Software design patterns
Part 3

Course: Object Oriented Programming (OOP)
CTU, FS, U12110
Matouš Cejnek

Contents

- Some design patterns

Builder

Creational pattern

Builder

- It is **creational** software design pattern
- The intent of the Builder design pattern is to separate the construction of a complex object from its representation.

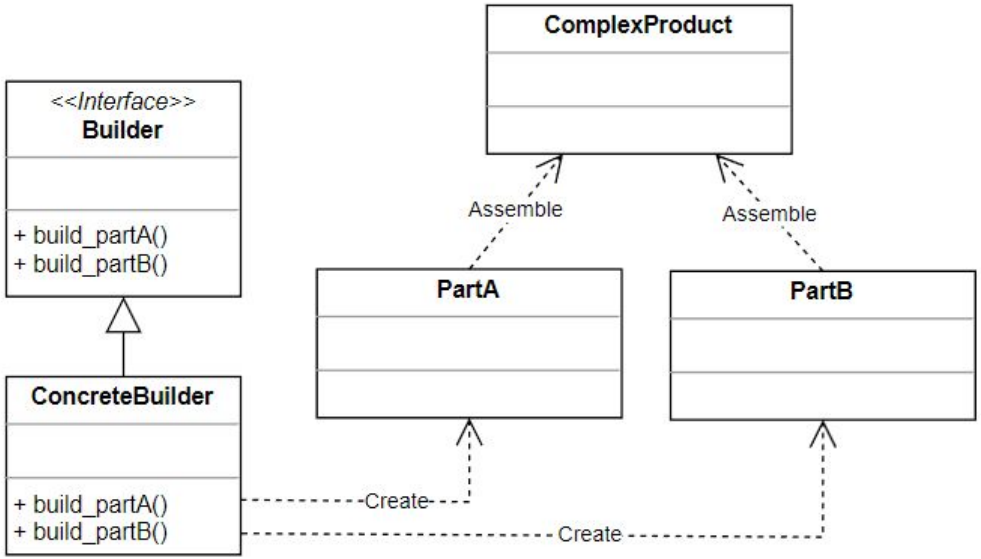
Builder - requirements

- How can a class create different representations of a complex object?
- How can a class that includes creating a complex object be simplified?

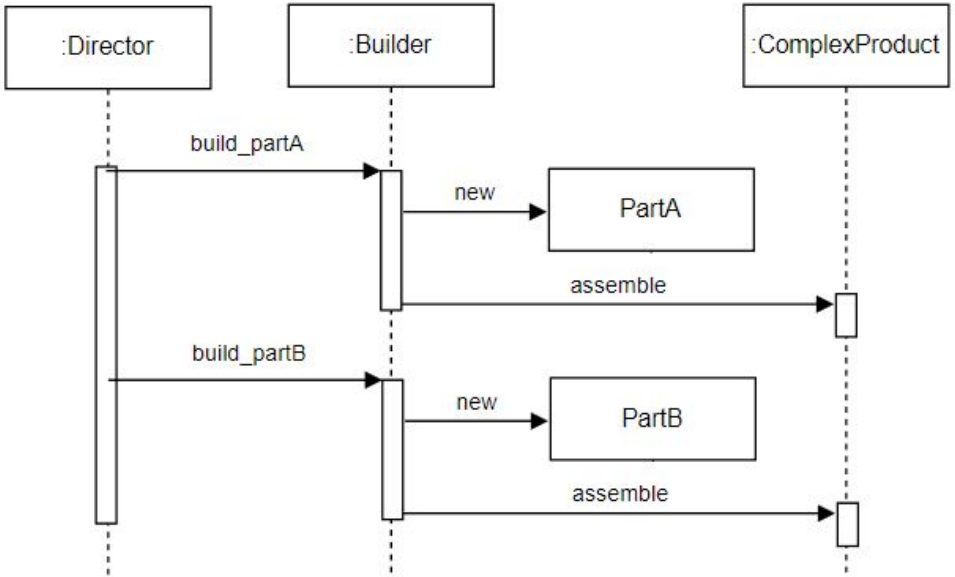
Builder - solution

1. Encapsulate creating and assembling the parts of a complex object in a separate Builder object.
2. A class delegates object creation to a Builder object instead of creating the objects directly.

Builder - class diagram



Builder - sequence diagram



Builder - example part 1

```
from abc import ABC, abstractmethod

class Builder(ABC):
    @abstractmethod
    def get_product(self):
        pass

    @abstractmethod
    def create_partA(self):
        pass

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

Builder - example part 2

```
class ProductBuilder1(Builder):

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

    def get_product(self):
        return self._product

    def create_partA(self):
        self._product.add("PartA1")

    def create_partB(self):
        self._product.add("PartB1")
```

Builder - example part 3

```
class Product():

    def __init__(self):
        self.parts = []

    def add(self, part):
        self.parts.append(part)

    def list_parts(self):
        print(f"Parts: {', '.join(self.parts)}")
```

Builder - example part 4

```
builder = ProductBuilder1()
builder.create_partA()
builder.create_partB()
builder.create_partA()

product001 = builder.get_product()
product001.list_parts()
```

>> Parts: PartA1, PartB1, PartA1

Builder with director - example part 1

```
class ProductBuilder1(Builder):

    def __init__(self):
        self.new()

    def new(self):
        self._product = Product()

    …
```

Builder with director - example part 2

```
class Director:

    def __init__(self, builder):
        self._builder = builder

    def build_AB_product(self):
        self._builder.new()
        self._builder.create_partA()
        self._builder.create_partB()
        return self._builder.get_product()
```

Builder with director - example part 3

```
director = Director(ProductBuilder1())

product001 = director.build_AB_product()
product001.list_parts()

product002 = director.build_ABA_product()
product002.list_parts()
```

>> Parts: PartA1, PartB1
>> Parts: PartA1, PartB1, PartA1

Flyweight

Structural pattern

Flyweight

- It is **structural** software design pattern
- The flyweight software design pattern refers to an object that minimizes memory usage by sharing some of its data with other similar objects.

Flyweight - requirements

- Dealing with large numbers of objects with simple repeated elements that would use a large amount of memory if individually stored.

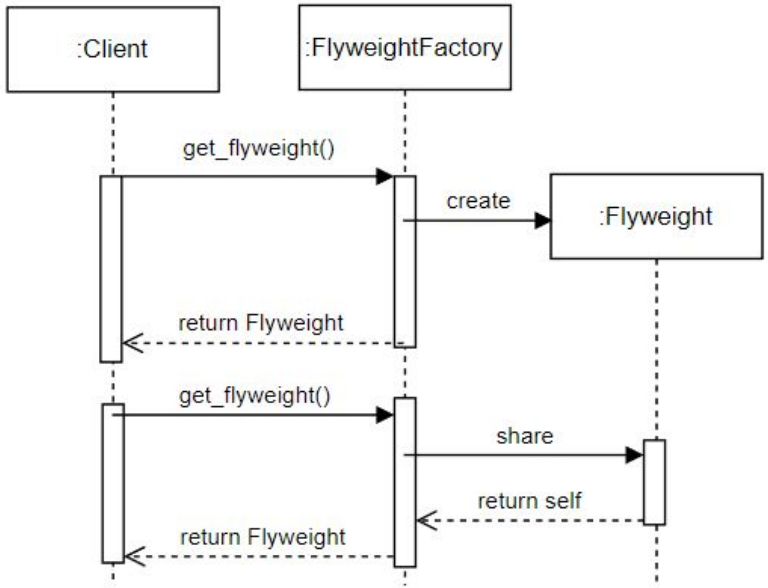
Flyweight - solution

- Hold shared data in external data structures and pass it to the objects temporarily when they are used.

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Flyweight - class diagram

Flyweight - sequence diagram



Flyweight - example part 1

```
class Pixel:

    def __init__(self, rgb_code):
        self.rgb_code = rgb_code
        self.values = {
            "red": int(self.rgb_code[1:3], base=16),
            "green": int(self.rgb_code[3:5], base=16),
            "blue": int(self.rgb_code[5:7], base=16)
        }
```

RGB code is something like: #abcdef (between #000000 and #ffffff)

Flyweight - example part 2

```
class Image:

    def __init__(self, array):
        self.height = len(array)
        self.width = len(array[0])
        self.palette = {}
        self.pixel_array = self.compress(array)

    def compress(self, array):
        return [[self.add_pixel(array[y][x])
                 for x in range(self.width)]
                 for y in range(self.height)]
```

Flyweight - example part 3

```
    def add_pixel(self, code):
        if not code in self.palette:
            self.palette[code] = Pixel(code)
        return self.palette[code]

    def __str__(self):
        msg = "Image of size {}x{} with {} unique pixels."
        return msg.format(
            self.height, self.width,
len(self.palette.keys()))

    def get_values(self, channel="red"):
        return [[self.pixel_array[y][x].values[channel]
                 for x in range(self.width)]
                 for y in range(self.height)]
```

Flyweight - example part 3

```
data = (
    ("#ff00ee", "#1345ab", "#ff00ee"),
    ("#ef001e", "#ff00ee", "#ff01ab"),
    ("#ff00ee", "#1345ab", "#ff00ee"),
    ("#ff00ee", "#1345ab", "#ff00ee"),
)

img = Image(data)
print(img)
print(img.get_values(channel="red"))
```

>> Image of size 4x3 with 4 unique pixels.
>> [[255, 19, 255], [239, 255, 255], [255, 19, 255], [255, 19, 255]]

Observer

Behavioral pattern

Observer

- It is **behavioral** software design pattern
- Other name is **Publish/subscribe**
- A subject maintains a list of its observers, and notifies them automatically of any state changes.

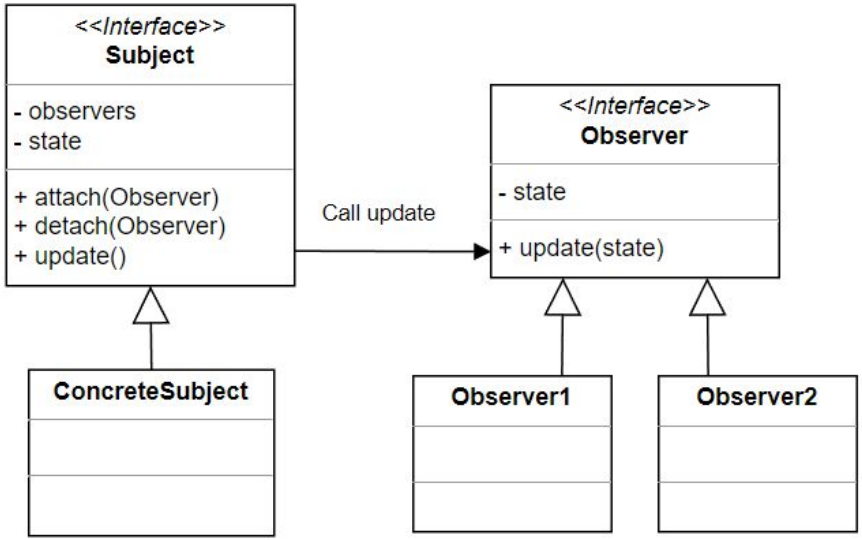
Observer - requirements

- A one-to-many dependency between objects should be defined without making the objects tightly coupled.
- It should be possible that one object can notify an open-ended number of other objects.

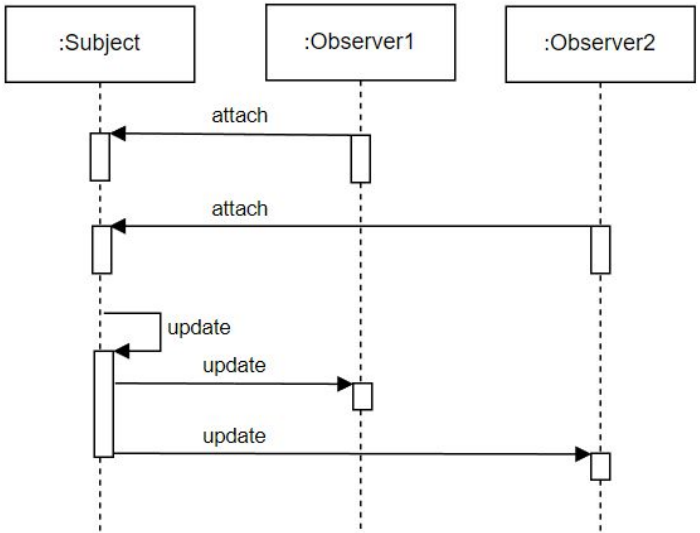
Observer - solution

- Define Subject and Observer objects.
- When a subject changes state, all registered observers are notified with update function.

Observer - class diagram



Observer - sequence diagram



Observer - example part 1

```
class SensorArray:
    def __init__(self):
        self._sensors = []

    def register_sensor(self, sensor):
        self._sensors.append(sensor)

    def set_precision(self, precision):
        for sensor in self._sensors:
            sensor.set_precision(precision)
```

Observer - example part 2

```
class Sensor:
    def __init__(self, name, sensor_array):
        self._name = name
        self._precision = "Low"
        sensor_array.register_sensor(self)

    def set_precision(self, precision):
        self._precision = precision
        print("{}: precision is set to {}".format(
            self._name, self._precision))
```

Observer - example part 3

```
sensor_array = SensorArray()
sensor1 = Sensor("S1", sensor_array)
sensor2 = Sensor("S2", sensor_array)

sensor_array.set_precision("High")
sensor_array.set_precision("Low")
```

>> S1: precision is set to High
>> S2: precision is set to High
>> S1: precision is set to Low
>> S2: precision is set to Low

Facade

Structural pattern

Facade

- It is **structural** software design pattern
- Facade is an object that serves as a front-facing interface masking more complex underlying or structural code

Facade - requirements

- To make a complex subsystem easier to use.
- It may perform additional functionality before/after forwarding a request.

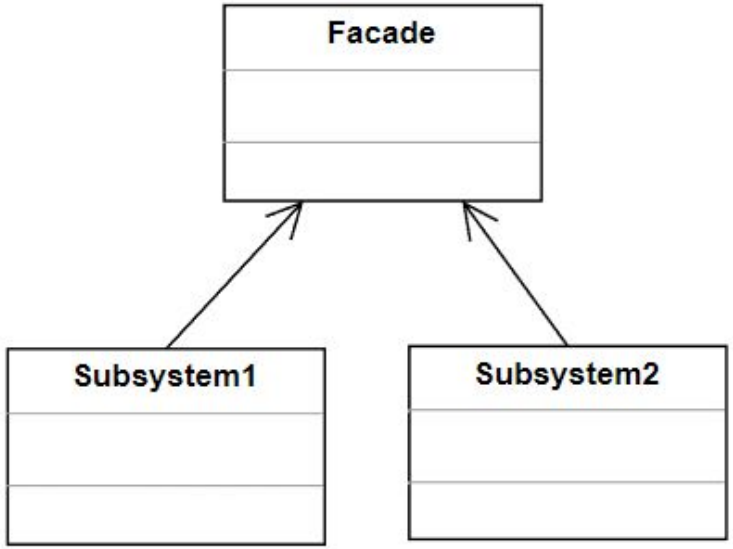
Facade - solution

- A simple interface should be provided for a set of interfaces in the subsystem.

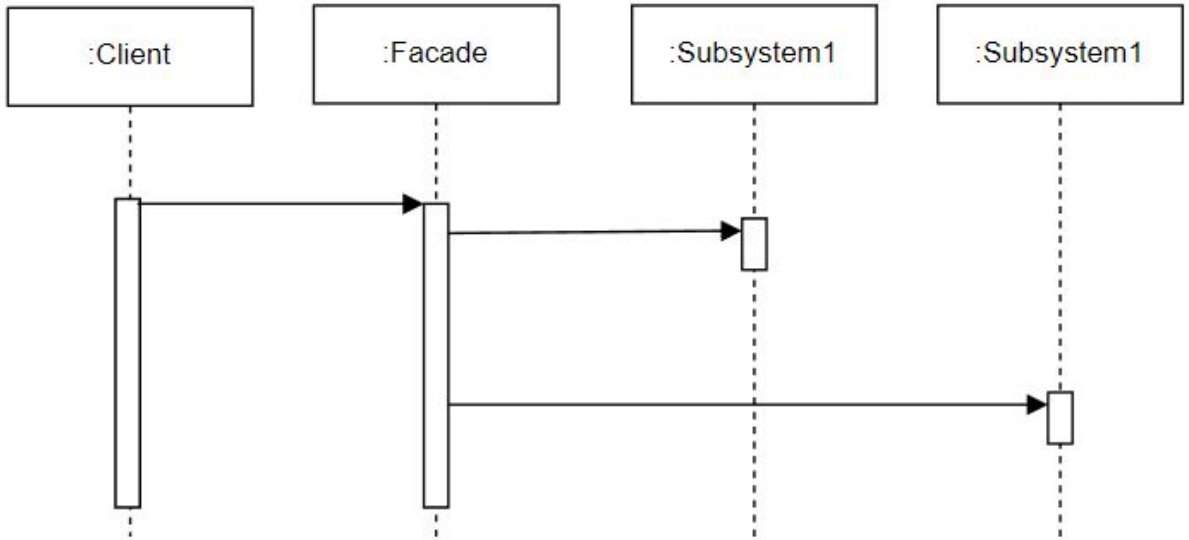
Facade - solution

- A simple interface should be provided for a set of interfaces in the subsystem.

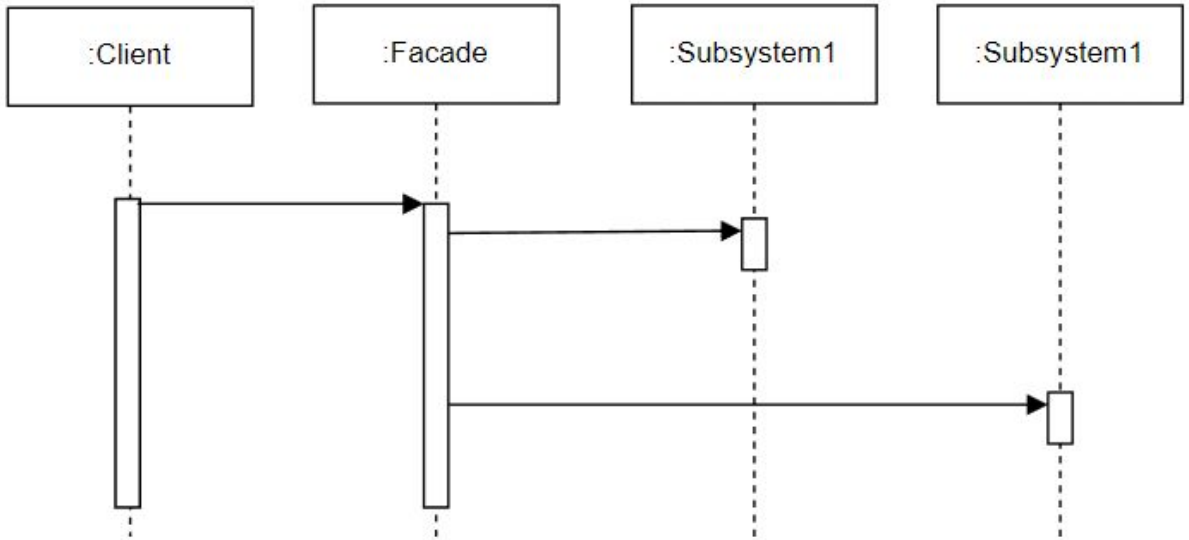
Facade - class diagram



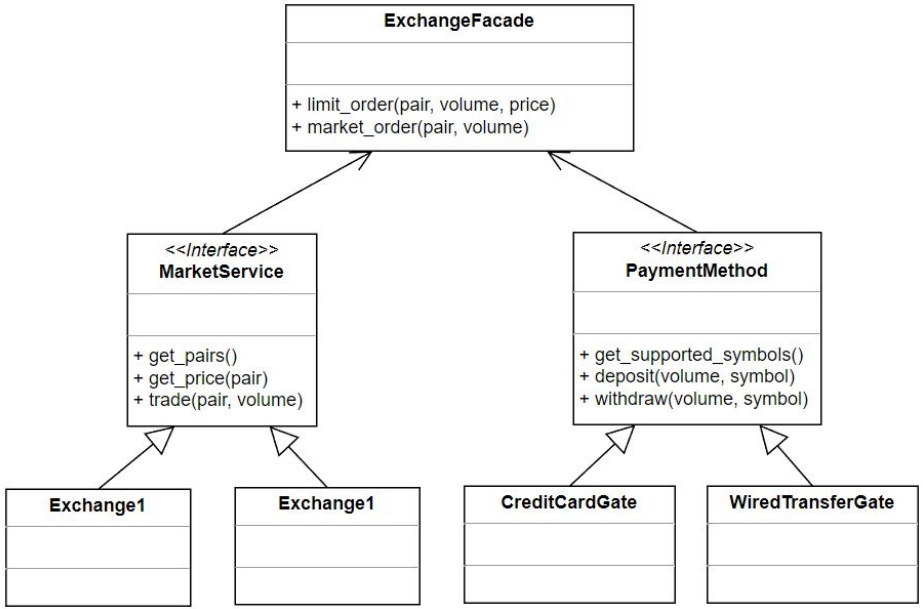
Facade - sequence diagram



Facade - sequence diagram



Facade - example - class diagram



Facade - example part 1

```
class MarketService():
    def get_pairs(self): pass # list all symbols
    def get_price(self, pair): pass
    def trade(self, pair, volume): pass

class ForexMarket(MarketService): pass
class ForexSuperExchange(MarketService): pass
class CryptoExchange(MarketService): pass
class CryptoAlternativeMarket(MarketService): pass
```

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Facade - example part 2

```
class PaymentMethod():
    def get_supported_symbols(self): pass
    def deposit(self, volume, symbol): pass
    def withdraw(self, volume, symbol): pass

class CreditCardGate(PaymentMethod): pass
class WiredTransfer(PaymentMethod): pass
class CryptoWallet(PaymentMethod): pass
```

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Facade - example part 3

```
class PaymentMethod():
    def get_supported_symbols(self): pass
    def deposit(self, volume, symbol): pass
    def withdraw(self, volume, symbol): pass

class CreditCardGate(PaymentMethod): pass
class WiredTransfer(PaymentMethod): pass
class CryptoWallet(PaymentMethod): pass
```

Facade - example part 4

```
class ExchangeFacade():

    def __init__(self):
        self._exchanges = [
            ForexMarket(),
            CryptoExchange(),
            ForexSuperExchange(),
            CryptoAlternativeMarket(),
        ]
        self._payment_gates = [
            CreditCardGate,
            WiredTransfer,
            CryptoWallet,
        ]
```

Facade - example part 5

```
def _find_payment_gate(self, symbol):
    """ Find gate that supports given symbol
    """

def _find_best_price(self, pair):
    """ Find best market service for the symbol
    """

def _make_order(self, pair, volume, exchange):
    """
    1. Call _find_payment_gate for both symbols in pair
    2. withdraw from source payment gate
    3. trade via exchange
    4. deposit to target payment gate
    """
```

Facade - example part 6

```
def limit_order(self, pair, volume, price):
    """
    1. Call _find_best_price till required price appears
    2. Call _make_order
    """

def market_order(self, pair, volume):
    """
    1. Call _find_best_price once
    2. Call _make_order
    """
```