Patron de conception en français, c'est une solution générique à un problème de conception récurrent.

> [!TIP] Liste exhaustive des design pattern
>
> Il y a même des exemple d'utilisation et d'implémentation dans différents langages.

Singleton

Le pattern *Singleton* permet d'obtenir une et une seule instance d'une classe.

On utilise communément une méthode statique et une méthode `get_instance` qui renverra l'instance existante, ou la créera si elle n'existe pas encore avant de la renvoyer.

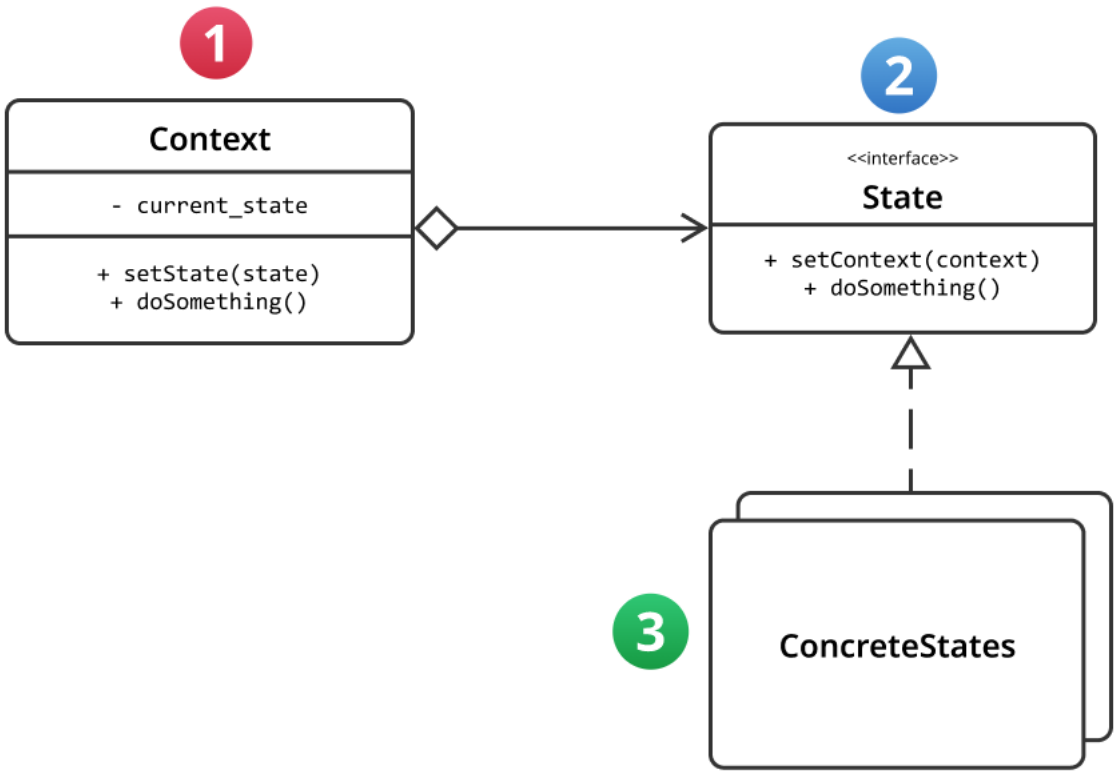
```
class Singleton:
 __conn = None

 @staticmethod
 def get_instance():
  if not Singleton.__conn:
   Singleton.__conn = Singleton()
  return Singleton.__conn

 def __init__(self):
  print("Singleton!")


obj1 = Singleton.get_instance()
print(obj1)

obj2 = Singleton.get_instance()
print(obj2)

print(obj1 == obj2)
```

> [!Tip] Singleton Ce design pattern est principalement utilisé pour gérer les connections

State

Au cours du cycle du vie d'un programme, un objet peut passer par plusieurs état. Ce pattern permet de gérer les variations d'état d'un objet sans passer par un long `if` dans l'implémentation de la classe.

Par exemple, un lecteur audio peut passer par les trois états suivants : l'état *playing* pendant lequel il joue un extrait vidéo, l'état *ready* lorsqu'il ne lit pas d'extrait mais est prêt à le faire, et l'état *locked* au cours duquel il est verrouillé.



Implémentation de `AudioPlayer`

```
class AudioPlayer:

 _state = None

 def __init__(self, state: State) -> None:
  self.transition_to(state)

 def transition_to(self, state: State):
  self._state = state
  self._state.context = self
```

Implémentation des états

```
from abc import abstractmethod, ABC


class State(ABC):

 _player: AudioPlayer

 def __init__(self, player):
  self._player = player
```

```
    @property
        def player(self) -> AudioPlayer:
            return self._player

        @context.setter
        def player(self, context: AudioPlayer) -> None:
            self._player = player

    @abstractmethod
    def click_lock(self):
     pass

    @abstractmethod
    def click_play(self):
     pass

    @abstractmethod
    def click_next(self):
     pass

    @abstractmethod
    def click_previous(self):
     pass
```

On va maintenant pouvoir créer différents états `LockedState`, `ReadyState` et `PlayingState` qui hériteront de cette classe abstraite `State`.

```
class LockedState(State):
 def __init__(self, player):
  super().__init__(player)

 def click_lock(self):
  if self.

 def click_play(self):
  pass

 def click_next(self):
  pass

 def click_previous(self):
  pass
```

Exercice : VehiculeFactory Exercice tutoré : VehiculeFactory accompagné