Bachelier en Informatique de Gestion

Programmation Orientée Objet

Enseignement supérieur économique de type court

Code FWB : 7525 21 U32 D3

Code ISFCE : 4IPO3

= BUROTIX ()

Table des matières

= BUROTIX ()

Partie 17
Programmation Orientée Objet
"Patron de Conception" (Design Pattern)

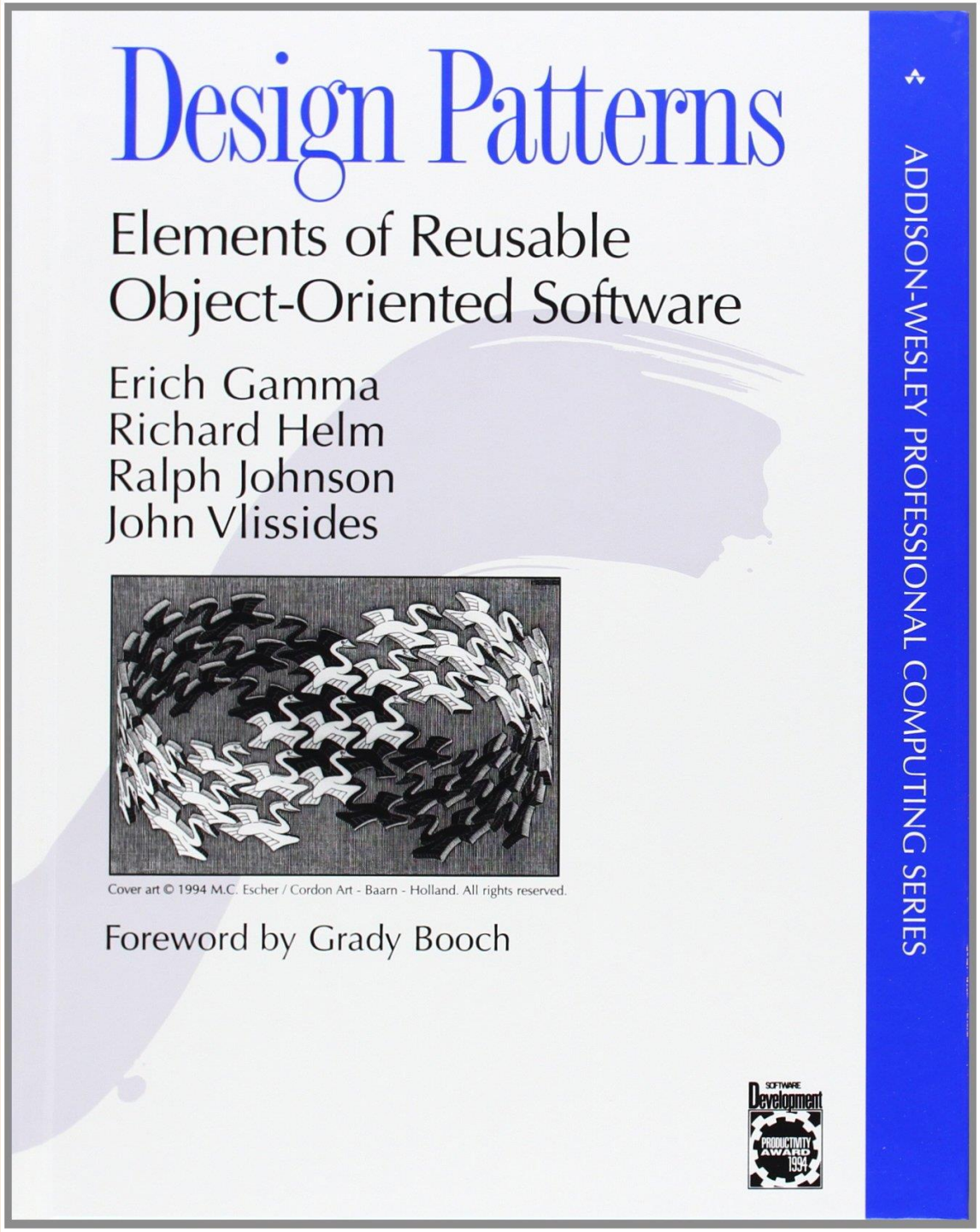
Syllabus & Exercices

17. POO Design Pattern

- 17-01 Decorator
- 17-02 Factory
- 17-13 Iterator
- 17-16 Event & Observer
- 17-19 Singleton

Design Pattern : références

- "Design Patterns – Elements of Reusable Object-Oriented Software"
- By the *Gang of Four*



Design Pattern : références

- https://refactoring.guru/design-patterns

Design Pattern : définition

- concept de génie logiciel
- résolution de problèmes récurrents de conception logicielle
- formalisation de bonnes pratiques
  - issu de l'expérience des concepteurs de logiciels
  - vocabulaire commun entre les intervenants (concepteur, programmeur, testeurs, ..)
- capitalisation de l'expérience
  - solution standard et (ré)utilisable
  - grandes lignes d'une solution, adaptable

= BUROTIX ()

Design Pattern : pro & con

PRO

- clarté du code
- réduction des interactions entre objets
  - maintainabilité accrue
- diminution du temps de développement
  - si appliqué judicieusement

CON

- suringénierie
  - complexité du code
  - or : agile => simplicité
- langages modernes intégrant de base certains DP

Design Pattern : familles

- Créateurs
  - instanciation et configuration des classes et des objets
- Structuraux
  - organisation des classes d'un programme dans une structure plus large
- Comportementaux
  - Organisation des objets en vue d'une collaboration

Les design patterns du *gang of four*

1. Décorateur
2. Fabrique
3. Fabrique abstraite
4. Adaptateur
5. Pont
6. Monteur
7. Chaîne de responsabilité
8. Commande
9. Composite
10. Façade
11. Flyweight
12. Interpreter
13. Iterator
14. Mediator
15. Memento
16. Observer
17. Prototype
18. Proxy
19. Singleton
20. Strategy
21. Template method
22. Visitor

= BUROTIX ()

Chapitre 17-01 : Décorateur

Design Pattern comportemental

Design Pattern : Decorator

- Utilité : Ajouter des fonctionnalités à un objet.
  - Par ex. lors du développement d'une nouvelle version d'un module
- Il faut savoir qu'en Python
  - Une fonction peut être passée en argument à une autre fonction.
  - On peut définir une fonction à l'intérieur d'une autre fonction.
  - Une fonction peut retourner une autre fonction.
- Décorateur en Python
  - Fonction recevant une autre fonction en argument
  - Comportement de la fonction argument étendu par le décorateur
- Syntaxe : `@decorator`

Design Pattern : Decorator

- Exo 17-01-05
- Référence
  - https://www.tutorialsteacher.com/python/decorators

Decorator : `@classmethod`

- Utilité : Pouvoir appeler une méthode **`MethodName`** sans créer l'objet de classe **`ClassName`**
  - Syntaxe pour la déclaration

```
@classmethod
def MethodName():
...
```

  - Syntaxe pour l'appel

```
ClassName.MethodName()
```

- Remarque
  - alternative à la fonction **`classmethod()`** (obsolète)

Decorator : `@classmethod`

- Décorateur d'une méthode de classe, p.ex. `MethodName`
  - Souvent d'une classe abstraite
- Premier paramètre : `cls`
  - utilisé pour accéder aux attributs de classe
- Accès uniquement aux attributs de classe
  - non aux attributs d'instance
- `MethodName` peut retourner un objet de la classe.
  - Application remarquable à certains design patterns : Factory, Singleton, …

Decorator : `@classmethod`

- Exo 17-01-07, 17-01-08
- Remarques:
  - Les `@classmethod` ne peuvent manipuler que les attributs de classe, pas les attributs d'objet.
- Référence
  - https://www.tutorialsteacher.com/python/classmethod-decorator

Remarque : décorateurs
`@classmethod` vs. `@staticmethod`

`@classmethod`

- 1er argument : cls
- accès en lecture et en écriture aux attributs de la classe
- application
  - modification des attributs de la classe
  - générateur d'objet (cf design pattern "factory")

`@staticmethod`

- pas de 1er argument spécifique
- aucun accès aux attributs de la classe
- application :
  - tâche utilitaire
  - calcul

https://www.tutorialspoint.com/class-method-vs-static-method-in-python

= BUROTIX ()

Remarque : décorateurs

@classmethod vs. @staticmethod

```
class Employee:
   __min_age = 25

   def __init__(self, name, age):
      self.name = name
      self.age = age

   @staticmethod
   def isAdult(age):
      if age > 18:
         return True
      else:
         return False
```

```
@classmethod
def factory(cls, name, year):
   current_age = today().year - year
   if cls.isAdult(current_age)
      and current_age >= cls.__min_age:
      return cls(name,current_age)

@classmethod
def set_age_to_work(cls, new_age):
   cls.__min_age = 18
```

Chapitre 17-02 : Factory

Design Pattern créateur

Design Pattern : Factory

- Utilité : Créer une instance à partir d'une famille de classes.
  - Isolement du processus de création d'un objet
  - En pratique, quand la création d'un objet et le choix de la classe adaptée est complexe, par ex. quand elle dépend de nombreux paramètres
  - Code simplifié, réutilisable et facile à entretenir.

Design Pattern : Factory

- Implantation
  - Via la méthode `factory()`
  - Elle crée l'objet sur base
    - des paramètres fournis
    - de l'algorithme contenu dans la méthode
  - Elle retourne l'objet.
  - Méthode statique, appelable depuis la classe parente
    - `@classmethod`

= BUROTIX ()

Exo 17-02-05 : translator (sample)

```
class Translator:

    @classmethod
    def factory(cls, language="English"):
        """Factory Method"""
        …
        return a_given_object

# MAIN
f = Translator.factory("French")
```

Exo 17-02-13 : quadrilatère

- Partez du fichier `17-02-13_quadrilateres_start`
  - La fonction `dump` est fournie.
- Implantez `factory`. Principe : `factory` retournera un quadrilatère, un losange ou un parallélogramme en fonction des valeurs de la liste fournie par l'utilisateur (input).

```
shape1 = Quadrilatere.factory([ 10, 15, 20, 25 ])
dump(shape1, "*** shape1") # quadrilatère

shape2 = Quadrilatere.factory([ 5, 5, 5, 5 ])
dump(shape2, "*** shape2") # losange

shape3 = Quadrilatere.factory([ 5, 15, 5, 15 ])
dump(shape3, "*** shape3") # parallélogramme
```

= BUROTIX ()

Exo 17-02-14 : formes géométriques

- Partez du fichier `17-02-14_shapes_start`
- Implantez les méthodes et le factory répondant à l'usage donné.
- Les paramètres (longueurs, etc.) sont donnés en input par l'utilisateur.

```
shape_name = input("Enter the name of the shape: ")
shape = Shape.factory(shape_name)

print("The type of object created: {}"
      .format(type(shape)))

print("The area of the {} is {}:"
      .format(shape_name, shape.calculate_area()))

print("The perimeter of the {} is: {}"
      .format(shape_name, shape.calculate_perimeter()))
```

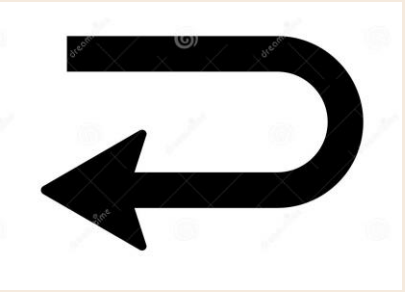
Chapitre 17-13 : Iterator

Design Pattern comportemental

Préalable : Generator

- Variante d'un itérateur classique (`for, while,` …)
- Mais pas de stockage en mémoire des éléments
- Dans une fonction : utiliser l'instruction `yield` ("céder") et non `return`
- `Return`
  - Retourne un ensemble de valeurs
  - La fonction est terminée
  - L'ensemble des valeurs a dû être mis en mémoire
- `Yield`
  - Retourne une valeur
  - La fonction continue
  - La fonction retourne toutes les valeurs par "paquets", itérativement.

https://realpython.com/introduction-to-python-generators/



Préalable : Generator : Exo 17-13-01 sample

Défi : On veut se procurer des nombres pairs, sans se préoccuper de les calculer.

```
def infinite_sequence():
  num = 0
  while True:
    yield num
     num += 2

for i in infinite_sequence():
  print(i, end=" ")
```

Ou

```
gen = infinite_sequence() # gen : generator
next(gen) # 0
next(gen) # 1 …
```

- Utile pour le debug en console

= BUROTIX ()

Préalable : Generator : Exo 17-13-02 application à la lecture d'un gros fichier

```
# opening file using return
# => tout le contenu est retourné en un coup
def csv_reader_1():
  file = open(fn)
  result = file.read().split("\n")
  return result

# opening file using yield
# => le contenu est retourné ligne par ligne
# => gain en mémoire
def csv_reader_2():
  for row in open(fn, "r"):
    yield row
```

= BUROTIX ()

Préalable : Generator : Exo 17-13-03 fonctions iter() et next()

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

while True:
  try:
    letter = next(myit)
  except StopIteration:
    break
  print(letter)
```

ou

```
while True:
  letter = next(myit, None)
  if letter is None : break
  print(letter)
```

arrêt de l'itération :

- try/except
- next, default value

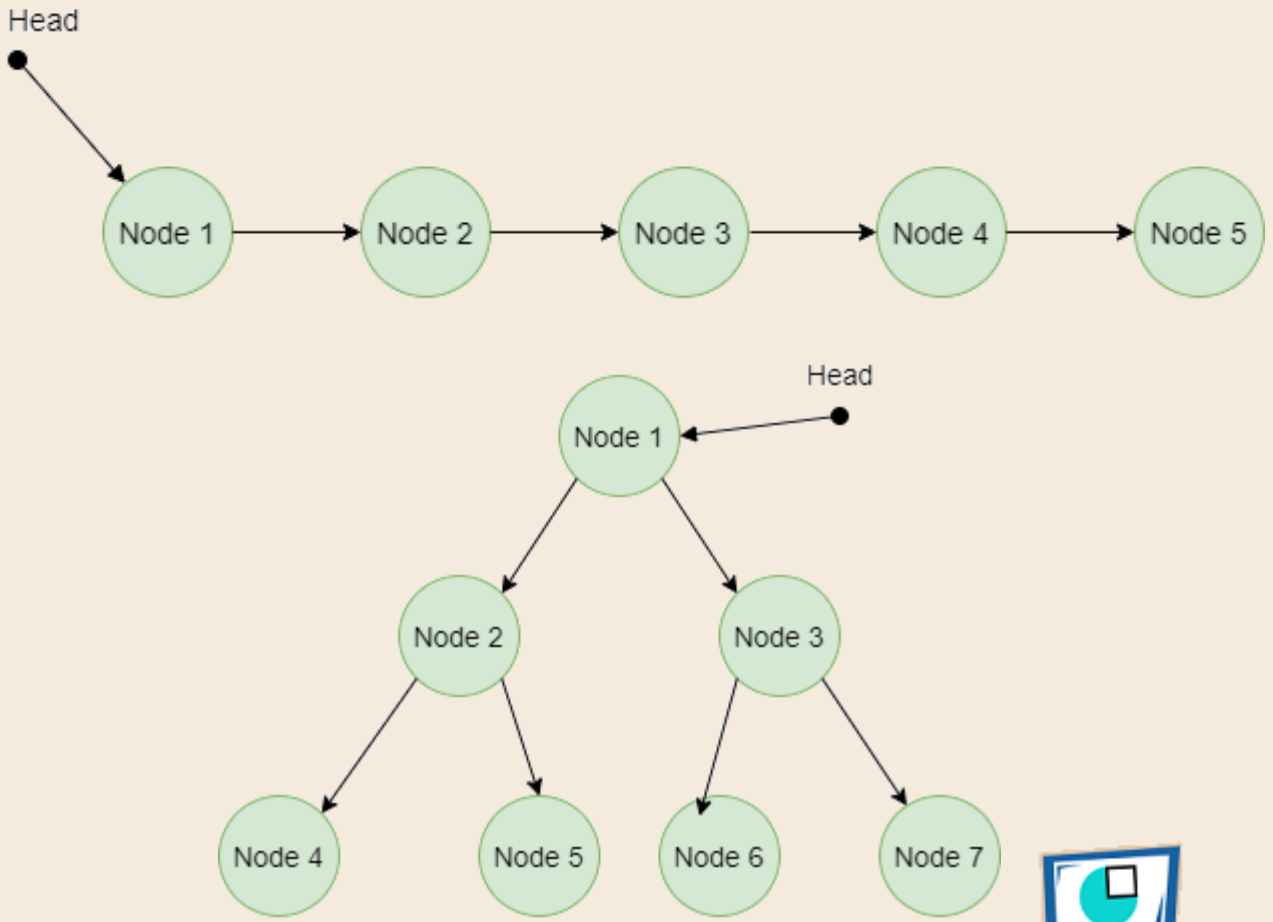
OUTPUT

```
apple
banana
cherry
```

= BUROTIX ()

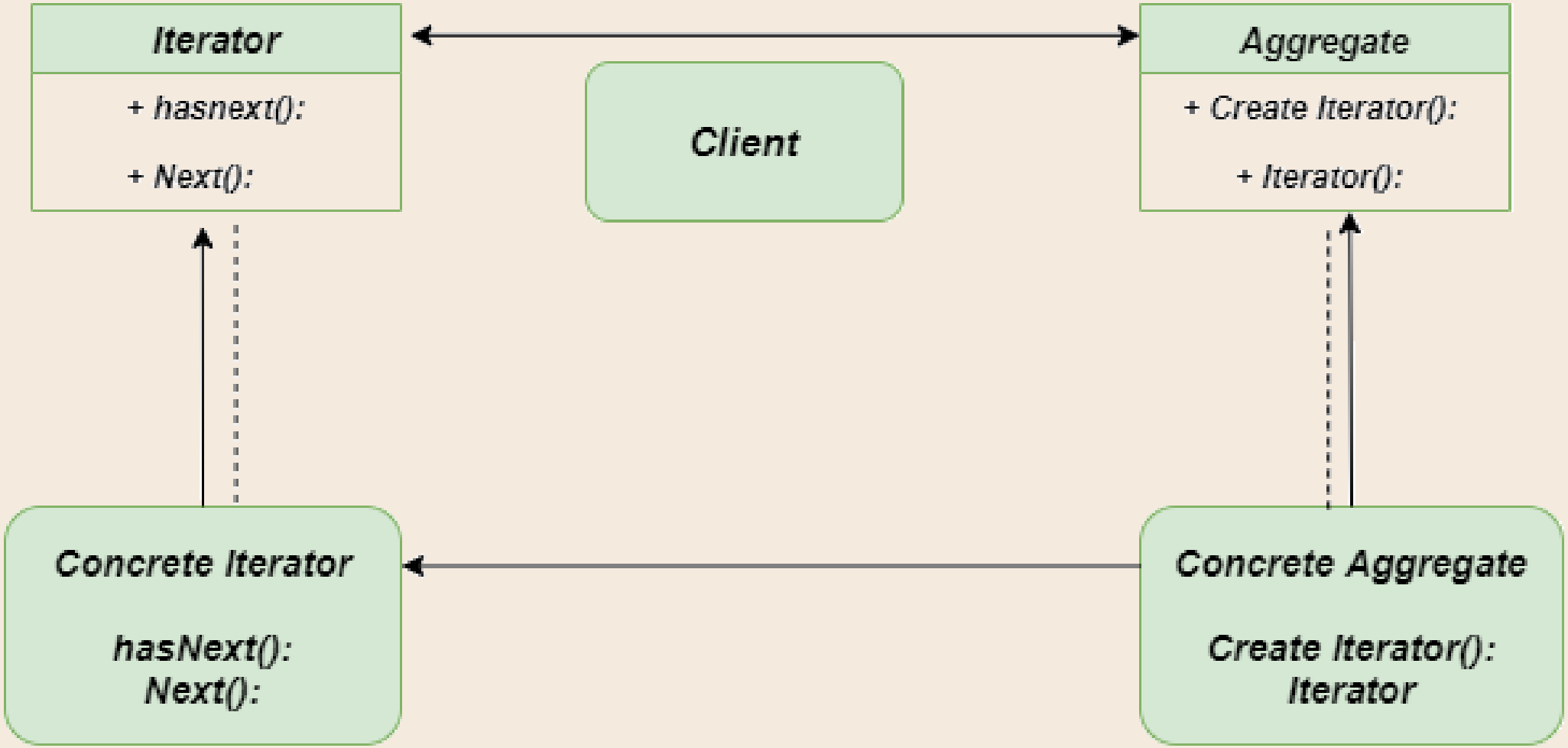
Design Pattern : Iterator

- Utilité : Accéder séquentiellement aux éléments d'un objet agrégé
  - Sans exposer son implémentation
  - On parcourt les éléments des collections sans prendre en compte les détails approfondis des éléments.



= BUROTIX ()

Design Pattern : Iterator



= BUROTIX ()

Design Pattern : Iterator : création

- Méthode `__iter__()`
  - Pour initialiser l'iterator
  - Similaire à `__init__()`
  - Retourne : l'objet (self) !
- Méthode `__next__()`
  - Pour atteindre l'élément suivant
  - Retourne : l'élément suivant de l'itération
  - Fin d'itération : `raise StopIteration`

Design Pattern : Iterator : Exo 17-13-06

```
# initialization du CountDown à 5
my_iter = iter(CountDown(5))

while True:
  try:
    # iteration
    i = next(my_iter)
    print(i)

  except StopIteration:
    # quand i atteint 0
    print("Go !")
    break
```

Ecrivez l'itérateur qui implante un compte à rebours.

OUTPUT

4

3

2

1

Go !

Design Pattern : Iterator : Exo 17-13-06

```
class CountDown:

def __init__(self):
  # Constructor
  ...

def __iter__(self):
  # creates iterator object
  self.__target = 0
  return self


def __next__(self):
  # move to next element

  # decrement
  self.__counter -= 1

  # Stop if target reached
  if self.__counter <= self.__target:
    raise StopIteration

  # Else return value
  return self.__counter
```

Exo 17-13-11 : feu de signalisation

- Partez de l'exo 17-13-11_start.py
- Redesignez la classe `Light` avec le Design Pattern `Iterator`
  - Méthode `__iter__()`
  - Méthode `__next__()`
  - Utilisation de `iter()`
  - Utilisation de `next()`
  - Quid de l'objet `feu01` ?

Exo 17-13-24 : la jungle

- Partez de l'exo 17-13-24_jungle_start.py
- Redesignez ce code avec le Design Pattern `Iterator`
  - Méthode `__iter__()`
  - Méthode `__next__()`
  - Remplacez les méthodes `go()`

Chapitre 17-16 : Observer

Design Pattern comportemental

Notion d' "évènement"

Couplage entre objets : à minimiser

Patron de conception Observer

- Un *observateur* observe des *observables*.

- En cas de **notification**, les observateurs effectuent une action en fonction des informations qui viennent des observables.
- La notion d'**observateur/observable** permet de coupler des modules de façon à réduire les dépendances aux seuls phénomènes observés.

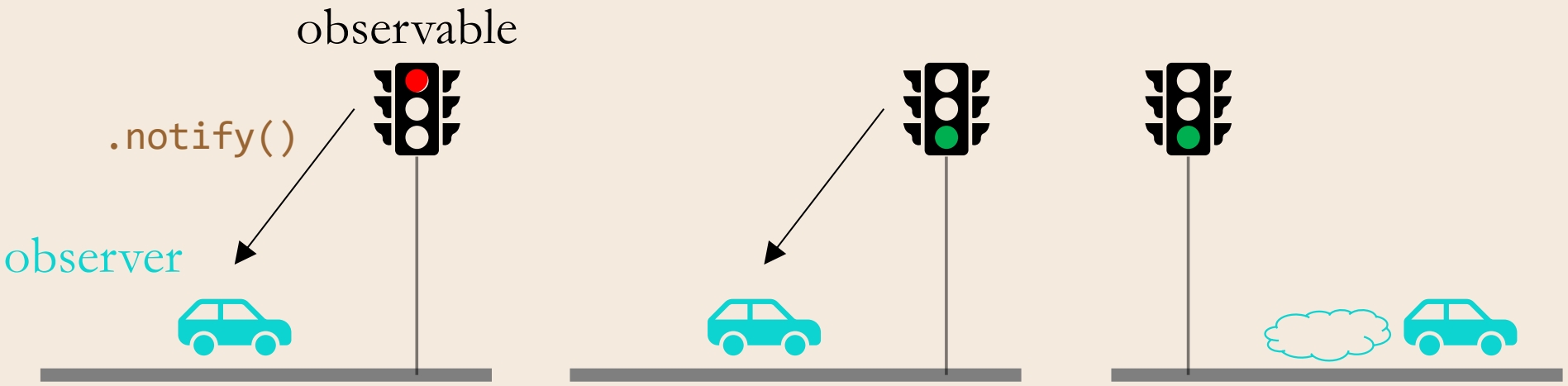
- Notions **d'événements (voir cours sur les événements)**

Design Pattern : Observer

- Motivation :
  - Prendre en compte le couplage entre objets
  - Réduire le couplage (la dépendance) entre objets
  - Rendre le code plus lisible et plus maintenable

Design Pattern : Observer

- Collaboration entre feu et voiture
  - Rapprochons-nous de la réalité
  - Le feu envoie une notification (sa couleur) à la voiture
  - La voiture réagit en conséquence.



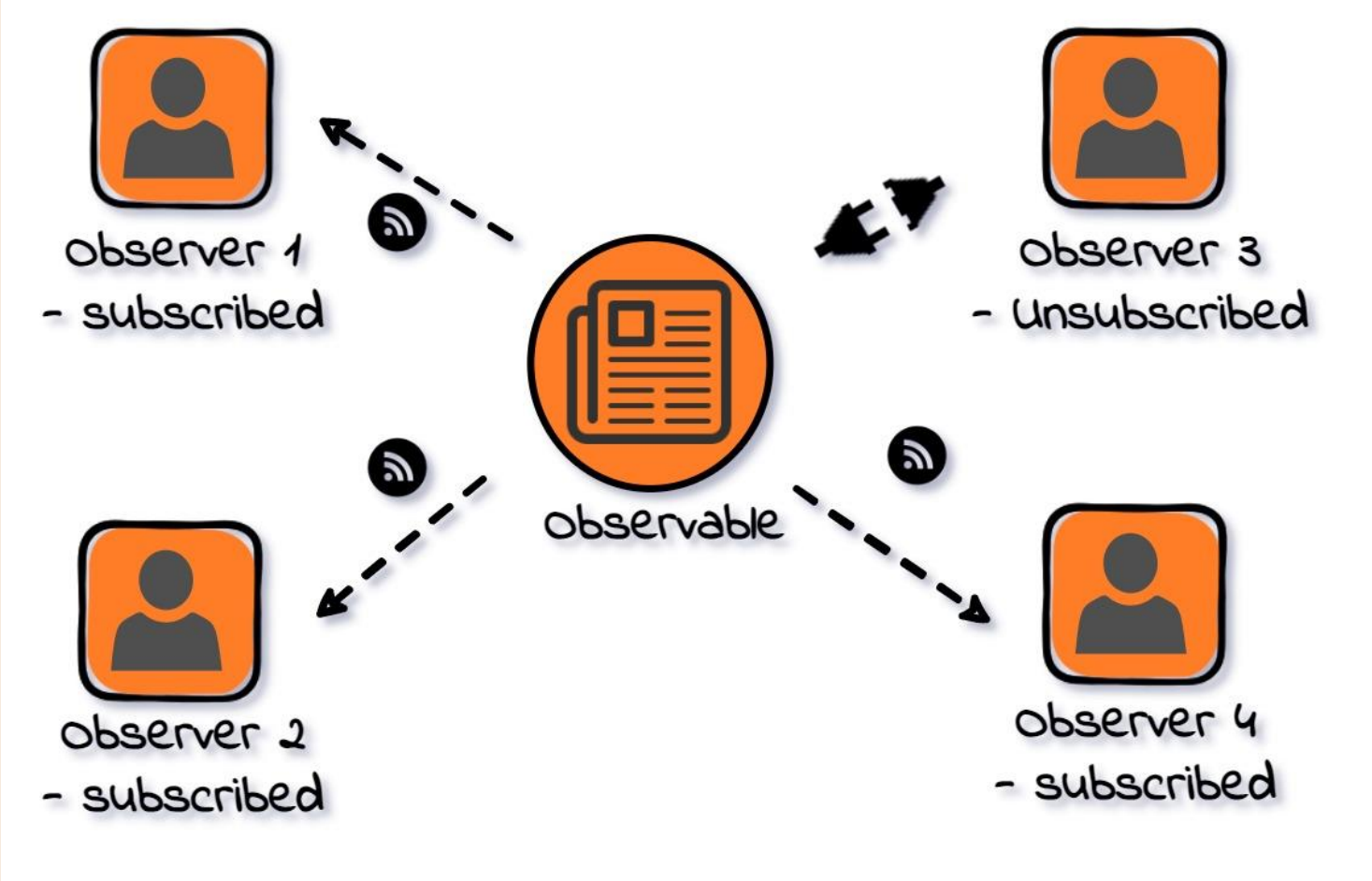
= BUROTIX ()

Design Pattern : Observer

- Utilité : Gestion des évènements
  - Lorsqu'un objet change, d'autres objets sont avisés du changement.
- Principe
  - Un objet sujet est déclaré "observable"
  - Les objets observateurs du sujet sont déclarés "observer".
  - Un mécanisme de souscription est mis en place.
  - Le sujet déclenche l'exécution d'une méthode de l'observateur.
  - Les observateurs ne sont activés que quand ils sont notifiés.
- Avantages
  - Limitation du couplage entre objets aux seuls phénomènes à observer.
  - Simplification si des observateurs multiples dépendent du même sujet

= BUROTIX ()

Design Pattern : Observer : Souscription



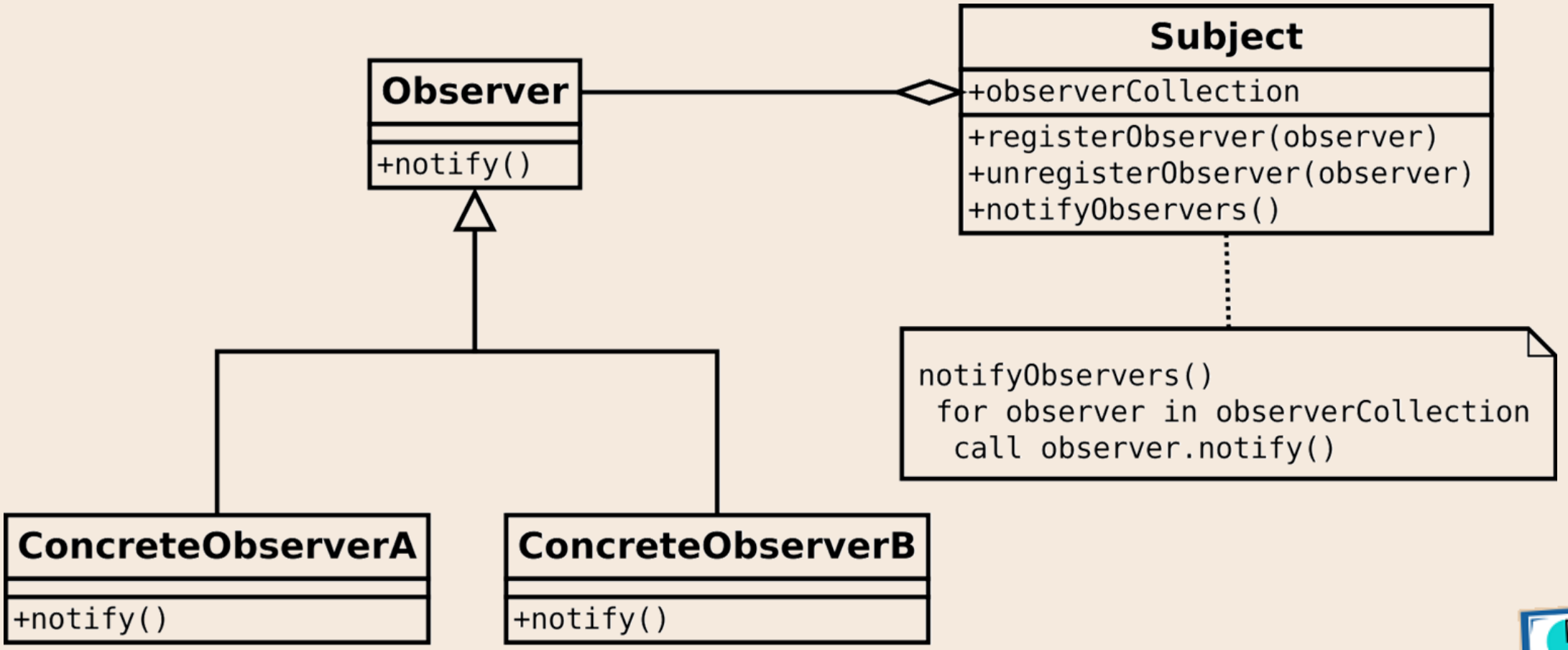
= BUROTIX ()

Design Pattern : Observer : Architecture

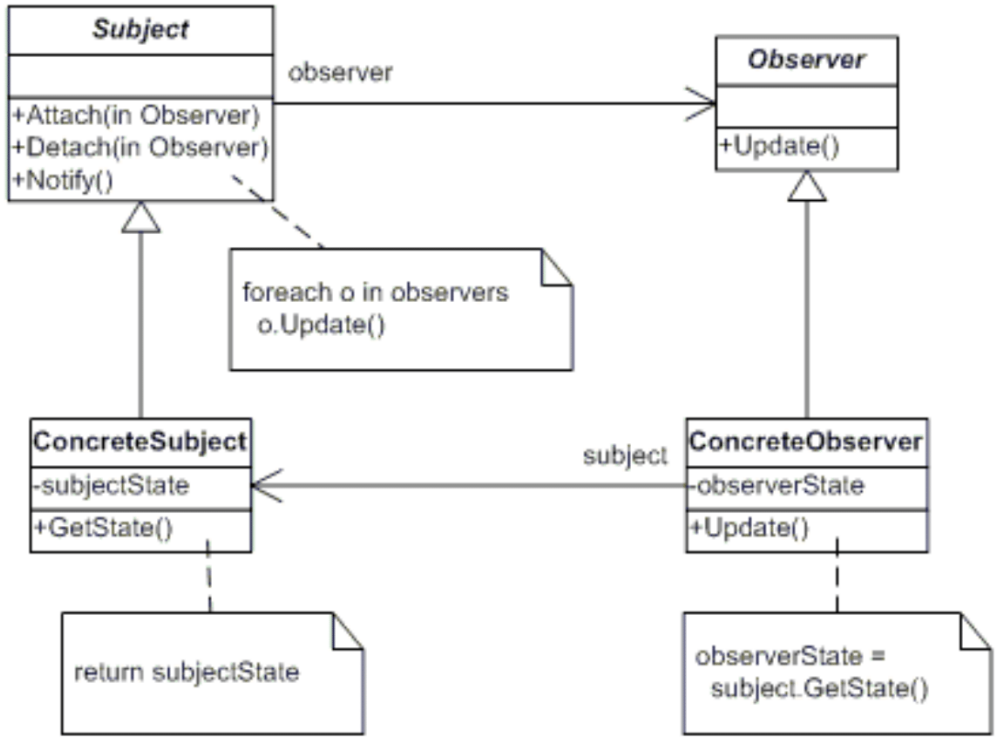
- Relation One-To-Many
  - un sujet, plusieurs observateurs
  - Pas de "event manager"
- Sujet
  - Class `Observable`
  - Il gère la liste des observateurs.
  - modification apportée au sujet => message émis vers les observateurs
  - méthode `notifyObservers()`
- Observateurs
  - Class `Observer`
  - modification apportée au sujet => message reçu de la part du sujet
  - méthode `notify()`, appelée lorsqu'un message est émis.
  - responsable de la mise à jour de son état

= BUROTIX ()

Design Pattern : Observer : UML



= BUROTIX ()



Exemple :

- Le sujet concret est *un cours de bourse*
- L'observer concret et un *Investisseur (Jean)*

L'investisseur (ConcreteObserver) veut être notifié à chaque fois que le cours d'IBM (ConcreteSubject) change

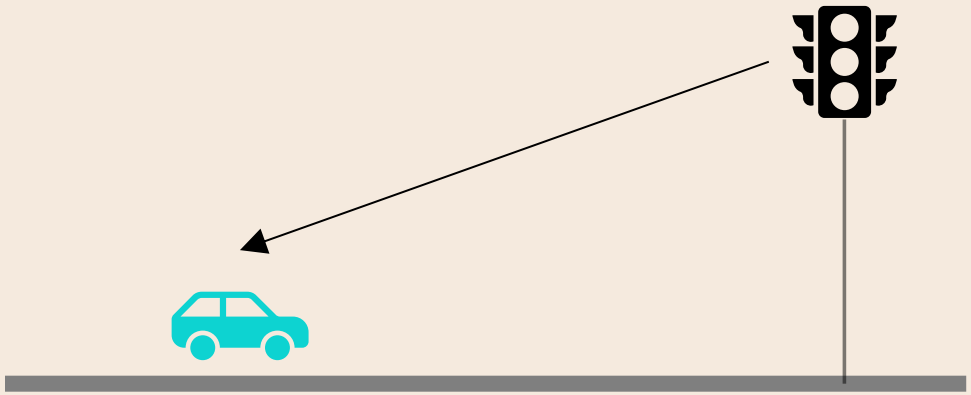
Observer : exo 17-16-02 : trivial … mais complet !

```
# à télécharger et à tester

class Observer(ABC):
  """ observateur """

  def __init__(self, observable):
    observable.subscribe(self)

  def notify():
    pass
```

```
class Observable(ABC):
  """ sujet """

  def __init__(self):
    self.observer_l = []

  def subscribe(self, observer):
    self.observer_l.append(observer)

  def notify_observers(self):
    for obs in self.observer_l:
      obs.notify(self)
```

Observer : exo 17-16-12 : trafic

- Commençons par une situation simple
  - 1 objet "feu"
  - 1 objet "voiture"
  - => 1 fil de message



= BUROTIX ()

Exo 17-16-12 : trafic, un feu, une voiture

- Partir de 17-16-12_trafic_one2one_start.py
- Spec additionnelle : la voiture A s'arrête au feu orange !
- Appliquez le design pattern `Observer` correspondant au scénario suivant :

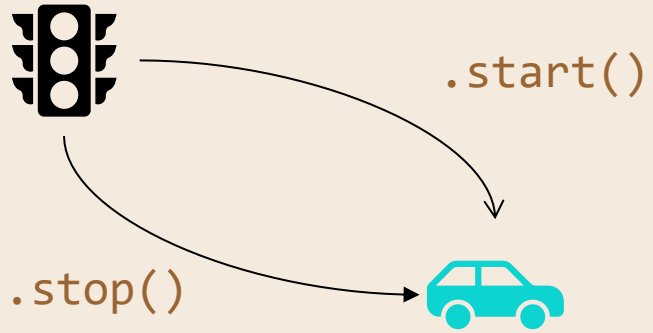
```
voiture_A = Car("Peugeot")
feu01 = Light(voiture_A)

for i in range(4):
  print(feu01)
  print(voiture_A)
  feu01.change()
```

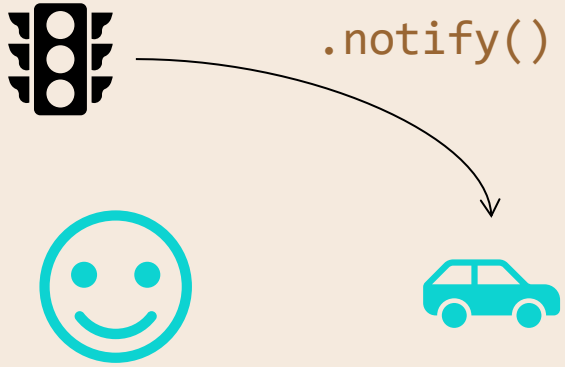
- En particulier, en vous aidant de l'exo 17-16-02 "trivia"
  - Identifiez la classe observable et la classe observateur
  - Implantez la méthode `.notify_observers()` dans l'objet observable
  - Implantez la méthode `.notify()` dans l'objet observateur

= BUROTIX ()

Exo 17-16-12 : la morale de l'histoire

- Sans observateur :
  - Le feu donne plusieurs ordres à la voiture
  - La voiture ne décide pas.
  - Couplage fort entre objets
- Avec observateur :
  - le feu informe la voiture de son nouvel état
  - La voiture décide d'avancer
  - Couplage faible entre objets

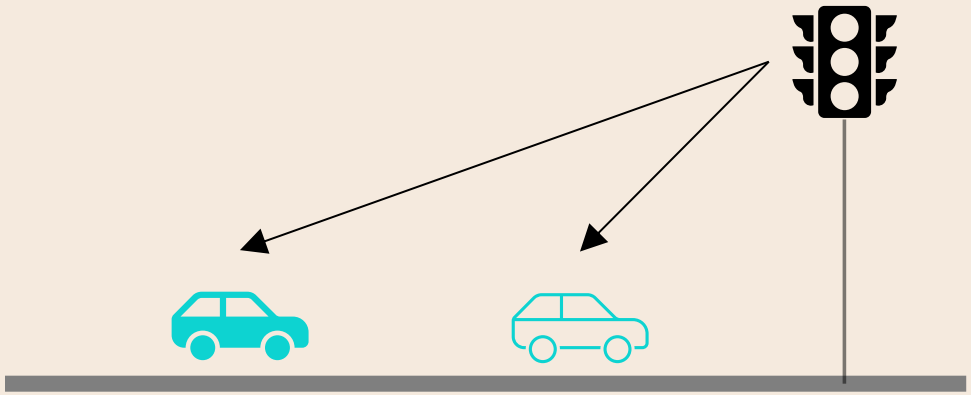




= BUROTIX ()

Observer : exo 17-16-14 : trafic

- Compliquons un peu
  - 1 objet "feu"
  - 2 objets "voiture"
  - => 2 fils de message



= BUROTIX ()

Exo 17-16-14 : trafic, un feu, deux voitures

- Partir de l'exo 17-16-12
- Spec : Au feu orange, la voiture A s'arrête, mais la voiture B continue.
- Appliquez le design pattern `Observer` correspondant au scénario suivant :

```
voiture_A = Car("Peugeot")
voiture_B = BadCar("BMW")
feu01 = Light()
feu01.subscribe(voiture_A)
feu01.subscribe(voiture_B)

for i in range(6):
  print(feu01)
  print(voiture_A)
  print(voiture_B)
  feu01.change()
```

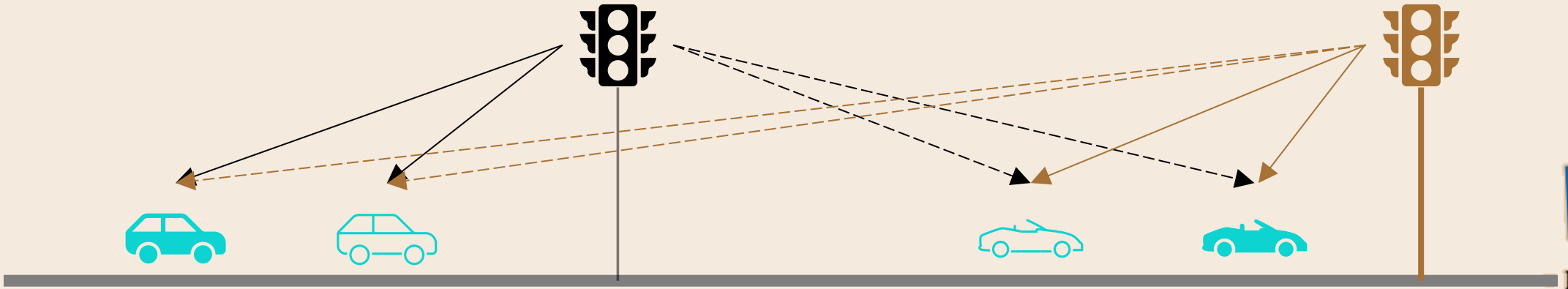
Exo 17-16-16 : trafic, un feu, trois voitures

- Partir de l'exo 17-16-14
- Appliquez le design pattern `Observer` correspondant au scénario suivant :
  - La voiture A s'arrête au feu rouge et au feu orange
  - La voiture B s'arrête au feu rouge uniquement
  - La voiture C ne s'arrête pas au feu rouge
- Implantation : A, B et C sont instances de trois classes enfants d'une classe (abstraite) `Vehicle`.

Observer : sujets multiples

Problème avancé et complexe

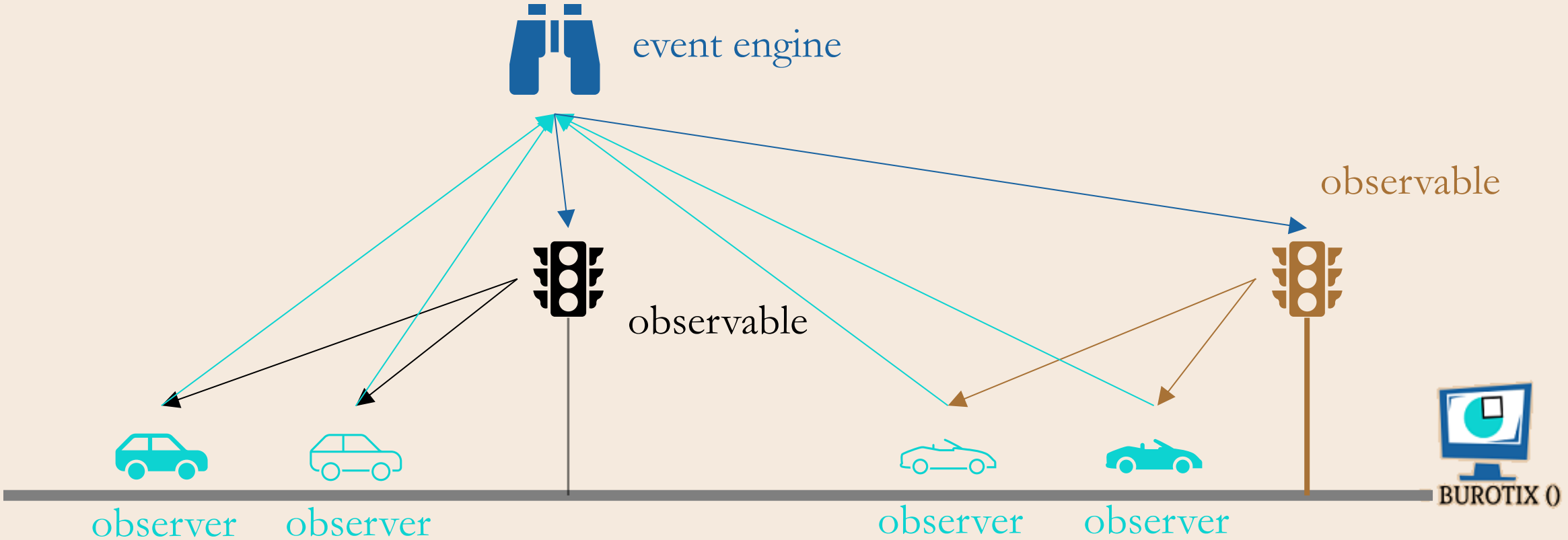
- Trafic : approche classique
  - F objets "feu"
  - V objets "voiture"
  - on dénombre F * V fils de message
  - Les observers devraient gérer des notifications inutiles.



BUROTIX ()

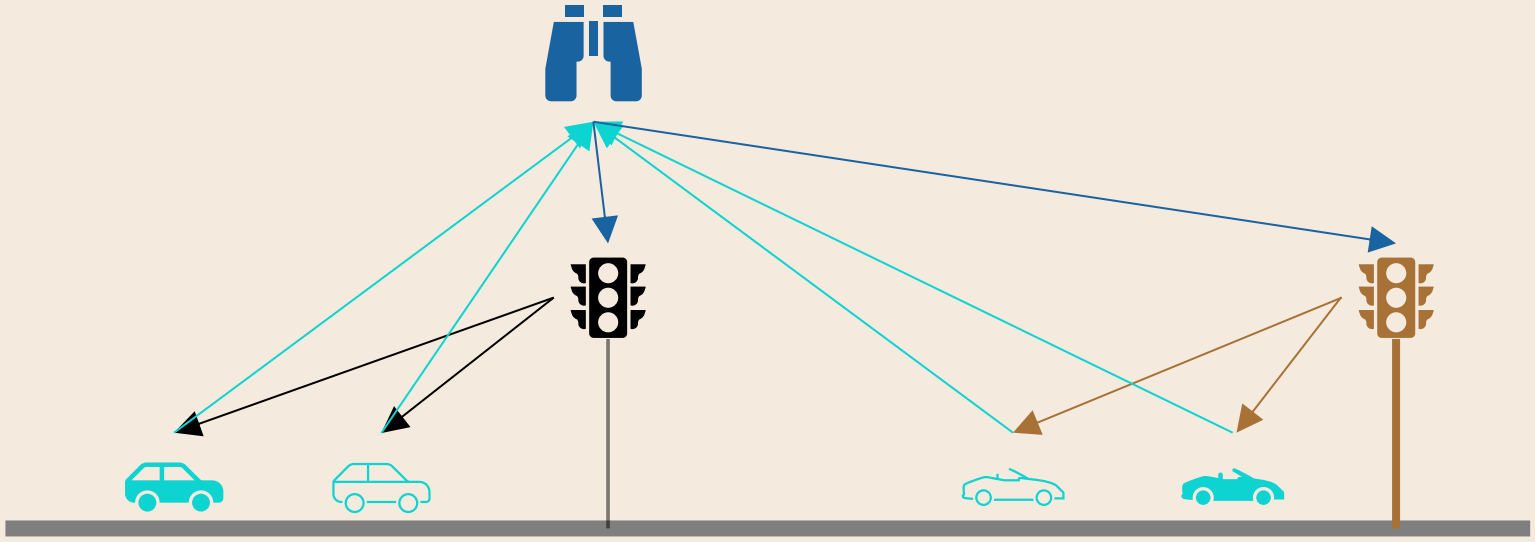
Observer : sujets multiples

- Solution : l' event engine
  - Gérer les souscriptions
  - Flux de messages: en triangle



Observer : sujets multiples

- event engine
  - F objets "feu"
  - V objets "voiture"
  - on dénombre approx. 2V+ F fils



= BUROTIX ()

Observer : sujets multiples : Architecture

- Relation Many-To-Many
  - plusieurs sujets
  - plusieurs observateurs
  - souscriptions variables
- Sujet (feu)
  - Class `Observable`
  - Si modification apportée au sujet => message émis vers les observateurs - call `.notifyObservers()`
  - Modifications des souscriptions reçues de l'Event Engine - def `.subscribe()`

= BUROTIX ()

Observer : sujets multiples : Architecture

- Observateur (voiture)
  - Class `Observer`
  - Si modification apportée au sujet => message reçu de la part du sujet - def `.notify()`
  - responsable de la mise à jour de son état
  - Si mise à jour état => message émis vers Event Engine - call `.event_engine.notify()`
- Event Engine (trafic)
  - Class `Event_Engine`
  - Si état observateur mis à jour => message reçu de la part de l'observateur - def `.notify()`
  - Modifier les souscriptions des sujets => message émis vers les sujets - call `.subscribe()`

= BUROTIX ()

Exo 17-16-18 : trafic, 2 feux, 3 voitures

- Partir de 13-01-11.
- Implantez la classe `Traffic` (event manager) correspondant au scénario suivant.

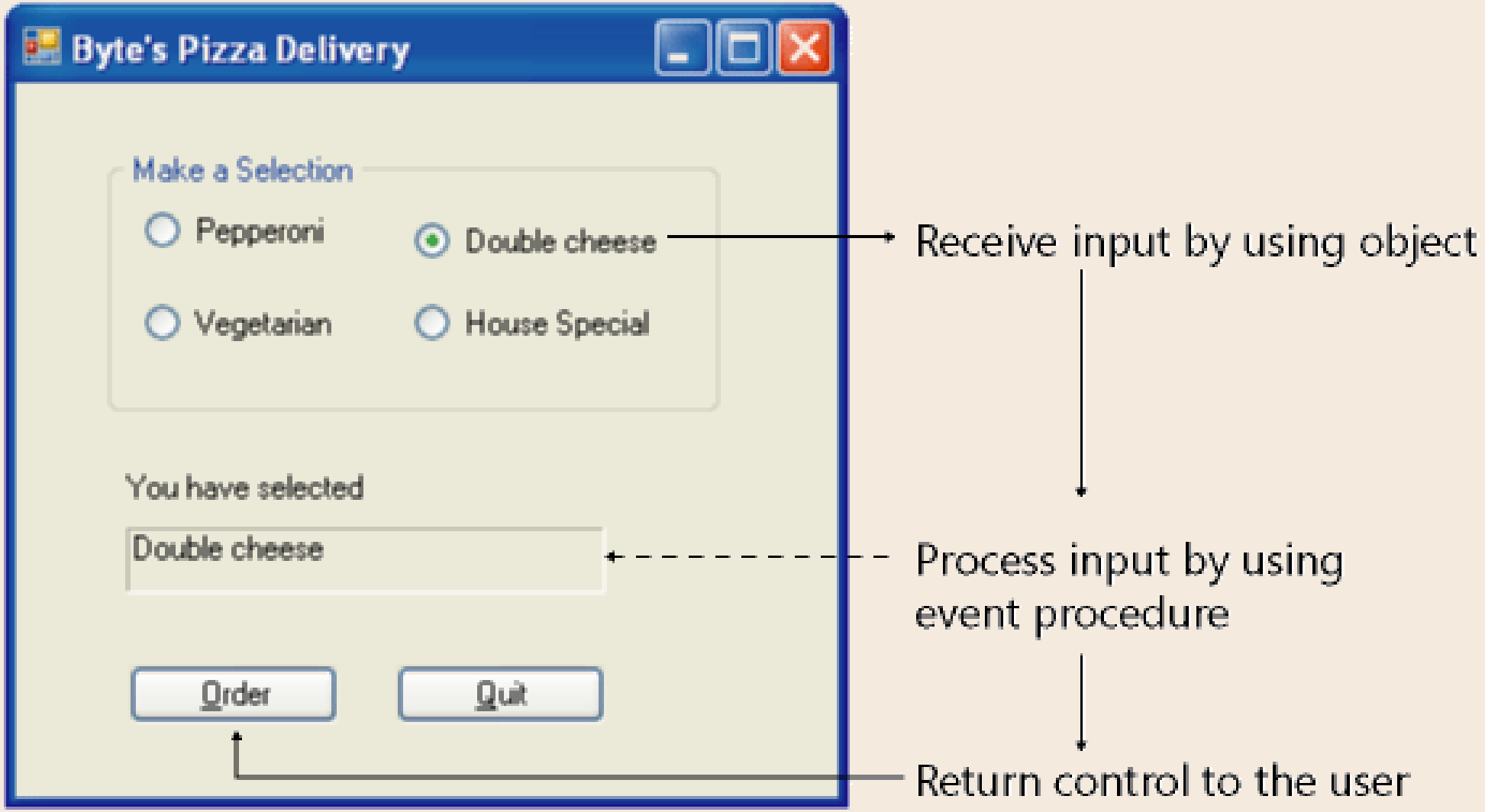
```
trafic = Traffic()
feu_l = [
  Light("Feu01", trafic, 1, 5), …
]
voiture_l = [
  Car("Peugeot", trafic), …
]
for i in range(25):
  for v in voiture_l:
    v.next()
    print(v)
  for f in feu_l:
    print(f)
  print()
  trafic.next()
```

= BUROTIX ()

Exo 17-16-18 : trafic, 2 feux, 3 voitures

- Scénario
  - `feu02` est situé après `feu01`.
  - Les voitures observent d'abord `feu01`.
  - Quand une voiture démarre, elle est désinscrite de `feu01` et s'inscrit à `feu02`.
  - Quand une voiture passe `feu02`, elle est désinscrite de `feu02` et continue sa route.
- Implantez les méthodes `.(un)subscribe()`, `.notifyObservers()`, et les différentes `.notify()`.

Observer : Application : GUI



Chapitre 17-19 : Singleton

Design Pattern créateur

Design Pattern : Singleton

- Utilité : Limiter l'instanciation d'une classe à un seul objet.
  - Quand la création de plusieurs objets issus de la même classe met l'application en danger.
  - Limiter l'accès simultané à une ressource partagée.
  - Créer un point d'accès global pour une ressource.

Design Pattern : Singleton

- Implantation
  - Via la méthode `singleton()`
  - Elle gère l'objet unique (création, enregistrement, vérification de l'existence).
  - Quand elle est appelée, elle renvoie l'objet unique.
  - Méthode statique, appelable depuis la classe
- Algorithme de `singleton()`
  - Si l'objet n'existe pas, alors il est créé, enregistré comme attribut, et retourné à l'appelant.
  - Si l'objet existe déjà, alors il est retourné à l'appelant.

Exo 17-19-05 : 2 objets, 2 dictionnaires

- Écrivez une classe `Storage` qui gère un dictionnaire.
- Méthodes :
  - `__init__()`
  - `__str__()`
  - `append( key, value )`
    - Ajoute une paire clé-valeur au dictionnaire
- Créez deux objets `Storage` et remplissez les différemment.
- Constatez si ces deux objets sont égaux.
  - Ils ne le sont pas. ☹
  - Approche KO

- Scénario

```
s = Storage()
s.append("fname","Kevin")

t = Storage()
t.append("lname","Costner")

print(s, t)
print(s is t)
```

```
s: Storage:
 - clé fname : valeur Kevin
t: Storage:
 - clé lname : valeur Brian
```

Exo 17-19-06 : 2 réf., 1 dict, `.singleton()`

- Partez de l'exo précédent.
- Appliquez le Design Pattern Singleton à la classe `Storage` afin que le dictionnaire soit unique.
- Méthodes
  - `singleton()`
  - `__init__()`
  - `__str__()`
  - `append( key, value )`

- Scénario légèrement modifié

```
s = Storage.singleton()
s.append("fname","Kevin")

t = Storage.singleton()
t.append("lname","Costner")

print(s, t)
print(s is t)
```

```
s: Storage:
 - clé fname : valeur Kevin
 - clé lname : valeur Brian
t: Storage:
 - clé fname : valeur Kevin
 - clé lname : valeur Brian
```

Exo 17-19-06 : 2 réf., 1 dict, .singleton()

```
class Storage:
  __instance = None

  @classmethod
  def singleton():
    if Storage.__instance is None:
      # If instance does not exist, then create it
      Storage.__instance = Storage()
    # return the instance
    return Storage.__instance

  def __init__(self):
    # If instance already exists, then exception
    if self.__instance is not None:
      raise Exception("This class is a singleton!")
```

Exo 17-19-06 : 2 réf., 1 dict, .singleton()

```
class Storage:
  __instance = None

  @classmethod
  def singleton(cls):
    if cls.__instance is None:
      # If instance does not exist, then create it
      cls.__instance = cls()
    # return the instance
    return cls.__instance

  def __init__(self):
    # If instance already exists, then exception
    if self.__instance is not None:
      raise Exception("This class is a singleton!")
```

Préalable : magic method `.__new__()`

- `__new__()`
  - Créer un objet d'une certaine classe
  - Retourne un objet
- `__init__()`
  - Initialiser un objet juste créé
  - Ne retourne rien

- Exo 17-19-02
  - Lire et exécuter
  - Type de `a` ? Pourquoi ?
  - Type de `b` ? Pourquoi ?

Pour programmeur avancé

= BUROTIX ()

Exo 17-19-07 : 2 réf., 1 dict, .__new__()

- Partez de l'exo précédent.
- Utilisez la méthode magique `__new__` afin de pouvoir appeler directement la classe, sans la méthode `singleton`.
- Méthodes
  - `__new__()`
  - `__init__()`
  - `__str__()`
  - `append( key, value )`

Pour programmeur avancé

- Scénario légèrement modifié

```
s = Storage()
s.append( "k_s", "v_s")
s.append("fname","Kevin")

t = Storage()
t.append("lname","Costner")

print(s, t)
print(s is t)
```

```
s: Storage:
 - clé fname : valeur Kevin
 - clé lname : valeur Brian
t: Storage:
 - clé fname : valeur Kevin
 - clé lname : valeur Brian
```

Exo 17-19-24 : la jungle

- Partez de l'exo 13-53-24
- Nouvelles hypothèses :
  - Le point d'eau est une ressource unique
  - Le buisson est une ressource unique
- Adaptez le code en conséquence.
  - On ne crée plus d'objet `lac` ou `buisson` à partir des classes `Water` ou `Archaeplastida`.
  - Les méthodes `set_water()` et `set_food()` disparaissent.
  - L'utilisation de ces resources se fera via un singleton.

Chapitre 17-20 : Iterator

Design Pattern comportemental

Design Pattern : Strategy

- Définir une famille d'algorithmes
  - encapsulés dans des classes séparées
  - interchangeables
- L'algorithme peut varier en fonction du code appelant.
- Utilité :
  - Éviter les conditions complexes (ex: "if/else") pour choisir un algorithme.
  - Rendre le code plus flexible et extensible
    - Ajouter ou modifier un algorithme n'impacte pas le code appelant
  - Isoler la logique métier

Design Pattern : Strategy

- trois acteurs principaux
  - Strategy (abstraite)
    - interface commune pour tous les algorithmes supportés.
  - Concrete Strategy
    - implémentation d'un algorithme spécifique
    - ex: "TriDirect", "TriInverse"
  - Context
    - sélection de la référence à une "Strategy" et
    - délégation du travail à cette stratégie.

Exo 17-20-02 : algorithme de tri : Strategy

```
class SortStrategy(ABC):
    @classmethod
    @abstractmethod
    def trier(self, data):
        pass


class TriAlphaDirect(SortStrategy):
    @classmethod
    def trier(self, data):
        return sorted(data)


class TriAlphaInverse(SortStrategy):
    @classmethod
    def trier(self, data):
        return sorted(data, reverse=True)
```

Exo 17-20-02 : algorithme de tri : Context

```
class SortContext:
    __tri_d = {
        "alpha-direct" : TriAlphaDirect,
        "alpha-inverse" : TriAlphaInverse,
    }

    def __init__(self):
        self.__strategy = None

    def set_strategy(self, strategy_name):
        self.__strategy = self.__tri_d[strategy_name]

    def sort(self, data):
        return self.__strategy.trier(data)
```

Exo 17-20-02 : algorithme de tri : Main

```
mylist = [5, 2, 8, 1, 3]
contexte = SortContext()

# Utilisation du tri direct
contexte.set_strategy("alpha-direct")
print(contexte.sort(mylist) )

# Changement dynamique de stratégie :
# utilisation du tri inverse
contexte.set_strategy("alpha-inverse")
print(contexte.sort(mylist))
```

```
OUTPUT
Tri direct appliqué :
[1, 2, 3, 5, 8]
Tri inverse appliqué :
[8, 5, 3, 2, 1]
```

Bachelier en Informatique de Gestion
Programmation Orientée Objet

= BUROTIX ()