Software design patterns
Part 1

Course: Object Oriented Programming (OOP)

CTU, FS, U12110

Matouš Cejnek

Contents

- Design patterns introduction
- Some design patterns

What we already know?

Polymorphism is the provision of a single interface to entities of different types.

Inheritance is the mechanism of basing an object or class upon another object or class.

Design patterns

Design patterns

- Software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.
- In other words, It is a general description, how to problem can be solved in a nice way.
- Design patterns can speed up the development process by providing tested, proven development paradigms

Design patterns

In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming

Design patterns gained popularity in computer science after the book ***Design Patterns: Elements of Reusable Object-Oriented Software*** was published in 1994

Design patterns classification

Design patterns are commonly classified according of problem they solve:

- Creational patterns
- Structural patterns
- Behavioral patterns
- *Concurrency patterns*

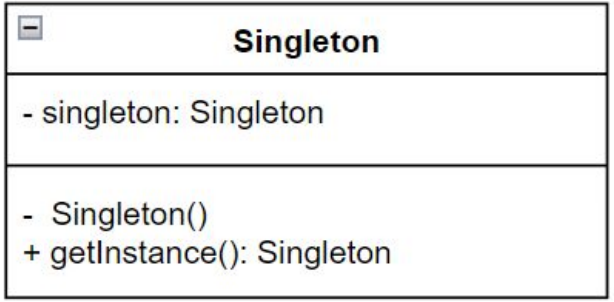
Singleton

Creational pattern

Singleton - requirements

- Ensure that a class only has one instance
- Easily access the sole instance of a class
- Control its instantiation
- Restrict the number of instances
- Access a global variable

Singleton - solution



Singleton

- singleton: Singleton

- Singleton()
+ getInstance(): Singleton

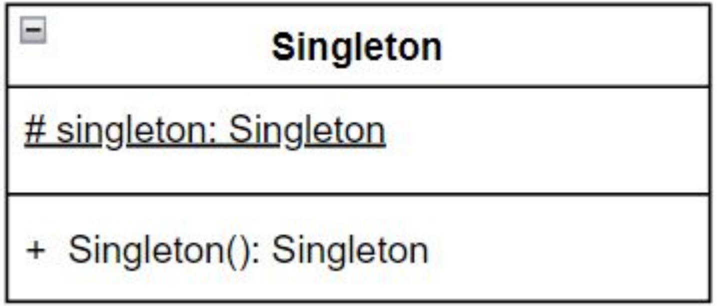
1. Hide the constructors of the class
2. Define a public static method that returns a reference to the sole instance.

Singleton - example implementation

Python code example:

```
class Singleton:
    __instance = None

    def __new__(cls, *args):
        if cls.__instance is None:
            cls.__instance = object.__new__(cls, *args)
        return cls.__instance
```



Singleton

singleton: Singleton

+ Singleton(): Singleton

Proxy

Structural pattern

Proxy

- It is **structural** software design pattern
- Proxy is a class functioning as an interface to something else (connection, a large object in memory, a file, …)
- It is kind of a wrapper for an object.

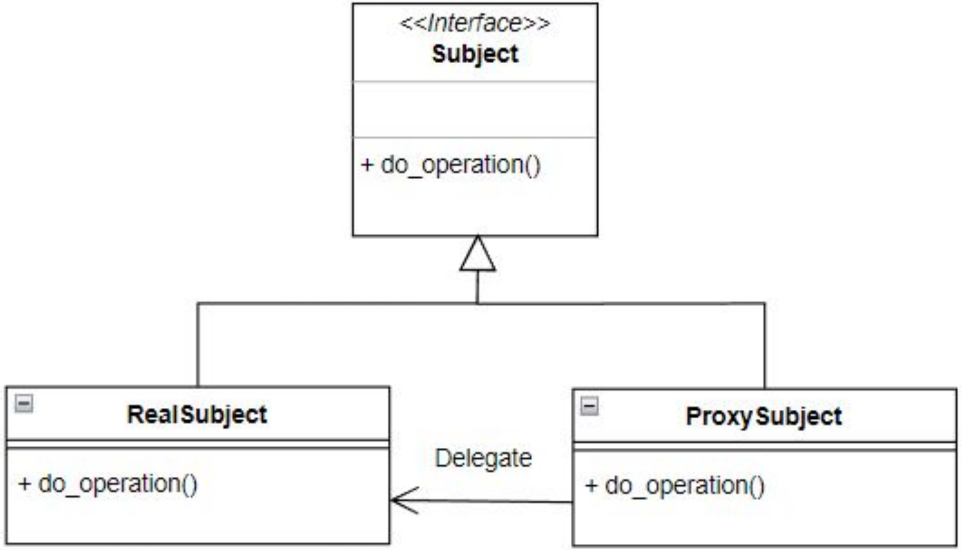
Proxy - requirements

- The access to an object should be controlled.
- Additional functionality should be provided when accessing an object.

Proxy - solution

1. Create a substitute (proxy) subject implementing the same interface as subject.
2. Add required logic to extend the original subject.

Proxy



<<Interface>>
Subject
+ do_operation()
RealSubject
+ do_operation()
Delegate
ProxySubject
+ do_operation()

Proxy - example part 1

```
from abc import ABC, abstractmethod

class AbstractConnection(ABC):
    @abstractmethod
    def connect(self): pass


class Connection(AbstractConnection):
    def connect(self):
        print("Connecting ...")
```

Proxy - example part 2

```
class ProxyConnection(AbstractConnection):
    def __init__(self):
        self.connection = Connection()

    def connect(self):
        print("Checking connection ...")
        self.connection.connect()

connection = ProxyConnection()
connection.connect()
```

Factory method

Creational pattern

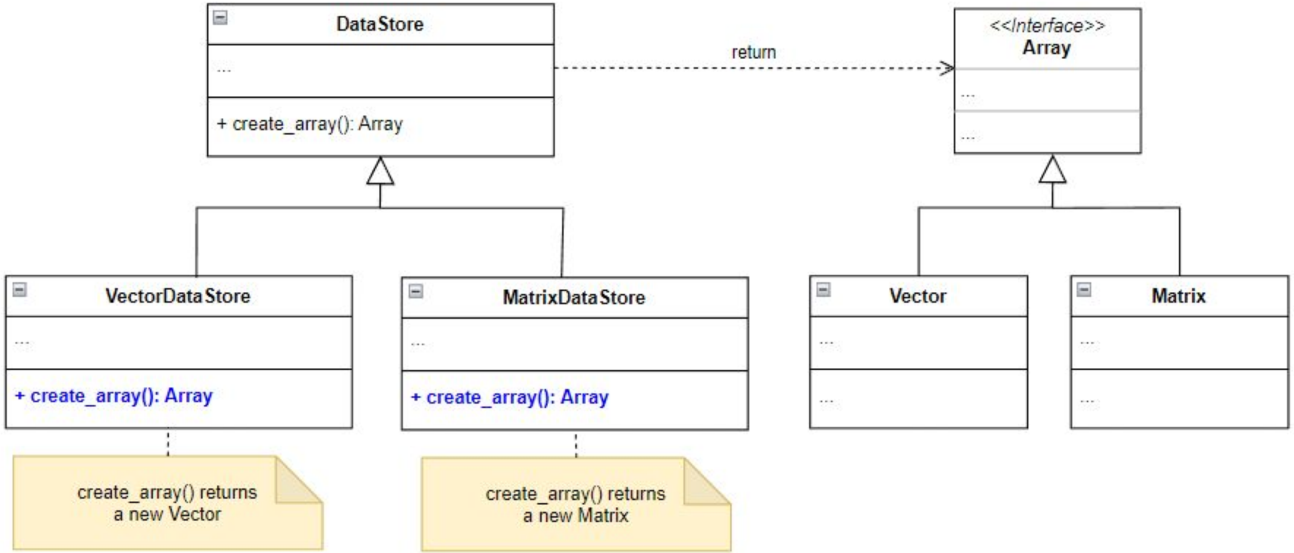
Factory method

- It is **creational** software design pattern.
- Factory method is creating objects without having to specify the exact class of the object that will be created.

Factory method - solution

- Define a separate operation (factory method) for creating an object.
- Create an object by calling a factory method.

Factory method - example



Factory method - example

Example how to use multiple storages of different type with factory method:

```
storages = [MatrixDataStore(), VectorDataStore()]:
for storage in storages:
    array = storage.create_array()
    # do something nice with array
```

Factory method summary

- It unifies the way how we obtain a new object from different classes
- Factory method heavily utilises polymorphism and inheritance

Abstract factory

Creational pattern

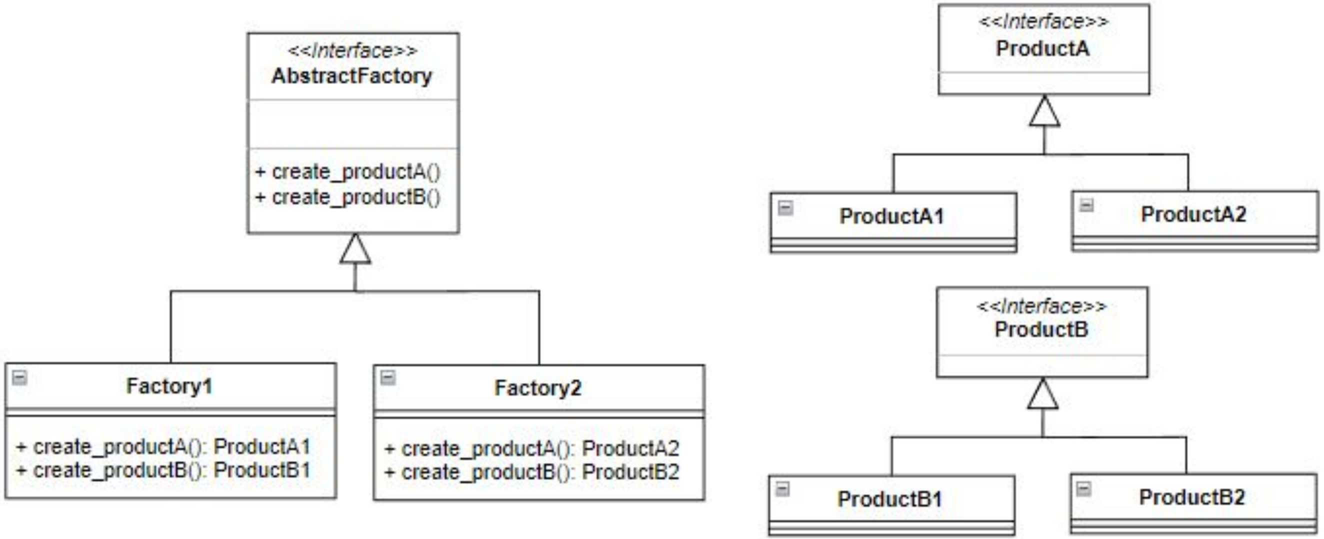
Abstract factory

- It is **creational** software design pattern
- The abstract factory is an encapsulation of a group of individual factories without specifying their concrete classes.
- A factory is an abstraction of a constructor of a class.

Abstract factory - solution

- We provide abstract class or interface, that defines how the individual factories should work

Abstract factory - example



Abstract factory - example

Example usage in Python:

```
if conditionA:
    factory = FactoryA()
elif conditionB:
    factory = FactoryB()

productA = factory.create_productA()
productB = factory.create_productB()
```

Abstract factory vs Factory method

- Factory method is a method
- Abstract factory is a class (meta class or interface)

Iterator

Behavioral pattern

Iterator

- It is **behavioral** software design pattern
- Iterator is used to traverse a container and access the container's elements.

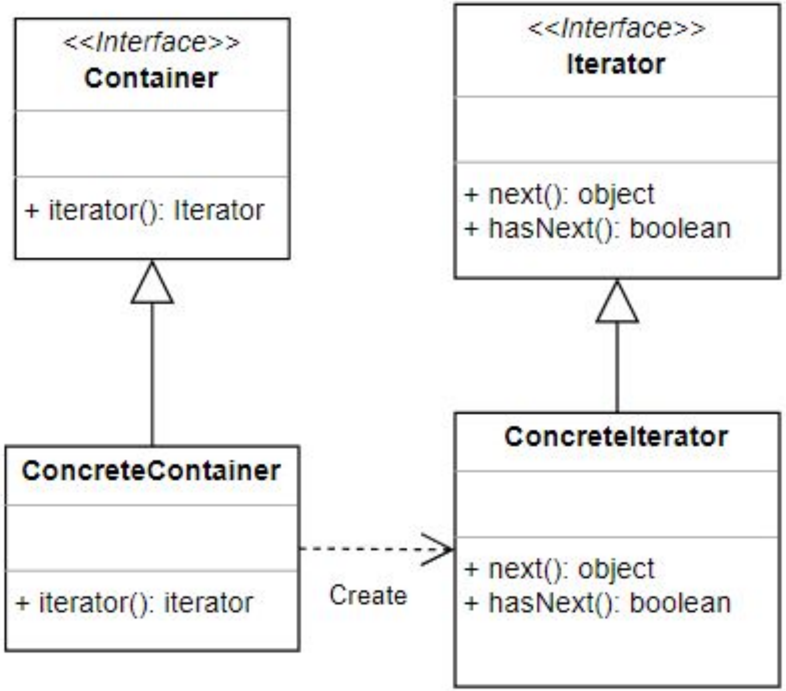
Iterator - requirements

- The elements of an aggregate object should be accessed and traversed without exposing its data structure.
- New traversal operations should be defined for an container object without changing its interface.

Iterator - solution

1. Define a separate (iterator) object that encapsulates accessing and traversing an container object.
2. Clients use an iterator to access and traverse an aggregate without knowing its data structure.

Iterator



<<Interface>>
Container

+ iterator(): Iterator

<<Interface>>
Iterator

+ next(): object
+ hasNext(): boolean

ConcreteContainer

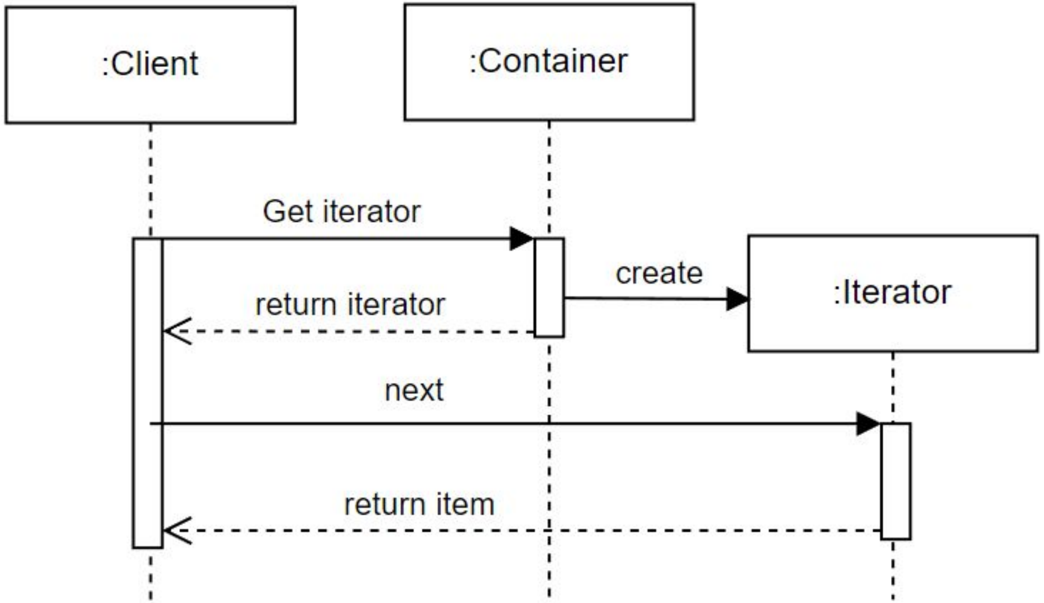
+ iterator(): iterator

Create

ConcreteIterator

+ next(): object
+ hasNext(): boolean

Iterator


:Client
:Container
:Iterator
Get iterator
create
return iterator
next
return item

Iterator - example (Python iterator)

Python default iterator in list:

```
container = [1, 2, 3]
iterator = container.__iter__()
print(iterator.__next__())          >>> 1
print(iterator.__next__())          >>> 2
print(iterator.__next__())          >>> 3
print(iterator.__next__())          >>> … StopIteration
```

Iterator - example part 1 (custom iterator)

```
class Iterator:
    def __init__(self, container):
        self.container = container
        self.idx = 0

    def __next__(self):
        if self.idx == len(self.container.content):
            raise StopIteration
        else:
            self.idx += 1
            return self.container.content[self.idx - 1]
```

Iterator - example part 2 (custom iterator)

```
class Container:
    def __init__(self):
        self.content = ["a", "b", "c"]

    def __iter__(self):
        return Iterator(self)

container = Container()
for item in container:
    print(item)
```

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

Memento

Behavioral pattern

Memento

- It is **behavioral** software design pattern
- Memento exposes the private internal state of an object.

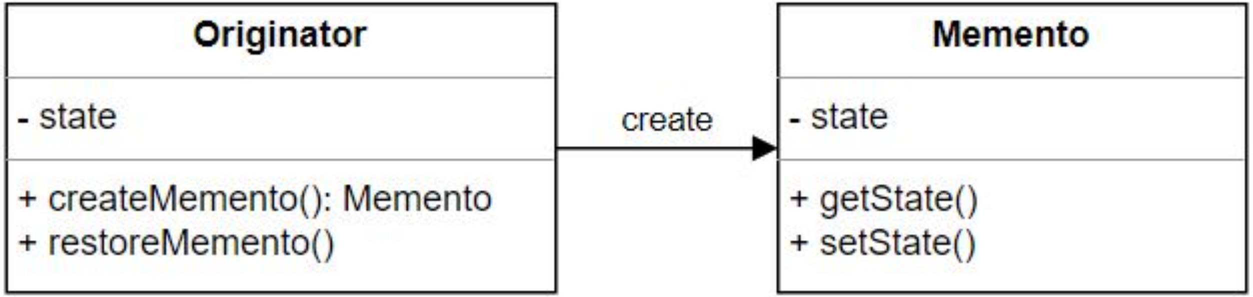
Memento - requirements

- The internal state of an object should be saved externally
- The object can be restored to this state later.
- The object's encapsulation must not be violated.

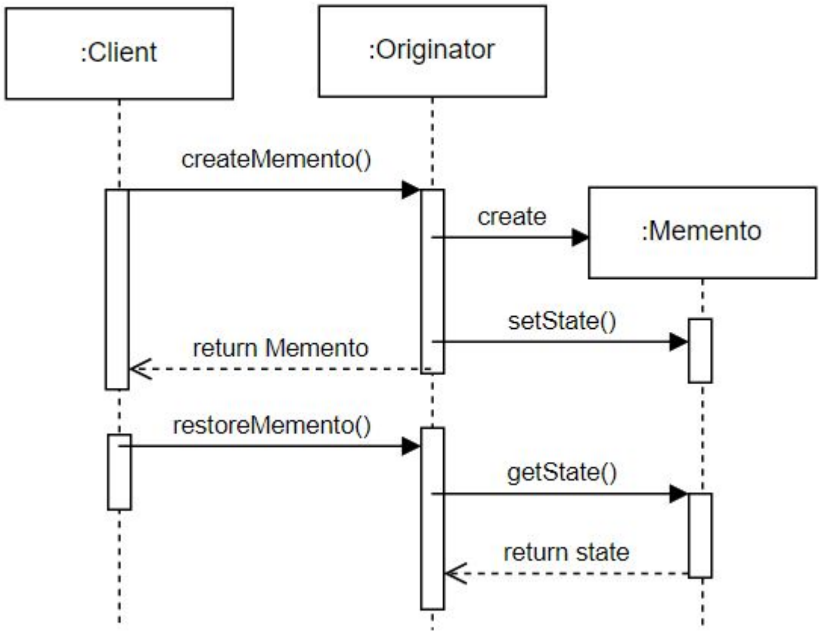
Memento - solution

- Object (originator) can save its internal state to a (memento) object and restore to a previous state from a (memento) object.
- Only the originator that created a memento is allowed to access it.

Memento - class diagram



Memento - sequence diagram



Memento - example part 1

```
class SavedLocation:

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

    def get_location(self):
        return self._location
```

Memento - example part 2

```python
class Maze:
    def set(self, state):
        self._location = state

    def display(self):
        print(self._location)

    def save_location(self):
        return SavedLocation(self._location)

    def load_location(self, location):
        self._location = location.get_location()
```

Memento - example part 3

```
maze = Maze()
maze.set("Room1")
maze.set("Room2")
maze.display()                          >> Room 2
saved_location = maze.save_location()
maze.set("Room3")
maze.display()                          >> Room 3
maze.load_location(saved_location)
maze.display()                          >> Room 2
```