Sakupljeni primjeri oblikovnih obrazaca u Pythonu

Izvor: internet
Metnuo na kup: **IcyTexx**

STRATEGY

```
class StrategyExample:
    def __init__(self, func=None):
        if func:
            self.execute = func

    def execute(self):
        print "Original execution"



def executeReplacement1(self):
    print "Strategy 1"

def executeReplacement2(self):
    print "Strategy 2"

if __name__ == “__main__":
    strat0 = StrategyExample()
    strat1 = StrategyExample(executeReplacement1)
    strat2 = StrategyExample(executeReplacement2)

    strat0.execute()
    strat1.execute()
    strat2.execute()



# -------------------- With classes --------------------


class AUsefulThing(object):
    def __init__(self, aStrategicAlternative):
        self.howToDoX = aStrategicAlternative

    def doX(self, someArg):
        self. howToDoX.theAPImethod(someArg, self)


class StrategicAlternative(object):
    pass


class AlternativeOne(StrategicAlternative):
    def theAPIMethod(self, someArg, theUsefulThing):
        pass  # an implementation


class AlternativeTwo(StrategicAlternative):
    def theAPImethod(self, someArg, theUsefulThing):
        pass  # another implementation



t = AUsefulThing(AlternativeOne())
t.doX('arg')
```

TEMPLATE

```
class AbstractGame:
     """An abstract class that is common to several games in which
     players play against the others, but only one is playing at a
     given time."""

     def __init__(self, *args, **kwargs):
         if self.__class__ is AbstractGame:
             raise TypeError('abstract class cannot be instantiated')

     def playOneGame(self, playersCount):
         self.playersCount = playersCount
         self.initializeGame()
         j = 0
         while not self.endOfGame():
             self.makePlay(j)
             j = (j + 1) % self.playersCount
         self.printWinner()

     def initializeGame(self):
         raise TypeError('abstract method must be overridden')

     def endOfGame(self):
         raise TypeError('abstract method must be overridden')

     def makePlay(self, player_num):
         raise TypeError('abstract method must be overridden')

     def printWinner(self):
         raise TypeError('abstract method must be overridden')

# Now to create concrete (non-abstract) games, you subclass AbstractGame
# and override the abstract methods.

class Chess(AbstractGame):
     def initializeGame(self):
         # Put the pieces on the board.
         pass

     def makePlay(player):
         # Process a turn for the player
         pass



# --------- second example ---------


class AbstractBase(object):
    def orgMethod(self):
        self.doThis()
        self.doThat()


class Concrete(AbstractBase):
    def doThis(self):
        pass

    def doThat(self):
        pass
```

OBSERVER

```
class AbstractSubject:
    def register(self, listener):
        raise NotImplementedError("Must subclass me")

    def unregister(self, listener):
        raise NotImplementedError("Must subclass me")

    def notify_listeners(self, event):
        raise NotImplementedError("Must subclass me")


class Listener:
    def __init__(self, name, subject):
        self.name = name
        subject.register(self)

    def notify(self, event):
        print self.name, "received event", event


class Subject(AbstractSubject):
    def __init__(self):
        self.listeners = []
        self.data = None

    def getUserAction(self):
        self.data = raw_input('Enter something to do:')
        return self.data

    # Implement abstract Class AbstractSubject

    def register(self, listener):
        self.listeners.append(listener)

    def unregister(self, listener):
        self.listeners.remove(listener)

    def notify_listeners(self, event):
        for listener in self.listeners:
            listener.notify(event)



if __name__ == "__main__":
    # Make a subject object to spy on
    subject = Subject()

    # Register two listeners to monitor it.
    listenerA = Listener("<listener A>", subject)
    listenerB = Listener("<listener B>", subject)

    # Simulated event
    subject.notify_listeners("<event 1>”)

    # Outputs:
    #     <listener A> received event <event 1>
    #     <listener B> received event <event 1>

    action = subject.getUserAction()
    subject.notify_listeners(action)

    # Enter something to do:hello
    # outputs:
    #     <listener A> received event hello
    #     <listener B> received event hello
```

DECORATOR

```
import time


def time_this(func):
    """The time_this decorator"""

    def decorated(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print 'Rain in', time.time() - start, 'seconds'
        return result
    return decorated


# Decorator syntax

@time_this
def count(until):
    """Counts to 'until', then returns the result"""

    print "Counting to", until, "…"
    num = 0
    for i in xrange(to_num(until)):
        num += 1
    return num

def to_num(numstr):
    """Turns a comma-separated number string to an int"""
    return int(numstr.replace(",", ""))



# Run count with various values

for number in ("10,000", "100,000", "1,000,000"):

    print count(number)

    print "-" * 20
```

COMMAND

pt. 1

```
from sys import stdout as console

# Handling 'exit' command
class SessionClosed(Exception):
    def __init__(self, value):
        self.value = value


# Interface
class Command:
    def execute(self):
        raise NotImplementedError()

    def cancel(self):
        raise NotImplementedError()

    def name():
        raise NotImplementedError()


# rm command
class RmCommand(Command):
    def execute(self):
        console.write("You are executed \"rm\" command\n")

    def cancel(self):
        console.write("You are canceled \"rm\" command\n")

    def name(self):
        return "rm"


# uptime command
class UptimeCommand(Command):
    def execute(self):
        console.write("You are executed \"uptime\" command\n”)

    def cancel(self):
        console.write("You are canceled \"uptime\" command\n")

    def name(self):
        return "uptime"


# undo command
class UndoCommand(Command):
    def execute(self):
        try:
            cmd = HISTORY.pop()
            TRASH.append(cmd)
            console.write("Undo command \"{0}\"\n".format(cmd.name()))
            cmd.cancel()

        except IndexError:
            console.write("ERROR: HISTORY is empty\n")

    def name(self):
        return "undo"


# redo command
class RedoCommand(Command):
    def execute(self):
        try:
            cmd = TRASH.pop()
            HISTORY.append(cmd)
            console.write("Redo command \"{0}\"\n".format(cmd.name()))
            cmd.execute()
        except IndexError:
            console.write("ERROR: TRASH is empty\n")


    def name(self):
        return "redo"
```

COMMAND

pt.2

```
# history command
class HistoryCommand(Command):
    def execute(self):
        i = 0
        for cmd in HISTORY:
            console.write("{0}: {1}\n".format(i, cmd.name()))
            i = i + 1

    def name(self):
        print "history"


# exit command
class ExitCommand(Command):
    def execute(self):
        raise SessionClosed("Good bay!")

    def name(self):
        return "exit"



# available commands
COMMANDS = {'rm': RmCommand(), 'uptime': UptimeCommand(), 'undo':
            UndoCommand(), 'redo': RedoCommand(), 'history': HistoryCommand(),
            'exit': ExitCommand()}

HISTORY = list()
TRASH = list()



# Shell
def main():
    try:
        while True:
            console.flush()
            console.write("pysh >> ")

            cmd = raw_input()

            try:
                command = COMMANDS[cmd]
                command.execute()
                if (not isinstance(command, UndoCommand) and not
                    isinstance(command, RedoCommand) and not
                    isinstance(command, HistoryCommand)):
                    TRASH = list()
                    HISTORY.append(command)

            except KeyError:
                console.write("ERROR: Command \"%s\" not found\n" % cmd)

    except SessionClosed as e:
        console.write(e.value)


if __name__ == "__main__":

    main()
```

SINGLETON

```
# ------- Real Singleton instance -------

class Singleton(object):
    def __new__(type):
        if not '_the_instance' in type.__dict__:
            type._the_instance = object.__new__(type)
        return type._the_instance

a = Singleton()
a.toto = 12

b = Singleton()
print b.toto

print id(a), id(b)  # The same !!




# ---------- Borg's singletone ----------


class Borg:
    __shared_state = {}

    def __init__(self):
        self.__dict__ = self.__shared_state


a = Borg()
a.toto = 12

b = Borg()
print b.toto

print id(a), id(b)  # different ! but states are sames




# ---------- more examples ----------


class Singleton(object):
    def __new__(cls, *a, **k):
        if not hasattr(cls, '_inst'):
            cls._inst = super(Singleton, cls).__new__(cls, *a, **k)
        return cls._inst


class Borg(object):
    """Subclassing is no problem."""

    _shared_state = {}

    def __new__(cls, *a, **k):
        obj = super(Borg, cls).__new__(cls, *a, **k)
        obj.__dict__ = cls._shared_state
        return obj
```

ITERATOR

```
# from a sequence
x = [42, "test", -12.34]
it = iter(x)
try:
    while True:
        x = next(it)  # in Python 2, you would use it.next()
        print x
except StopIteration:
    pass



# a generator
def foo(n):
    for i in range(n):
        yield i

it = foo(5)
try:
    while True:
        x = next(it)  # in Python 2, you would use it.next()
        print x
except StopIteration:
    pass
```

ADAPTER

```
class Adaptee:
    def specific_request(self):
        return 'Adaptee'


class Adapter:
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def request(self):
        return self.adaptee.specific_request()


client = Adapter(Adaptee())
print client.request()



# ---------- Second example -------------



class UppercasingFile:
    def __init__(self, *a, **k):
        self.f = file(*a, **k)

    def write(self, data):
        self.f.write(data.upper())

    def __getattr__(self, name):

        return getattr(self.f, name)
```

COMPOSITE

```
class Component(object):
    def __init__(self, *args, **kw):
        pass

    def component_function(self):
        pass


class Leaf(Component):
    def __init__(self, *args, **kw):
        Component.__init__(self, *args, **kw)

    def component_function(self):
        print "some function"


class Composite(Component):
    def __init__(self, *args, **kw):
        Component.__init__(self, *args, **kw)
        self.children = []

    def append_child(self, child):
        self.children.append(child)

    def remove_child(self, child):
        self.children.remove(child)

    def component_function(self):
        map(lambda x: x.component_function(), self.children)



c = Composite()
l = Leaf()
l_two = Leaf()
c.append_child(l)
c.append_child(l_two)
c.component_function()
```

STATE

```
"""Implementation of the state pattern"""

import itertools


class State(object):
    """Base state. This is to share functionality"""

    def scan(self):
        """Scan the dial to the next station"""
        print "Scanning... Station is", self.stations.next(), self.name


class AmState(State):
    def __init__(self, radio):
        self.radio = radio
        self.stations = itertools.cycle(["1250", "1380", "1510"])
        self.name = "AM"

    def toggle_amfm(self):
        print "Switching to FM"
        self.radio.state = self.radio.fmstate


class FmState(State):
    def __init__(self, radio):
        self.radio = radio
        self.stations = itertools.cycle(["81.3", "89.1", "103.9"])
        self.name = "FM"

    def toggle_amfm(self):
        print "Switching to AM"
        self.radio.state = self.radio.amstate


class Radio(object):
    """A radio.
    It has a scan button, and an AM/FM toggle switch."""

    def __init__(self):
        """We have an AM state and an FM state"""

        self.amstate = AmState(self)
        self.fmstate = FmState(self)
        self.state = self.amstate

    def toggle_amfm(self):
        self.state.toggle_amfm()

    def scan(self):
        self.state.scan()


def main():
    ''' Test our radio out '''
    radio = Radio()
    actions = ([radio.scan] * 2 + [radio.toggle_amfm] + [radio.scan] * 2) * 2
    for action in actions:
        action()
```

PROXY

```
class IMath:
    """Interface for proxy and real subject."""

    def add(self, x, y):
        raise NotImplementedError()

    def sub(self, x, y):
        raise NotImplementedError()

    def mul(self, x, y):
        raise NotImplementedError()

    def div(self, x, y):
        raise NotImplementedError()


class Math(IMath):
    """Real subject."""

    def add(self, x, y):
        return x + y

    def sub(self, x, y):
        return x - y

    def mul(self, x, y):
        return x * y

    def div(self, x, y):
        return x / y


class Proxy(IMath):
    """Proxy."""

    def __init__(self):
        self.math = Math()

    def add(self, x, y):
        return self.math.add(x, y)

    def sub(self, x, y):
        return self.math.sub(x, y)

    def mul(self, x, y):
        return self.math.mul(x, y)

    def div(self, x, y):
        if y == 0:
            return float('inf') # Вернуть positive infinity
        return self.math.div(x, y)


p = Proxy()

x, y = 4, 2
print '4 + 2 = ' + str(p.add(x, y))
print '4 - 2 = ' + str(p.sub(x, y))
print '4 * 2 = ' + str(p.mul(x, y))
print '4 / 2 = ' + str(p.div(x, y))
```

BRIDGE

```
# Implementor
class DrawingAPI:
    def drawCircle(x, y, radius):
        pass


# ConcreteImplementor 1/2
class DrawingAPI1(DrawingAPI):
    def drawCircle(self, x, y, radius):
            print "API1.circle at %f:%f radius %f" % (x, y, radius)


# ConcreteImplementor 2/2
class DrawingAPI2(DrawingAPI):
    def drawCircle(self, x, y, radius):
            print "API2.circle at %f:%f radius %f" % (x, y, radius)


# Abstraction
class Shape:
    # Low-level
    def draw(self):
        pass

    # High-level
    def resizeByPercentage(self, pct):
        pass


# Refined Abstraction
class CircleShape(Shape):
    def __init__(self, x, y, radius, drawingAPI):
        self.__x = x
        self.__y = y
        self.__radius = radius
        self.__drawingAPI = drawingAPI

    # low-level i.e. Implementation specific
    def draw(self):
        self.__drawingAPI.drawCircle(self.__x, self.__y, self.__radius)

    # high-level i.e. Abstraction specific
    def resizeByPercentage(self, pct):
        self.__radius *= pct



def main():
    shapes = [
        CircleShape(1, 2, 3, DrawingAPI1()),
        CircleShape(5, 7, 11, DrawingAPI2())
    ]

    for shape in shapes:
        shape.resizeByPercentage(2.5)
        shape.draw()


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

PROTOTYPE

```
from copy import deepcopy, copy


copyfunc = deepcopy


def Prototype(name, bases, dict):
    class Cls:
        pass

    Cls.__name__ = name
    Cls.__bases__ = bases
    Cls.__dict__ = dict
    inst = Cls()
    inst.__call__ = copyier(inst)
    return inst


class copyier:
    def __init__(self, inst):
        self._inst = inst

    def __call__(self):
        newinst = copyfunc(self._inst)
        if copyfunc == deepcopy:
            newinst.__call__._inst = newinst
        else:
            newinst.__call__ = copyier(newinst)
        return newinst


class Point:
    __metaclass__ = Prototype
    x = 0
    y = 0

    def move(self, x, y):
        self.x += x
        self.y += y



a = Point()
print a.x, a.y          # prints 0 0
a.move(100, 100)
print a.x, a.y          # prints 100 100

Point.move(50, 50)
print Point.x, Point.y  # prints 50 50
p = Point()
print p.x, p.y          # prints 50 50

q = p()
print q.x, q.y          # prints 50 50
```

VISITOR

```
class CodeGeneratorVisitor(object):
    @dispatch.on('node')
    def visit(self, node):
        """This is the generic method"""

    @visit.when(ASTNode)
    def visit(self, node):
        map(self.visit, node.children)

    @visit.when(EchoStatement)
    def visit(self, node):
        self.visit(node.children)
        print "print"

    @visit.when(BinaryExpression)
    def visit(self, node):
        map(self.visit, node.children)
        print node.props['operator']

    @visit.when(Constant)
    def visit(self, node):
        print "push %d" % node.props['value']


sometree = None
CodeGeneratorVisitor().visit(sometree)

# Output:
# push 1
# print
# push 2
# push 4
# push 3
# multiply
# plus
# print
```

FACTORY METHOD

```
class Pizza(object):
    def __init__(self):
        self._price = None

    def get_price(self):
        return self._price

class HamAndMushroomPizza(Pizza):
    def __init__(self):
        self._price = 8.5


class DeluxePizza(Pizza):
    def __init__(self):
        self._price = 10.5

class HawaiianPizza(Pizza):
    def __init__(self):
        self._price = 11.5


class PizzaFactory(object):
    @staticmethod
    def create_pizza(pizza_type):
        if pizza_type == 'HamMushroom':
            return HamAndMushroomPizza()
        elif pizza_type == 'Deluxe':
            return DeluxePizza()
        elif pizza_type == 'Hawaiian':
            return HawaiianPizza()


if __name__ == '__main__':
    for pizza_type in ('HamMushroom', 'Deluxe', 'Hawaiian'):
        print 'Price of {0} is {1}'.format(pizza_type, \
               PizzaFactory.create_pizza(pizza_type).get_price())


# ------------------- Second example -----------------


class JapaneseGetter:
    """A simple localizer a la gettext"""

    def __init__(self):
        self.trans = dict(dog="犬", cat="猫")

    def get(self, msgid):
        """We'll punt if we don't have a translation"""

        try:
            return unicode(self.trans[msgid], "utf-8")
        except KeyError:
            return unicode(msgid)


class EnglishGetter:
    """Simply echoes the msg ids"""

    def get(self, msgid):
        return unicode(msgid)

def get_localizer(language="English"):
    """The factory method"""

    languages = dict(English=EnglishGetter, Japanese=JapaneseGetter)

    return languages[language]()


# Create our localizers
e, j = get_localizer("English"), get_localizer("Japanese")

# Localize some text
for msgid in "dog parrot cat".split():
    print e.get(msgid), j.get(msgid)
```

ABSTRACT FACTORY

```
"""Implementation of the abstract factory pattern"""

import random


class PetShop:
    """A pet shop"""

    def __init__(self, animal_factory=None):
        """pet_factory is our abstract factory. We can set it at will."""
        self.pet_factory = animal_factory

    def show_pet(self):
        """Creates and shows a pet using the abstract factory"""
        pet = self.pet_factory.get_pet()
        print "This is a lovely", pet
        print "It says", pet.speak()
        print "It eats", self.pet_factory.get_food()


# Stuff that our factory makes

class Dog:
    def speak(self):
        return "woof"

    def __str__(self):
        return "Dog"


class Cat:
    def speak(self):
        return "meow"

    def __str__(self):
        return "Cat"


# Factory classes

class DogFactory:
    def get_pet(self):
        return Dog()

    def get_food(self):
        return "dog food"


class CatFactory:
    def get_pet(self):
        return Cat()

    def get_food(self):
        return "cat food"


# Create the proper family

def get_factory():
    """Let's be dynamic!"""
    return random.choice([DogFactory, CatFactory])()


# Show pets with various factories

shop = PetShop()

for i in range(3):
    shop.pet_factory = get_factory()
    shop.show_pet()
    print "=" * 10
```