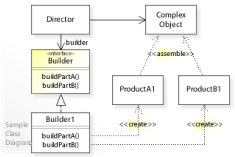
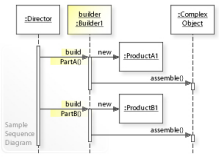
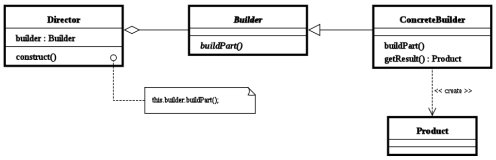
① Design pattern description

Builder : Creational Pattern







② Design pattern example

```
from __future__ import print_function
from abc import ABCMeta, abstractmethod


class Car(object):
    def __init__(self, wheels=4, seats=4, color="Black"):
        self.wheels = wheels
        self.seats = seats
        self.color = color

    def __str__(self):
        return "This is a {0} car with {1} wheels and {2} seats.".format(
            self.color, self.wheels, self.seats
        )
```

```
class Builder:
    __metaclass__ = ABCMeta

    @abstractmethod
    def set_wheels(self, value):
        pass

    @abstractmethod
    def set_seats(self, value):
        pass

    @abstractmethod
    def set_color(self, value):
        pass

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

```
class CarBuilder(Builder):
    def __init__(self):
        self.car = Car()

    def set_wheels(self, value):
        self.car.wheels = value
        return self

    def set_seats(self, value):
        self.car.seats = value
        return self

    def set_color(self, value):
        self.car.color = value
        return self

    def get_result(self):
        return self.car
```

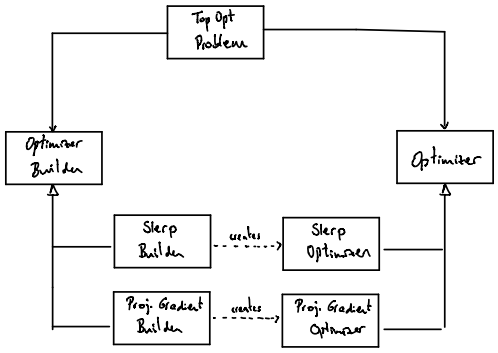
```
class CarBuilderDirector(object):
    @staticmethod
    def construct():
        return CarBuilder()
                .set_wheels(8)
                .set_seats(4)
                .set_color("Red")
                .get_result()


car = CarBuilderDirector.construct()
print(car)
```


```
class CarBuilderDirector(object):
    @staticmethod
    def construct():
        return CarBuilder()
                .set_wheels(8)
                .set_seats(4)
                .set_color("Red")
                .get_result()
```

(3) Existing Example in FEM-MAT-OO ?

Not explicitly.

(4) Design pattern proposal in FEM-MAT-OO ?



function top_opt_problem

builder = sterpbuilder
builder.set kappa (5)
builder.set
builder.construct()
optimizer = builder.getoptimizer()

- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
- Builder often builds a Composite.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- Sometimes creational patterns are complementary: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.

A factory is simply a wrapper function around a constructor (possibly one in a different class). The key difference is that a factory method pattern requires the entire object to be built in a single method call, with all the parameters pass in on a single line. The final object will be returned.

A builder pattern, on the other hand, is in essence a wrapper object around all the possible parameters you might want to pass into a constructor invocation. This allows you to use setter methods to slowly build up your parameter list. One additional method on a builder class is a build() method, which simply passes the builder object into the desired constructor, and returns the result.