FRUIZIONE E UTILIZZO DEI MATERIALI DIDATTICI

- **E’ vietata** la **copia**, la **rielaborazione**, la **riproduzione** dei contenuti e immagini presenti nelle lezioni in qualsiasi forma

- **E’ inoltre vietata** la **diffusione**, la **redistribuzione** e la **pubblicazione** dei contenuti e immagini, incluse le registrazioni delle videolezioni con qualsiasi modalità e mezzo non autorizzati espressamente
dall'autore o da Unica

STRATEGY DESIGN PATTERN

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it”

[Gamma, Erich, et al. "Elements of reusable object-oriented software.", 1995.]

Example

Write a program that manages drink orders in a pub.

Each `customer` can order one or more drinks declaring, in his `add_drink()` method, the number of drinks and the unit cost. The total cost of the various orders is stored in a `customer` attribute.

```
customer1.add_drink(1, 7) # 1 drink, 7 euros for each drink
```

During the ***Happy Hour*** the customer declares the same price, but the drink is discounted by 50%.

```
customer1. add_drink(1, 7) #  it pays 3.5 euros
```

Moreover, the type of clothing will be appropriate for the situation. The `get_actual_dress()` method will print **normal dress** or **happy hour dress**, depending on the time.

Classic approach

```python
class CustomerNoStrategy:
    def __init__(self):
        self.cost = 0

    def print_bill(self):
        print(self.cost)

    def add_drink(self, n, unit_cost, happy_hour=False):
        cost = n * unit_cost
        if happy_hour:
            cost = cost / 2
        self.cost += cost

    @staticmethod
    def get_actual_dress(happy_hour=False):
        if happy_hour:
            print('happy hour dress')
        else:
            print('normal dress')
```

```python
# Code in `strategy/pub`

# NORMAL BILLING
customer1 = CustomerNoStrategy()
customer1.add_drink(1, 7)
customer1.get_actual_dress()

# START HAPPY HOUR (50% discount)
customer1.add_drink(2, 5, happy_hour=True)
customer1.get_actual_dress(happy_hour=True)

# FINAL BILL
customer1.get_actual_dress(happy_hour=True)
customer1.print_bill()  # 12
```

Discussion and Issues [1]

- Potentially complicated, nested conditional logic with k branch in n different methods.
- Risk of introducing a *maintenance challenge*. Modifying existing code is often an error-prone activity.
- It doesn't scale!
  - What happens if we have several methods that depend on a condition, and not only `get_actual_dress()` and `add_drink()`?
  - What if we have other situations, not only (`happy_hour`, `not happy_hour`) ?
  - What if we **add** other new methods and other new situations?

---

[1] For this toy-problem the solution is adequate. We can also store the `happy_hour` flag (`True` or `False`) in a class variable or in a 'time' object. In this way, all customer objects can read the same flag. Problems arise when the program is more complex, i.e., we have **many methods that depend on multiple conditions** (time of the day, season, weather conditions...)

Discussion and Issues

- Coupling

The coupling characterizes the relationship between modules. Two modules have a **high coupling** if they are strictly dependent on each other. A low level of coupling (or loose coupling) makes it possible to analyze, understand, modify, test, or reuse each module separately.

The worst form of coupling is **control coupling**.
The client passes a flag (*happy_hour*, in this case) used to control the behavior and the server module's inner logic.
Loss of encapsulation: the client knows about the server's internals.

Inheritance approach

```
class CustomerNormalHour:
    def __init__(self):
        self.cost = 0

    def print_bill(self):
        print(self.cost)

    def add_drink(self, n, unit_cost):
        self.cost += n * unit_cost

    @staticmethod
    def get_actual_dress():
        print('normal dress')

class CustomerHappyHour(CustomerNormalHour):
    def add_drink(self, n, unit_cost):
        discount = 0.5
        self.cost += discount * (n * unit_cost)

    @staticmethod
    def get_actual_dress():
        print('happy hour dress')
```

```
# Code in `strategy/pub`

# NORMAL BILLING
customer1 = CustomerNormalHour()
customer1.add_drink(1, 7)
customer1.get_actual_dress()

# START HAPPY HOUR (50% discount)
customer1.__class__ = CustomerHappyHour
# You are changing class at runtime!
# It is possible, but are we sure it is a good idea?

customer1.add_drink(2, 5)
customer1.get_actual_dress()

# FINAL BILL
customer1.get_actual_dress()
customer1.print_bill()  # 12
```

Discussion and Issues

Allowing an object to **change class** will lead to code that is difficult to understand, maintain, modify, test, and debug. Moreover, it can also cause a **loss of consistency** of the involved objects. If you change the class of an object from A to B, you will obtain an **object of class B** that responds to instance methods of class B but **maintains the attributes of class A**.

```
class A:
  def __init__(self):
    self.attribute_of_A = 10
  def f_A(self):
    print("Instance method of class A")

class B:
  def __init__(self):
    self.attribute_of_B = 10
  def f_B(self):
    print("Instance method of class B")
```

```
print("object of class A")
obj = A()
print([m for m in dir(obj)
    if not m.startswith('__')])

print("Now the class is B")
obj.__class__ = B
print([m for m in dir(obj)
    if not m.startswith('__')])

# object of class A
# ['attribute_of_A', 'f_A']
# Now the class is B
# ['attribute_of_A', 'f_B']
```

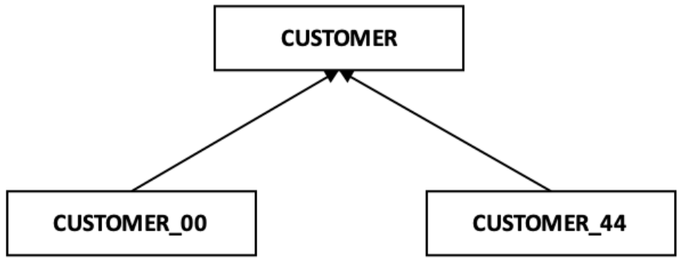
Discussion and Issues

If we want to model **two different types of objects** it can be correct and appropriate to use inheritance. But this solution is totally inadequate if **the behavior is not related to the class**, for example if the behavior of an object **depends on the 'environment'**, and can **change at runtime** due to an external event. For example, the amount to be paid changes depending on whether we are in the 'happy hour' or not.

Why?
If we have 5 types of payment that depend on the time (`morning`, `afternoon`, `happy hour`, and so on), and 5 types of clothing that depend on the weather (`cold`, `hot`,...), we have 25 possible behaviors to be implemented in 25 classes.

		CLOTHING				
		0	1	2	3	4
PAYMENT	0	Class_00	Class_01	Class_02	Class_03	Class_04
	1	Class_10	Class_11	Class_12	Class_13	Class_14
	2	Class_20	Class_21	Class_22	Class_23	Class_24
	3	Class_30	Class_31	Class_32	Class_33	Class_34
	4	Class_40	Class_41	Class_42	Class_43	Class_44

Defining a class for each configuration would force us to **greatly complicate our architecture**. This is one of the many cases where **composition is better than inheritance**.

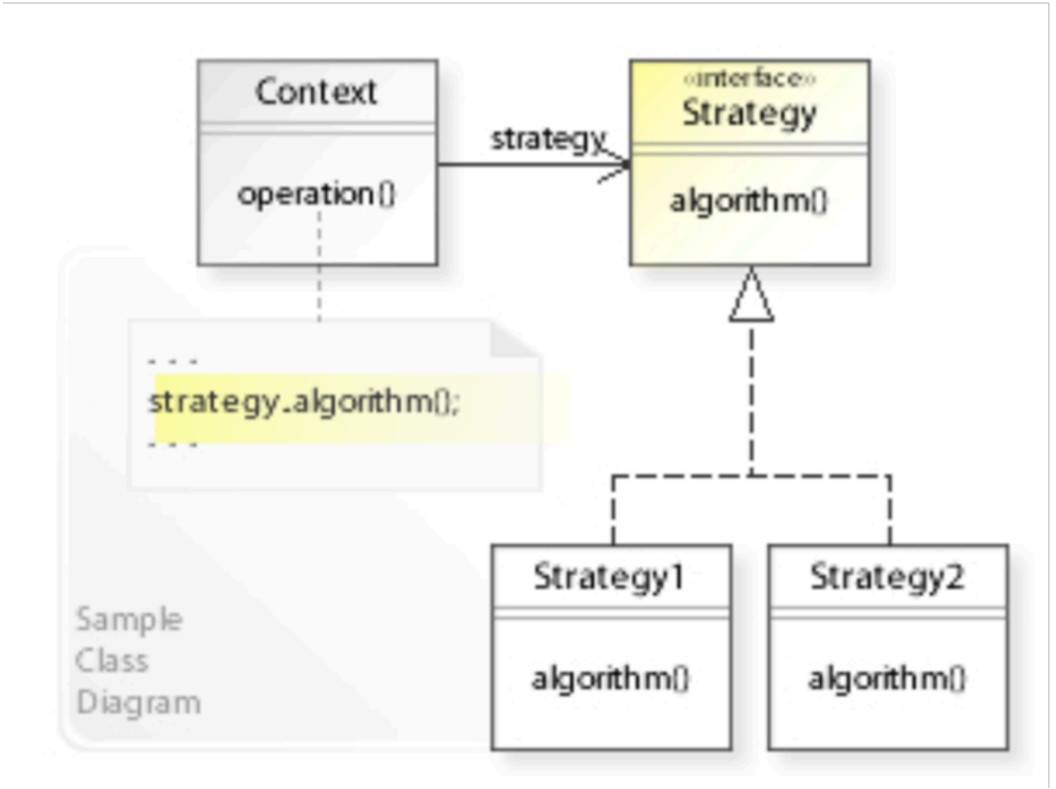


Solution: STRATEGY DESIGN PATTERN

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it” [Gamma, et al]

The strategy pattern enables **selecting an algorithm at runtime**. Instead of implementing a single algorithm directly, code receives runtime instructions about the algorithm to be used.

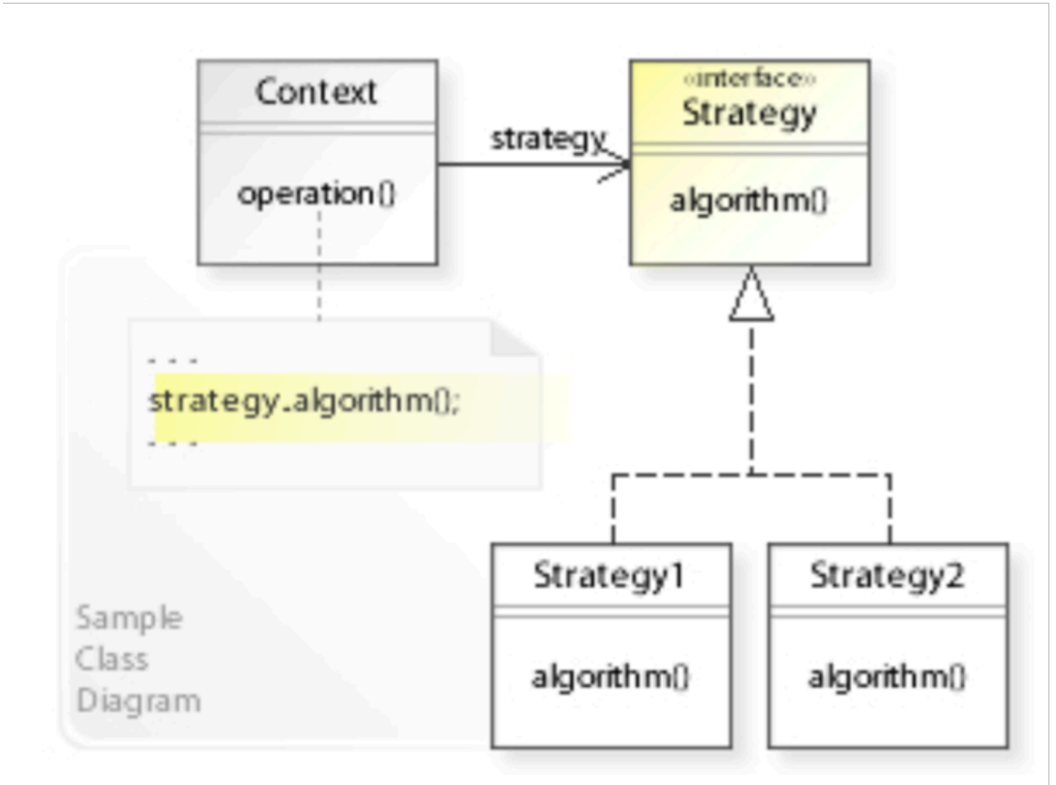
Deferring which algorithm to use at runtime allows the calling code to be more flexible and reusable.



Class diagram. The **Context** class doesn't implement an algorithm directly.

Context refers to the **Strategy** interface for performing an algorithm (`strategy.algorithm()`), which makes **Context independent of how an algorithm is implemented**.

The `Strategy1` and `Strategy2` classes implement the Strategy interface, that is, implement (encapsulate) the concrete algorithm.



Example. One algorithm works better with small input sizes, while the other works better with large input sizes. You can use Strategy to decide which algorithm to use based on the input data at runtime.

Implementation.
Typically the strategy pattern stores a reference to some code in a data structure and retrieves it. This can be achieved using

- function pointer
- first-class function
- classes or class instances

STRATEGY Design Pattern provides a way to apply the **open-closed principle**.

OPEN CLOSED PRINCIPLE

"Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**" [Martin 2002].

Extending the behaviour of a module is achieved by **adding code instead of changing the existing source**.

Following this principle

- minimizes the risk of introducing bugs in existing, tested code
- raises the quality of the design by introducing loose coupling.

NB. it is *impossible* to design a module totally closed against all kinds of changes.

Solution (code in `strategy/pub`)

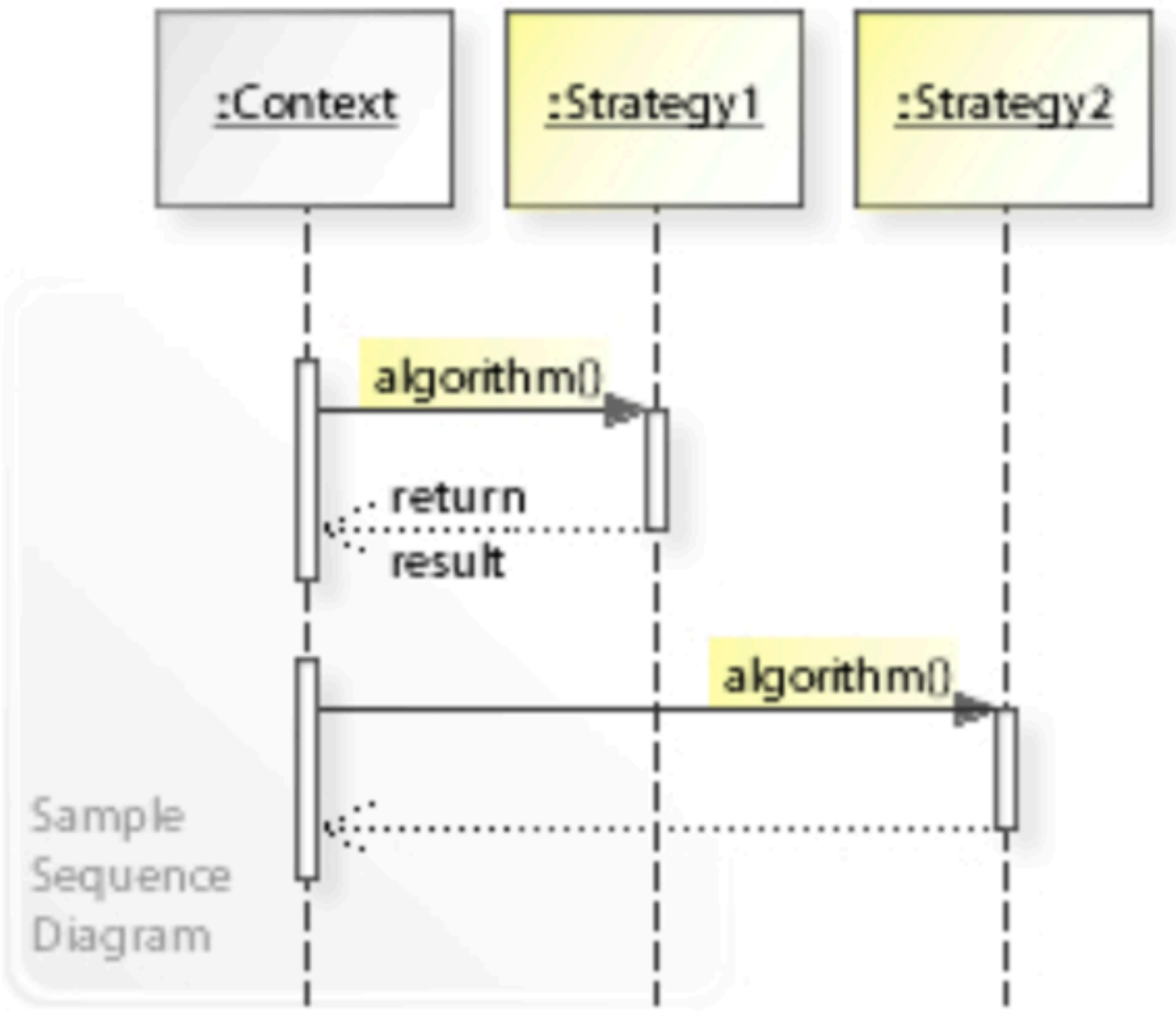
```
class NormalStrategy(Strategy):
    def get_actual_price(self, value):
        return value

    def get_actual_dress(self):
        print('normal dress')


class HappyHourStrategy(Strategy):
    def get_actual_price(self, value):
        # (50 % discount)
        return value * 0.5

    def get_actual_dress(self):
        print('happy hour dress')
```

```
class Customer:
    def __init__(self, strategy):
        self.cost = 0
        self._strategy = strategy

    def print_bill(self):
        print(self.cost)

    def add_drink(self, n, unit_cost):
        self.cost +=
            self._strategy.get_actual_price(n * unit_cost)

    def set_strategy(self, strategy):
        self._strategy = strategy

    def get_actual_dress(self):
        self._strategy.get_actual_dress()
```

The **Sequence diagram** shows the run-time interactions.

The **Context** object delegates an algorithm to different **Strategy** objects.

- **Context** calls `algorithm()` on a **Strategy1** object, which performs the algorithm and returns the result to Context.
- **Context** changes its strategy and calls `algorithm()` on a **Strategy2** object, which performs the algorithm and returns the result to Context.



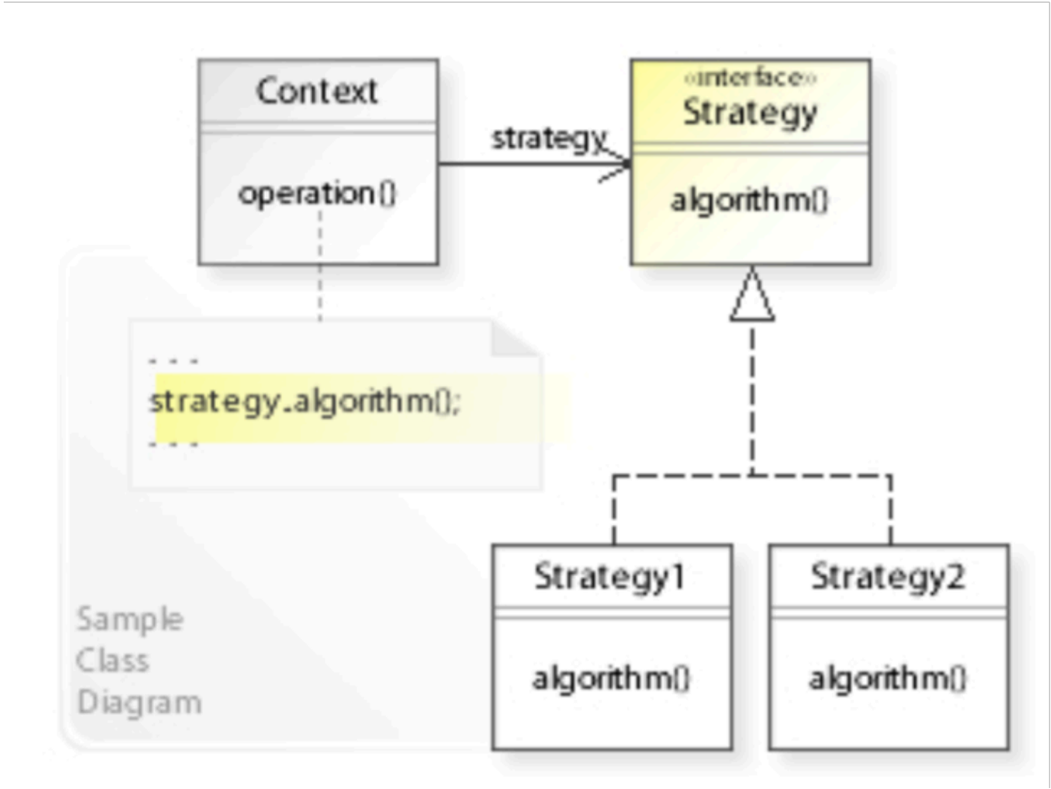
The use of Strategy is an **alternative to inheritance** because it allows us to vary the behavior of a class, without having to override it.

According to the strategy pattern, **the behaviors of a class should not be inherited**. Instead they should be encapsulated using interfaces. This is compatible with the open/closed principle, which proposes that **classes should be open for extension but closed for modification**.

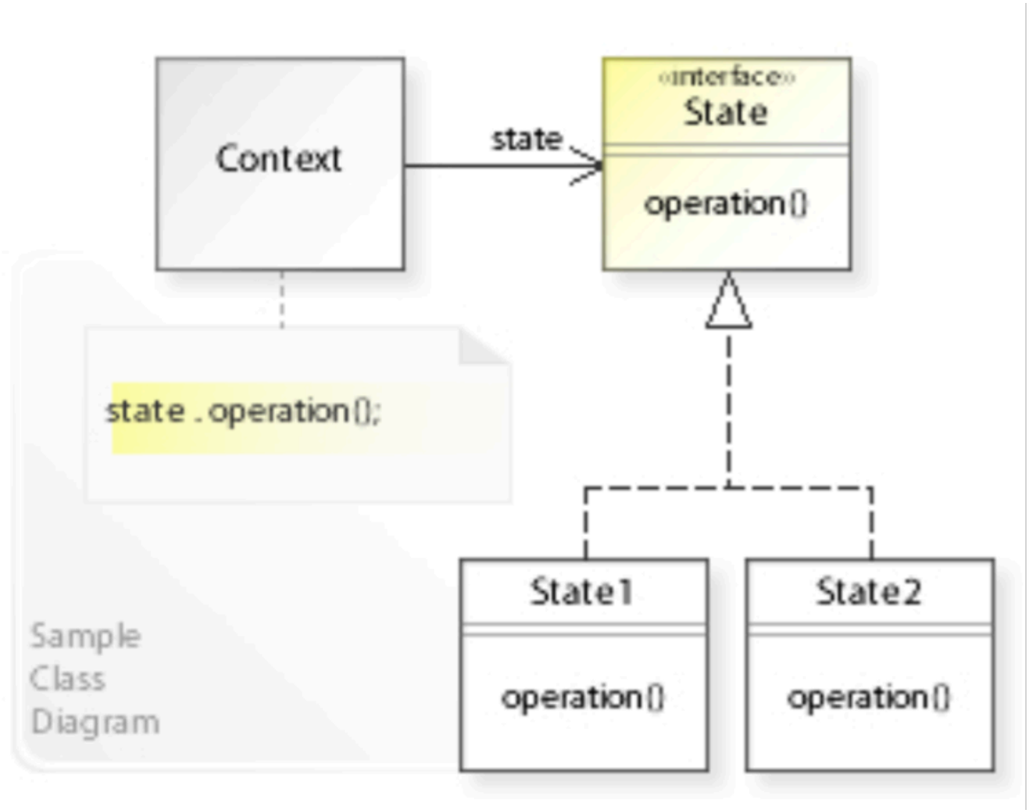
A simplified version using first-class function is in `strategy/pub_simplified`.

Difference between STATE and STRATEGY

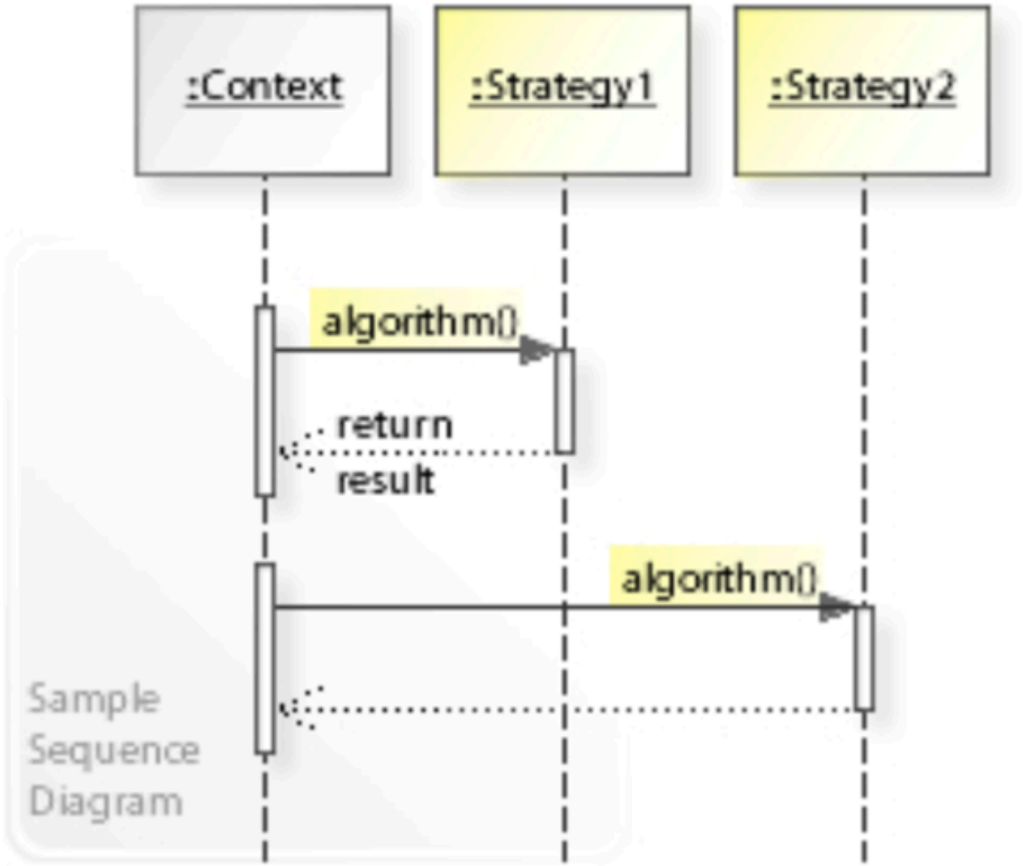
STRATEGY design pattern



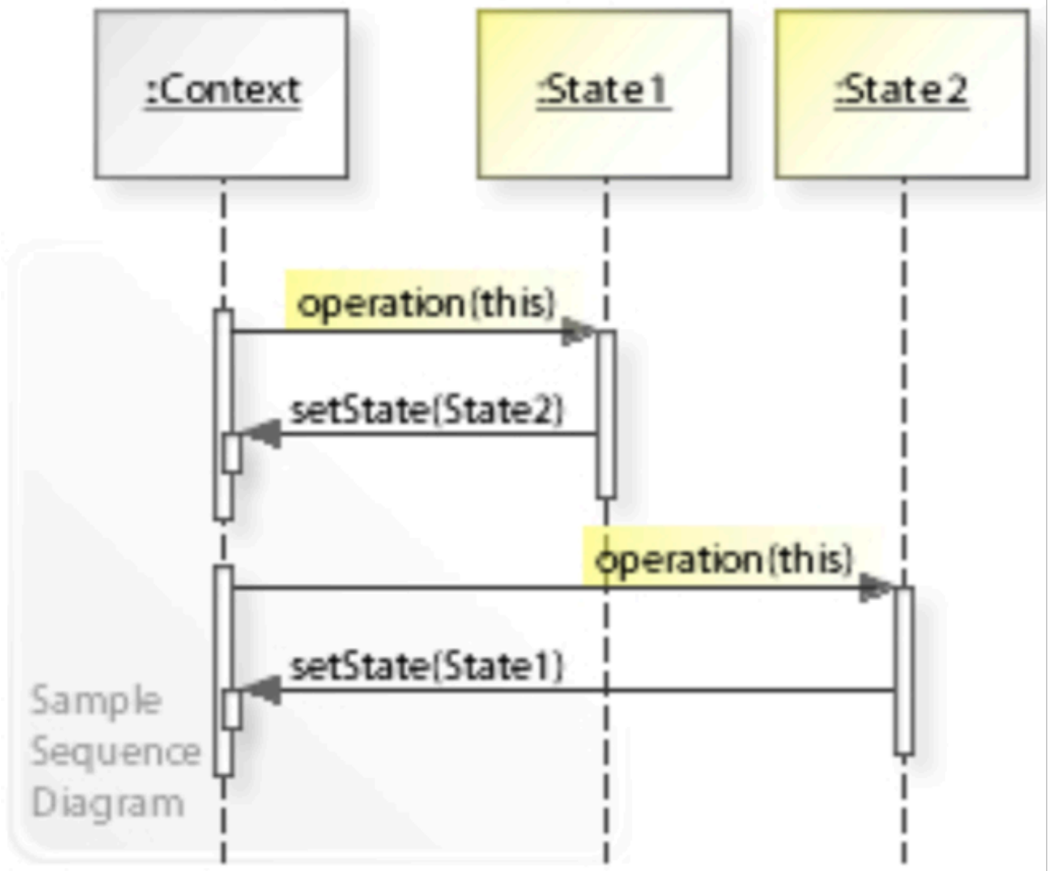
STATE design pattern



STRATEGY design pattern



STATE design pattern



Difference between STATE and STRATEGY

- Both design patterns show different behaviors depending on the context
- The biggest difference is that in STATE there is an **explicit state transition**, and the state transition is explicitly managed by the State object.
- In the STRATEGY pattern the transition between different strategies is driven by an external action.

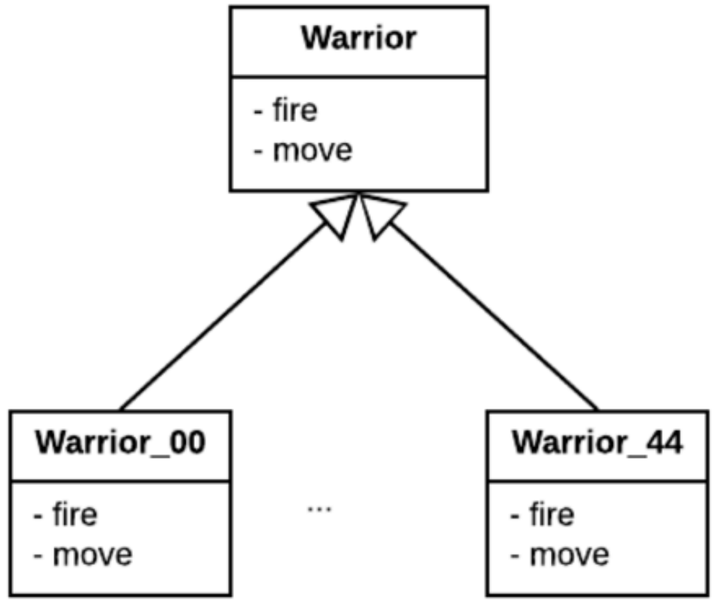
STATE and **STRATEGY** solve different problems.

- The strategy pattern is used to choose an algorithm at runtime; generally, only one of those algorithms is going to be chosen for a particular use case.
- The state pattern is designed to allow **switching between different states dynamically**, as some process evolves.
- Regarding the implementation, the primary difference is that the strategy pattern is not typically aware of other strategy objects. In the state pattern, either the state or the context needs to know what other states exist to handle the transitions.

Composition vs. inheritance

- In order to achieve different behaviors, **inheritance** can be used.
- If we have 5 types of behavior for **firepower** and 5 for **speed**, we have 25 possible behaviors to be implemented in 25 different classes.
- In addition to the large number of classes, I have to consider the fact that objects can change behavior, and therefore they can change class.

SPEED \ FIREPOWER	0	1	2	3	4
0	Class_00	Class_01	Class_02	Class_03	Class_04
1	Class_10	Class_11	Class_12	Class_13	Class_14
2	Class_20	Class_21	Class_22	Class_23	Class_24
3	Class_30	Class_31	Class_32	Class_33	Class_34
4	Class_40	Class_41	Class_42	Class_43	Class_44



Composition vs. inheritance

- If we have many different types of behaviors it may be easier to use the **composition** - that is, incorporate into the 'main' object (i.e. the character), an object that provides the desired behavior (i.e. a State or a Strategy object).
- In this way we avoid the multiplication of classes
- The different behavior can be implemented by **dynamically changing the functions** that implement the behavior.