Software design patterns
Part 2

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

Contents

- Some design patterns

Adapter

Structural pattern

Adapter

- It is **structural** software design pattern
- It allows the interface of an existing class to be used as another interface.
- It is often used to make existing classes work with others without modifying their source code.

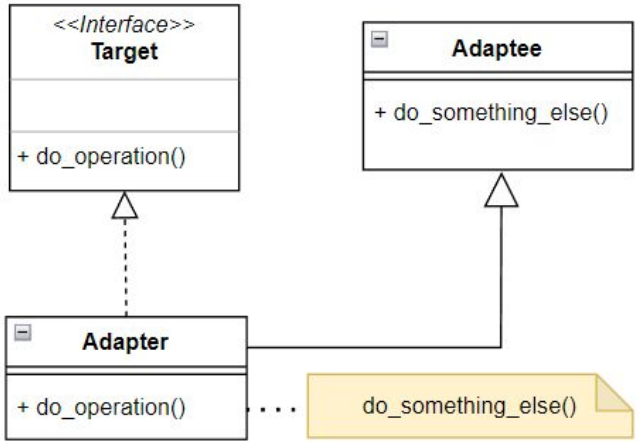
Adapter - requirements

- How can a class be reused that does not have an interface that a client requires?
- How can classes that have incompatible interfaces work together?

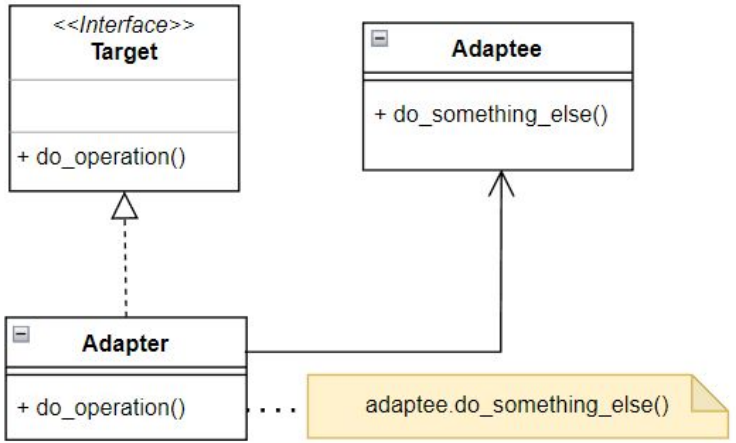
Adapter - solution

1. Create a new adapter class that converts the original interface into another interface.
2. Use the adapter to work with classes that do not have the required interface.

Adapter - class diagram



Inheritance based



Instance based

Adapter - example part 1

```
from abc import ABC, abstractmethod

class NumberInterface(ABC):

    @abstractmethod
    def do_operation(self, value: int): pass

class ComplexNumberService():

    def do_the_complex_trick(self, value):
        if isinstance(value, complex):
            return complex(value.real * 2, value.imag * 3)
        else:
            raise Exception("Value is not complex")
```

Adapter - example part 2

```
class ComplexNumberService():

    def do_the_complex_trick(self, value):
        if isinstance(value, complex):
            return complex(value.real * 2, value.imag * 3)
        else:
            raise Exception("Value is not complex")

class ServiceAdapter(NumberInterface, ComplexNumberService):

    def do_operation(self, value):
        value = complex(value)
        value = self.do_the_complex_trick(value)
        return value.real
```

Adapter - example part 3

```
a1 = ComplexNumberService()
print(a1.do_the_complex_trick(10+5j))

a2 = ServiceAdapter()
print(a2.do_operation(10))
```

Output:

```
(20+15j)
20.0
```

Decorator

Structural pattern

Decorator

- It is **structural** software design pattern
- Decorator allows to add behavior to an individual object, dynamically, without affecting the behavior of other objects from the same class.

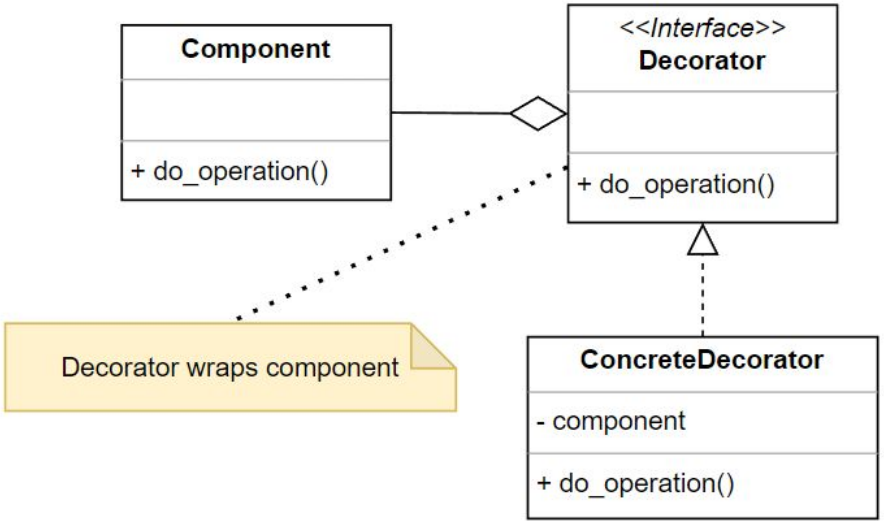
Decorator - requirements

- Features should be added to an object dynamically at run-time.
- A flexible alternative to subclassing for extending functionality should be provided.

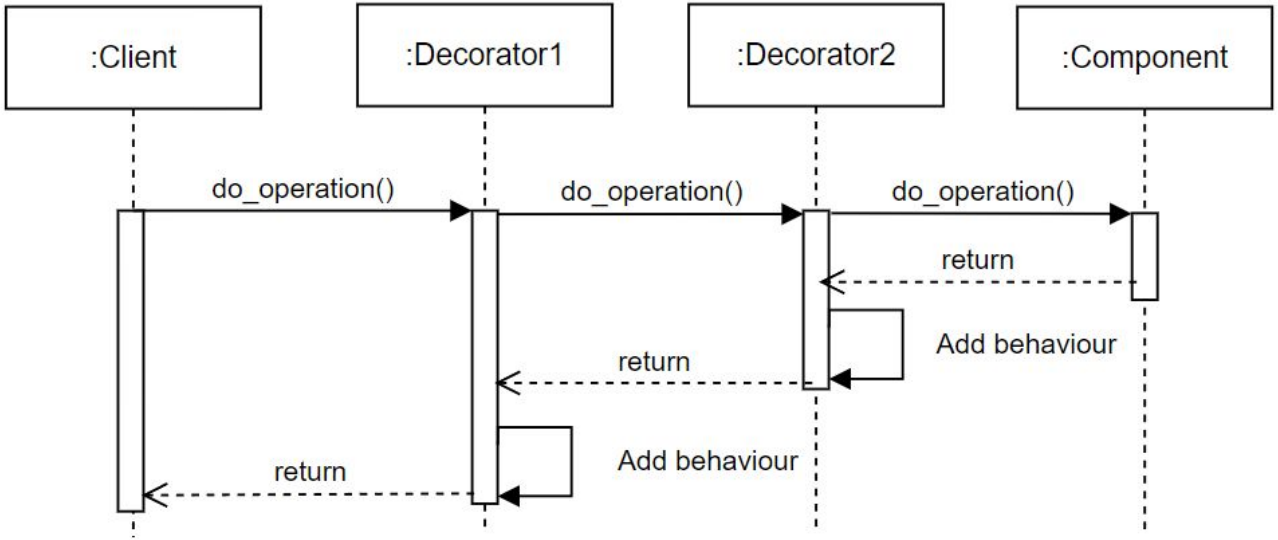
Decorator - solution

1. Subclass the original Component as a Decorator
2. In the Decorator, add a Component pointer as a field
3. Attach the Component to Decorator during init.
4. In Decorator, Forward all Component methods
5. In Decorator, override any Component methods whose behavior needs to be modified.

Decorator - class diagram



Decorator - sequence diagram



Decorator - example part 1

```
class TitledName():

    def __init__(self, name_str):
        self._name_str = name_str

    def get(self):
        return self._name_str
```

Decorator - example part 2

```
class NameDecorator(ABC):

    def __init__(self, name):
        self._name = name

    @abstractmethod
    def get(self):
        return self._name.get()
```

Decorator - example part 3

```
class MscDecorator(NameDecorator):

    def get(self):
        return "Msc. {}".format(self._name.get())

class PhdDecorator(NameDecorator):

    def get(self):
        return "{}, Ph.D.".format(self._name.get())
```

Decorator - example part 4

```
name1 = TitledName("Jane Doe")
name1 = MscDecorator(name1)
name1 = PhdDecorator(name1)

name2 = TitledName("John Doe")
name2 = MscDecorator(name2)

print(name1.get())
print(name2.get())
```

>>> Msc. Jane Doe, Ph.D

>>> Msc. John Doe

Lazy initialization

Creational pattern

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Lazy initialization

- It is **creational** software design pattern
- Lazy initialization is the tactic of delaying the creation of an object.
- Object (or other expensive process) is delayed till the first time it is needed.

Lazy initialization - requirements

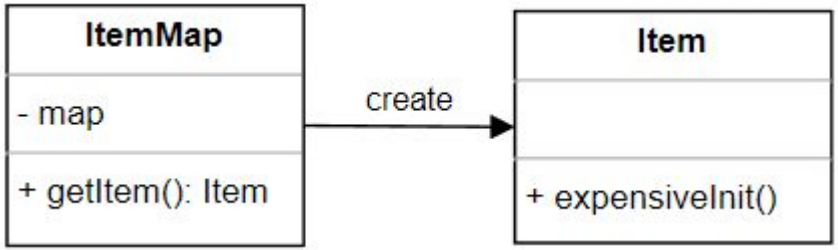
- Avoid multiple expensive object creation at the same time.
- Spread the expensive object creations in time.

Lazy initialization - solution

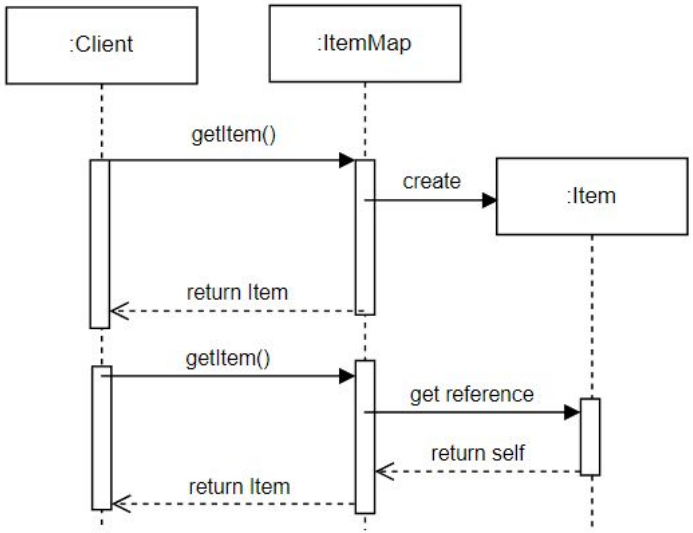
1. Create map object that works as a map of lazy objects.
2. Access lazy objects via map object.

Map object creates lazy objects if they do not exist.

Lazy initialization - class diagram



Lazy initialization - sequence diagram



Lazy initialization - example part 1

```
class NumberFactorial:

    def __init__(self, key):
        self.key = key
        self.value = self.factorial(key)

    def __str__(self):
        return "Factorial of {} is {}".format(self.key, self.value)

    def factorial(self, n):
        return 1 if (n==1 or n==0) else n * self.factorial(n - 1)
```

Lazy initialization - example part 2

```
class Factorials:
    def __init__(self) -> None:
        self.numbers = {}

    def get(self, key):
        if key not in self.numbers:
            print("Calculating factorial for value: {}".format(key))
            self.numbers[key] = NumberFactorial(key)
        return self.numbers[key]
```

Lazy initialization - example part 3

```
factorials = Factorials()
print(factorials.get(5))
print(factorials.get(6))
print(factorials.get(5))
```

>> Calculating factorial for value: 5
>> Factorial of 5 is 120
>> Calculating factorial for value: 6
>> Factorial of 6 is 720
>> Factorial of 5 is 120

Prototype

Creational pattern

Prototype

- It is **creational** software design pattern.
- New object is created by cloning of existing object.
- It is used to avoid subclasses.

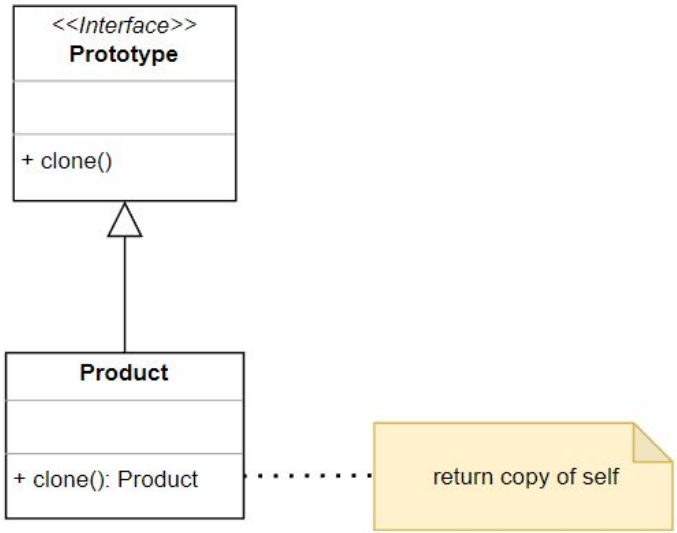
Prototype - requirements

- How can objects be created so that which objects to create can be specified at run-time?
- How can dynamically loaded classes be instantiated?

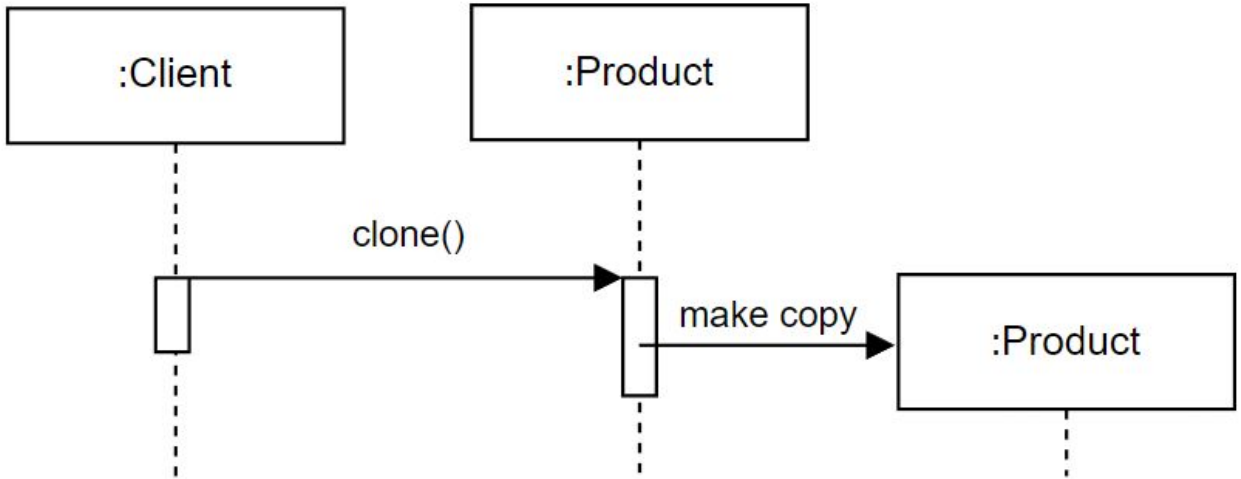
Prototype - solution

1. Define a Prototype object that returns a copy of itself.
2. Create new objects by copying a Prototype object.

Prototype - class diagram



Prototype - sequence diagram



Prototype - example part 1

```
import copy

class Prototype:

    def clone(self):
        return copy.deepcopy(self)
```

Prototype - example part 2

```
class Individual(Prototype):

    def __init__(self):
        self.genome = "000"

    def evolve(self, new_gene):
        self.genome += new_gene

    def __str__(self):
        return "My genome is: {}".format(self.genome)
```

Prototype - example part 3

```
genome1 = Individual()
genome1.evolve("101")
genome2 = genome1.clone()
genome2.evolve("111")
genome1.evolve("000")
print(genome1)        >> My genome is: 000101000
print(genome2)        >> My genome is: 000101111
```

Bridge

Structural pattern

Bridge

- It is **structural** software design pattern
- Bridge is used to decouple an abstraction from its implementation.

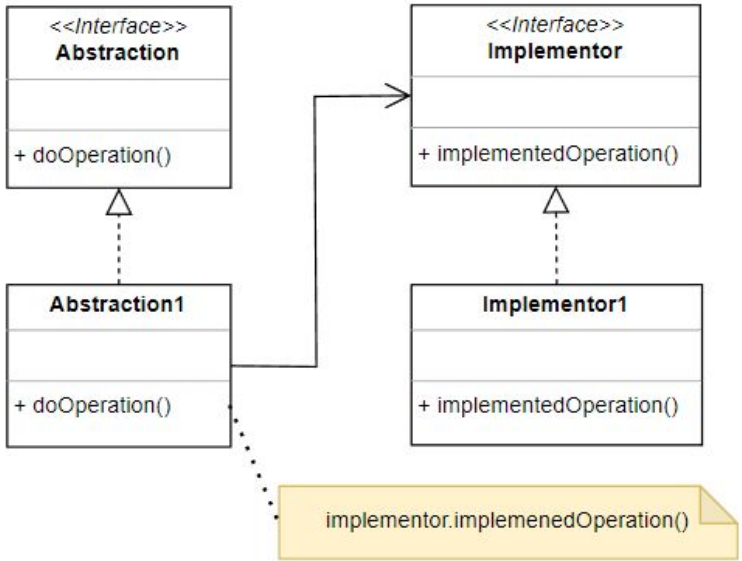
Bridge - requirements

- An abstraction and its implementation should be defined and extended independently from each other.
- A compile-time binding between an abstraction and its implementation should be avoided so that an implementation can be selected at run-time.

Bridge - solution

- Separate an abstraction from its implementation by putting them in separate class hierarchies.
- Delegate abstraction objects methods to implementation objects.

Bridge - class diagram



Bridge - example part 1 (implementor)

```
class Reporter(metaclass=ABCMeta):

    @abstractmethod
    def report_good_news(self):
        raise NotImplementedError()

    @abstractmethod
    def report_bad_news(self):
        raise NotImplementedError()
```

Bridge - example part 2

```
class FairReporter(Reporter):

    def report_good_news(self, news):
        print(news)

    def report_bad_news(self, news):
        print(news)
```

Bridge - example part 3

```
class PositiveBiasReporter(Reporter):

    def report_good_news(self, news):
        print(news)

    def report_bad_news(self, news):
        print("Nothing to report.")
```

Bridge - example part 4 (abstraction)

```
class News(metaclass=ABCMeta):

    def __init__(self, reporter, news):
        self._reporter = reporter
        self.news = news

    @abstractmethod
    def report(self):
        raise NotImplementedError()
```

Bridge - example part 5

```
class BadNews(News):
    def report(self):
        return self._reporter.report_bad_news(self.news)

class GoodNews(News):
    def report(self):
        return self._reporter.report_good_news(self.news)
```

Bridge - example part 6

```
reporter1 = FairReporter()
reporter2 = PositiveBiasReporter()
BadNews(reporter1, "Out of money").report()
BadNews(reporter2, "Out of money").report()
```

>> Out of money
>> Nothing to report.