Aucune référence à l'affichage
  - Modèle autonome: peut fonctionner sans affichage
- Visualisation
  - Branchement depuis le main



Observer

- Le code sera développé dans un nouveau package: `li260.view.observeurs`
- VoitureObserveur sait afficher la voiture dans l'image
- Le système d'affichage est découplé du modèle: on peut le débrancher ou en mettre un autre... Depuis le main



Une vue qui regroupe les éléments à afficher

Dans un cadre tkinter:

```
1  from tkinter import *
2
3  class Vue(Tk): # extension d'une fenetre
4      def __init__(self):
5          Tk.__init__(self)
6          self.toDraw = []
7          # frame 1
8          self.can = Canvas(self, width=500, height=500, background='blue')
9          self.can.pack(side = LEFT, padx = 5, pady = 5)
10
11     def draw_scene(self):
12         self.can.delete("all")
13         for obj in self.toDraw:
14             obj.draw(self.can)
15
16     def add(self, obj):
17         self.toDraw.append(obj)
```

Qu'est ce qu'un observeur de robot?

Etant donné un `canvas` tkinter et un robot **très** basique

- Position `_x`,`_y`, Direction `_dx`,`_dy`
- Fonction d'interaction unique:

```
    def avance(self):
        self._x += self._dx
        self._y += self._dy
```

```
from model.Robot import *
from tkinter import *


class RobotVue:
    def __init__(self, robot):
        self.rob = robot

    def draw(self, can):
        can.create_rectangle(self.rob._x, self.rob._y,\
         self.rob._x + self.rob._larg, \
         self.rob._y + self.rob._haut)
```

Usage depuis le main

```
1  from tkinter import *
2  from model.Robot import Robot
3  from vue.Vue import Vue
4  from vue.RobotVue import *
5
6  fen = Vue()
7  r = Robot()
8  rvue = RobotVue(r)
9  fen.add(rvue)
10
11 def update():
12     r.avance()
13     fen.draw_scene()
14     print(r._x, r._y)
15     fen.after(20, update) # planification récursive
16
17 fen.after(20,update) # lancement de la boucle de planif
18 fen.mainloop() # fenetre
```

Pourquoi cette architecture est intéressante?

- EVOLUTIVITE :
  - Si on veut changer la représentation de la voiture, c'est facile
    - Exemple: changer la couleur de la trace
  - Changer la représentation ne modifie pas la simulation
  - Changer la simulation ne modifie pas la représentation

- LIMITE :
  - Les observeurs forment un ensemble cohérent avec le controleur: il faut les faire évoluer ensemble
  - Il faut encore gérer la dynamique: les observers doivent être appeler régulièrement!