Strategy Design Pattern

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

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
```

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
```

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 - NO INHERITANCE

```python
# NORMAL BILLING
customer1 = Customer()
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
```

```python
class Customer:

  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 /= 2
      self._cost += cost

  def f1(self, happy_hour=False):
    # if happy_hour:
        pass

  def f2(self, happy_hour=False):
    # if happy_hour:
        pass

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

Discussion and Issues

- Potentially complicated, nested conditional logic with k branch in n different methods.

Discussion and Issues

- 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.

Discussion and Issues

- 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 ?
  - What if we have other situations, i.e., other conditions to evaluate?
  - What if we **add** other new methods and other new situations?

Discussion and Issues

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, ensuring that all customer objects share the same flag.

Discussion and Issues

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, ensuring that all customer objects share the same flag.

Problems arise in more complex programs, particularly when dealing with **numerous methods dependent on multiple conditions** (time of day, season, weather conditions...).

Discussion and Issues

Coupling

Coupling delineates the interdependence between modules. Modules exhibit **high coupling** when they are strictly dependent on each other.

Discussion and Issues

Coupling

Coupling delineates the interdependence between modules. Modules exhibit **high coupling** when they are strictly dependent on each other.

A low level of coupling (or loose coupling) facilitates the independent analysis, comprehension, modification, testing, or reuse of each module.

Discussion and Issues

Coupling

Coupling delineates the interdependence between modules. Modules exhibit **high coupling** when they are strictly dependent on each other.

A low level of coupling (or loose coupling) facilitates the independent analysis, comprehension, modification, testing, or reuse of each module.

The worst form of coupling is **control coupling**.
In this scenario, the client passes a flag (e.g., `happy_hour`) that controls the behavior and influences the inner logic of the server module. This results in a **loss of encapsulation**, as the client gains insight into the internal workings of the server.

Inheritance approach

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

# START HAPPY HOUR (50% discount)
customer1.change_class(CustomerHappyHour)
# You are changing class at runtime!
# It is technically 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
```

```python
from abc import ABC, abstractmethod


class Customer(ABC):

  def __init__(self):
    self._cost = 0

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

  @abstractmethod
  def add_drink(self, n, unit_cost):
    self._cost += n * unit_cost

  def change_class(self, new_class):
    self.__class__ = new_class

  @staticmethod
  def get_actual_dress():
    print("normal dress")
```

```python
class CustomerNormalHour(Customer):

  def add_drink(self, n, unit_cost):
    super().add_drink(n, unit_cost)


class CustomerHappyHour(Customer):

  def add_drink(self, n, unit_cost):
    discount = 0.5
    d_price = unit_cost * discount
    super().add_drink(n, d_price)

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

Discussion and Issues

Enabling an object to **switch classes** results in code that is challenging to comprehend, maintain, modify, test, and debug. Additionally, it may introduce a **lack of consistency** among the involved objects.

Discussion and Issues

Enabling an object to **switch classes** results in code that is challenging to comprehend, maintain, modify, test, and debug. Additionally, it may introduce a **lack of consistency** among the involved objects.

When you change the class of an object from A to B, you end up with an **object of class B** that adheres to the instance methods of class B but still retains the **attributes of class A**.

```python
class A:

  def __init__(self):
    self.a = "attribute of A"

  def f(self):
    print("method of class A")


class B:

  def __init__(self):
    self.b = "attribute of B"

  def f(self):
    print("method of class B")
```

```python
print("object of class A")
obj = A()
print(obj.__dict__)
obj.f()
```

```
Output
--------------------
"object of class A"
{'a': 'attribute of A'}
"method of class A"
```

```python
print("Now the class is B")
obj.__class__ = B
print(obj.__dict__)
obj.f()
```

```
Output
--------------------
"Now the class is B"
{'a': 'attribute of A'}
"method of class B"
```

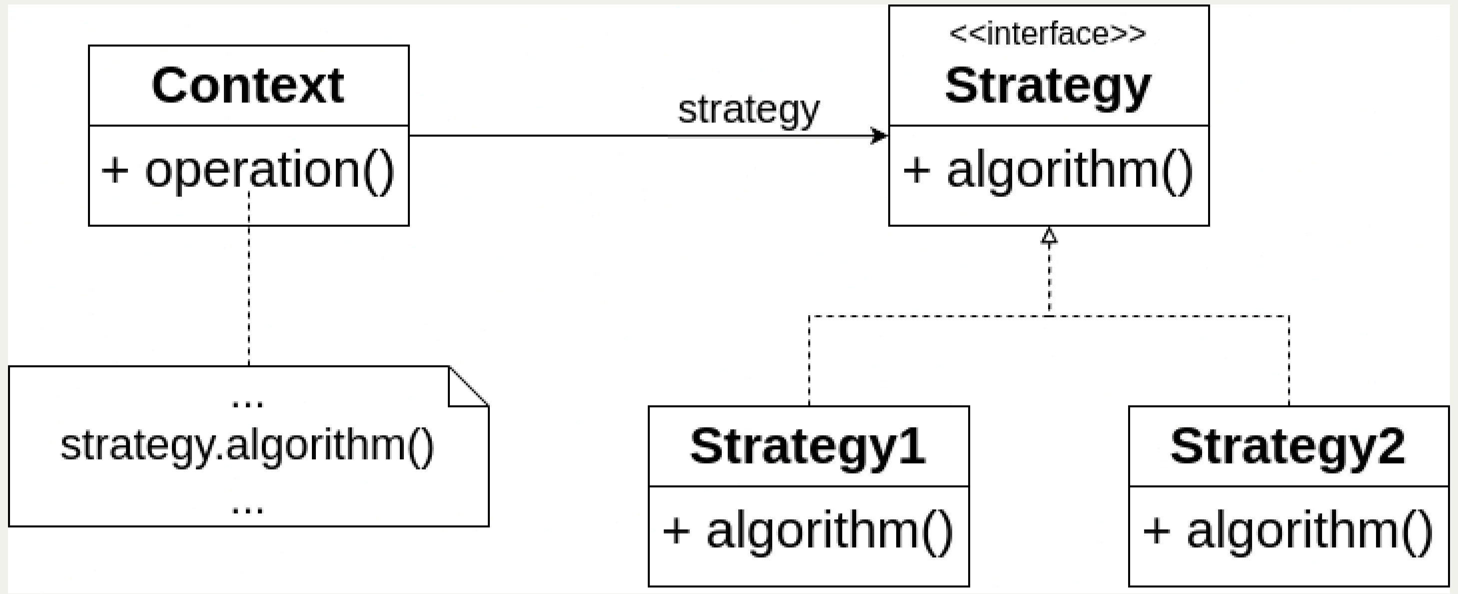
Discussion and Issues

If we aim to model **two distinct types of objects**, using inheritance might be appropriate.
However, this approach becomes insufficient when the **behavior is unrelated to the class**, especially when an object's behavior **depends on the 'environment'** and can **dynamically change at runtime due to an external event**.

For instance, the amount to be paid may change based on whether it is currently the 'happy hour' or not.

Solution: Strategy Design Pattern

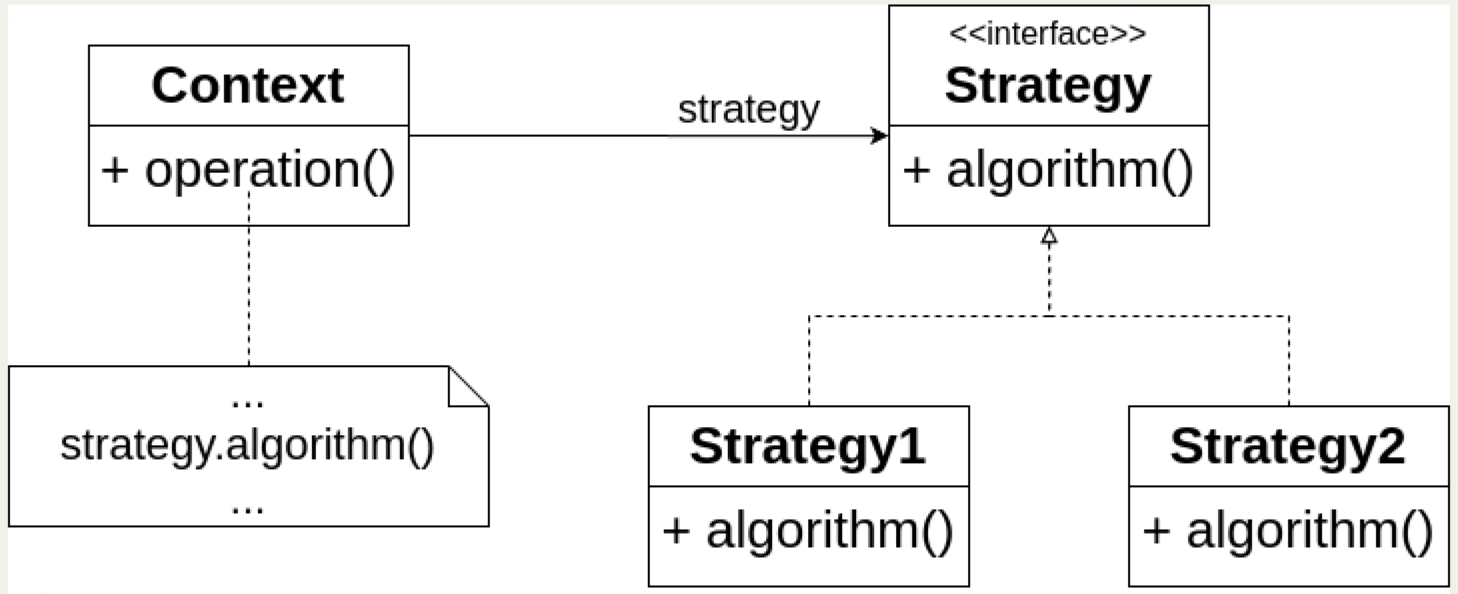
“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it” [Gamma, et al]



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 facilitates the dynamical selection of an algorithm. Rather than directly implementing a single algorithm, the code receives runtime instructions regarding the algorithm to be employed.

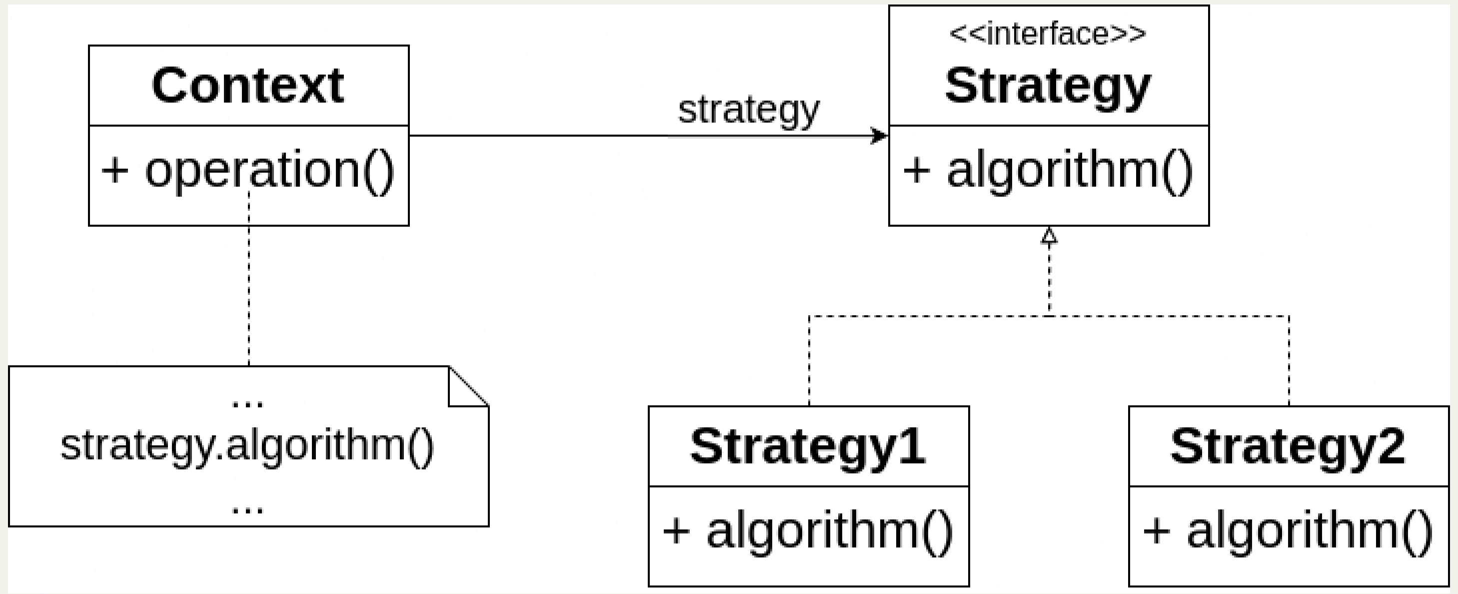


Solution: Strategy Design Pattern

“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 facilitates the dynamical selection of an algorithm. Rather than directly implementing a single algorithm, the code receives runtime instructions regarding the algorithm to be employed.

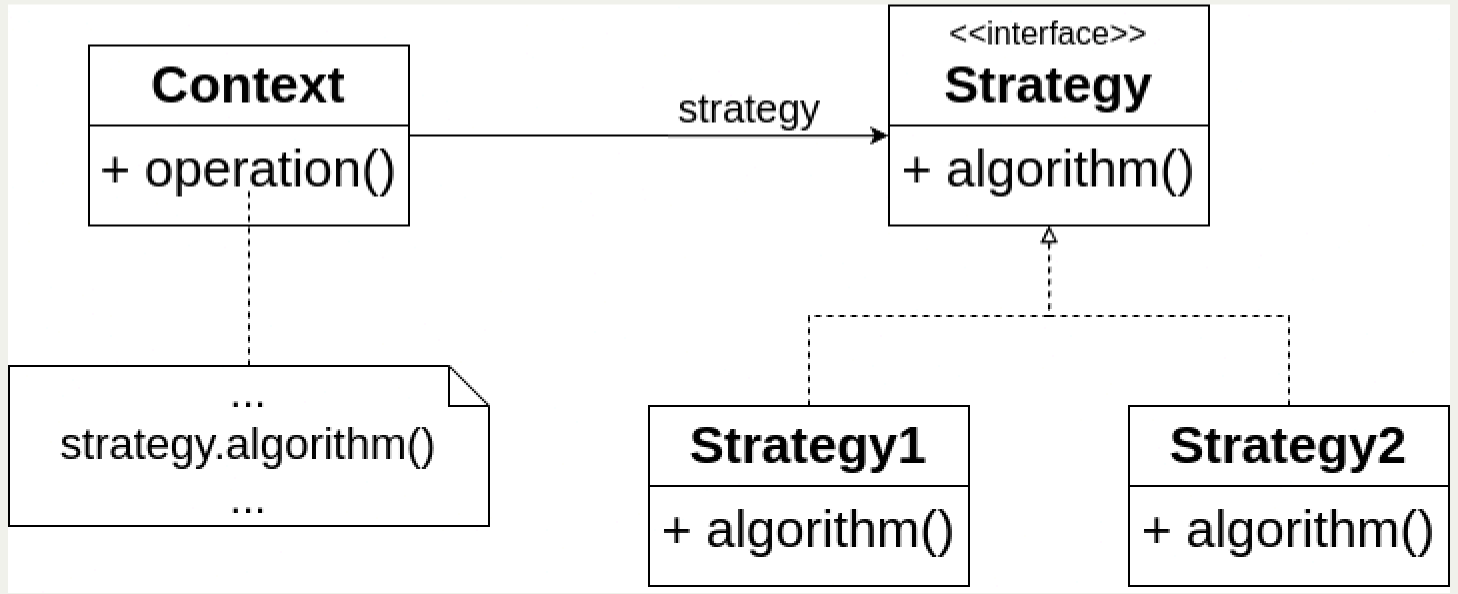
Postponing the decision on which algorithm to use at runtime enhances the flexibility and reusability of the code.



Class diagram

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

Context refers to the **Strategy** interface for executing an algorithm (`strategy.algorithm()`), thereby making **`Context`** **independent of the algorithm's specific implementation**.

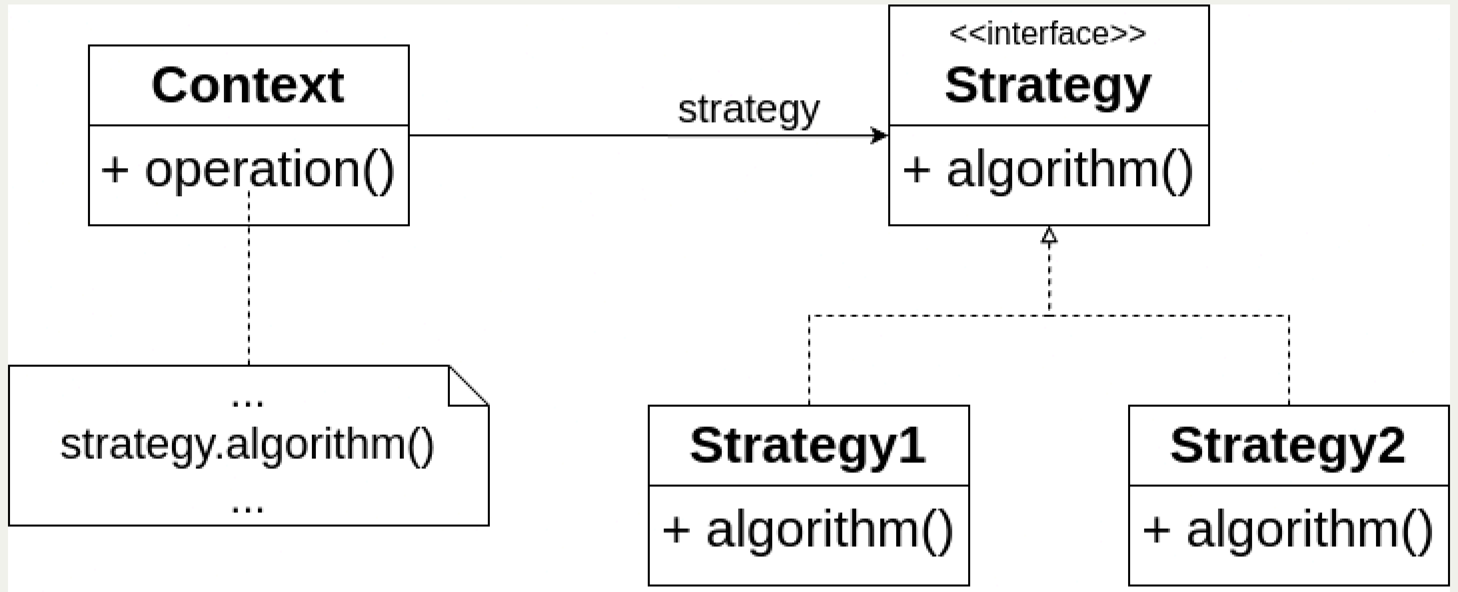


Class diagram

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

Context refers to the **Strategy** interface for executing an algorithm (`strategy.algorithm()`), thereby making **`Context` independent of the algorithm's specific implementation**.

The `Strategy1` and `Strategy2` classes implement the Strategy interface, effectively providing concrete implementations for the algorithms.



Example

One algorithm performs more effectively with small input sizes, whereas the other excels with larger input sizes. Or one algorithm functions effectively with entirely disordered data, while others perform better when only a few data deviate from the required order.

Example

One algorithm performs more effectively with small input sizes, whereas the other excels with larger input sizes. Or one algorithm functions effectively with entirely disordered data, while others perform better when only a few data deviate from the required order.

The Strategy pattern can be employed to dynamically determine which algorithm to use based on the input data during runtime.

Implementation

The strategy pattern typically involves storing a **reference to code** in a data structure and retrieving it.

Implementation

The strategy pattern typically involves storing a **reference to code** in a data structure and retrieving it.

This can be done using:

- Function pointer
- First-class function
- Classes or class instances

Implementation

The strategy pattern typically involves storing a **reference to code** in a data structure and retrieving it.

This can be done using:

- Function pointer
- First-class function
- Classes or class instances

The **STRATEGY Design Pattern** offers a method 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].

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**.

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
- enhances the quality of the design by promoting loose coupling.

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**.

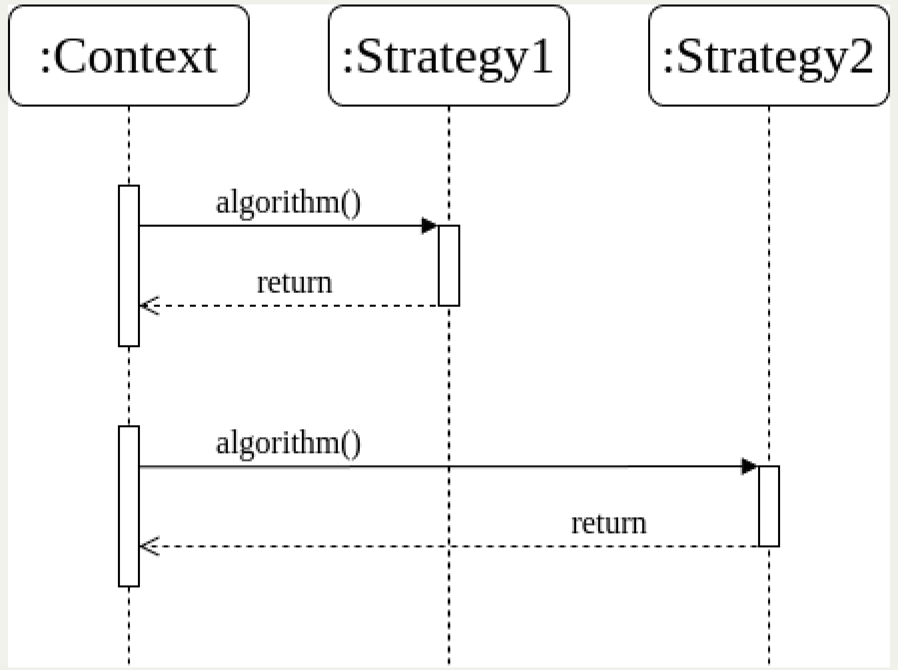
Following this principle

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

NB: It is *impossible* to design a module that is entirely closed against all forms of changes.

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

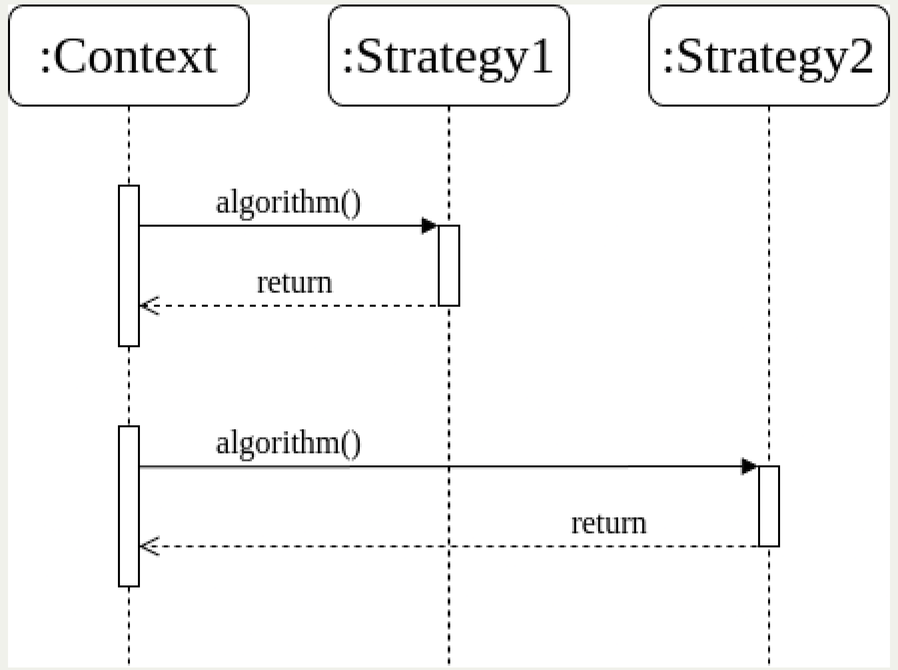
The **Context** object (the `Customer` in the current example) delegates an algorithm to various **Strategy** objects.



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

The **Context** object (the `Customer` in the current example) delegates an algorithm to various **Strategy** objects.

Context invokes `algorithm()` on a **Strategy1** object, executing the algorithm and receiving the result.

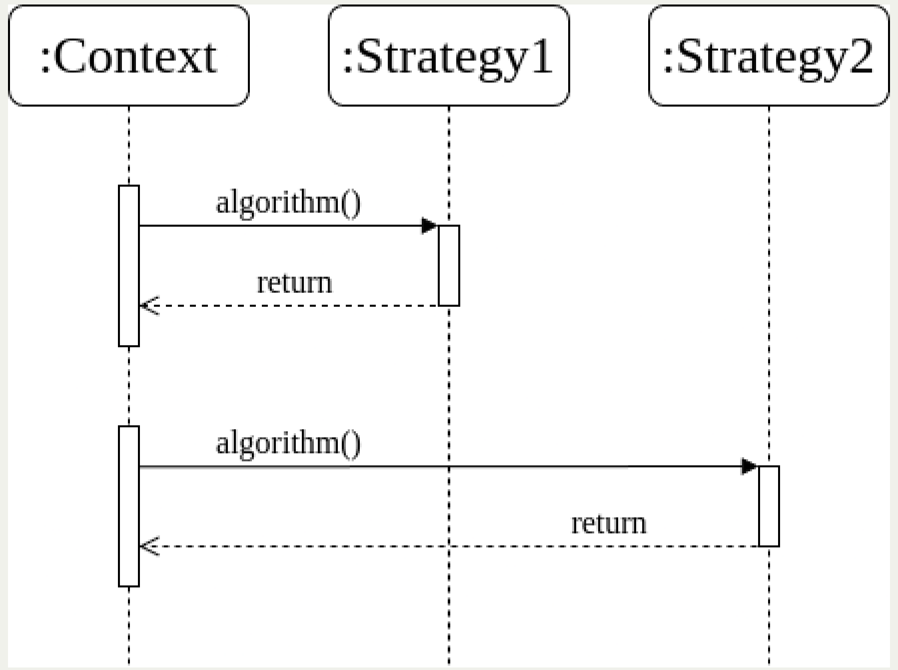


The **Sequence diagram** illustrates the run-time interactions.

The **Context** object (the `Customer` in the current example) delegates an algorithm to various **Strategy** objects.

Context invokes `algorithm()` on a **Strategy1** object, executing the algorithm and receiving the result.

Context subsequently alters its strategy and invokes `algorithm()` on a **Strategy2** object, executing the algorithm and receiving the result.



In summary:

We can define two `Strategy` objects that implement distinct strategies, each comprising different versions of methods such as `compute_price()`, `get_actual_dress()`, and others.

In summary:

We can define two `Strategy` objects that implement distinct strategies, each comprising different versions of methods such as `compute_price()`, `get_actual_dress()`, and others.

The `Customer` object has **a `Strategy` object as an attribute,** and delegates to the `Strategy` the execution of the desired action.

In summary:

We can define two `Strategy` objects that implement distinct strategies, each comprising different versions of methods such as `compute_price()`, `get_actual_dress()`, and others.

The `Customer` object has **a `Strategy` object as an attribute,** and delegates to the `Strategy` the execution of the desired action.

The `Customer` object can dynamically **switch** its (inner object) **strategy** based on changes in the environment (e.g., during Happy Hour) or in response to external signals.

CLIENT CODE

```python
# Prepare strategies
# strategies implement the methods
# `compute_price` and `get_actual_dress`
normal_strategy = NormalStrategy()
happy_hour_strategy = HappyHourStrategy()

# NORMAL BILLING
customer1 = Customer(normal_strategy)
customer1.add_drink(1, 7)
customer1.get_actual_dress()
```

```python
# START HAPPY HOUR (50% discount)
customer1.strategy = happy_hour_strategy
customer2=Customer(happy_hour_strategy)

customer2.add_drink(1, 7)
customer1.add_drink(2, 5)
customer2.add_drink(2, 5)
customer1.get_actual_dress()

# FINAL BILL
customer1.print_bill()  # 12
customer2.print_bill()  # 8.5
```

Implement classes `NormalStrategy`, `HappyHourStrategy`, `Customer`

Implement classes `NormalStrategy`, `HappyHourStrategy`, `Customer`

```python
from abc import ABC, abstractmethod


class Strategy(ABC):

  @abstractmethod
  def compute_price(self, value):
    pass  # DO NOTHING

  @abstractmethod
  def get_actual_dress(self):
    pass  # DO NOTHING


class NormalStrategy(Strategy):
  def compute_price(self, value):
    return value

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


class HappyHourStrategy(Strategy):

  def compute_price(self, value):
    discount = 0.5
    return value * discount

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

Implement classes `NormalStrategy`, `HappyHourStrategy`, `Customer`

```python
class Customer:

  def __init__(self, strategy):
    self._cost = 0
    self.strategy = strategy

  @property
  def strategy(self):
    return self._strategy

  @strategy.setter
  def strategy(self, strategy):
    self._strategy = strategy

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

  def add_drink(self, n, unit_cost):
    self._cost += self.strategy.compute_price(n * unit_cost)

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

The utilization of the Strategy pattern enables us to alter the behavior of a class without the need to override the methods of the base class through inheritance.

The utilization of the Strategy pattern enables us to alter the behavior of a class without the need to override the methods of the base class through inheritance.

In adherence to the strategy pattern, the object's behavior should be encapsulated using interfaces.
This adheres to the open/closed principle. The code should be open for extension but closed for modification.

The utilization of the Strategy pattern enables us to alter the behavior of a class without the need to override the methods of the base class through inheritance.

In adherence to the strategy pattern, the object's behavior should be encapsulated using interfaces.
This adheres to the open/closed principle. The code should be open for extension but closed for modification.

If the various strategies involve a single function, for instance, distinct implementations of the `compute_price` method, creating a class for each strategy becomes superfluous.

The utilization of the Strategy pattern enables us to alter the behavior of a class without the need to override the methods of the base class through inheritance.

In adherence to the strategy pattern, the object's behavior should be encapsulated using interfaces.
This adheres to the open/closed principle. The code should be open for extension but closed for modification.

If the various strategies involve a single function, for instance, distinct implementations of the `compute_price` method, creating a class for each strategy becomes superfluous.

Instead, we can directly create a function for each strategy. In this scenario, the object will store a reference or a pointer to the function, depending on the language employed.

```python
def compute_price_normal(value):
  return value

def compute_price_happy_hour(value):
  discount = 0.5
  return value * discount


class Customer:

  def __init__(self, strategy_price):
    self._cost = 0
    self.compute_price = strategy_price

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


customer1 = Customer(compute_price_normal)
customer1.add_drink(1, 7)

# START HAPPY HOUR (50% discount)
customer1.compute_price = compute_price_happy_hour
customer1.add_drink(2, 5)
```

Consider having two distinct sets of strategies, each independently selectable based on specific criteria.
For instance, one set involves **clothing strategies** dependent on season and dress code, while the other set involves **pricing strategies** dependent on time.

Consider having two distinct sets of strategies, each independently selectable based on specific criteria.
For instance, one set involves **clothing strategies** dependent on season and dress code, while the other set involves **pricing strategies** dependent on time.

We can define two separate class hierarchies and, through composition, construct an object that encapsulates a strategy from the first set and one from the second.
This approach enables the maintenance of a simplified class hierarchy.

Consider having two distinct sets of strategies, each independently selectable based on specific criteria.
For instance, one set involves **clothing strategies** dependent on season and dress code, while the other set involves **pricing strategies** dependent on time.

We can define two separate class hierarchies and, through composition, construct an object that encapsulates a strategy from the first set and one from the second.
This approach enables the maintenance of a simplified class hierarchy.

Opting for a solution based **only on inheritance** would require implementing **a class for each combination of strategies**, resulting in a more intricate and likely superfluous architecture (as some classes may never be utilized).

Example

If we have 5 types of payment based on time (`morning`, `afternoon`, `happy hour`, etc.), and 5 types of clothing based on weather conditions (`cold`, `hot`, ...), we would need to implement 25 possible behaviors with 25 classes.

Creating a class for each configuration would significantly complicate our architecture.

		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