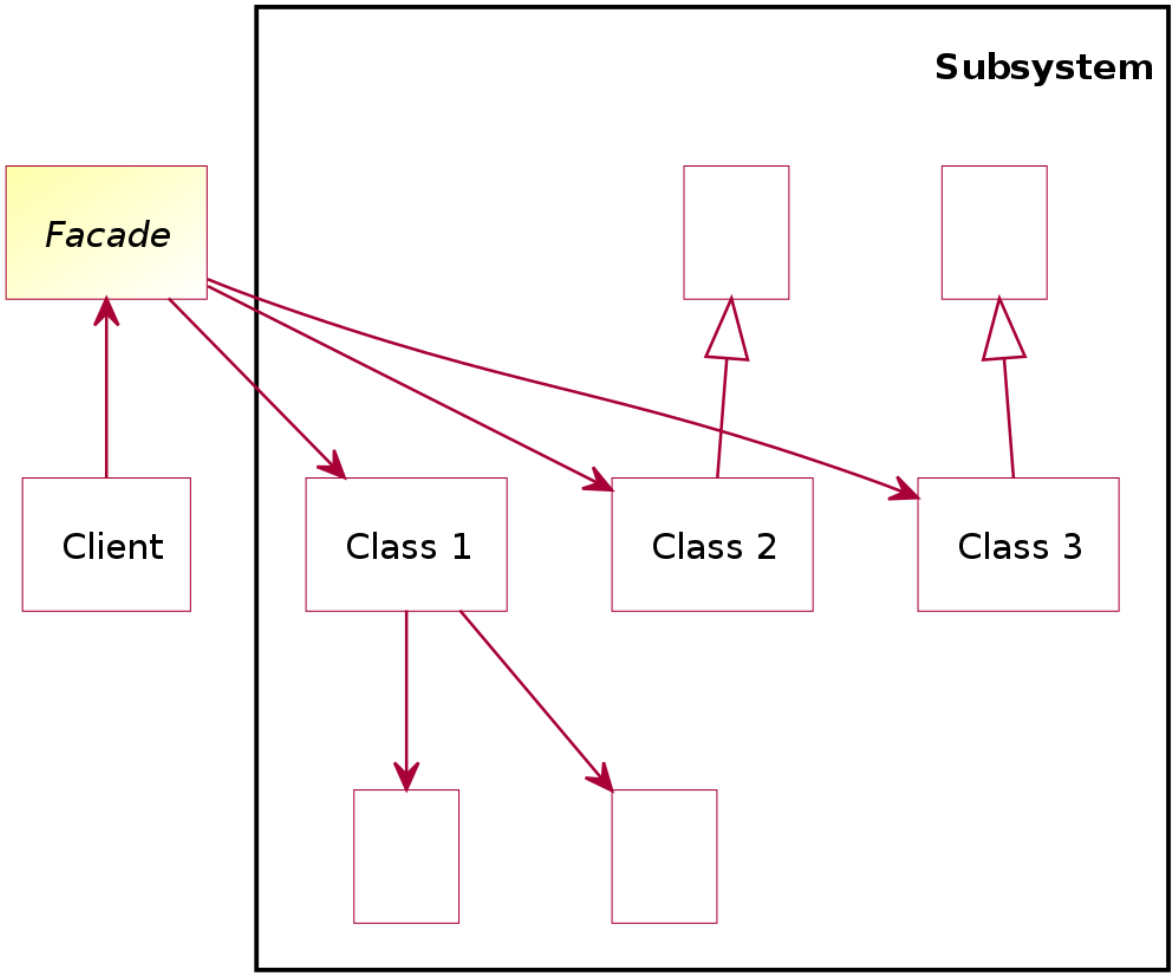
Паттерны (шаблоны) проектирования

Gang of Four (GoF)
«Банда четырёх»

- Эрих Гамм
- Ричард Хелм
- Ральф Джонсон
- Джон Влиссидес

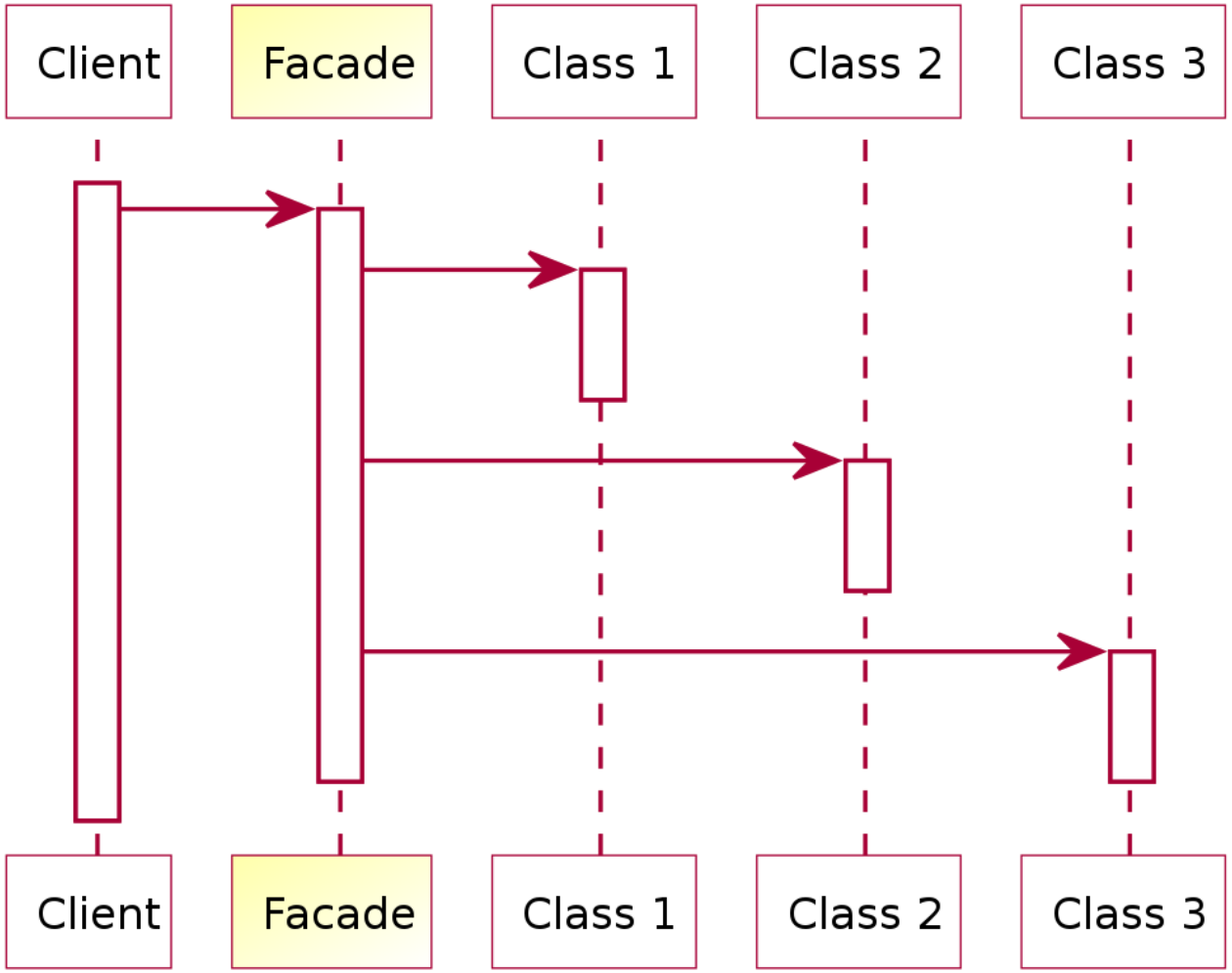
ФАСАД (FACADE)

Диаграмма классов

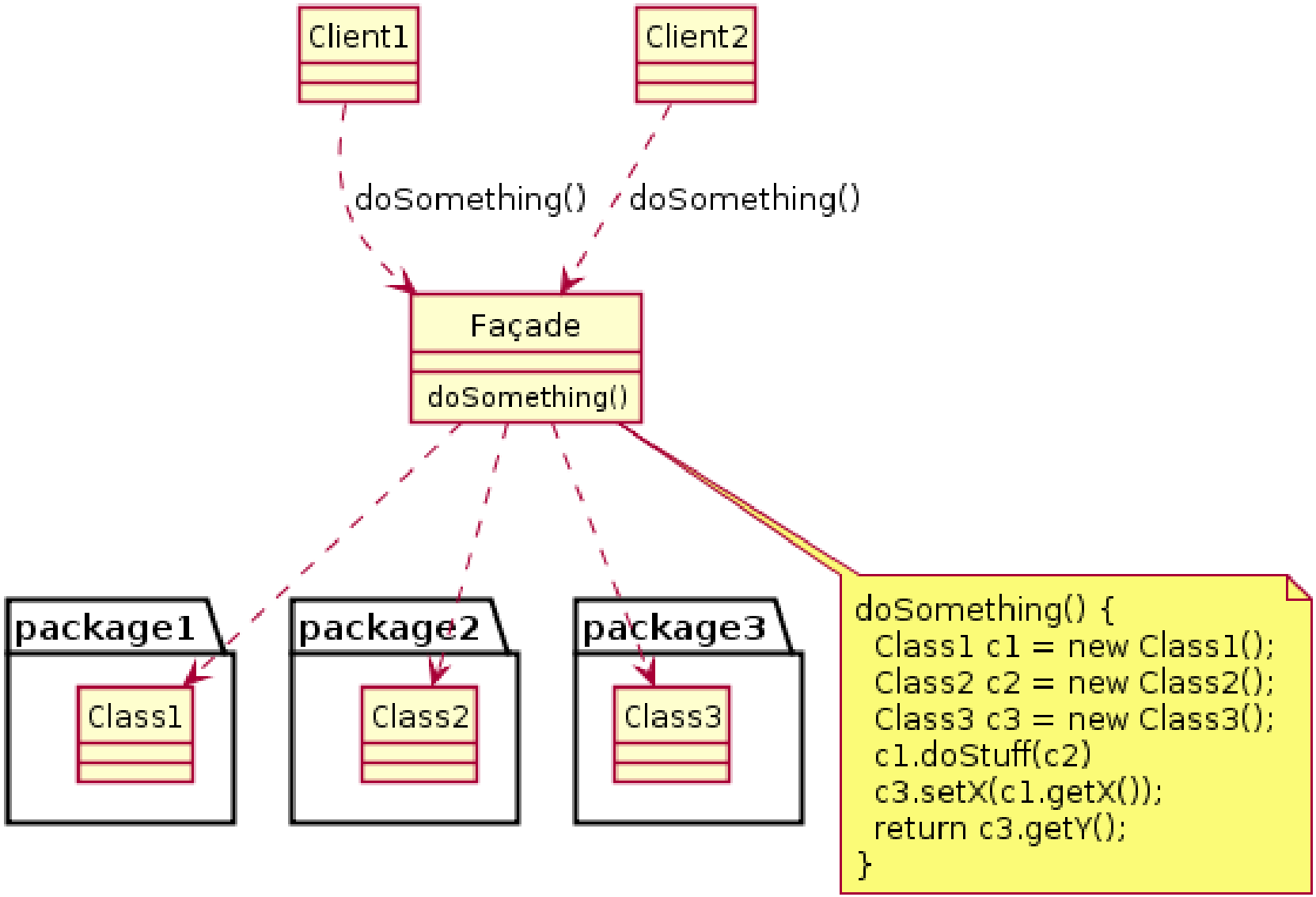


Sample class diagram

Диаграмма последовательности



Sample sequence diagram


Client1
Client2
doSomething()
doSomething()
Façade
doSomething()
package1
Class1
package2
Class2
package3
Class3
doSomething() {
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
c1.doStuff(c2)
c3.setX(c1.getX());
return c3.getY();
}

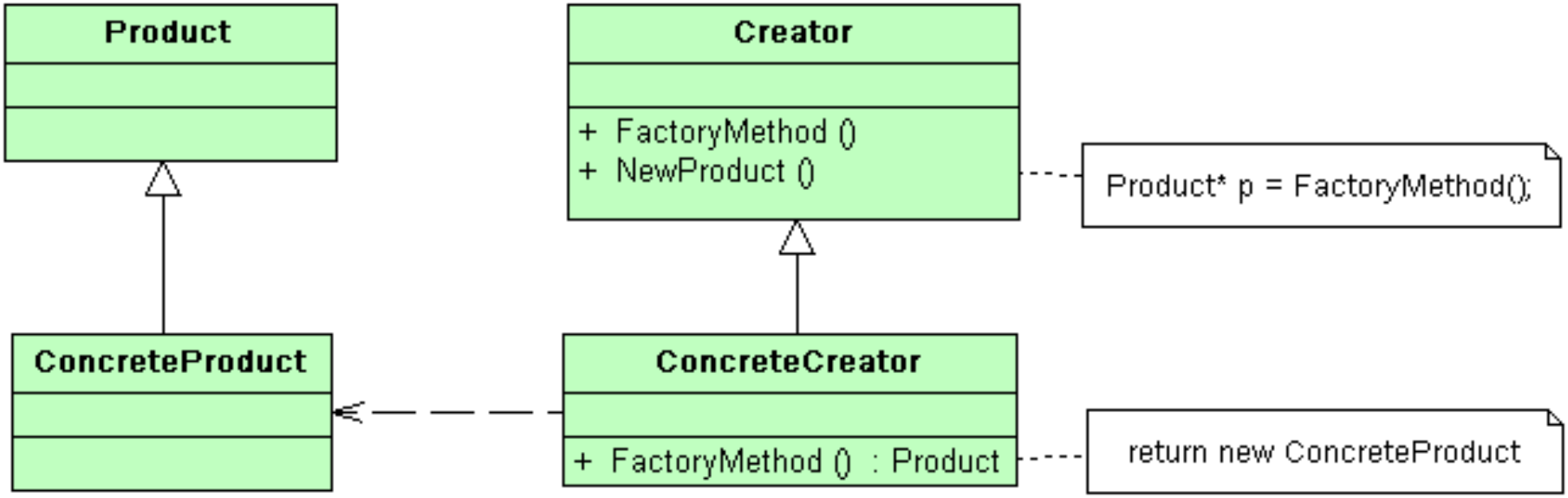
Пример

```
class VideoFile
// ...

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...
```

```
class VideoConverter {
    method convert(filename, format):File {
        file = new VideoFile(filename)
        sourceCodec = new CodecFactory.extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)}}
```

```
class Application {
    method main() {
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()}}
```

ФАБРИКА (FACTORY METHOD)


Product
Creator
+ FactoryMethod ()
+ NewProduct ()
Product* p = FactoryMethod();
ConcreteProduct
ConcreteCreator
+ FactoryMethod () : Product
return new ConcreteProduct

Класс универсального продукта

```
class Coffee:
    def __init__(self):
        self.Title = ''

    def grindCoffee(self):        # перемалываем кофе
        …

    def makeCoffee(self):         # делаем кофе
        …

    def pourIntoCup(self):        # наливаем в чашку
        …

    def getName(self):            # передаем клиенту
        return 'Ваш кофе - ' + self.Title
```

Конкретные продукты

```
class Americano(Coffee):
    def __init__(self):
        self.Title = 'AMERICANO'
```

```
class CaffeLatte(Coffee):
    def __init__(self):
self.Title = 'CAFFE_LATTE'
```

```
class Cappuccino(Coffee):
    def __init__(self):
        self.Title =
'CAPPUCCINO'
```

```
class Espresso(Coffee):
    def __init__(self):
self.Title = 'ESPRESSO'
```

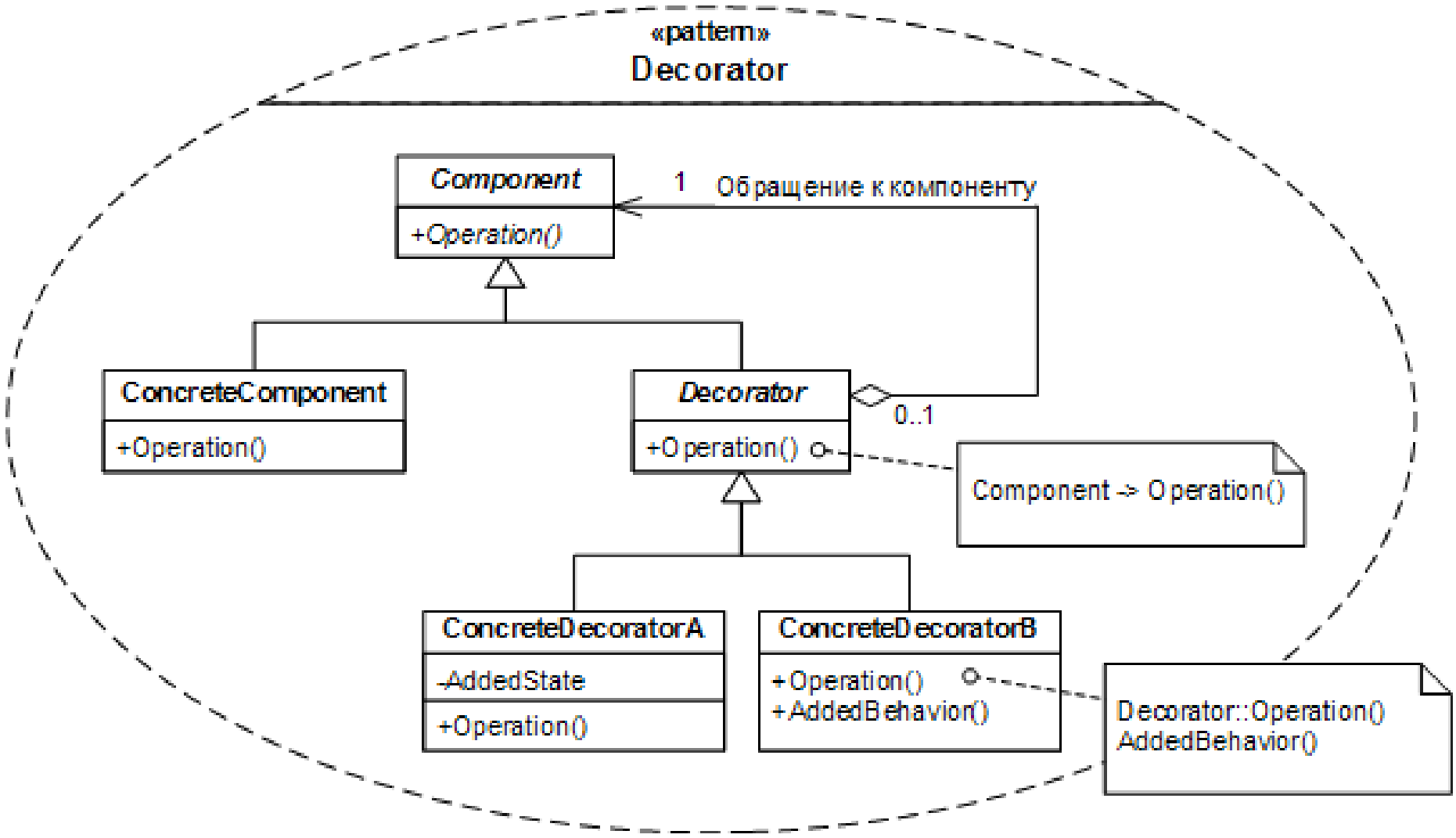
Фабрика

```
class SimpleCoffeeFactory():

    def CreateCoffee(self, TypeCoffee):
        coffee = None
        if TypeCoffee == 'AMERICANO':
            coffee = Americano()
        if TypeCoffee == 'ESPRESSO':
            coffee = Espresso()
        if TypeCoffee == 'CAPPUCCINO':
            coffee = Cappuccino()
        if TypeCoffee == 'CAFFE_LATTE':
            coffee = CaffeLatte()
        return coffee;
```

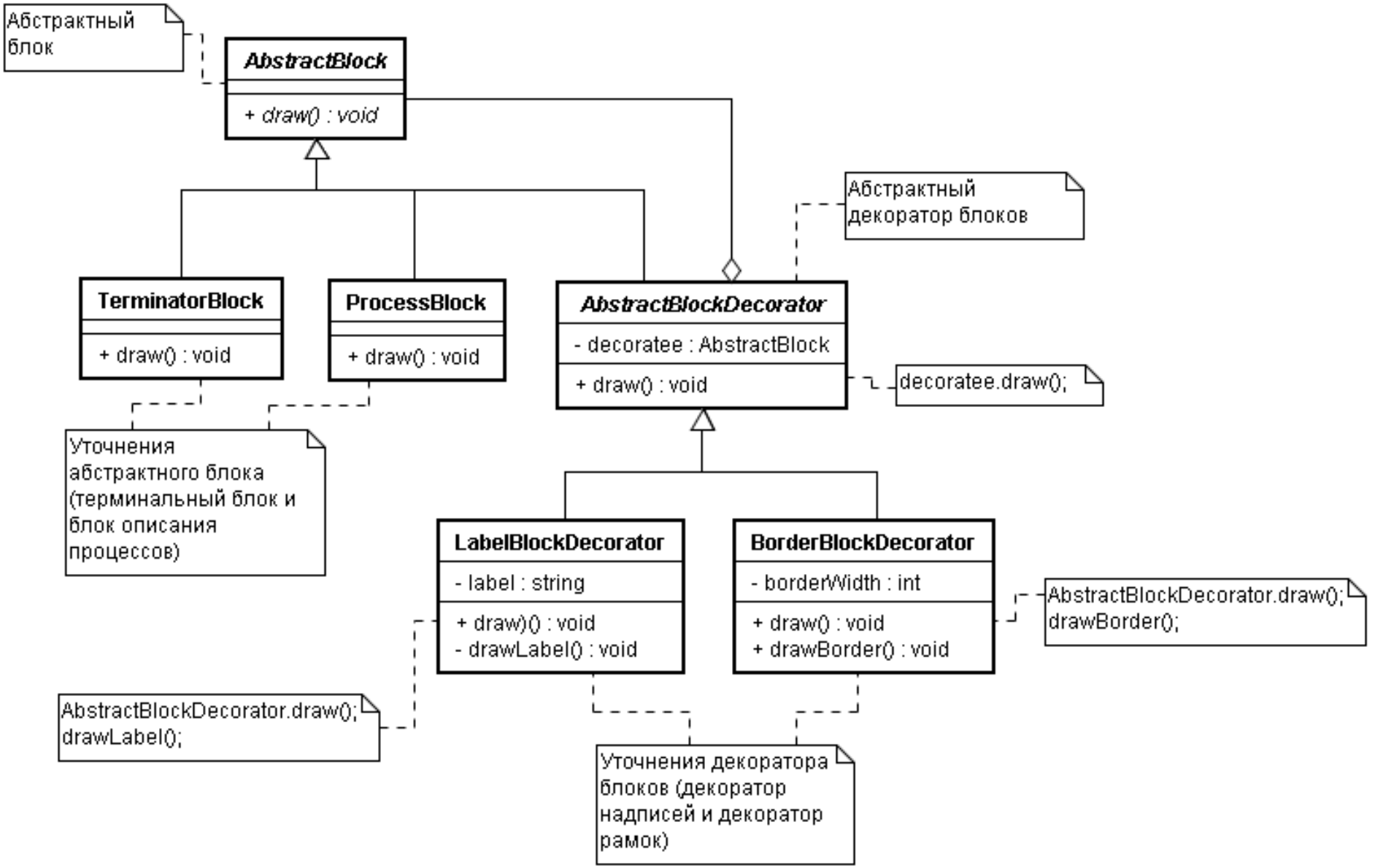
Использование

```
CoffeList =
['ESPRESSO','AMERICANO','CAFFE_LATTE','CAPPUCCINO’]

import random

f = SimpleCoffeeFactory()
coffee = f.CreateCoffee(CoffeList[random.choice(range(0, 4))])
coffee.grindCoffee();
coffee.makeCoffee();
coffee.pourIntoCup();
print(coffee.getName())  # Ваш кофе - CAPPUCCINO
```

ДЕКОРАТОР (DECORATOR)


«pattern»
Decorator
Component
+Operation()
1
Обращение к компоненту
ConcreteComponent
+Operation()
Decorator
+Operation()
0..1
Component -> Operation()
ConcreteDecoratorA
-AddedState
+Operation()
ConcreteDecoratorB
+Operation()
+AddedBehavior()
Decorator::Operation()
AddedBehavior()

Пример


Абстрактный блок
AbstractBlock
+ draw() : void
Абстрактный декоратор блоков
TerminatorBlock
+ draw() : void
ProcessBlock
+ draw() : void
AbstractBlockDecorator
- decoratee : AbstractBlock
+ draw() : void
decoratee.draw();
Уточнения абстрактного блока (терминальный блок и блок описания процессов)
LabelBlockDecorator
- label : string
+ draw)() : void
- drawLabel() : void
BorderBlockDecorator
- borderWidth : int
+ draw() : void
+ drawBorder() : void
AbstractBlockDecorator.draw();
drawBorder();
AbstractBlockDecorator.draw();
drawLabel();
Уточнения декоратора блоков (декоратор надписей и декоратор рамок)

Блоки диаграммы

```
class AbstractBlock:
  # Абстрактный блок
    def draw(self):
        raise NotImplementedError();

class TerminatorBlock(AbstractBlock):
  # Терминальный блок (начало/конец, вход/выход)
    def draw(self):
        print('Terminator block drawing ... ')

class ProcessBlock(AbstractBlock):
  # Блок - процесс (один или несколько операторов)
    def draw(self):
        print("Process block drawing ... ")
```

Абстрактный декоратор

```
class AbstractBlockDecorator(AbstractBlock):
  # Абстракный декоратор блоков

  def __init__(self, decoratee):
    # _decoratee - ссылка на декорируемый объект
        self._decoratee = decoratee

  def draw(self):
        self._decoratee.draw()
```

Декоратор надписи

```
class LabelBlockDecorator(AbstractBlockDecorator):
  #  Декорирует блок текстовой меткой

  def __init__(self, decoratee, label):
    self._decoratee = decoratee
    self._label = label

  def draw(self):
    AbstractBlockDecorator.draw(self)
    self._drawLabel()

  def _drawLabel(self):
    print(" ... drawing label  " + self._label)
```

Декоратор рамки

```
class BorderBlockDecorator(AbstractBlockDecorator):
  # Декорирует блок специальной рамкой

  def __init__(self, decoratee, borderWidth):
    self._decoratee = decoratee
    self._borderWidth = borderWidth

  def draw(self):
    AbstractBlockDecorator.draw(self)
    self._drawBorder()

  def _drawBorder(self):
    print(" ... drawing border with width " + str(self._borderWidth))
```

Использование

```
# терминальный блок
tBlock = TerminatorBlock()
# блок – процесс
pBlock = ProcessBlock()

# Применим LabelDecorator к терминальному блоку
labelDecorator = LabelBlockDecorator(tBlock, "Label222")

# Применим BorderDecorator к терминальному блоку, после применения LabelDecorator
borderDecorator1 = BorderBlockDecorator(labelDecorator, 22)

# Применим BorderDecorator к блоку – процессу
borderDecorator2 = BorderBlockDecorator(pBlock, 22)

labelDecorator.draw()  # Terminator block drawing ...  drawing label Label222
borderDecorator1.draw()  # Terminator block drawing ... drawing label Label222  drawing border with width 22
borderDecorator2.draw()  #  Process block drawing ...  drawing border with width 22
```

ПОСЕТИТЕЛЬ (VISITOR)

Определяем классы

```
class Vector():
    def __init__(self):
        self.Ptype = ''

    def getType(self):
        return self.Ptype
```

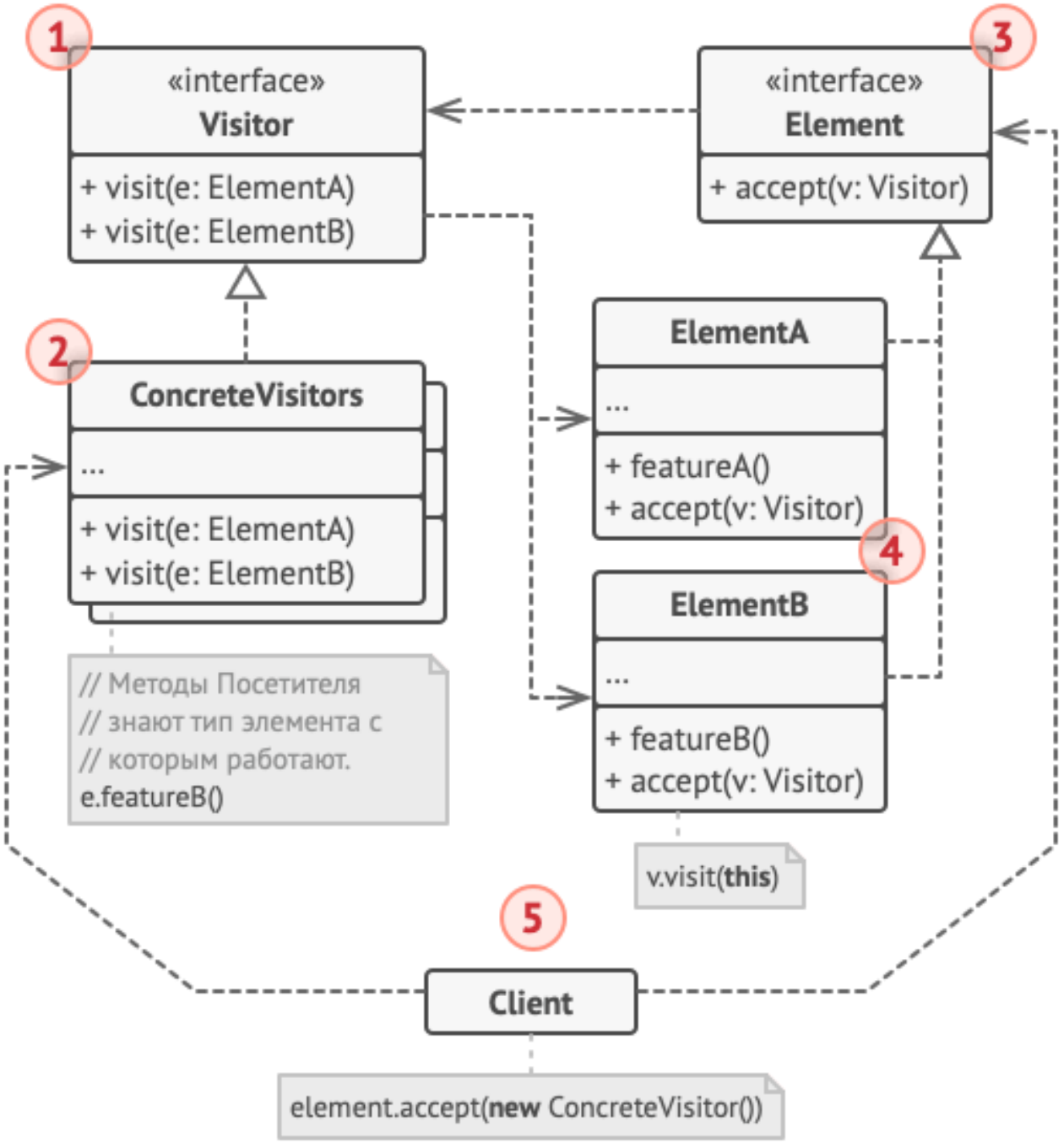
```
class Vector2D(Vector):
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.Ptype = '2D'

class Vector3D(Vector):
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
        self.Ptype = '3D'
```

Добавляем новое поведение

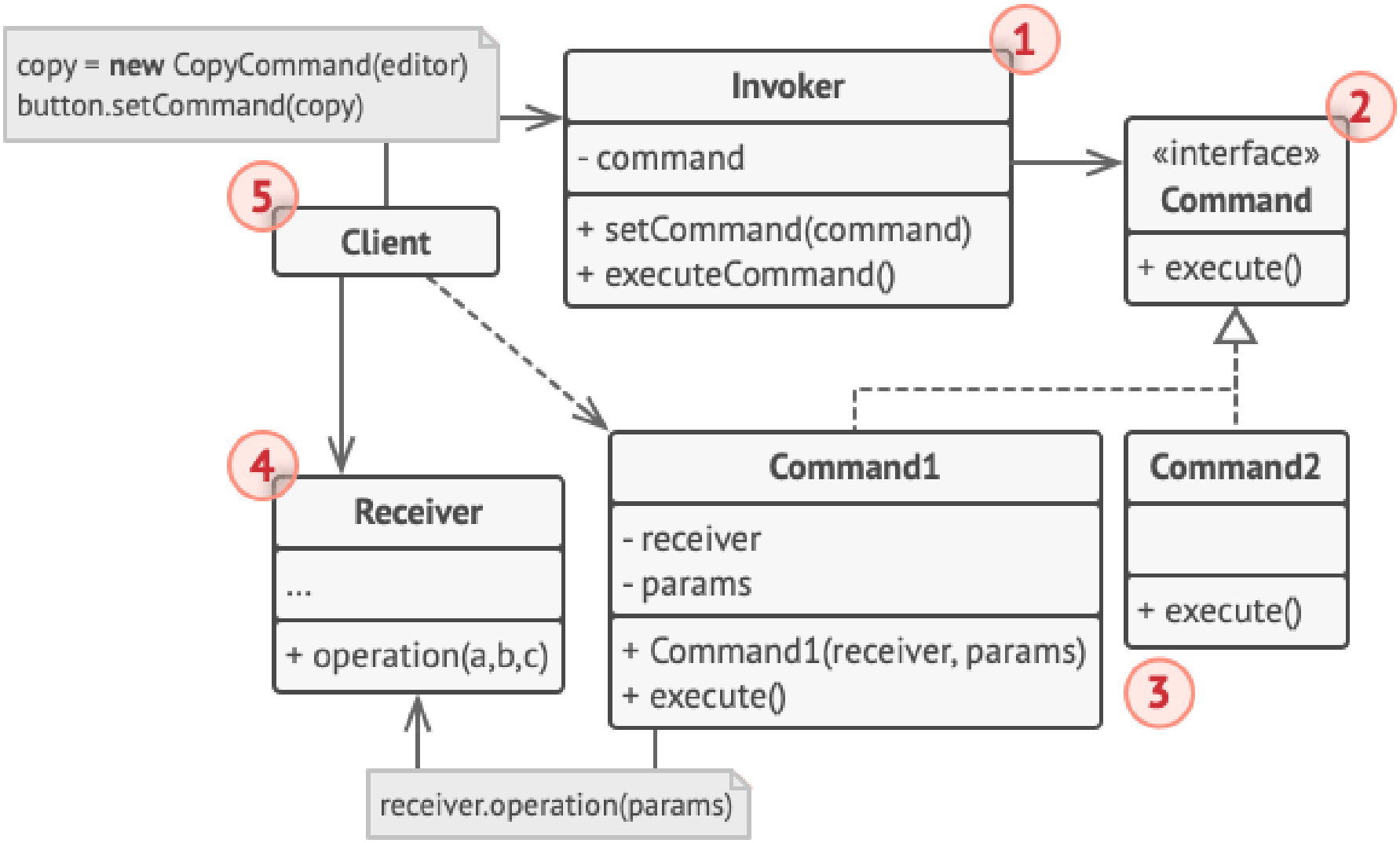
```
class Euklid():  # Visitor
    def metric(self, p):
        if type(p) is Vector2D:
            return math.sqrt(p.x*p.x + p.y*p.y)
        if type(p) is Vector3D:
            return math.sqrt(p.x*p.x + p.y*p.y + p.z*p.z)
        return None
```

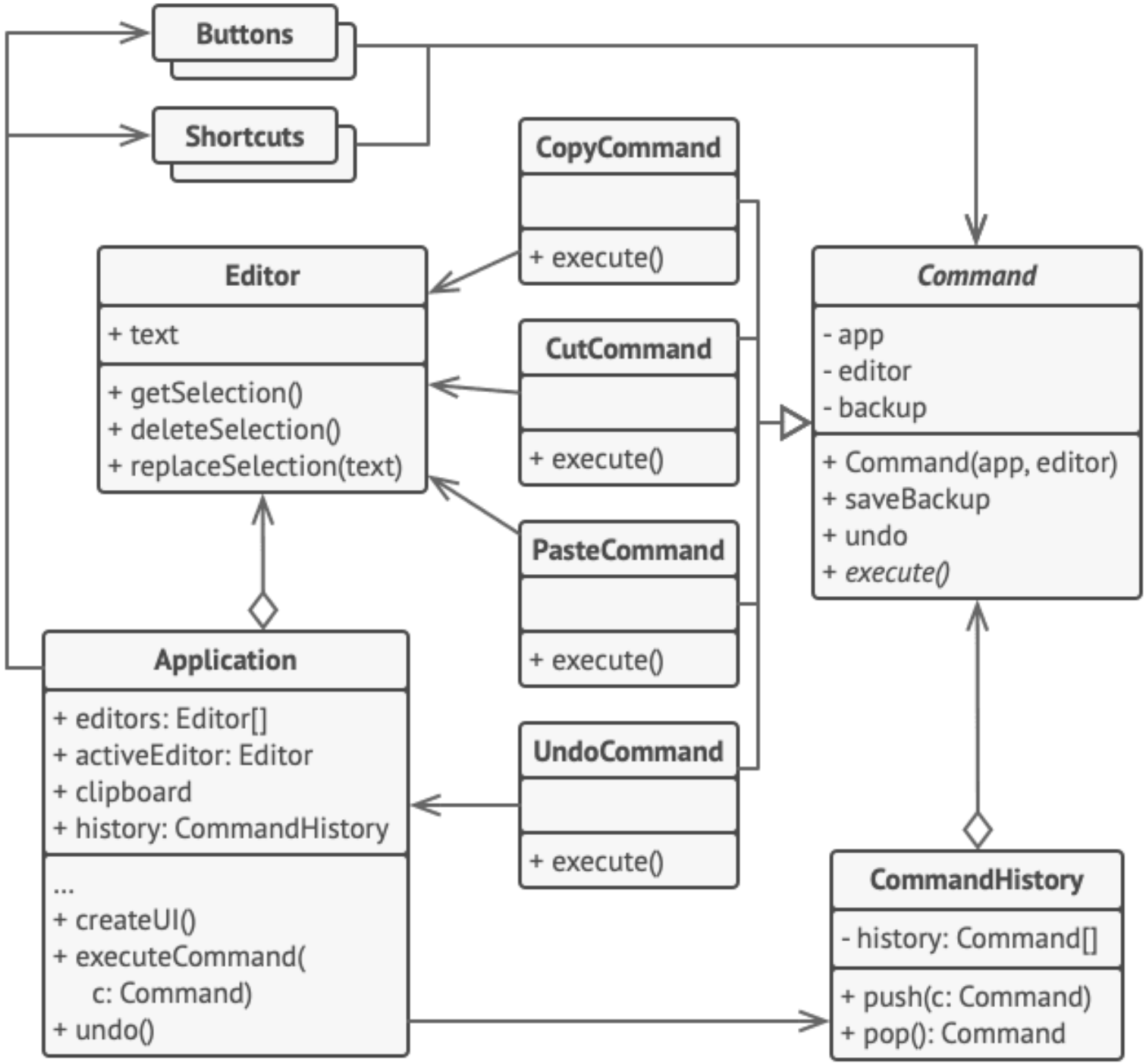
Использование

```
e = Euklid()
p1 = Vector2D(2, 3)
p2 = Vector3D(2, 1, 3)
print(p1.getType())  # 2D
print(e.metric(p1))  # 3.605551275463989
print(p2.getType())  # 3D
print(e.metric(p2))  # 3.7416573867739413
```


1
«interface»
Visitor
+ visit(e: ElementA)
+ visit(e: ElementB)
3
«interface»
Element
+ accept(v: Visitor)
2
ConcreteVisitors
...
+ visit(e: ElementA)
+ visit(e: ElementB)
// Методы Посетителя
// знают тип элемента с
// которым работают.
e.featureB()
ElementA
...
+ featureA()
+ accept(v: Visitor)
4
ElementB
...
+ featureB()
+ accept(v: Visitor)
v.visit(this)
5
Client
element.accept(new ConcreteVisitor())

КОМАНДА (COMMAND)


copy = new CopyCommand(editor)
button.setCommand(copy)
1
Invoker
- command
+ setCommand(command)
+ executeCommand()
2
«interface»
Command
+ execute()
5
Client
4
Receiver
...
+ operation(a,b,c)
Command1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Command2
+ execute()
3
receiver.operation(params)


Buttons
Shortcuts
CopyCommand
+ execute()
Editor
+ text
+ getSelection()
+ deleteSelection()
+ replaceSelection(text)
CutCommand
+ execute()
Command
- app
- editor
- backup
+ Command(app, editor)
+ saveBackup
+ undo
+ execute()
PasteCommand
+ execute()
Application
+ editors: Editor[]
+ activeEditor: Editor
+ clipboard
+ history: CommandHistory
...
+ createUI()
+ executeCommand(
c: Command)
+ undo()
UndoCommand
+ execute()
CommandHistory
- history: Command[]
+ push(c: Command)
+ pop(): Command

Глобальная история команд — стек

```
from abc import ABCMeta, abstractmethod

class CommandHistory():
    def __init__(self):
        self.history = []

    def push(self, c):
        # Добавить команду в конец массива-истории.
        self.history.append(c)

    def pop(self):
        # Достать последнюю команду из массива-истории.
        return self.history.pop()
```

Класс Команда

```
class Command(metaclass=ABCMeta):
    def __init__(self, d):
        self.d = d
        self.backup = ''
        self.name = ''

    def saveBackup(self):        # Сохраняем состояние редактора.
        self.backup = self.d.text
        self.d.history.push(self)

    def undo(self):   # Восстанавливаем состояние редактора.
        self.d.text = self.backup

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

Конкретные команды – добавление текста

```
class AddCommand(Command):
    def execute(self, text):
        self.name = 'ADD'
        self.saveBackup()
        self.d.text = self.d.text + text
        print('Add: ' + text)
        return True
```

Конкретные команды – замена текста

```
class ReplaceCommand(Command):
    def execute(self, text1,text2):
        self.name = 'REPLACE'
        self.saveBackup()
        self.d.text = self.d.text.replace(text1, text2)
        print('Replace: ' + text1)
        return True
```

Конкретные команды – отмена команды

```
class UndoCommand(Command):
    def execute(self):
        command = self.d.history.pop()
        if (command != None):
            command.undo()
            print('Undo ' + command.name)
        return True
```

Документ

```
class Document():
    def __init__(self):
        self.text = ''
        self.history = CommandHistory()

doc = Document()
AddC = AddCommand(doc)
RepC = ReplaceCommand(doc)
UndC = UndoCommand(doc)
```

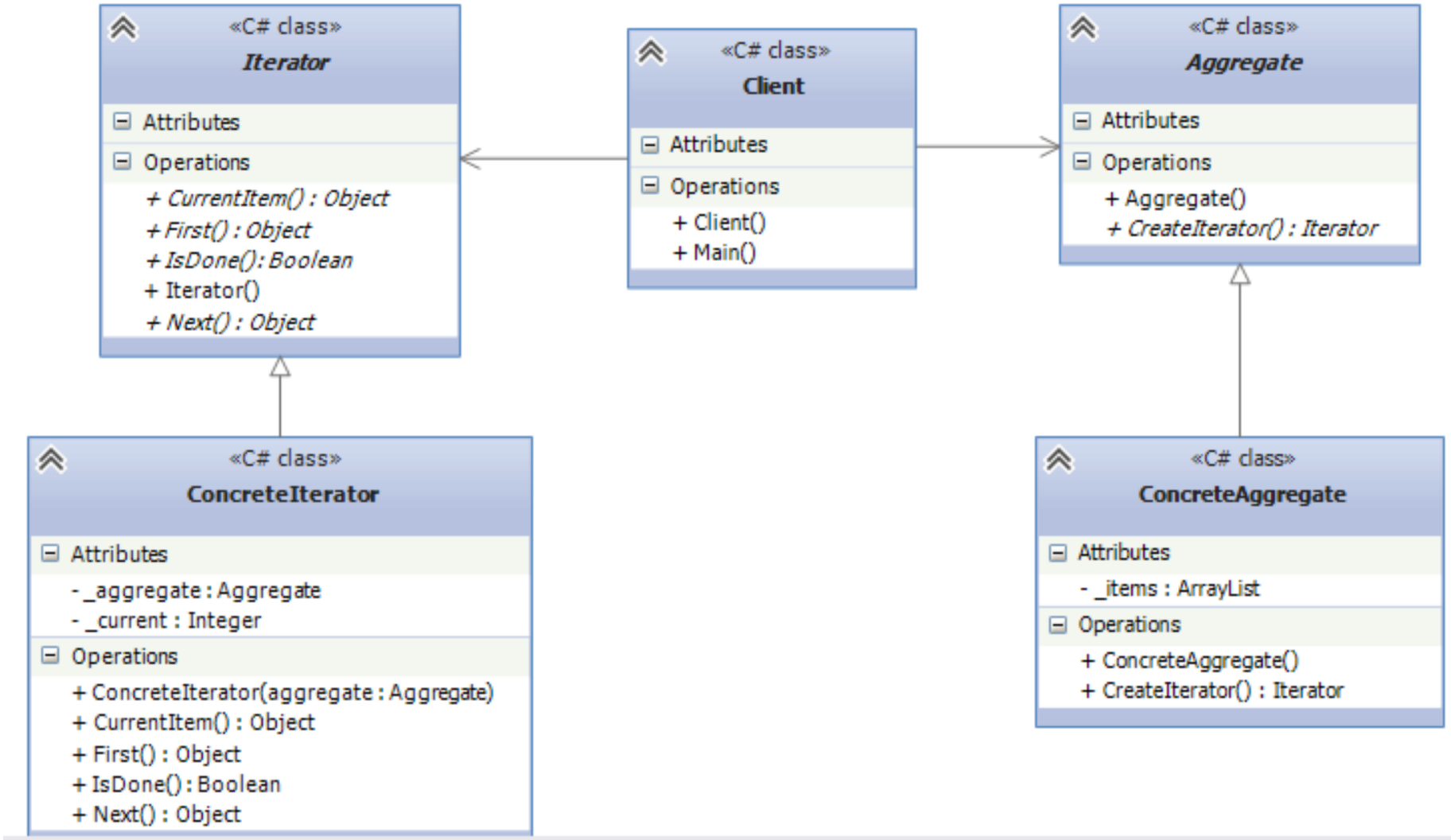
Использование

```
AddC.execute('aaa')                print(doc.text)
AddC.execute('bbb')                #  aaa
AddC.execute('ccc')                #  aaabbb
print(doc.text)                    #  aaabbbccc
RepC.execute('bb','rrrr')          #  aaabbbccc
AddC.execute('fff')                #  aaarrrrbccc
UndC.execute()                     #  aaarrrrbcccfff
UndC.execute()                     #  aaarrrrbccc
                                   #  aaabbbccc
```

ИТЕРАТОР (ITERATOR)

Итератор

- Поведенческий шаблон проектирования
- Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов
- Итератор может ничего не знать о типе итерируемого агрегата
- Итераторы позволяют абстрагироваться от типа и признака окончания агрегата, используя полиморфный Next() и полиморфный end(), возвращающий значение «конец агрегата»


«C# class»
Iterator
Attributes
Operations
+ CurrentItem() : Object
+ First() : Object
+ IsDone(): Boolean
+ Iterator()
+ Next() : Object
«C# class»
Client
Attributes
Operations
+ Client()
+ Main()
«C# class»
Aggregate
Attributes
Operations
+ Aggregate()
+ CreateIterator() : Iterator
«C# class»
ConcreteIterator
Attributes
- _aggregate : Aggregate
- _current : Integer
Operations
+ ConcreteIterator(aggregate : Aggregate)
+ CurrentItem() : Object
+ First() : Object
+ IsDone() : Boolean
+ Next() : Object
«C# class»
ConcreteAggregate
Attributes
- _items : ArrayList
Operations
+ ConcreteAggregate()
+ CreateIterator() : Iterator

Абстрактный итератор

```
from abc import ABCMeta, abstractmethod
class Iterator(metaclass=ABCMeta):
  @abstractmethod
  def __init__(self, collection, cursor):
      self._collection = collection # коллекция, по которой производится проход итератором
      self._cursor = cursor  # изначальное положение курсора в коллекции (ключ)
  @abstractmethod
  def current(self): # Вернуть текущий элемент, на который указывает итератор
       pass
  @abstractmethod
  def next(self): # Сдвинуть курсор на следующий элемент коллекции и вернуть его
      pass
  @abstractmethod
  def has_next(self): # Проверить, существует ли следующий элемент коллекции
      pass
```

Абстрактная коллекция

```
class Collection(metaclass=ABCMeta):
  @abstractmethod
  def iterator(self):
        pass
```

Итератор списка книг

```
class BookIterator(Iterator):

    def __init__(self, collection: list):
        super().__init__(collection, 0)

    def current(self):
        if self._cursor < len(self._collection):
            return self._collection[self._cursor]

    def next(self):
        if len(self._collection) >= self._cursor + 1:
            self._cursor += 1
            return self._collection[self._cursor]
```

```
def has_next(self):
    return len(self._collection) >=
    self._cursor + 1
```

Коллекция книг

```
class BookCollection(Collection):

    def __init__(self, collection: list):
        self._collection = collection

    def iterator(self):
        return BookIterator(self._collection)
```

Работа с итератором

```
Shelf = BookCollection(["Война и мир",
                        "Отцы и дети",
                        "Вишневый сад"])
iterator = Shelf.iterator()
print(iterator.current()) # "Война и мир"
iterator.next()
print(iterator.next()) # "Отцы и дети"
print(iterator.has_next())  # True
```

Класс книга

```
class Book():
    def __init__(self, T, A):
        self.Title = T
        self.Author = A

Shelf = BookCollection([Book('Война и мир','Л.Н.Толстой'), Book('Отцы и дети','И.С.Тургенев'), Book('Вишневый сад','А.П.Чехов')])

class BookIterator(Iterator):
   def current(self):
        if self._cursor < len(self._collection):
            return self._collection[self._cursor].Title

    def next(self):
        if len(self._collection) >= self._cursor + 1:
            self._cursor += 1
            return self._collection[self._cursor].Title
```