Design Patterns 2

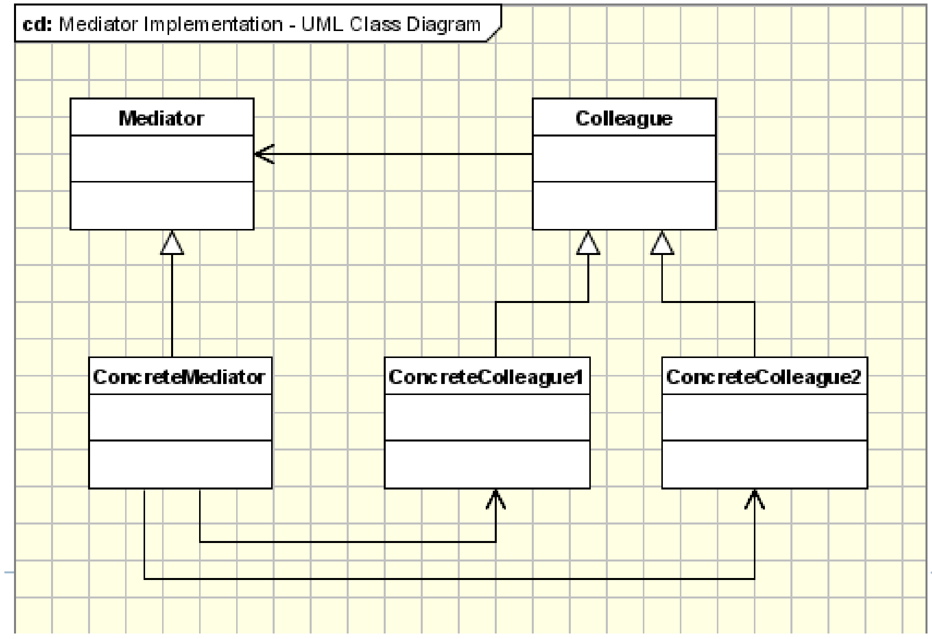
Padrões para Comportamento e Criação

Padrões para Comportamento

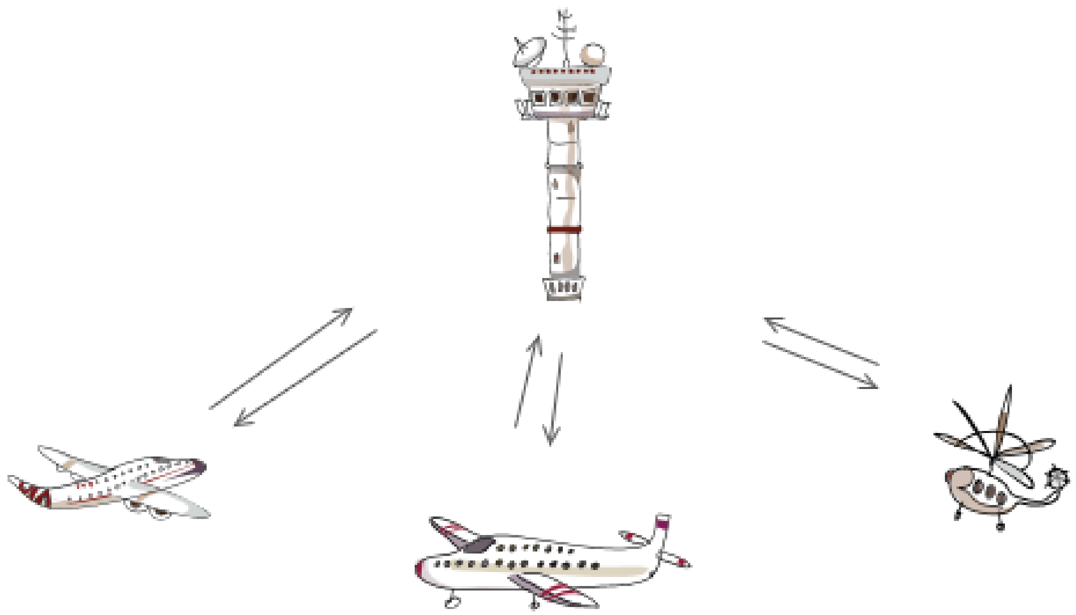
- Os padrões para comportamento proveem soluções para comunicação entre objetos.

Mediador

- Intenção: ter um objeto que intermedia a interação com um grupo de objetos.
- Motivação: Quando tenho um conjunto de objetos torna-se necessário uma forma de interação entre os objetos sem que seja necessário a criação de referencias desses objetos dentro dos diferentes objetos.



ATC Mediator



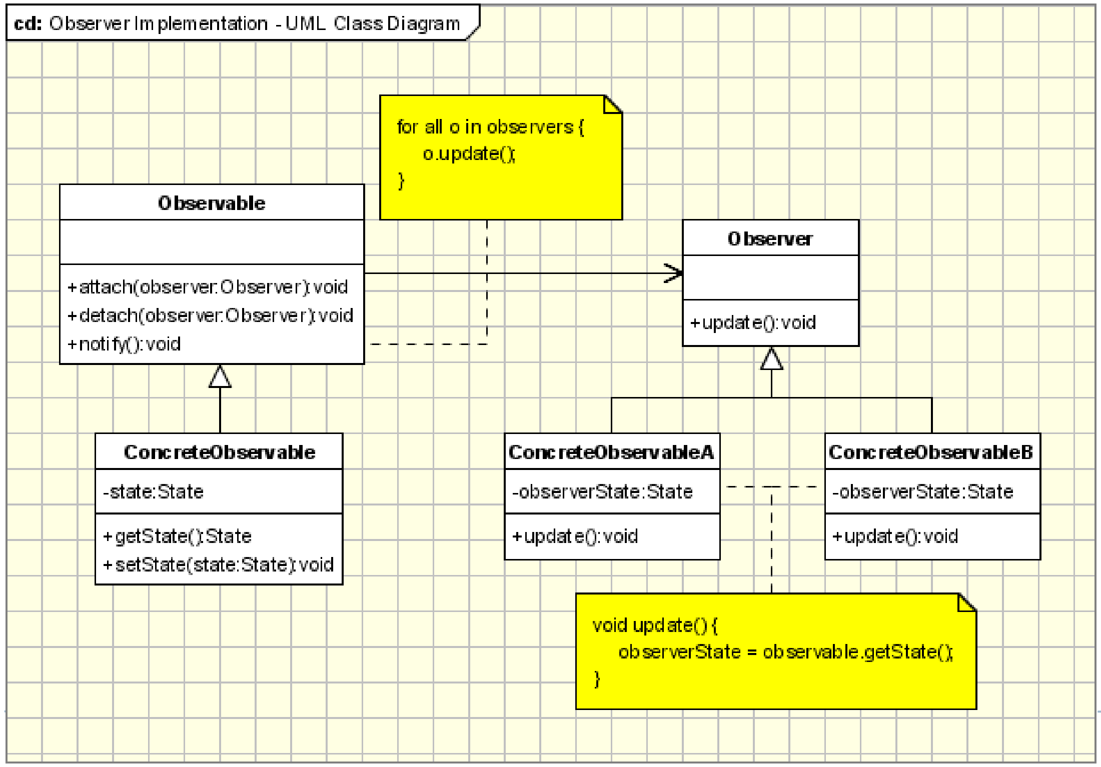
```
class Mediator:
 """ Implement cooperative behavior by coordinating
Colleague objects. Know and maintains its colleagues. ""“

def __init__(self):
    self._colleague_1 = Colleague1(self)
    self._colleague_2 = Colleague2(self)
```

```
class Colleague1:
 """ Know its Mediator object.
Communicate with its mediator
whenever it would have
otherwise communicated with
another colleague. """

def __init__(self, mediator):
    self._mediator = mediator
```

```
class Colleague2:
""" Know its Mediator object.
Communicate with its mediator
whenever it would have otherwise
communicated with another
colleague. ""“

def __init__(self, mediator):
    self._mediator = mediator
```

Observador

- Intenção: Permitir a criação de dependencia um para muitos. De modo que se o estado de um objeto muda. Todos os outros são notificados.
- Motivação: Um objeto gera um evento e ele notifica todos os clientes do evento.


cd: Observer Implementation - UML Class Diagram
for all o in observers { o.update(); }
Observable
+attach(observer:Observer):void
+detach(observer:Observer):void
+notify():void
Observer
+update():void
ConcreteObservable
-state:State
+getState():State
+setState(state:State):void
ConcreteObservableA
-observerState:State
+update():void
ConcreteObservableB
-observerState:State
+update():void
void update() { observerState = observable.getState(); }

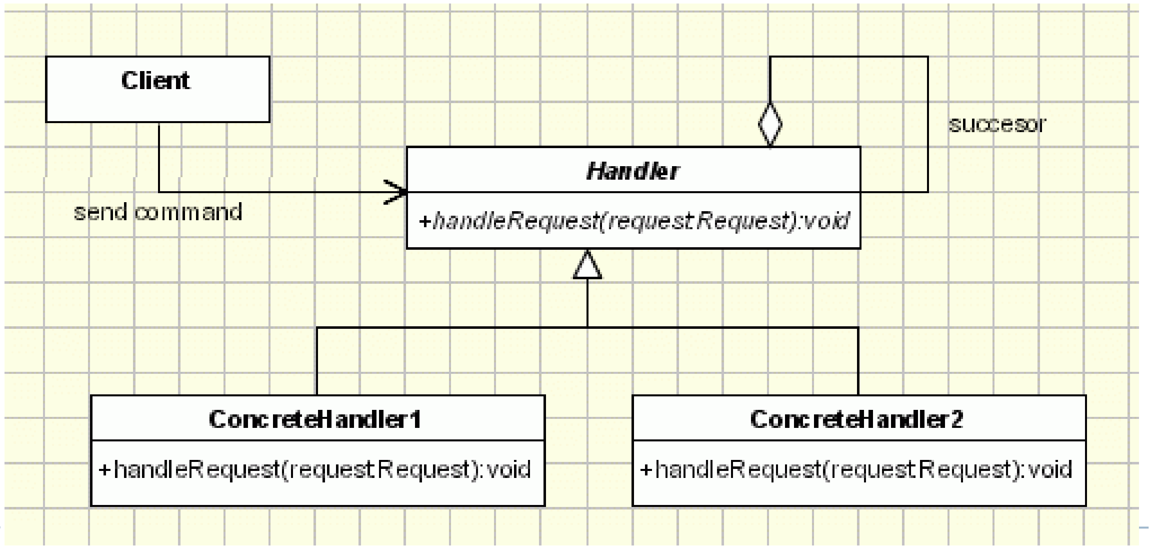
```
import abc

class Subject:
 """ Know its observers. Any number of Observer objects may observe a subject.
Send a notification to its observers when its state changes. ""“

def __init__(self):
      self._observers = set()
      self._subject_state = None

def attach(self, observer):
      observer._subject = self
      self._observers.add(observer)

def detach(self, observer):
     observer._subject = None
     self._observers.discard(observer)

def _notify(self):
     for observer in self._observers:
          observer.update(self._subject_state)
@property
def subject_state(self):
      return self._subject_state
@subject_state.setter
def subject_state(self, arg):
      self._subject_state = arg
      self._notify()
```

```
class Observer(metaclass=abc.ABCMeta):
""" Define an updating interface for objects that
should be notified of changes in a subject. ""“

def __init__(self):
    self._subject = None
    self._observer_state = None

@abc.abstractmethod
def update(self, arg):
     pass

class ConcreteObserver(Observer):
""" Implement the Observer updating interface to
keep its state consistent with the subject's. Store
state that should stay consistent with the subject's.
""“

def update(self, arg):
    self._observer_state = arg

# ...
```

```
def main():
    subject = Subject()
    concrete_observer =
ConcreteObserver()
    subject.attach(concrete_observer)
    subject.subject_state = 123

if __name__ == "__main__":
    main()
```

Cadeia de Responsabilidades

- Intenção: muitas vezes o atendimento a um evento não é executado pelo primeiro objeto receptor, que passa o evento para outro objeto.
- Motivação: Desacoplar o cliente do servidor, permitindo o servidor selecionar o melhor objeto para atender o cliente.


Client
send command
Handler
+handleRequest(request:Request):void
succesor
ConcreteHandler1
+handleRequest(request:Request):void
ConcreteHandler2
+handleRequest(request:Request):void

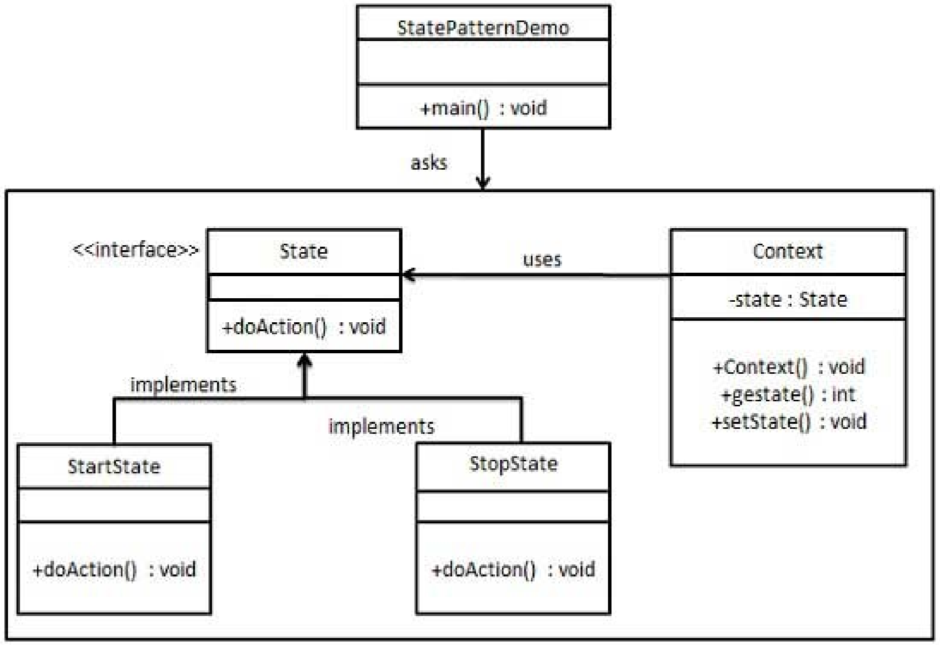
```
import abc class

Handler(metaclass=abc.ABCMeta):
""" Define an interface for handling
requests. Implement the successor link. ""“

def __init__(self, successor=None):
    self._successor = successor

@abc.abstractmethod
    def handle_request(self):
        pass
```

```
class ConcreteHandler1(Handler):
""" Handle request, otherwise forward
it to the successor. ""“

def handle_request(self):
    if True: # if can_handle:
        pass
    elif self._successor is not None:
        self._successor.handle_request()
```

```
class ConcreteHandler2(Handler):
""" Handle request, otherwise forward
it to the successor. ""“

def handle_request(self):
    if True: # if can_handle:
        pass
    elif self._successor is not None:
        self._successor.handle_request()



def main():
    concrete_handler_1 = ConcreteHandler1()
    concrete_handler_2 =
ConcreteHandler2(concrete_handler_1)
    concrete_handler_2.handle_request()

if __name__ == "__main__":
    main()
```

State

- Intenção: Alterar o comportamento de um objeto quando o estado interno é modificado.
- Motivação: Usar uma classe de contexto para representar a interface única para o mundo. Criar uma familia de classes State para representar os diferentes estados. Cada estado tem a sua implementa para o comportamento.



```
import abc

class Context:
""" Define the interface of interest to clients. Maintain an
instance of a ConcreteState subclass that defines the current
state. ""“

    def __init__(self, state):
        self._state = state

    def request(self):
        self._state.handle()

class State(metaclass=abc.ABCMeta):
""" Define an interface for encapsulating the behavior
associated with a particular state of the Context. """

    @abc.abstractmethod
    def handle(self):
        pass
```

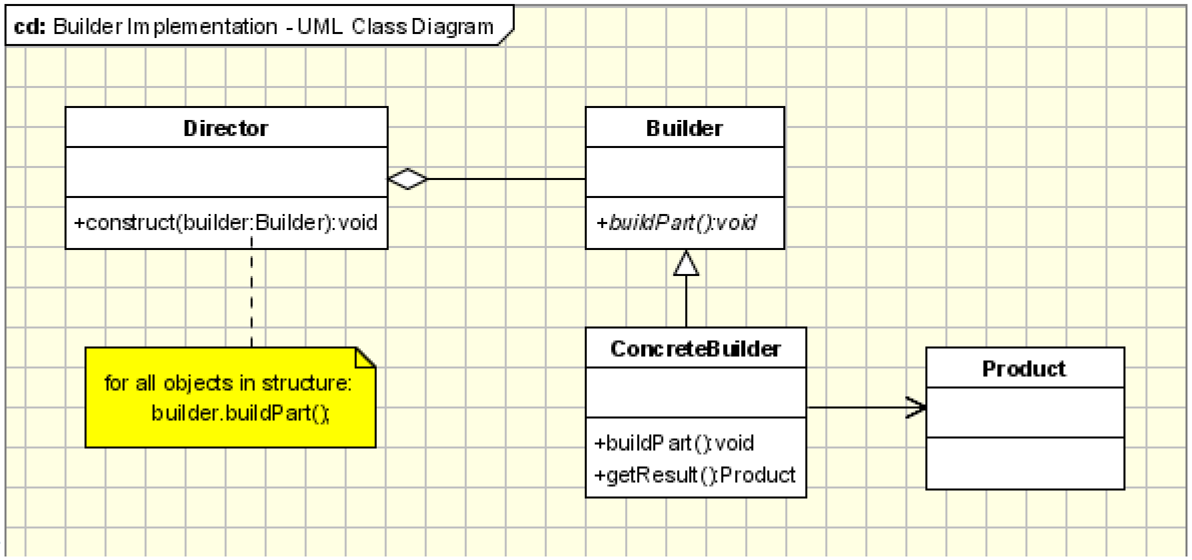
```
class ConcreteStateA(State):
    """
    Implement a behavior associated with a state of the Context.
    """

    def handle(self):
        pass


class ConcreteStateB(State):
    """
    Implement a behavior associated with a state of the Context.
    """

    def handle(self):
        pass

def main():
    concrete_state_a = ConcreteStateA()
    context = Context(concrete_state_a)
    context.request()

if __name__ == "__main__":
    main()
```

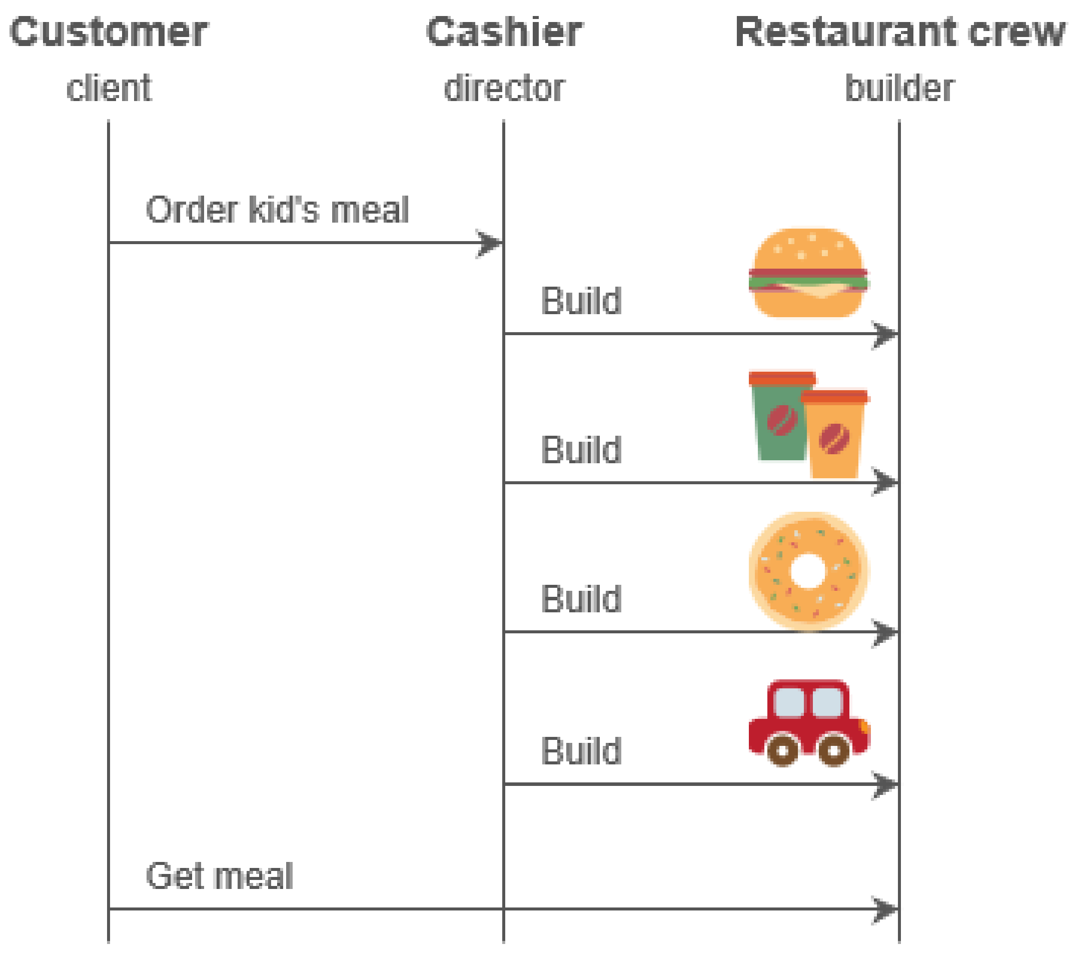
Padrões para Criação

- Padrões para criação são soluções para a instanciação de objetos.

Builder

- Intenção: ter uma interface comum para a criação de objetos. Cada sub-classe cuida dos detalhes de criação de objetos.
- Motivação: a criação de novos objetos pode se tornar uma atividade complexa. Pode ser necessário separar o processo de criação do objeto do uso em si do objeto.
- Implementação:




Customer
client
Cashier
director
Restaurant crew
builder
Order kid's meal
Build
Build
Build
Build
Get meal

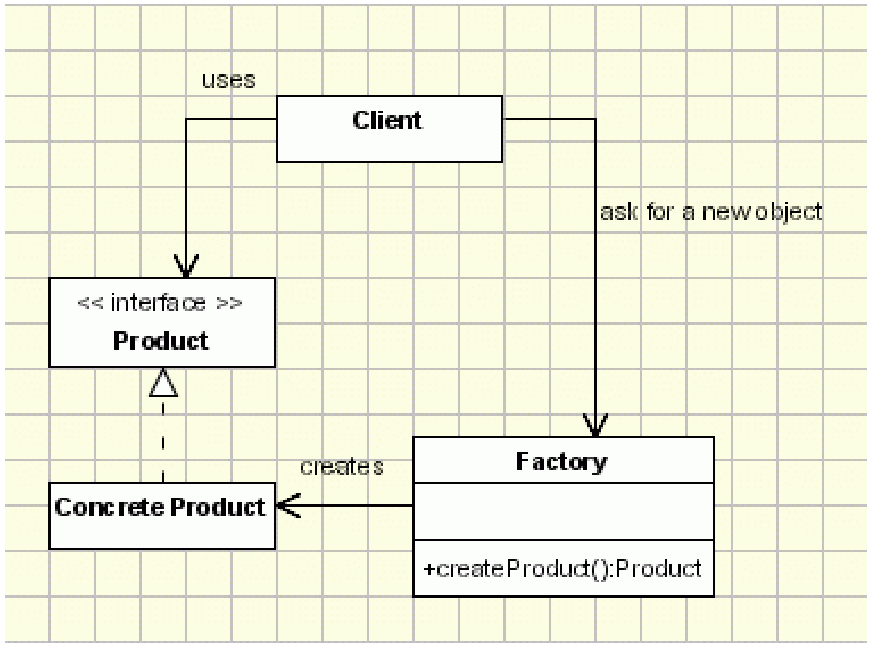
```
"""
Separate the construction of a complex object from its
representation so
that the same construction process can create different
representations.
"""

import abc


class Director:
    """
    Construct an object using the Builder interface.
    """

    def __init__(self):
        self._builder = None

    def construct(self, builder):
        self._builder = builder
        self._builder._build_part_a()
        self._builder._build_part_b()
        self._builder._build_part_c()
```

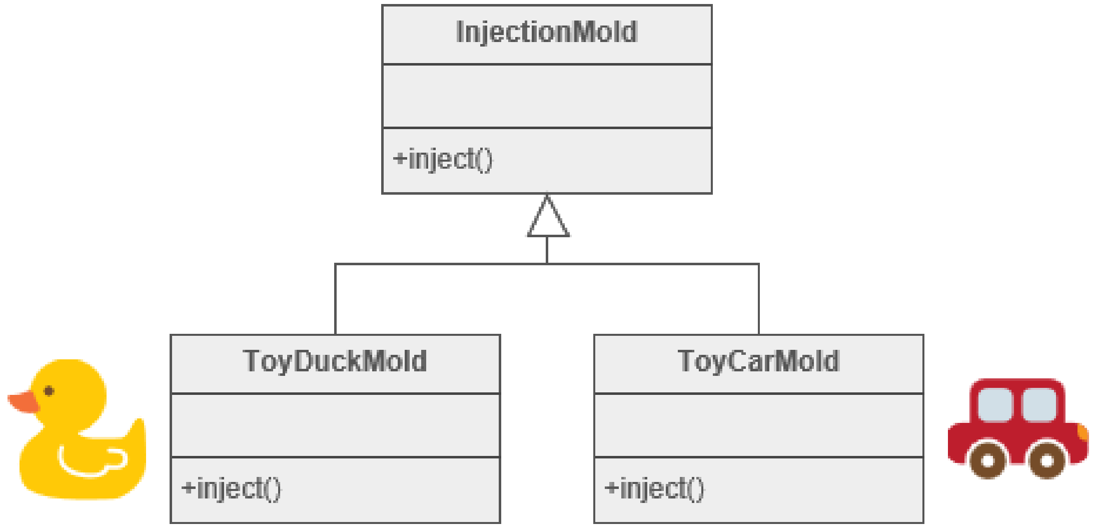
```
class Builder(metaclass=abc.ABCMeta):
    "“”    Specify an abstract interface for creating parts of a Product object.    "“”

    def __init__(self):
        self.product = Product()

    @abc.abstractmethod
    def _build_part_a(self):
        pass

    @abc.abstractmethod
    def _build_part_b(self):
        pass

    @abc.abstractmethod
    def _build_part_c(self):
        pass

class ConcreteBuilder(Builder):
    "“”    Construct and assemble parts of the product by implementing the Builder interface.
    Define and keep track of the representation it creates.
    Provide an interface for retrieving the product.    """

    def _build_part_a(self):
        pass

    def _build_part_b(self):
        pass

    def _build_part_c(self):
        pass
```

```
class Product:
    """
    Represent the complex object under construction.
    """

    pass


def main():
    concrete_builder = ConcreteBuilder()
    director = Director()
    director.construct(concrete_builder)
    product = concrete_builder.product


if __name__ == "__main__":
    main()
```

Fábrica

- Intenção: Abstrair a construção de objetos complexos.
- Motivação: Criar objetos sem expor a lógica de instanciação para o cliente. Clientes interagem com o objeto criado através de uma interface comum.




InjectionMold
+inject()
ToyDuckMold
+inject()
ToyCarMold
+inject()

```
"""
Define an interface for creating an object, but let subclasses decide
which class to instantiate. Factory Method lets a class defer instantiation to
subclasses.
"""“
import abc

class Creator(metaclass=abc.ABCMeta):
    """
    Declare the factory method, which returns an object of type Product.
    Creator may also define a default implementation of the factory  method
that returns a default ConcreteProduct object.  Call the factory method to
create a Product object.
    """


    def __init__(self):
        self.product = self._factory_method()

    @abc.abstractmethod
    def _factory_method(self):
        pass

    def some_operation(self):
        self.product.interface()
```

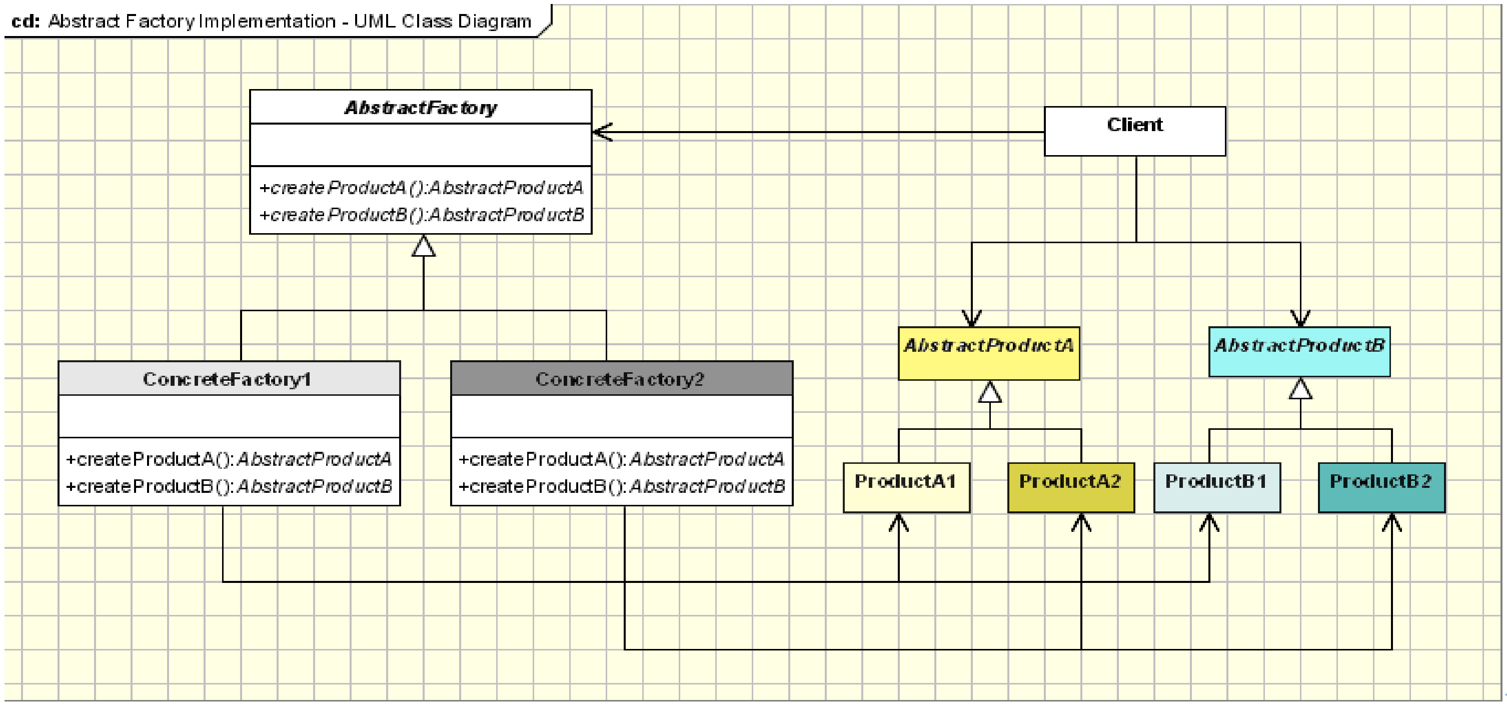
```
class ConcreteCreator1(Creator):
    """
    Override the factory method to return an instance of a
    ConcreteProduct1.
    """

    def _factory_method(self):
        return ConcreteProduct1()


class ConcreteCreator2(Creator):
    """
    Override the factory method to return an instance of a
    ConcreteProduct2.
    """

    def _factory_method(self):
        return ConcreteProduct2()
```

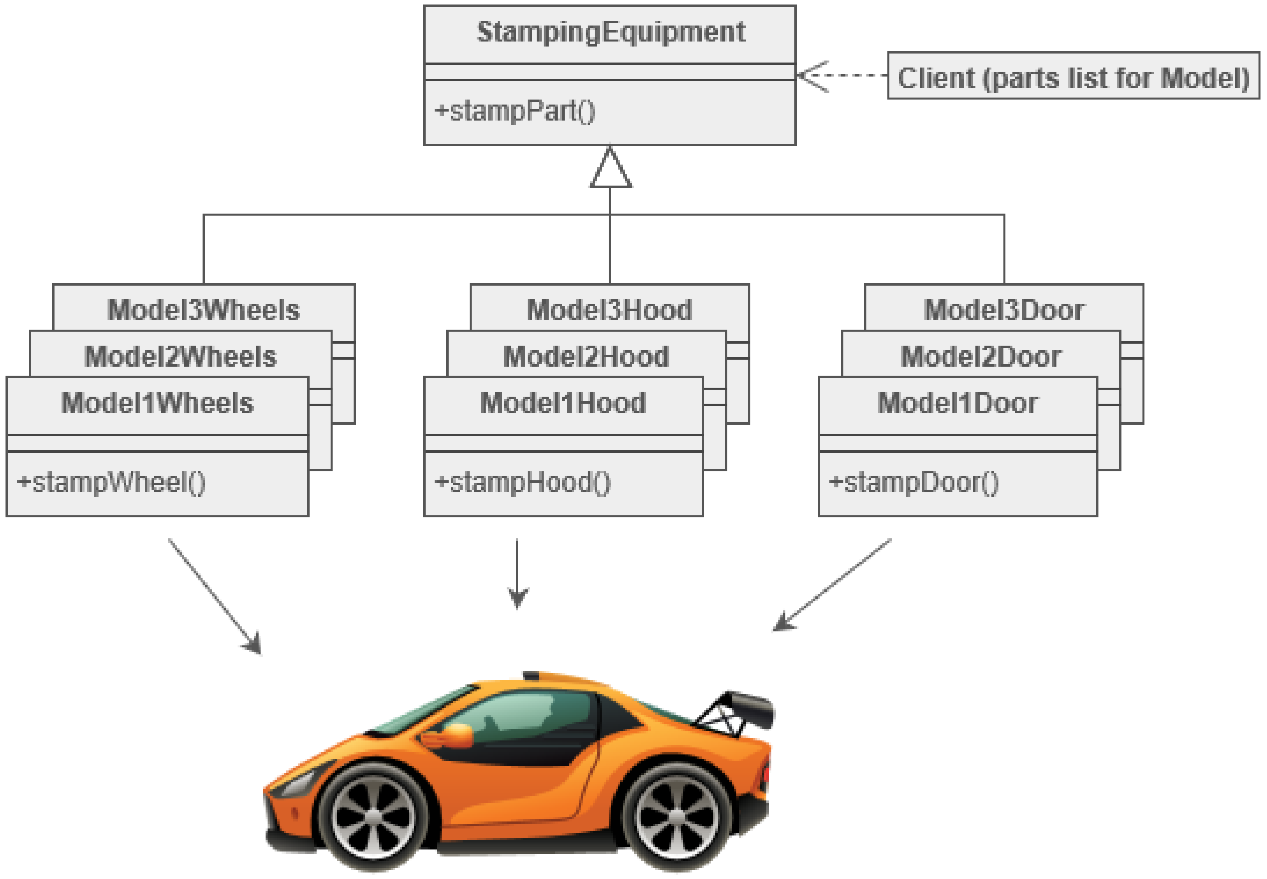
```
class Product(metaclass=abc.ABCMeta):
    """
    Define the interface of objects the factory method creates.
    ""“
    @abc.abstractmethod
    def interface(self):
        pass

class ConcreteProduct1(Product):
    """
    Implement the Product interface.
    """


    def interface(self):
        pass

class ConcreteProduct2(Product):
    """
    Implement the Product interface.
    """


    def interface(self):
        pass
```

```
def main():
    concrete_creator =
ConcreteCreator1()

concrete_creator.product.interface()
    concrete_creator.some_operation()


if __name__ == "__main__":
    main()
```

Fábrica Abstrata

- Intenção: Os objetos podem sofrer modificações futuras o que pode afetar o seu uso e criação. De modo a possibilitar um nível adicional de isolamento para uso e criação é definido o conceito de fábrica abstrata.
- Motivação: Oferecer interfaces para uso e criação sem especificar suas classes.


cd: Abstract Factory Implementation - UML Class Diagram
AbstractFactory
+createProductA():AbstractProductA
+createProductB():AbstractProductB
Client
ConcreteFactory1
+createProductA():AbstractProductA
+createProductB():AbstractProductB
ConcreteFactory2
+createProductA():AbstractProductA
+createProductB():AbstractProductB
AbstractProductA
AbstractProductB
ProductA1
ProductA2
ProductB1
ProductB2


StampingEquipment
+stampPart()
Client (parts list for Model)
Model3Wheels
Model2Wheels
Model1Wheels
+stampWheel()
Model3Hood
Model2Hood
Model1Hood
+stampHood()
Model3Door
Model2Door
Model1Door
+stampDoor()

```
"""
Provide an interface for creating families of related or
dependent
objects without specifying their concrete classes.
"""

import abc


class AbstractFactory(metaclass=abc.ABCMeta):
    """
    Declare an interface for operations that create abstract
product
    objects.
    """

    @abc.abstractmethod
    def create_product_a(self):
        pass

    @abc.abstractmethod
    def create_product_b(self):
        pass
```

```
class ConcreteFactory1(AbstractFactory):
    ""
    Implement the operations to create concrete product
objects.
    ""

    def create_product_a(self):
        return ConcreteProductA1()

    def create_product_b(self):
        return ConcreteProductB1()


class ConcreteFactory2(AbstractFactory):
    ""
    Implement the operations to create concrete product
objects.
    ""

    def create_product_a(self):
        return ConcreteProductA2()

    def create_product_b(self):
        return ConcreteProductB2()
```

```
class AbstractProductA(metaclass=abc.ABCMeta):
    """
    Declare an interface for a type of product object.
    ""“
    @abc.abstractmethod
    def interface_a(self):
        pass

class ConcreteProductA1(AbstractProductA):
    """
    Define a product object to be created by the corresponding concrete 
factory.
    Implement the AbstractProduct interface.
    ""“
    def interface_a(self):
        pass

class ConcreteProductA2(AbstractProductA):
    """
    Define a product object to be created by the corresponding concrete 
factory.
    Implement the AbstractProduct interface.
    ""“
    def interface_a(self):
```

```
class AbstractProductB(metaclass=abc.ABCMeta):
    """
    Declare an interface for a type of product object.
    ""“
    @abc.abstractmethod
    def interface_b(self):
        pass

class ConcreteProductB1(AbstractProductB):
    """
    Define a product object to be created by the corresponding concrete
factory.
    Implement the AbstractProduct interface.
    ""“
    def interface_b(self):
        pass

class ConcreteProductB2(AbstractProductB):
    """
    Define a product object to be created by the corresponding concrete
factory.
    Implement the AbstractProduct interface.
    ""“
    def interface_b(self):
```

```
def main():
    for factory in (ConcreteFactory1(), ConcreteFactory2()):
        product_a = factory.create_product_a()
        product_b = factory.create_product_b()
        product_a.interface_a()
        product_b.interface_b()


if __name__ == "__main__":
    main()
```

Cuidados

- Todo bom profissional domina os patterns da sua área. Entretanto:
  - O uso de Design Pattern não é obrigatório.
  - Utilizar quando houver necessidades concretas para a obtenção de flexibilidade e facilidades para modificação.

Exercício

- Criar exemplos de programas com os seguintes padrões:
  - Fábrica Abstrata
  - Mediador
  - State