МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт радиоэлектроники и информационной безопасности

Кафедра информационной безопасности

М.А. Артеменко, А.В. Лебеденко

ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ Python

Учебное пособие

Севастополь
СевГУ
2018

УДК 004.432
А86

Р е ц е н з е н т:
И.П. Шумейко - кандидат физико-математических наук, доцент

Артеменко М.А.
А86 Технологии и методы программирования на языке Python: учеб. пособие / М.А. Артеменко, А.В. Лебеденко. – Севастополь: СевГУ, 2018. – 78 с.

В пособии изложены технологии и методы разработки программного обеспечения на языке программирования python3. Рассмотрен объектно-ориентированный подход проектирования приложений, использование паттернов и технологии разработки сетевых приложений.

Может быть использовано для самостоятельной работы студентов направления подготовки «Информационная безопасность» очной и очно-заочной форм обучения.

УДК 004.432

Учебное пособие утверждено на заседании кафедры информационной безопасности, протокол № 2 от 2 сентября 2015 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Технологией программирования называют совокупность методов и средств, используемых в процессе разработки программного обеспечения. Как любая другая технология она представляет собой набор технологических инструкций, включающих:

– указание последовательности выполнения технологических операций;

– перечисление условий, при которых выполняется та или иная операция;

– описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т.п.

Кроме того, технология программирования определяет способ описания модели, используемой на конкретном этапе разработки проектируемой системы.

Различают технологии, используемые на конкретных этапах разработки или для решения отдельных задач этих этапов, и технологии, охватывающие несколько этапов или весь процесс разработки. В основе первых, как правило, лежит ограниченно применимый метод, позволяющий решить конкретную задачу. В основе вторых обычно лежит методология, определяющая совокупность методов, используемых на разных этапах разработки.

Раздел 1. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД НА ПРИМЕРЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ PYTHON

1.1. Области видимости и пространства имен в Python

Пространство имен (*namespace*) – это набор связей имен с объектами. Нет абсолютно никакой связи между именами в разных пространствах имен: например, два разных модуля могут без проблем определять функцию «*maximize*», так как пользователи модулей будут использовать имена модулей в качестве префиксов.

Пространства имен создаются в различные моменты и имеют разное время жизни. Пространство имен, содержащее встроенные имена создается при запуске интерпретатора и не удаляется никогда. Глобальное пространство имен модуля создается при вычитке определения модуля. Обычно, пространства имен модулей также «живут» до выхода из интерпретатора. Выражения, выполняемые верхнеуровневым порождением интерпретатора, прочитанные из файла сценария или интерактивно, рассматриваются как часть модуля под названием ***__main__***, поэтому у них есть свое собственное глобальное пространство имен. (Встроенные имена по факту также находятся в модуле, он называется ***builtins***).

Локальное пространство имен функции создается при ее вызове и удаляется когда функция возвращает значение либо порождает исключение, внутри нее не перехваченное.

Область видимости (*scope*) – это текстовая область в программе на Python, на которой прямым образом доступно пространство имен. «Доступно прямым образом» подразумевает, что явная ссылка на имя вынуждает интерпретатор искать это имя в пространстве имен.

Несмотря на то, что области видимости определяются статически, используются они динамически. В любой момент во время выполнения существует как минимум три вложенных области видимости, чьи пространства имен доступны прямым образом: самая внутренняя область видимости (по ней поиск осуществляется в первую очередь) содержит локальные имена; пространства имен всех объемлющих [данный код] функций, поиск по которым начинается с ближайшей объемлющей [код] области видимости; область видимости среднего уровня, по ней следующей проходит поиск и она содержит глобальные имена текущего модуля; и самая внешняя область видимости (заключительный поиск) – это пространство имен, содержащее встроенные имена.

Если имя определено глобально, тогда все ссылки и присваивания уходят прямо в область видимости среднего уровня, содержащую глобальные имена модуля. Чтобы сменить привязку у всех переменных, найденных вне самой внутренней области видимости, можно использовать оператор *nonlocal*; если такая переменная не объявлена как *nonlocal*, то она ис-

пользуется только для чтения (попытка записать значение в такую переменную создаст новую локальную переменную в самой внутренней области видимости, оставляя идентично названную вовне переменную без изменений).

Обычно локальная область видимости ссылается на локальные имена текущей (на уровне текста) функции. Вне функций локальная область видимости ссылается на то же пространство имен, что и глобальная область видимости: пространство имен модуля. Определения классов помещают в локальную область видимости еще одно пространство имен.

Оператор ***global*** можно использовать для того, чтобы объявить определенные переменные как привязанные к глобальной области видимости и указывает, что их переназначения должны происходить в ней; оператор ***nonlocal*** помечает переменные как привязанные к окружающей их области видимости и указывает, что их переназначения должны происходить в ней.

Пример:

```
def scope_test():
    def do_local():
        spam = "локальный спам"
    def do_nonlocal():
        nonlocal spam
        spam = "нелокальный спам"
    def do_global():
        global spam
        spam = "глобальный спам"

    spam = "тестовый спам"
    do_local()
    print("После локального присваивания:", spam)
    do_nonlocal()
    print("После нелокального присваивания:", spam)
    do_global()
    print("После глобального присваивания:", spam)

scope_test()
print("В глобальной области видимости:", spam)
```

1.2. Основные понятия

Python проектировался как объектно-ориентированный язык программирования. Это означает, что он построен с учетом следующих принципов:

1. Все данные в нем представляются объектами.
2. Программу можно составить как набор взаимодействующих объектов, посылающих друг другу сообщения.
3. Каждый объект имеет собственную часть памяти и может состоять из других объектов.
4. Каждый объект имеет тип.
5. Все объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия).

ООП – это методология написания кода.

Объектно-ориентированный подход использует объектную декомпозицию, то есть поведение системы описывается в терминах взаимодействия объектов.

Также существует определение, более приближенное к программированию. **Объектно-ориентированное**, или **объектное**, **программирование** – парадигма программирования, в которой основными концепциями являются понятия объектов и классов. В случае языков с прототипированием вместо классов используются объекты-прототипы.

Класс – это абстракция множества сущностей реального мира, объединенных общностью структуры и поведения.

Объект – это элемент класса, то есть абстракция определенной сущности.

С точки зрения ООП в Python, класс представляет собой коллекцию данных.

Синтаксис определения класса.

Простейшая форма определения класса выглядит так:

```
class ИмяКласса:
["""  Строка документирования """]
    <оператор-1>
    …
    …
    …
    <оператор-N>
```

Определения классов, как и определения функций (операторы `def`), должны быть исполнены для того, чтобы возыметь действие

На практике, внутри определения класса обычно помещаются определения функций, но позволено использовать и другие. –

При вводе определения класса создается новое пространство имен, которое и используется в качестве локальной области видимости. Таким образом, все присваивания локальным переменным происходят в этом новом пространстве имен.

При успешном окончании парсинга определения класса (по достижении конца определения), создается *объект-класс* (*class object*). По существу, это обертка вокруг содержимого пространства имен, созданного во время определения класса; подробнее объекты классов мы изучим в следующем разделе.

1.3. Классы в Python

1.3.1. Объекты-классы

Объекты-классы поддерживают два вида операций: ссылки на атрибуты и создание экземпляра.

Ссылки на атрибуты (*Attribute references*) используют стандартный синтаксис, использующийся для всех ссылок на атрибуты в Python: *объект.имя*. Корректными именами атрибутов являются все имена, которые находились в пространстве имен класса при создании объекта-класса. Таким образом, если определение класса выглядело так:

```
class MyClass:
    """Простой пример класса"""
    i = 12345
    def f(self):
        return 'привет мир'
```

то ***MyClass.i и MyClass.f*** являются корректными ссылками на атрибуты, возвращающими целое и *объект-функцию* (function object) соответственно. Атрибутам класса можно присваивать значение, так что вы можете изменить значение MyClass.i через присваивание. __doc__ также является корректным атрибутом, возвращающим строку документации, принадлежащей классу: "Простой пример класса".

Синтаксис определения экземпляра классса выглядит следующим образом:

```
<Экземпляр класса> = <Название класса>([Параметры])
```

Создание экземпляра класса использует синтаксис вызова функции. Просто представьте, что объект-класс – это непараметризированная функция, которая возвращает новый экземпляр класса. Например (предполагая класс, приведенный выше):

```
x = MyClass()
```

создает новый экземпляр класса и присваивает этот объект локальной переменной x.

Операция *создания экземпляра* (`instantiation`) создает объект данного класса. Большая часть классов предпочитает создавать экземпляры, имеющие определенное начальное состояние. Для этого класс может определять специальный метод под именем *__init__()*, например так:

```
def __init__(self):
    self.data = []
```

В других языках программирования этот метод называется конструктором и вызывается в момент создания экземпляра класса.

Метод *__init__()* может иметь параметры. В этом случае аргументы, переданные оператору создания экземпляра класса, передаются методу *__init__()*.

Например:

```
>>> class Complex:
     def __init__(self, realpart, imagpart):
         self.r = realpart
         self.i = imagpart

>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

1.3.2. Объекты-экземпляры

Теперь, что же мы можем делать с объектами-экземплярами? Единственные операции, доступные объектам-экземплярам – это ссылки на атрибуты. Есть два типа корректных имен атрибутов – это атрибуты-данные и методы.

Атрибуты-данные (*data attributes*) аналогичны «переменным экземпляров» в Smalltalk и «членам-данным» в C++. Атрибуты-данные не нужно описывать: как и переменные, они начинают существование в момент первого присваивания. Например, если *x* – экземпляр созданного выше *MyClass*, следующий отрывок кода выведет значение *16*, не вызвав ошибок:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

Другой тип ссылок на атрибуты экземпляра – это *метод* (*method*). Метод – это функция, «принадлежащая» объекту. (В Python термин не уникален для экземпляров класса: другие объекты также могут иметь методы. Например, объекты-списки имеют методы *append, insert, remove, sort* и т.п.

Корректные имена методов объектов-экземпляров зависят от их класса. По определению, все атрибуты класса, являющиеся объектами-функциями, описывают соответствующие методы его экземпляров. Так, в нашем примере, *x.f* является корректной ссылкой на метод, а *x.i* ей не является, поскольку не является и *MyClass.i*. Но при этом *x.f* – это не то же самое, что *MyClass.f*: это объект-метод, а не объект-функция.

1.3.3. Функции доступа к атрибутам классов

Для доступа к атрибутам и методам можно также использовать следующие функции:

getattr() - возвращает значение атрибута по его названию, заданному в виде строки. С помощью этой функции можно сформировать название атрибута динамически во время выполнения программы.

setattr() - задает значение атрибута. Название атрибута указывается в виде строки.

delattr() - удаляет указанный атрибут. Название атрибута указывается в виде строки.

hasattr() - проверяет наличие указанного атрибута. Если атрибут существует, функция возвращает значение True.

```
class Class1:
    var1 = 10
    def f_test(self):
        return self.var1

c1 = Class1()                    # Создаем экземпляр класса
print getattr(c1, "var1")        # Выведет: 10
print getattr(c1, "f_test")()    # Выведет: 10
```

```
print getattr(c1, "var2", 0)      # Выведет: 0, т.к. атрибут не найден
setattr(c1, "var2", 20)           # Создаем атрибут var2
print getattr(c1, "var2", 0)      # Выведет: 20
delattr(c1, "var2")               # Удаляем атрибут var2
print getattr(c1, "var2", 0)      # Выведет: 0, т.к. атрибут не найден
print hasattr(c1, "var1")         # Выведет: True
print hasattr(c1, "var2")         # Выведет: False
```

Все атрибуты класса в языке Python является открытыми (public), т.е. доступными для непосредственного изменения. Кроме того, атрибуты можно создавать динамически после создания класса. Можно создать как атрибут объекта класса, так и атрибут экземпляра класса.

Прицепы ООП применяются аналогична и в других языках программирования. Создание игр, программ, скриптов и сайтов все используют объектно-ориентированное программирование.

1.4. Конструкторы и деструкторы или методы __init__() и __del__() в Python

При создании экземпляра класса интерпретатор автоматически вызывает метод инициализации **__init__()**. В некоторых языках программирования данные методы принято называть конструктором и деструктором класса. Формат метода:

```
def __init__(self[, <Значение1>[, <ЗначениеN>]]):
    <Выражение>
```

С помощью метода **__init__()** можно присвоить значения по умолчанию для атрибутов класса. При создании экземпляра класса начальные значения указываются после имени класса в круглых скобках:

```
<Экземпляр класса> = <Имя класса>(<Значение1>[, ..., <ЗначениеN>])
```

Пример использования метода __init__() приведен ниже:

```
# -*- coding: utf-8 -*-
class HexColor:
    def __init__(self):
        self.colors = {
            'red'   :   '#ff0000',
            'greed' :   '#7cfc00',
            'blue'  :   '#4169e1'
        }

hex_color = HexColor()
print hex_color.colors['red']   # Получим: #ff0000


# Пример 2: Удваиваем число
class DoubleMe:
    def __init__(self, number):
        self.result = number * 2

Double = DoubleMe(2)
print Double.result             # Получим 4
```

Если конструктор вызывается при создании объекта, то перед уничтожением объекта автоматически вызывается метод, называемый деструктором т.е. "уничтожитель". В языке программирования Python деструктор

реализуется в виде предопределенного метода __del__(). Следует заметить, что метод не будет вызван, если на экземпляр класса существует хотя бы одна ссылка. Кроме того, т.к. интерпретатор самостоятельно заботится об удалении объектов, использование деструктора в языке программирования Python не имеет особого смысла.

Применение метода __del__():

```
class Class1:
    def __init__(self): # Конструктор класса
        print "Вызван метод __init__()"
    def __del__(self):  # Деструктор класса
        print "Вызван метод __del__()"
c1 = Class1()           # Выведет: Вызван метод __init__()
del c1                  # Выведет: Вызван метод __del__()
c2 = Class1()           # Выведет: Вызван метод __init__()
c3 = c2                 # Создаем ссылку на экземпляр класса
del c2                  # Ничего не выведет, т.к. существует ссылка
del c3                  # Выведет: Вызван метод __del__()
```

Объектно-ориентированный поход поможет вам быстрее понять принципы более сложного программирования не только на языке Python, но так же и на Java, C++. Если хорошо знать один язык программирования то любой другой язык будет в изучении более легким чем новичку.

1.5. Свойства класса - ООП Python

Классы нового стиля позволяют создать идентификатор, через который можно получить, изменить или удалить значение атрибута класса. Создается такой идентификатор с помощью функции property(), форма функции:

```
<Свойства> = property(<Чтение>[, <Запись>[, <Удаление>[, <Строка документирования>]]])
```

В первых трех параметрах указывается ссылка на соответствующий метод класса. При попытке получить значение будет вызван метод, указанный в первом параметре. При операции присваивания значения будет вызван метод, указанный во втором параметре. Этот метод должен принимать один параметр. В случае удаления атрибута вызывается метод, указанный в третьем параметре. Если в качестве какого-либо параметра задано значение None, то это означает, что соответствующий метод не поддерживается. Рассмотрим свойства класса на примере.

Свойства класса:

```
class Class1(object):
    def __init__(self, value):
        self.__var = value
    def getVar(self):        # Чтение
        return self.__var
    def setVar(self, value): # Запись
        self.__var = value
    def delVar(self):        # Удаление
        del self.__var
    v = property(getVar, setVar, delVar, "Строка документирования")
c1 = Class1(5)
c1.v = 35                    # Вызывается метод setVar()
```

```
print c1.v                    # Вызывается метод getVar()
del c1.v                      # Вызывается метод delVar()
```

В Python 2.6 были добавлены методы getter(), setter() и deleter(), позволяющие создавать свойства классов с помощью декораторов функций. Пример использования декораторов приведен ниже.

Методы getter(), setter() и deleter():

```
class Class1(object): # Работает, начиная с версии Python 2.6
    def __init__(self, value):
        self.__var = value
    @property
    def v(self):                          # Чтение
        return self.__var
    @v.setter
    def v(self, value):                   # Запись
        self.__var = value
    @v.deleter
    def v(self):                          # Удаление
        del self.__var
c1 = Class1(5)
c1.v = 35                                 # Запись
print c1.v                                # Чтение
del c1.v                                  # Удаление
```

1.6. Статические методы и методы класса

Внутри класса можно создать метод, который будет доступен без создания экземпляра класса. Для этого перед определением метода внутри класса следует указать декоратор @staticmethod. Вызов статического метода без создания экземпляра класса осуществляется следующим образом:

```
<Название класса>.<Название метода>(<Параметры>)
```

Кроме того, можно вызвать статический метод через экземпляр класса:

```
<Экземпляр класса>.<Название метода>(<Параметры>)
```

Пример использования статических методов приведен ниже:

```
class Class1(object):
    @staticmethod
    def sum1(x, y):               # Статический метод
        return x + y
    def sum2(self, x, y):         # Обычный метод в классе
        return x + y
    def sum3(self, x, y):
        return Class1.sum1(x, y) # Вызов из метода класса

print Class1.sum1(10, 20)         # Вызываем статический метод
c1 = Class1()
print c1.sum2(15, 6)              # Вызываем метод класса
print c1.sum1(50, 12)             # Вызываем статический метод
                                  # через экземпляр класса
print c1.sum3(23, 5)              # Вызываем статический метод
                                  # внутри класса
```

Обратите внимание на то, что в определении статического метода нет параметра self. Это означает, что внутри статического метода нет доступа к атрибутам и методам экземпляра класса. Методы класса создаются с помощью декоратора @classmethod. В качестве первого параметра в метод

класса передается ссылка на класс, а не на экземпляр класса. Вызов метода класса осуществляется следующим образом:

```
<Название класса>.<название метода>(<Параметры>)
```

Кроме того, можно вызывать метод класса через экземпляр класса:

```
<Экземпляр класса>.<Название метода>(<Параметры>)
```

Методы класса:

```
class Class1(object):
    @classmethod
    def test(cls, x): # Метод класса
        print cls, x
Class1.test(10)       # Вызываем метод через название класса
c1 = Class1()
c1.test(50)           # Вызываем метод класса через экземпляр
```

1.7. Наследование. Объектно-ориентированное программирование в Python

Наследование в Python является важным фактором для понимания принципа работы ООП. Предположим, у вас есть класс (Пример Class1). При помощи наследования мы можем создать новый класс (Например Class2), в котором будет доступ ко всем атрибутам и методам класса Class1, а также к некоторым атрибутам и методам.

Наследование:

```
# -*- coding: utf-8 -*-
class Class1:         # Базовый класс
    def f_func1(self):
        print "Метод f_func1() класса Class1"

    def f_func2(self):
        print "Метод f_func2() класса Class1"

class Class2(Class1): # Класс Class2 наследует класс Class1
    def f_func3(self):
        print "Метод f_func3() класса Class2"

c1 = Class2()         # Создаем экземпляр класса Class2
c1.f_func1()          # Выведет: Метод f_func1() класса Class1
c1.f_func2()          # Выведет: Метод f_func2() класса Class1
c1.f_func3()          # Выведет: Метод f_func3() класса Class2
```

Как видно из примера, класс Class1 указывается внутри круглых скобок в определение класса Class2. Таким образом, класс Class2 наследует все атрибуты и методы класса Class1. Класс Class1 вызовется базовым классом ими суперклассом, а класс Class2 – производным классом или подклассом.

Если имя метода в классе Class2 совпадает с именем метода класса Class1, то будет использоваться метод из класса Class2. Чтобы вызвать одноименный метод из базового класса, следует указать перед методом название базового класса. Кроме того, в первом параметре метода необходимо явно указать ссылку на экземпляр класса. Рассмотрим это на примере:

Переопределение метода:

```
class Class1:              # Базовый класс
```

```
    def __init__(self):
        print "Конструктор базового класса"
    def f_func1(self):
        print "Метод f_func1() класса Class1"

class Class2(Class1):       # Класс Class2 наследует класс Class1
    def __init__(self):
        print "Конструктор производного класса"
        Class1.__init__(self) # Вызываем конструктор базового класса
    def f_func1(self):
        print "Метод f_func1() класса Class2"
        Class1.f_func1(self)  # Вызываем метод базового класса

c1 = Class2()                 # Создаем экземпляр класса Class2
c1.f_func1()                  # Вызываем метод f_func1()
```

Результат работы:

```
Конструктор производного класса
Конструктор базового класса
Метод f_func1() класса Class2
Метод f_func1() класса Class1
```

1.8. Множественное наследование в Python

В определенных классах в круглых скобках можно указать сразу несколько базовых классов через запятую. В этом случае поиск идентификаторов производится вначале в производном классе, затем в базовом классе, расположенном первым в списке, далее просматриваются все базовые классы базового класса. Только после этого просматривается базовый класс, расположенный в списке правее, и все его базовые классы. Список базовых классов просматривается слева направо. Результатом поиска будет первый найденный идентификатор. Рассмотрим множественное наследование на примере.

Множественное наследование:

```
class Class1:         # Базовый класс для класса Class2
    def f_func1(self):
        print "Метод f_func1() класса Class1"

class Class2(Class1): # Класс Class2 наследует класс Class1
    def f_func2(self):
        print "Метод f_func2() класса Class2"

class Class3(Class1): # Класс Class3 наследует класс Class1
    def f_func1(self):
        print "Метод f_func1() класса Class3"
    def f_func2(self):
        print "Метод f_func2() класса Class3"
    def f_func3(self):
        print "Метод f_func3() класса Class3"
    def f_func4(self):
        print "Метод f_func4() класса Class3"

class Class4(Class2, Class3): # Множественное наследование
    def f_func4(self):
        print "Метод f_func4() класса Class4"

c1 = Class4()             # Создаем экземпляр класса Class4
```

```
c1.f_func1()              # Выведет: Метод f_func1() класса Class1
c1.f_func2()              # Выведет: Метод f_func2() класса Class2
c1.f_func3()              # Выведет: Метод f_func3() класса Class3
c1.f_func4()              # Выведет: Метод f_func4() класса Class4
```

Итак, метод f_func1() определен в двух классах - Class1 и Class3. Так как класс Class2 стоит первым в списке базовых классов, вначале просматривается этот класс, а затем все его базовые классы. Поэтому метод f_func1() будет найден в классе Class1, а не в классе Class3.

Метод f_func2() также определен в двух классах - Class2 и Class3. Так как класс Class2 стоит первым в списке базовых классов, то метод будет найден именно в этом классе. Чтобы наследовать метод из класса Class3, следует указать это явным образом. Передаем определение класса Class4 из предыдущего примера и наследуем метод f_func2() из класса Class3.

Указание класса при наследовании метода:

```
class Class4(Class2, Class3): # Множественное наследование
    # Наследуем f_func2() из класса Class3, а не из класса Class2
    f_func2 = Class3.f_func2
    def f_func4(self):
        print "Метод f_func4() класса Class4"
```

Метод f_func3() определен только в классе Class3, поэтому метод наследуется от этого класса. Метод f_func4(), определенный в классе Class3, переопределяется в производном классе. Если метод найден в производном классе, то вся иерархия наследования просматривается не будет.

Если необходимо получить перечень базовых классов, то можно воспользоваться атрибутом __base__. В качестве значения атрибут возвращает кортеж.

1.9. Абстрактные методы в Python

Абстрактные методы содержат только определение метода без реализации. Предполагается, что класс-потомок должен переопределить метод и реализовать его функциональность. Чтобы такое предположение сделать более очевидным, часто внутри абстрактного метода возбуждают исключение.

Абстрактные методы:

```
class Class1(object):
    def test(self, x):     # Абстрактный метод
        # Возбуждаем исключение с помощью raise
        raise NotImplementedError("Необходимо переопределить метод")
class Class2(Class1):      # Наследуем абстрактный метод
    def test(self, x):     # Переопределяем метод
        print x
class Class3(Class1):      # Класс не переопределяет метод
    pass
c2 = Class2()
c2.test(50)                # Выведет: 50
c3 = Class3()
try:                       # Перехватываем исключения
    c3.test(50)            # Ошибка. Метод test() не переопределен
except NotImplementedError, msg:
    print msg              # Выведет: Необходимо переопределить метод
```

Начиная с версии Python 2.6, в состав библиотеки входит модуль abc. В этом модуле определен декоратор @abstractmethod, который позволяет указать, что метод, перед которым расположен декоратор, является абстрактным. При попытке создать экземпляр класса-потомка, в котором не переопределен абстрактный метод, возбуждается исключение TypeErrror. Рассмотрим использование декоратора @abstractmethod на примере.

Использование декоратора @abstractmethod:

```
from abc import ABCMeta, abstractmethod
class Class1(object):
    __metaclass__ = ABCMeta
    @abstractmethod
    def test(self, x):      # Абстрактный метод
        pass
class Class2(Class1):       # Наследуем абстрактный метод
    def test(self, x):      # Переопределяем метод
        print x
class Class3(Class1):       # Класс не переопределяет метод
    pass
c2 = Class2()
c2.test(50)                 # Выведет: 50
try:
    c3 = Class3()           # Ошибка. Метод test() не переопределен
    c3.test(50)
except TypeError, msg:
    print msg               # Can't instantiate abstract class Class3
                            # with abstract methods test
```

Раздел 2. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

При создании программных систем перед разработчиками часто встает проблема выбора тех или иных проектных решений. В этих случаях на помощь приходят паттерны. Дело в том, что почти наверняка подобные задачи уже решались ранее и уже существуют хорошо продуманные элегантные решения, составленные экспертами. Если эти решения описать и систематизировать в каталоги, то они станут доступными менее опытным разработчикам, которые после изучения смогут использовать их как шаблоны или образцы для решения задач подобного класса. Паттерны как раз описывают решения таких повторяющихся задач.

2.1. Классификация паттернов

В силу популярности каталога GoF часто под паттернами проектирования подразумевают все виды паттернов программной индустрии, что является не совсем корректным. В области разработки программных систем существует множество паттернов, которые отличаются областью применения, масштабом, содержимым, стилем описания. Например, в зависимости от сферы применения существуют такие паттерны как паттерны анализа, проектирования, тестирования, документирования, организации процесса разработки, планирования проектов и другие.

В настоящее время наиболее популярными паттернами являются паттерны проектирования. Одной из распространенных классификаций таких паттернов является классификация по степени детализации и уровню абстракции рассматриваемых систем. Согласно [2], паттерны проектирования программных систем делятся на следующие категории:

- архитектурные паттерны;
- паттерны проектирования;
- идиомы.

Архитектурные паттерны, являясь наиболее высокоуровневыми паттернами, описывают структурную схему программной системы в целом. В данной схеме указываются отдельные функциональные составляющие системы, называемые подсистемами, а также взаимоотношения между ними. Примером архитектурного паттерна является хорошо известная программная парадигма "модель-представление-контроллер" (model-view-controller – MVC).

В свою очередь, подсистемы могут состоять из архитектурных единиц уровнем ниже. **Паттерны проектирования** описывают схемы детализации программных подсистем и отношений между ними, при этом они не влияют на структуру программной системы в целом и сохраняют независимость от реализации языка программирования. Паттерны GoF относятся именно к этой категории. Согласно [5], под паттернами проектирования

объектно-ориентированных систем понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

В русскоязычной литературе обычно встречаются несколько вариантов перевода оригинального названия design patterns – **паттерны проектирования**, **шаблоны проектирования**, **образцы**. Здесь в основном используется первый вариант, иногда второй.

Идиомы, являясь низкоуровневыми паттернами, имеют дело с вопросами реализации какой-либо проблемы с учетом особенностей данного языка программирования. При этом часто одни и те же идиомы для разных языков программирования выглядят по-разному или не имеют смысла вовсе. Например, в C++ для устранения возможных утечек памяти могут использоваться интеллектуальные указатели. Интеллектуальный указатель содержит указатель на участок динамически выделенной памяти, который будет автоматически освобожден при выходе из зоны видимости. В среде Java такой проблемы просто не существует, так как там используется автоматическая сборка мусора. Обычно, для использования идиом нужно глубоко знать особенности применяемого языка программирования.

Следует отметить, что в программной области существуют и другие виды паттернов, не относящиеся к проектированию вообще, например, паттерны анализа, тестирования, документирования и др.

2.2. Описание паттернов

Задача каждого паттерна – дать четкое описание проблемы и ее решения в соответствующей области. Для этого могут использоваться разные форматы описаний от художественно-описательного до строгого, академического. В общем случае описание паттерна всегда содержит следующие элементы:

1. Название паттерна. Представляет собой уникальное смысловое имя, однозначно определяющее данную задачу или проблему и ее решение.

2. Решаемая задача. Здесь дается понимание того, почему решаемая проблема действительно является таковой, четко описывает ее границы.

3. Решение. Здесь указывается, как именно данное решение связано с проблемой, приводится пути ее решения.

4. Результаты использования паттерна. Обычно приводятся достоинства, недостатки и компромиссы.

2.3. Результаты применения паттернов

Один из соавторов GoF, Джон Влиссидес приводит следующие преимущества применения паттернов проектирования:

1. Они (паттерны) позволяют суммировать опыт экспертов и сделать его доступным рядовым разработчикам.

2. Имена паттернов образуют своего рода словарь, который позволяет разработчикам лучше понимать друг друга.

3. Если в документации системы указано, какие паттерны в ней используются, это позволяет читателю быстрее понять систему.

4. Паттерны упрощают реструктуризацию системы независимо от того, использовались ли паттерны при ее проектировании.

Правильно выбранные паттерны проектирования позволяют сделать программную систему более гибкой, ее легче поддерживать и модифицировать, а код такой системы в большей степени соответствует концепции повторного использования.

Паттернами проектирования (Design Patterns) называют решения часто встречающихся проблем в области разработки программного обеспечения. Паттерны проектирования не являются готовыми решениями, которые можно трансформировать непосредственно в код, а представляют общее описание решения проблемы, которое можно использовать в различных ситуациях.

Существуют несколько типов паттернов проектирования, каждый из которых предназначен для решения своего круга задач:

– порождающие паттерны, предназначенные для создания новых объектов в системе:

– структурные паттерны, решающие задачи компоновки системы на основе классов и объектов;

– паттерны поведения, предназначенные для распределения обязанностей между объектами в системе.

В контексте данной методички будут рассмотрены в качестве примеров только некоторые паттерны из каждого подвида.

2.3. Диаграмма классов как средство отображения архитектуры объектно-ориентированной системы

Диаграммы классов – это только один аспект UML, но, вероятно, они чаще всего используются.

Классы – это главные составные элементы диаграмм классов. Класс представляется в виде прямоугольника с именем, как показано на рис. 1.

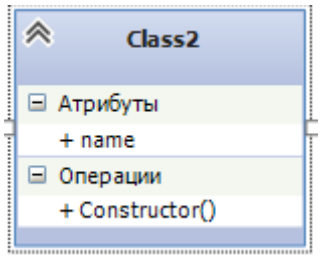


Рис. 1. Класс с именем Class2

Класс делится на три раздела, в первом из которых отображается имя. Разделительные линии необязательны, если, помимо имени класса, больше никакой дополнительной информации не предоставляется. Создавая диаграммы, вы увидите, что для некоторых классов вполне достаточно того уровня подробностей, который представлен на рис. 1. Иногда вовсе не обязательно представлять все поля и методы, и даже все классы, на диаграмме классов.

Абстрактные классы представляют, либо выделяя имя класса курсивом, как показано на рис. 2, либо добавляя к имени класса уточнение {abstract}.

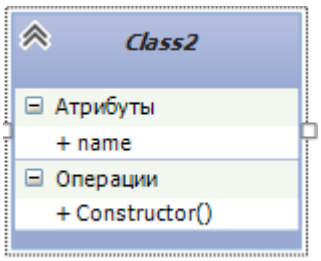


Рис. 2. Абстрактный класс

Интерфейсы определяются таким же образом, как классы, за исключением того, что они должны включать стереотип (т.е. расширение словаря UML), как показано на рис. 3. Интерфейсы реализованы в таких языках программирования как java, C#, PHP. В Языке программирования Python, C++, вместо интерфейсов реализованы абстрактные классы и поддерживается множественное наследование.

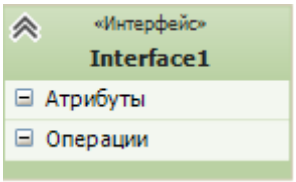


Рис. 3. Интерфейс в нотации UML

Атрибуты.

Если говорить в общем, то атрибуты описывают свойства класса. Атрибуты перечисляются в разделе, который расположен непосредственно под именем класса, как показано на рис. 4.

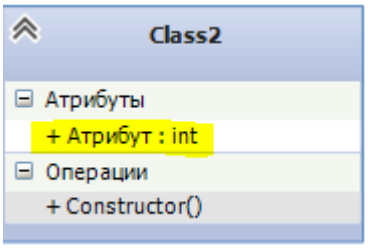


Рис. 4. Атрибут класса в нотации UML

Давайте рассмотрим атрибут из этого примера повнимательнее. Первый символ (+) обозначает уровень видимости, или контроль доступа, для атрибута. В табл. 1 перечислены три возможных варианта символа видимости.

Таблица 1 – Символы видимости

Символ	Видимость	Описание
+	Общедоступный	Доступный для всего кода
-	Закрытый	Доступный только для текущего класса
#	Защищенный	Доступный только для текущего класса и его подклассов

За символом видимости следует имя атрибута. В данном случае мы описываем свойство Атрибут. Двоеточие используется для отделения имени атрибута от его типа (и, возможно, от его стандартного значения).

Можно включать ровно столько деталей, сколько это необходимо для ясности.

Операции.

Операции описывают методы или, точнее, вызовы, которые могут быть сделаны к экземпляру класса. На рис. 5 показаны две операции класса ShopProduct.

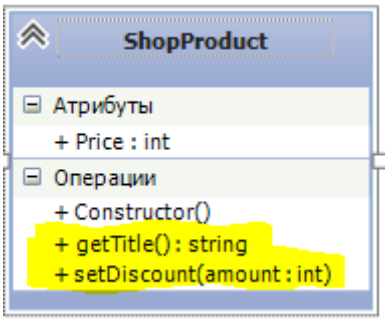


Рис. 5. Операции класса в нотации UML

Для операций используется такой же синтаксис, как и для атрибутов. Символ видимости предшествует имени метода. Список параметров заключается в круглые скобки. Тип, возвращаемый методом, если он есть, отделен двоеточием. Параметры разделяются запятыми, и для них исполь-

зуется такой же синтаксис, как и для атрибута: имя атрибута отделяется от его типа двоеточием.

Как и можно было ожидать, это достаточно гибкий синтаксис. Вы можете опустить признак видимости и возвращаемый тип. Параметры часто представляются только их типом, поскольку имя аргумента обычно не имеет особого значения.

Описание наследования и реализации.

В UML отношения наследования описываются в виде обобщений. Это отношение обозначается линией, ведущей от подкласса к родительскому классу. Эта линия заканчивается не закрашенной замкнутой стрелкой.

На рис. 6 показана связь между классом ShopProduct и его дочерними классами.

С помощью UML описывается также связь между интерфейсом и классами, которые его реализуют. Так, если бы класс ShopProduct реализовывал интерфейс Chargeable, мы бы добавили его к диаграмме класса, как показано на рис. 6.

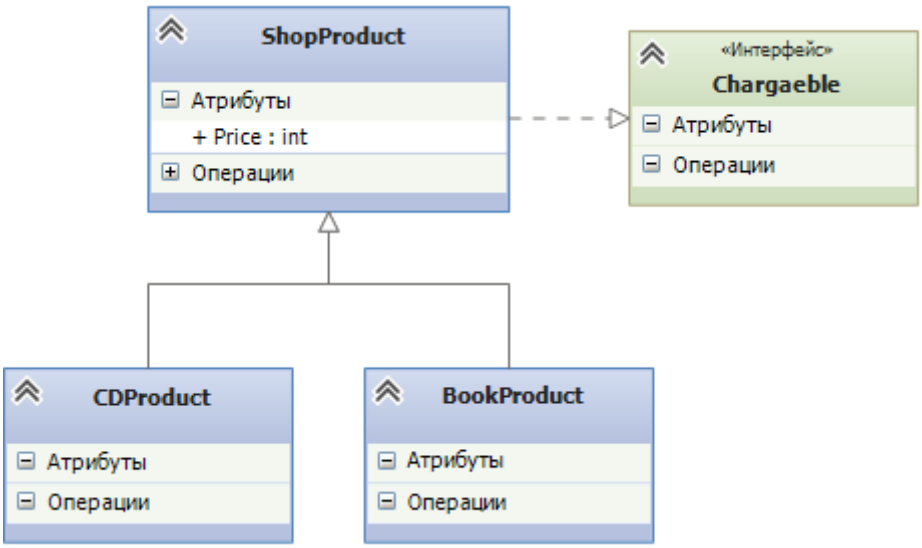


Рис. 6. Операции класса в нотации UML

Ассоциации.

Наследование – это только одно из отношений в объектно-ориентированной системе. Ассоциация происходит, когда объявляется, что свойство класса содержит ссылку на экземпляр (или экземпляры) другого класса.

Стрелки используются для описания направления ассоциации. Если в классе **Корзина** существует ссылка на экземпляр класса **Товар**, но не наоборот, то мы должны определить ассоциацию с помощью стрелки, направленной от класса **Корзина** к классу **Товар**. Эта ассоциация, которая называется однонаправленной, показана на рис. 7.

Рис. 7. Отношение ассоциации в нотации UML

Если в каждом классе существует ссылка на другой класс, то для описания двунаправленного отношения используется двунаправленная стрелка.

Можно указать также количество экземпляров класса, на которые ссылается другой класс в ассоциации. Для этого нужно поместить число или интервал чисел рядом с каждым классом. Можно также использовать звездочку (*), обозначающую любое число. На рис. 8 показано, что может существовать только один объект **Корзина** и нуль или больше объектов **Товар**.

Рис. 7. Кратность отношения асоциации в нотации UML

Агрегирование и композиция.

Агрегирование и композиция похожи на ассоциацию. Все эти термины служат для описания ситуации, когда в классе содержится постоянная ссылка на один или более экземпляров другого класса. Но с помощью агрегирования и композиции экземпляры, на которые ссылаются, формируют внутреннюю часть ссылающегося объекта.

В случае агрегирования, содержащиеся объекты составляют основную часть объекта-контейнера (который их содержит), но они могут также одновременно содержаться и в других объектах. Отношение агрегирования обозначается линией, которая начинается с не закрашенного ромба.

Композиция представляет собой еще более сильное отношение. В композиции на содержащийся объект может ссылаться только его объект-контейнер. И он должен быть удален при удалении объекта-контейнера. Отношения композиции изображаются так же, как и отношения агрегации, только с закрашенным ромбом.

Описание отношений использования.

В UML отношение использования описывается в виде зависимости. Это самое неустойчивое из всех отношений, рассматриваемых в данном разделе, потому что оно не описывает постоянную связь между классами.

Используемый класс может передаваться в качестве аргумента или быть получен в результате вызова метода.

Класс Report на рис. 8 использует объект ShopProductWriter. Отношение использования изображается с помощью пунктирной линии со стрелкой с незамкнутым контуром на конце; эта линия соединяет рассматриваемые класс и объект. Тем не менее при этом отношении ссылка на объект не хранится в виде свойства, как, например, в объекте ShopProductWriter, где хранится массив ссылок на объекты типа ShopProduct.

Рис 8. Диаграмма классов

Использование примечаний.

На диаграммах классов отображается структура системы, но они не дают представления о самом процессе.

Примечание представляет собой прямоугольник с загнутым уголком. Обычно в нем содержатся фрагменты псевдокода.

В некоторых случаях даже примечание может не дать достаточной информации. Но, к счастью, мы можем моделировать взаимодействие объектов в нашей системе, а также структуру классов.

2.4. Порождающие паттерны

Порождающие паттерны проектирования предназначены для создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов.

К порождающим паттернам относятся:

Паттерн **Factory Method**. В его классическом варианте вводится полиморфный класс Factory, в котором определяется интерфейс фабричного метода, создание экземпляра, а ответственность за создание объектов кон-

кретных классов переносится на производные от Factory классы, в которых этот метод переопределяется.

Паттерн **Abstract Factory** использует несколько фабричных методов и предназначен для создания целого семейства или группы взаимосвязанных объектов.

Паттерн **Builder** определяет процесс поэтапного конструирования сложного объекта, в результате которого могут получаться разные представления этого объекта.

Паттерн **Prototype** создает новые объекты с помощью прототипов. Прототип – некоторый объект, умеющий создавать по запросу копию самого себя.

Паттерн **Singleton** контролирует создание единственного экземпляра некоторого класса и предоставляет доступ к нему.

Паттерн **Object Pool** используется в случае, когда создание объекта требует больших затрат или может быть создано только ограниченное количество объектов некоторого класса.

2.4.1. Порождающий паттерн Singleton

Назначение паттерна Singleton

Часто в системе могут существовать сущности только в единственном экземпляре, например, система ведения системного журнала сообщений или драйвер дисплея. В таких случаях необходимо уметь создавать единственный экземпляр некоторого типа, предоставлять к нему доступ извне и запрещать создание нескольких экземпляров того же типа.

Описание паттерна Singleton

Архитектура паттерна Singleton основана на идее использования глобальной переменной, имеющей следующие важные свойства:

1. Такая переменная доступна всегда. Время жизни глобальной переменной - от запуска программы до ее завершения.

2. Предоставляет глобальный доступ, то есть, такая переменная может быть доступна из любой части программы.

Для решения этой проблемы паттерн Singleton возлагает контроль над созданием единственного объекта на сам класс. Доступ к этому объекту осуществляется через статическую функцию-член класса, которая возвращает указатель или ссылку на него. Этот объект будет создан только при первом обращении к методу, а все последующие вызовы просто возвращают его адрес.

UML-диаграмма классов паттерна Singleton

Пример:

```
# coding: utf-8
"""
Одиночка (Singleton) - паттерн, порождающий объекты.
Гарантирует, что у класса есть только один экземпляр, и предоставляет к не-
му глобальную точку доступа.
С помощью паттерна одиночка могут быть реализованы многие паттерны (абст-
рактная фабрика, строитель, прототип).
"""
class SingletonMeta(type):
    def __init__(cls, *args, **kwargs):
        cls._instance = None
        # глобальная точка доступа `Singleton.get_instance()`
        cls.get_instance = classmethod(lambda c: c._instance)
        super(SingletonMeta, cls).__init__(*args, **kwargs)

    def __call__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(SingletonMeta,
                                  cls).__call__(*args, **kwargs)
        return cls._instance

class Singleton(object):
    __metaclass__ = SingletonMeta

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

    def get_name(self):
        return self._name

obj1 = Singleton('MyInstance 1')
print obj1.get_name()  # MyInstance 1

obj2 = Singleton('MyInstance 2')
print obj2.get_name()  # MyInstance 1

print obj1 is obj2 is Singleton.get_instance()  # True
```

2.4.2. Порождающий паттерн Abstract Factory (абстрактная фабрика)

Назначение паттерна Abstract Factory.

Используйте паттерн Abstract Factory (абстрактная фабрика) если:

- система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов;

- необходимо создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.

Приведем примеры групп взаимосвязанных объектов.

Пусть некоторое приложение с поддержкой графического интерфейса пользователя рассчитано на использование на различных платформах, при этом внешний вид этого интерфейса должен соответствовать принятому стилю для той или иной платформы. Например, если это приложение уста-

новлено на Windows-платформу, то его кнопки, меню, полосы прокрутки должны отображаться в стиле, принятом для Windows. Группой взаимосвязанных объектов в этом случае будут элементы графического интерфейса пользователя для конкретной платформы.

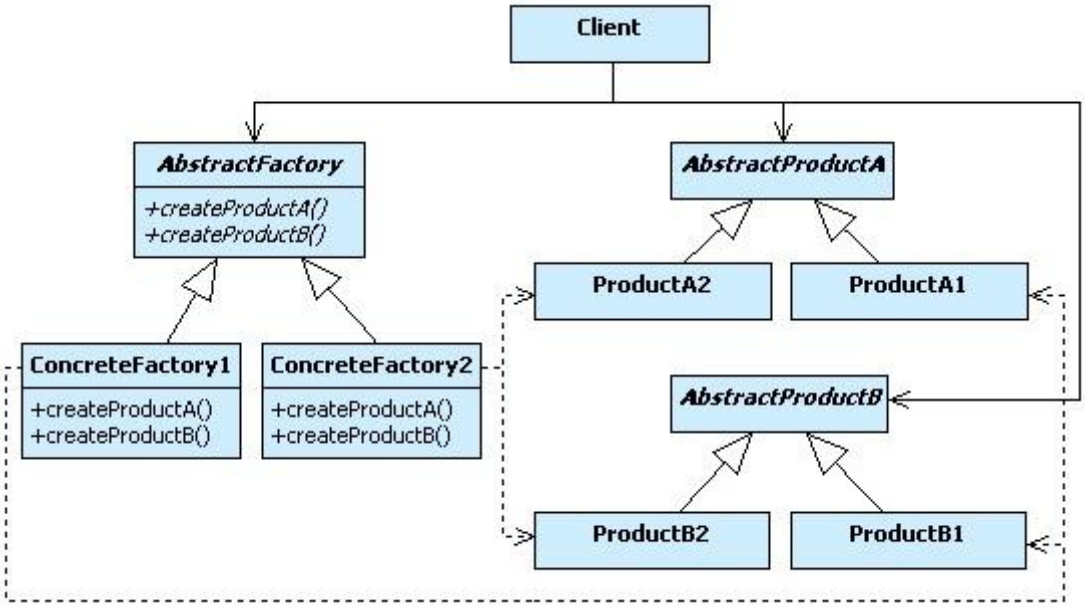
Описание паттерна Abstract Factory.

Любое семейство или группа взаимосвязанных объектов характеризуется несколькими общими типами создаваемых продуктов, при этом сами продукты таких типов будут различными для разных семейств. Например, для случая стратегической игры общими типами создаваемых продуктов будут пехота, лучники и конница, при этом каждый из этих родов войск римской армии может существенно отличаться по внешнему виду и боевым характеристикам от соответствующих родов войск армии Карфагена.

Для того чтобы система оставалась независимой от специфики того или иного семейства продуктов необходимо использовать общие интерфейсы для всех основных типов продуктов. В случае стратегической игры это означает, что необходимо использовать три абстрактных базовых класса для каждого типа воинов: пехоты, лучников и конницы. Производные от них классы будут реализовывать специфику соответствующего типа воинов той или иной армии.

Для решения задачи по созданию семейств взаимосвязанных объектов паттерн *Abstract Factory* вводит понятие абстрактной фабрики. Абстрактная фабрика представляет собой некоторый полиморфный базовый класс, назначением которого является объявление интерфейсов фабричных методов, служащих для создания продуктов всех основных типов (один фабричный метод на каждый тип продукта). Производные от него классы, реализующие эти интерфейсы, предназначены для создания продуктов всех типов внутри семейства или группы.

UML-диаграмма классов паттерна Abstract Factory



Реализация паттерна Abstract Factory:

```
# coding: utf-8

"""
Абстрактная фабрика (Abstract factory, Kit) - паттерн, порождающий объекты.
Предоставляет интерфейс для создания семейств взаимосвязанных или взаимоза-
висимых объектов,
не специфицируя их конкретных классов.
Классы абстрактной фабрики часто реализуются фабричными методами,
но могут быть реализованы и с помощью паттерна прототип.
"""

class AbstractFactory(object):
    def create_drink(self):
        raise NotImplementedError()

    def create_food(self):
        raise NotImplementedError()

class Drink(object):
    def __init__(self, name):
        self._name = name

    def __str__(self):
        return self._name

class Food(object):
    def __init__(self, name):
        self._name = name

    def __str__(self):
        return self._name

class ConcreteFactory1(AbstractFactory):
    def create_drink(self):
        return Drink('Coca-cola')

    def create_food(self):
        return Food('Hamburger')

class ConcreteFactory2(AbstractFactory):
    def create_drink(self):
        return Drink('Pepsi')

    def create_food(self):
        return Food('Cheeseburger')

def get_factory(ident):
    if ident == 0:
        return ConcreteFactory1()
    elif ident == 1:
        return ConcreteFactory2()

factory = get_factory(1)
print (factory.create_drink()) # Pepsi
print (factory.create_food()) # Cheeseburger
```

2.5. Структурные паттерны

Структурные паттерны решают задачи компоновки системы на основе классов и объектов.

При этом могут использоваться следующие механизмы:

- Наследование, когда базовый класс определяет интерфейс, а подклассы - реализацию. Структуры на основе наследования получаются статичными.
- Композиция, когда структуры строятся путем объединения объектов некоторых классов. Композиция позволяет получать структуры, которые можно изменять во время выполнения.

Рассмотрим несколько примеров реализации структурных паттернов.

2.5.1. Паттерн Decorator (декоратор, wrapper, обертка)

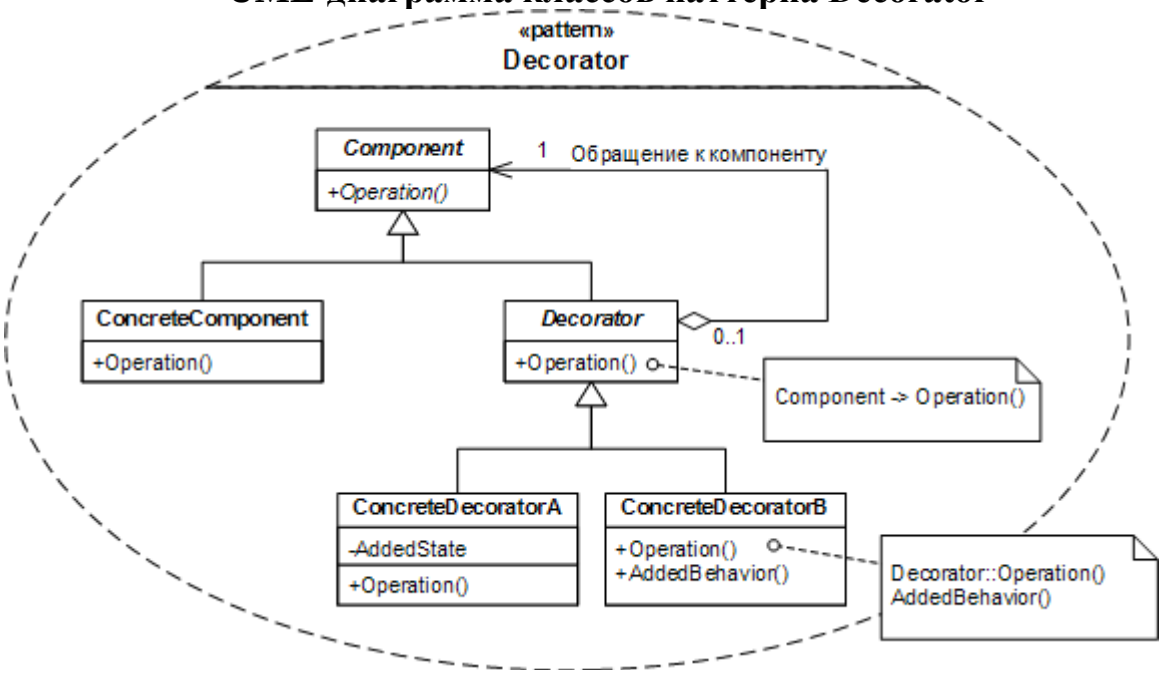
Назначение паттерна Decorator:

- паттерн Decorator динамически добавляет новые обязанности объекту. Декораторы являются гибкой альтернативой порождению подклассов для расширения функциональности;
- рекурсивно декорирует основной объект;
- паттерн Decorator использует схему "обертываем подарок, кладем его в коробку, обертываем коробку".

Решаемая проблема.

Вы хотите добавить новые обязанности в поведении или состоянии отдельных объектов во время выполнения программы. Использование наследования не представляется возможным, поскольку это решение статическое и распространяется целиком на весь класс.

UML-диаграмма классов паттерна Decorator



Пример реализации паттерна Decorator на python:

```
# coding: utf-8

"""
Декоратор (Decorator, Wrapper) - паттерн, структурирующий объекты.
Динамически добавляет объекту новые обязанности.
Является гибкой альтернативой порождению подклассов с целью расширения
функциональности.
"""


class Man(object):
    """Человек"""
    def __init__(self, name):
        self._name = name

    def say(self):
        print ('Привет! Меня зовут %s!' % self._name)


class Jetpack(object):
    """Реактивный ранец"""
    def __init__(self, man):
        self._man = man

    def __getattr__(self, item):
        return getattr(self._man, item)

    def fly(self):
        # расширяем функциональность объекта добавляя возможность летать
        print ('%s летит на реактивном ранце!' % self._man._name)


man = Man('Виктор')

man_jetpack = Jetpack(man)
man_jetpack.say()  # Привет! Меня зовут Виктор!
man_jetpack.fly()  # Виктор летит на реактивном ранце!
```

2.5.2. Паттерн Facade (фасад)

Назначение паттерна Façade.

Паттерн Facade предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Facade определяет интерфейс более высокого уровня, упрощающий использование подсистемы.

Паттерн Facade "обертывает" сложную подсистему более простым интерфейсом.

Решаемая проблема.

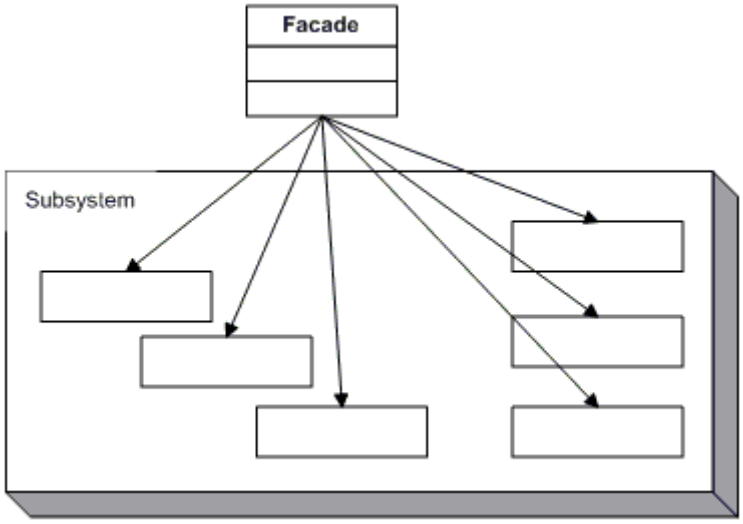
Клиенты хотят получить упрощенный интерфейс к общей функциональности сложной подсистемы.

Паттерн Facade инкапсулирует сложную подсистему в единственный интерфейсный объект. Это позволяет сократить время изучения подсистемы, а также способствует уменьшению степени связанности между подсистемой и потенциально большим количеством клиентов. С другой сто-

роны, если фасад является единственной точкой доступа к подсистеме, то он будет ограничивать возможности, которые могут понадобиться "продвинутым" пользователям.

Клиенты общаются с подсистемой через Facade. При получении запроса от клиента объект Facade переадресует его нужному компоненту подсистемы. Для клиентов компоненты подсистемы остаются "тайной, покрытой мраком".

UML-диаграмма классов паттерна Façade



Пример реализации паттерна Facade на python:

```
"""
Фасад (Facade) - паттерн, структурирующий объекты.
Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой
подсистемы.
Фасад определяет интерфейс более высокого уровня, который упрощает исполь-
зование подсистемы.
"""

class Paper(object):
    """Бумага"""
    def __init__(self, count):
        self._count = count

    def get_count(self):
        return self._count

    def draw(self, text):
        if self._count > 0:
            self._count -= 1
            print (text)

class Printer(object):
    """Принтер"""
    def error(self, msg):
        print ('Ошибка: %s' % msg)

    def print_(self, paper, text):
        if paper.get_count() > 0:
```

```
            paper.draw(text)
        else:
            self.error('Бумага закончилась')

class Facade(object):
    def __init__(self):
        self._printer = Printer()
        self._paper = Paper(1)

    def write(self, text):
        self._printer.print_(self._paper, text)

f = Facade()
f.write('Hello world!')  # Hello world!
f.write('Hello world!')  # Ошибка: Бумага закончилась
```

2.6. Паттерны поведения

Паттерны поведения рассматривают вопросы о связях между объектами и распределением обязанностей между ними. Для этого могут использоваться механизмы, основанные как на наследовании, так и на композиции.

2.6.1. Паттерн Observer

Паттерн **Observer** Определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.

Назначение паттерна Observer:

- паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически;
- паттерн Observer инкапсулирует главный (независимый) компонент в абстракцию Subject и изменяемые (зависимые) компоненты в иерархию Observer;
- паттерн Observer определяет часть "View" в модели Model-View-Controller (MVC) .

Паттерн Observer находит широкое применение в системах пользовательского интерфейса, в которых данные и их представления ("виды") отделены друг от друга. При изменении данных должны быть изменены все представления этих данных (например, в виде таблицы, графика и диаграммы).

Решаемая проблема.

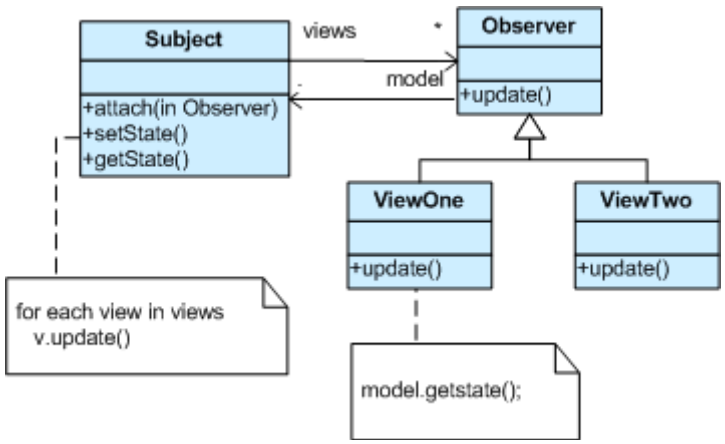
Имеется система, состоящая из множества взаимодействующих классов. При этом взаимодействующие объекты должны находиться в согласованных состояниях. Вы хотите избежать монолитности такой системы, сделав классы слабо связанными (или повторно используемыми).

Структура паттерна Observer.

Subject представляет главную (независимую) абстракцию. Observer представляет изменяемую (зависимую) абстракцию. Субъект извещает на-

блюдателей о своем изменении, на что каждый наблюдатель может запросить состояние субъекта.

UML-диаграмма классов паттерна Observer



Пример реализации паттерна Observer на python:

```
"""
Наблюдатель (Observer, Dependents, Publish-Subscribe) - паттерн поведения
объектов.
Определяет зависимость типа "один ко многим" между объектами таким образом,
что при изменении состояния одного объекта все зависящие от него оповещают-
ся об этом
и автоматически обновляются.
"""

class Subject(object):
    """Субъект"""
    def __init__(self):
        self._data = None
        self._observers = set()

    def attach(self, observer):
        # подписаться на оповещение
        if not isinstance(observer, ObserverBase):
            raise TypeError()
        self._observers.add(observer)

    def detach(self, observer):
        # отписаться от оповещения
        self._observers.remove(observer)

    def get_data(self):
        return self._data

    def set_data(self, data):
        self._data = data
        self.notify(data)

    def notify(self, data):
        # уведомить всех наблюдателей о событии
        for observer in self._observers:
            observer.update(data)
```

```
class ObserverBase(object):
    """Абстрактный наблюдатель"""
    def update(self, data):
        raise NotImplementedError()
class Observer(ObserverBase):
    """Наблюдатель"""
    def __init__(self, name):
        self._name = name

    def update(self, data):
        print ('%s: %s' % (self._name, data))

subject = Subject()
subject.attach(Observer('Наблюдатель 1'))
subject.attach(Observer('Наблюдатель 2'))
subject.set_data('данные для наблюдателя')
# Наблюдатель 2: данные для наблюдателя
# Наблюдатель 1: данные для наблюдателя
```

2.6.2. Паттерн Command (команда)

Используйте паттерн Command, если:

- система управляется событиями. При появлении такого события (запроса) необходимо выполнить определенную последовательность действий;
- необходимо параметризировать объекты выполняемым действием, ставить запросы в очередь или поддерживать операции отмены (undo) и повтора (redo) действий;
- нужен объектно-ориентированный аналог функции обратного вызова в процедурном программировании.

Пример событийно-управляемой системы – приложение с пользовательским интерфейсом. При выборе некоторого пункта меню пользователем вырабатывается запрос на выполнение определенного действия (например, открытия файла).

Описание паттерна Command.

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

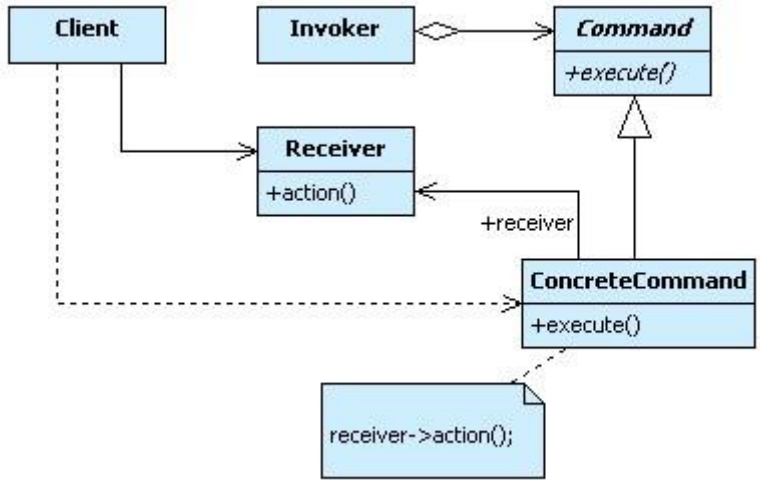
Интерфейс командного объекта определяется абстрактным базовым классом **Command** и в самом простом случае имеет единственный метод **execute().** Производные классы определяют получателя запроса (указатель на объект-получатель) и необходимую для выполнения операцию (метод этого объекта). Метод **execute()** подклассов **Command** просто вызывает нужную операцию получателя.

В паттерне Command может быть до трех участников:

- клиент, создающий экземпляр командного объекта;

- инициатор запроса, использующий командный объект;
- получатель запроса.

UML-диаграмма классов паттерна Command



Сначала клиент создает объект **ConcreteCommand**, конфигурируя его получателем запроса. Этот объект также доступен инициатору. Инициатор использует его при отправке запроса, вызывая метод **execute().** Этот алгоритм напоминает работу функции обратного вызова в процедурном программировании – функция регистрируется, чтобы быть вызванной позднее.

Паттерн Command отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Пример реализации паттерна Command на python:

```
"""
Команда (Command, Action, Transaction) - паттерн поведения объектов.
Инкапсулирует запрос как объект, позволяя тем самым задавать параметры кли-
ентов
для обработки соответствующих запросов, ставить запросы в очередь или про-
токолировать их,
а также поддерживать отмену операций.
"""


class Light(object):
    def turn_on(self):
        print ('Включить свет')

    def turn_off(self):
        print ('Выключить свет')


class CommandBase(object):
    def execute(self):
```

```
        raise NotImplementedError()


class LightCommandBase(CommandBase):
    def __init__(self, light):
        self.light = light


class TurnOnLightCommand(LightCommandBase):
    def execute(self):
        self.light.turn_on()


class TurnOffLightCommand(LightCommandBase):
    def execute(self):
        self.light.turn_off()


class Switch(object):
    def __init__(self, on_cmd, off_cmd):
        self.on_cmd = on_cmd
        self.off_cmd = off_cmd

    def on(self):
        self.on_cmd.execute()

    def off(self):
        self.off_cmd.execute()


light = Light()
switch = Switch(on_cmd=TurnOnLightCommand(light),
                off_cmd=TurnOffLightCommand(light))
switch.on() # Включить свет
switch.off() # Выключить свет
```

Раздел 3. РАБОТА С СЕТЬЮ

3.1. Работа с сокетами

Применяемая в IP-сетях архитектура клиент-сервер использует IP-пакеты для коммуникации между клиентом и сервером. Клиент отправляет запрос серверу, на который тот отвечает. В случае с TCP/IP между клиентом и сервером устанавливается соединение (обычно с двусторонней передачей данных), а в случае с UDP/IP - клиент и сервер обмениваются пакетами (дейтаграммамми) с негарантированной доставкой.

Каждый сетевой интерфейс IP-сети имеет уникальный в этой сети адрес (**IP-адрес**). Упрощенно можно считать, что каждый компьютер в сети Интернет имеет собственный IP-адрес. При этом в рамках одного сетевого интерфейса может быть несколько сетевых **портов**. Для установления сетевого соединения приложение клиента должно выбрать свободный порт и установить соединение с серверным приложением, которое слушает (listen) порт с определенным номером на удаленном сетевом интерфейсе. Пара IP-адрес и порт характеризуют **сокет** (гнездо) – начальную (конечную) точку сетевой коммуникации. Для создания соединения TCP/IP необходимо два сокета: один на локальной машине, а другой – на удаленной. Таким образом, каждое сетевое соединение имеет IP-адрес и порт на локальной машине, а также IP- адрес и порт на удаленной машине.

Модуль **socket** обеспечивает возможность работать с сокетами из Python. Сокеты используют транспортный уровень согласно семиуровневой модели OSI (Open Systems Interconnection, взаимодействие открытых систем), то есть относятся к более низкому уровню, чем большинство описываемых в этом разделе протоколов.

Уровни модели OSI:

Физический

Поток битов, передаваемых по физической линии. Определяет параметры физической линии.

Канальный (Ethernet, PPP, ATM и т.п.)

Кодирует и декодирует данные в виде потока битов, справляясь с ошибками, возникающими на физическом уровне в пределах физически единой сети.

Сетевой (IP)

Маршрутизирует информационные пакеты от узла к узлу.

Транспортный (TCP, UDP и т.п.)

Обеспечивает прозрачную передачу данных между двумя точками соединения.

Управляет сеансом соединения между участниками сети. Начинает, координирует и завершает соединения.

Представления

Обеспечивает независимость данных от формы их представления путем преобразования форматов. На этом уровне может выполняться прозрачное (с точки зрения вышележащего уровня) шифрование и дешифрование данных.

Приложений (HTTP, FTP, SMTP, NNTP, POP3, IMAP и т.д.)

Поддерживает конкретные сетевые приложения. Протокол зависит от типа сервиса.

Каждый сокет относится к одному из коммуникационных доменов. Модуль **socket** поддерживает домены UNIX и Internet. Каждый домен подразумевает свое семейство протоколов и адресацию. Данное изложение будет затрагивать только домен Internet, а именно протоколы TCP/IP и UDP/IP, поэтому для указания коммуникационного домена при создании сокета будет указываться константа *socket.AF_iNET*.

В качестве примера следует рассмотреть простейшую клиент-серверную пару. Сервер будет принимать строку и отвечать клиенту. Сетевое устройство иногда называют хостом (host), поэтому будет употребляться этот термин по отношению к компьютеру, на котором работает сетевое приложение.

Сервер:

```
#Модуль socket для сетевого программирования
from socket import *

#данные сервера
host = 'localhost'
port = 777
addr = (host,port)

#socket - функция создания сокета
#первый параметр socket_family может быть AF_INET или AF_UNIX
#второй параметр socket_type может быть SOCK_STREAM(для TCP) или
SOCK_DGRAM(для UDP)
tcp_socket = socket(AF_INET, SOCK_STREAM)
#bind - связывает адрес и порт с сокетом
tcp_socket.bind(addr)
#listen - запускает прием TCP
tcp_socket.listen(1)

#Бесконечный цикл работы программы
while True:

    #Если мы захотели выйти из программы
    question = input('Do you want to quit? y\\n: ')
    if question == 'y': break

    print('wait connection...')

    #accept - принимает запрос и устанавливает соединение, (по умолчанию
работает в блокирующем режиме)
    #устанавливает новый сокет соединения в переменную conn и адрес клиента
в переменную addr
    conn, addr = tcp_socket.accept()
```

```
    print('client addr: ', addr)
    #recv - получает сообщение TCP
    data = conn.recv(1024)
    #если ничего не прислали, завершим программу
    if not data:
        conn.close()
        break
    else:
        print(data)
        #send - передает сообщение TCP
        conn.send(b'Hello from server!')
        #close - закрывает сокет
        conn.close()

tcp_socket.close()
```

Клиент:

```
from socket import *
import sys

host = 'localhost'
port = 777
addr = (host,port)

tcp_socket = socket(AF_INET, SOCK_STREAM)
tcp_socket.connect(addr)


data = input('write to server: ')
if not data :
    tcp_socket.close()
    sys.exit(1)

#encode - перекодирует введенные данные в байты, decode - обратно
data = str.encode(data)
tcp_socket.send(data)
data = bytes.decode(data)
data = tcp_socket.recv(1024)
print(data)


tcp_socket.close()
```

Примечание:

В примере использованы русские буквы: необходимо указывать кодировку.

Прежде всего, нужно запустить сервер. Сервер открывает сокет на локальной машине на порту 777, и адресе 127.0.0.1. После этого он слушает **listen()** порт. Когда на порту появляются данные, принимается (**accept()**) входящее соединение. Метод **accept()** возвращает пару - Socket-объект и адрес удаленного компьютера, устанавливающего соединение (пара - IP-адрес, порт на удаленной машине). После этого можно применять методы **recv()** и **send ()** для общения с клиентом. В **recv()** задается число байтов в очередной порции. От клиента может прийти и меньшее количество данных.

Код программы-клиента достаточно очевиден. Метод **connect()** устанавливает соединение с удаленным хостом (в приведенном примере он рас-

положен на той же машине). Данные передаются методом **send()** и принимаются методом **recv()** - аналогично тому, что происходит на сервере.

Модуль **socket** имеет несколько вспомогательных функций. В частности, функции для работы с системой доменных имен (**DNS**):

```
>>> import socket
>>> socket.gethostbyname('www.onego.ru')
('www.onego.ru', [],   ['195.161.136.4'])
>>> socket.gethostbyaddr('195.161.136.4')
('www.onego.ru', [],   ['195.161.136.4'])
>>> socket.gethostname()
'rnd.onego.ru'
```

В новых версиях Python появилась такая функция как *socket.getservbyname()*. Она позволяет преобразовывать наименования Интернет-сервисов в общепринятые номера портов:

```
>>> for srv in 'http', 'ftp', 'imap', 'pop3', 'smtp':
... print socket.getservbyname(srv, 'tcp'), srv
   80
   http
   21
   ftp
   143
   imap
   110
   pop3
   25
   smtp
```

Модуль также содержит большое количество констант для указания протоколов, типов сокетов, коммуникационных доменов и т.п. Другие функции модуля **socket** можно при необходимости изучить по документации.

3.2. Модуль smtplib

Сообщения электронной почты в Интернете передаются от клиента к серверу и между серверами в основном по протоколу **SMTP** (**S** imple **M** ail **T** ransfer **P** rotocol, простой протокол передачи почты). Протокол SMTP и ESMTP (расширенный вариант SMTP) описаны в RFC 821 и RFC 1869. Для работы с SMTP в стандартной библиотеке модулей имеется модуль **smtplib**. Для того чтобы начать SMTP-соединение с сервером электронной почты, необходимо в начале создать объект для управления SMTP-сессией с помощью конструктора класса **SMTP**:

```
smtplib.SMTP([host[, port]])
```

Параметры *host* и *port* задают адрес и порт SMTP-сервера, через который будет отправляться почта. По умолчанию, *port=25*. Если *host* задан, конструктор сам установит соединение, иначе придется отдельно вызывать метод *connect()*. Экземпляры класса *smtp* имеют методы для всех распространенных команд SMTP- протокола, но для отправки почты достаточно вызова конструктора и методов *sendmail()* и *quit()*:

```
from smtplib import SMTP
fromaddr = "student@mail.ru"   # От кого
toaddr = "rnd@onego.ru"    # Кому
message = """From: Student <%(fromaddr)s>
```

```
To: Lecturer <%(toaddr)s>
Subject: From Python course student MIME-Version: 1.0
Content-Type: text/plain; charset=Windows-1251 Content-Transfer-Encoding:
8bit
Здравствуйте! Я изучаю курс по языку Python и отправляю письмо его автору.
"""
connect = SMTP('mail.onego.ru')
connect.set_debuglevel(1)
connect.sendmail(fromaddr, toaddr, message % vars())
connect.quit()
```

Следует заметить, что *toaddr* в сообщении (в поле *To*) и при отправке могут не совпадать. Дело в том, что получатель и отправитель в ходе SMTP-сессии передается командами SMTP-протокола. При запуске указанного выше примера на экране появится отладочная информация (ведь уровень отладки задан равным 1):

```
send:   'ehlo   rnd.onego.ru\r\n'
reply:  '250-mail.onego.ru Hello as3-042.dialup.onego.ru [195.161.147.4],
pleased to meet you\r\n'
send:   'mail   FROM:<student@mail.ru> size=270\r\n'
reply:  '250    2.1.0 <student@mail.ru>... Sender ok\r\n'
send:   'rcpt   TO:<rnd@onego.ru>\r\n'
reply:  '250    2.1.5 <rnd@onego.ru>... Recipient ok\r\n'
send:   'data\r\n'
reply:  '354    Enter mail, end with "." on a line by itself\r\n'
send:   'From:  Student <student@mail.ru>\r\n .       .       .       '
reply:  '250    2.0.0 iBPFgQ7q028433 Message accepted for delivery\r\n'
send:   'quit\r\n'
reply:  '221    2.0.0 mail.onego.ru closing connection\r\n'
```

Из этой (несколько сокращенной) отладочной информации можно увидеть, что клиент отправляет (send) команды SMTP-серверу (*EHlO, MAIL FROM, RCPT TO, DATA, QUIT*), а тот выполняет команды и отвечает (reply), возвращая код возврата.

В ходе одной SMTP-сессии можно отправить сразу несколько писем подряд, если не вызывать *quit()* .

В принципе, команды SMTP можно подавать и отдельно: для этого у объекта-соединения есть методы *(helo(), ehlo(), expn(), help(), mail(), rcpt(), vrfy(), send(), noop(), data()*), соответствующие одноименным командам SMTP-протокола.

Можно задать и произвольную команду SMTP-серверу с помощью метода *docmd().* В следующем примере показан простейший сценарий, который могут использовать те, кто время от времени принимает почту на свой сервер по протоколу SMTP от почтового сервера, на котором хранится очередь сообщений для некоторого домена:

```
from smtplib import SMTP
connect = SMTP('mx.abcde.ru')
connect.set_debuglevel(1)
connect.docmd("ETRN rnd.abcde.ru")
connect.quit()
```

Этот простенький сценарий предлагает серверу *mx.abcde.ru* попытаться связаться с основным почтовым сервером домена *rnd.abcde.ru* и переслать всю накопившуюся для него почту.

При работе с классом *smtplib.SMTP* могут возбуждаться различные исключения. Назначение некоторых из них приведено ниже:

smtplib.SMTPException

Базовый класс для всех исключений модуля.

smtplib.SMTPServerDisconnected

Сервер неожиданно прервал связь (или связь с сервером не была установлена).

smtplib.SMTPResponseException

Базовый класс для всех исключений, которые имеют код ответа SMTP-сервера.

smtplib.SMTPSenderRefused

Отправитель отвергнут

smtplib.SMTPRecipientsRefused

Все получатели отвергнуты сервером.

smtplib.SMTPDataError

Сервер ответил неизвестным кодом на данные сообщения.

smtplib.SMTPConnectError

Ошибка установления соединения.

smtplib.SMTPHeloError

Сервер не ответил правильно на команду `helo` или отверг ее.

3.3. Модуль poplib

Еще один протокол - **POP3** (**P** ost **O** ffice **P** rotocol, почтовый протокол) - служит для приема почты из почтового ящика на сервере (протокол определен в RFC 1725).

Для работы с почтовым сервером требуется установить с ним соединение и, подобно рассмотренному выше примеру, с помощью SMTP-команд получить требуемые сообщения. Объект-соединение POP3 можно установить посредством конструктора класса *POP3* из модуля *poplib*:

```
poplib.POP3(host[, port])
```

где *host* – адрес POP3-сервера, *port* – порт на сервере (по умолчанию 110), *pop_obj* – объект для управления сеансом работы с POP3-сервером.

Следующий пример демонстрирует основные методы для работы с POP3-соединением:

```
import poplib, email
# Учетные данные пользователя:
SERVER = "pop.server.com"
USERNAME = "user"
USERPASSWORD = "secretword"
p = poplib.POP3(SERVER)
print (p.getwelcome())
#   этап идентификации
```

```
print (p.user(USERNAME) )
print (p.pass_(USERPASSWORD))
#   этап транзакций
response, lst, octets = p.list()
print (response)
for msgnum, msgsize in [i.split() for i in lst]:
    print ("Сообщение %(msgnum)s имеет длину %(msgsize)s" % vars())
    print ("UIDL = ", p.uidl(int(msgnum)).split()[2])
    if int(msgsize) > 32000:
        (resp, lines, octets) = p.top(msgnum, 0)
    else:
        (resp, lines, octets) = p.retr(msgnum)
        msgtxt = "\n".join(lines)+"\n\n"
        msg = email.message_from_string(msgtxt)
        print ("* От: %(from)s\n* Кому: %(to)s\n* Тема: %(subject)s\n" %
msg)
        # msg содержит заголовки сообщения или все сообщение (если оно не-
большое)
#   этап обновления
print (p.quit())
```

Примечание:

Разумеется, чтобы пример сработал корректно, необходимо внести реальные учетные данные.

При выполнении сценарий выведет на экран примерно следующее.

```
+OK POP3 pop.server.com server ready
+OK User name accepted, password please
+OK Mailbox open, 68 messages
+OK Mailbox scan listing follows
Сообщение 1 имеет длину 4202
UIDL = 4152a47e00000004
 От: online@kaspersky.com
 Кому: user@server.com
 Тема: KL Online Activation

            ...
+OK Sayonara
```

Методы экземпляров класса POP3 представлены в табл. 3.1.

Таблица 3.1 – Методы экземпляров класса POP3

Метод	**Команда POP3**	**Описание**
getwelcome()		Получает строку s с приветствием POP3-сервера
user(name)	USER name	Посылает команду user с указанием имени пользователя name. Возвращает строку с ответом сервера
pass (pwd)	PASS pwd	Отправляет пароль пользователя в команде PASS. После этой команды и до выполнения команды QUIT почтовый ящик блокируется
apop(user, secret)	APOP user secret	Идентификация на сервере по APOP

rpop(user) stat()	RPOP user STAT	Идентификация по методу RPOP Возвращает кортеж с информацией о почтовом ящике. В нем m - количество сообщений, l - размер почтового ящика в байтах
list([num])	LIST [num]	Возвращает список сообщений в формате (resp, ['num octets', ...]), если не указан num, и "+OK num octets", если указан. Список lst состоит из строк в формате "num octets".
retr(num)	RETR num	Загружает с сервера сообщение с номером num и возвращает кортеж с ответом сервера (resp, lst, octets)
dele(num)	DELE num	Удаляет сообщение с номером num
rset()	RSET	Отменяет пометки удаления сообщений
noop()	NOOP	Ничего не делает (поддерживает соединение)
quit()	QUIT	Отключение от сервера. Сервер выполняет все необходимые изменения (удаляет сообщения) и снимает блокировку почтового ящика
top(num, lines)	TOP num lines	Команда аналогична RETR, но загружает только заголовок и lines строк тела сообщения. Возвращает кортеж (resp, lst, octets)
uidl([num])	UIDL [num]	Сокращение от "unique-id listing" (список уникальных идентификаторов сообщений). Формат результата: (resp, lst, octets),если num не указан, и "+OK num uniqid", если указан. Список lst состоит из строк вида "+ok num uniqid"

В этой таблице *num* обозначает номер сообщения (он не меняется на протяжении всей сессии), *resp* -- ответ сервера, возвращается для любой команды, начинается с *"+OK "* для успешных операций (при неудаче возбуждается исключение *poplib.proto_error*). Параметр *octets* обозначает количество байт в принятых данных. *uniqid* − идентификатор сообщения, генерируемый сервером.

Работа с POP3-сервером состоит из трех фаз: идентификации, транзакций и обновления. На этапе идентификации сразу после создания POP3-объекта разрешены только команды USER, PASS (иногда APOP и RPOP). После идентификации сервер получает информацию о пользователе и наступает этап транзакций. Здесь уместны остальные команды. Этап обновления вызывается командой QUIT, после которой POP3- сервер обновляет почтовый ящик пользователя в соответствии с поданными командами, а именно – удаляет помеченные для удаления сообщения.

3.4. Модули для клиента WWW

Стандартные средства языка Python позволяют получать из программы доступ к объектам WWW как в простых случаях, так и при сложных

обстоятельствах, в частности при необходимости передавать данные формы, идентификации, доступа через прокси и т.п.

Стоит отметить, что при работе с WWW используется в основном протокол HTTP, однако WWW охватывает не только HTTP, но и многие другие схемы (FTP, gopher, HTTPS и т.п.). Используемая схема обычно указана в самом начале URL.

3.4.1. Функции для загрузки сетевых объектов

Простой случай получения WWW-объекта по известному URL показан в следующем примере:

```
import urllib.request as urllib2
doc = urllib2.urlopen("http://www.yandex.ru").read()
print (doc[:100])
```

Сначала мы импортируем *urllib.request*, как *urllib2*. Функция *urllib2.urlopen()* создает файлоподобный объект, который читает методом *read()*. Другие методы этого объекта: *readline(), readlines(), fileno(), close()* работают как и у обычного файла, а также есть метод *info()*, который возвращает соответствующий полученному с сервера Message-объект. Этот объект можно использовать для получения дополнительной информации:

```
f = urllib2.urlopen("http://www.yandex.ru")
print (f.info())

Date: Mon, 19 Oct 2015 12:22:23 GMT
Content-Type: text/html; charset=UTF-8
Cache-Control: no-cache,no-store,max-age=0,must-revalidate
Expires: Mon, 19 Oct 2015 12:22:23 GMT
Last-Modified: Mon, 19 Oct 2015 12:22:23 GMT
Content-Security-Policy: default-src 'self' 'unsafe-inline' 'unsafe-eval'
https://yastatic.net *.yandex.ru yandex.ru *.yandex.net https://*.yandex.ru
https://yandex.ru https://*.yandex.net yandex.st yastatic.net
*.yastatic.net ws://portal-xiva.yandex.net wss://portal-xiva.yandex.net
wss://push.yandex.ru; img-src data: 'self' https://yastatic.net www.tns-
counter.ru https://www.tns-counter.ru *.yandex.ru yandex.ru *.tns-
counter.ru *.gemius.pl https://*.yandex.ru https://yandex.ru https://*.tns-
counter.ru https://*.gemius.pl yandex.st *.yandex.net yastatic.net
*.yastatic.net; report-uri
https://csp.yandex.net/csp?from=big.ru&showid=22891.11022.1445257343.3959&h
=n55&yandexuid=2274833241445257343;
P3P: policyref="/w3c/p3p.xml", CP="NON DSP ADM DEV PSD IVDo OUR IND STP PHY
PRE NAV UNI"
Set-Cookie: yp=; Expires=Fri, 21-Oct-2005 12:22:23 GMT; Path=/
Set-Cookie: yp=; Expires=Fri, 21-Oct-2005 12:22:23 GMT; Do-
main=.www.yandex.ru; Path=/
Set-Cookie: yp=1447849343.ygu.1; Expires=Thu, 16-Oct-2025 12:22:23 GMT; Do-
main=.yandex.ru; Path=/
Set-Cookie: yandex_gid=959; Expires=Wed, 18-Nov-2015 12:22:23 GMT; Do-
main=.yandex.ru; Path=/
Set-Cookie: S=; Expires=Fri, 21-Oct-2005 12:22:23 GMT; Path=/
Set-Cookie: yandexuid=2274833241445257343; Expires=Thu, 16-Oct-2025
12:22:23 GMT; Domain=.yandex.ru; Path=/
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Content-Length: 82463
Connection: Close
```

С помощью функции *urllib.urlopen()* можно делать и более сложные вещи, например, передавать web-серверу данные формы. Как известно, данные заполненной web-формы могут быть переданы на web-сервер с использованием метода **GET** или метода **POST**. Метод **GET** связан с кодированием всех передаваемых параметров после знака "?" в **URL**, а при методе **POST** данные передаются в теле HTTP-запроса. Оба варианта передачи представлены ниже:

```
import urllib.request as urllib2
import urllib
data = {"seach":    "Python"}
enc_data = urllib.parse.urlencode(data)
#    метод GET
print("http://searchengine.com/search" + "?" + enc_data)
f = urllib2.urlopen("http://searchengine.com/search" + "?" + enc_data)
print (f.read())
#   метод POST
f = urllib2.urlopen("http://searchengine.com/search", enc_data)
print (f.read())
```

В некоторых случаях данные имеют повторяющиеся имена. В этом случае в качестве параметра *urllib.parse.urlencode* можно использовать вместо словаря последовательность пар имя-значение:

```
import urllib
data = [("n", "1"), ("n", "3"),    ("n", "4"), ("button", "Привет")]
enc_data = urllib.parse.urlencode(data)
print(enc_data)
```

Результат выполнения:

```
n=1&n=3&n=4&button=%D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D1%82
```

Модуль *urllib* позволяет загружать web-объекты через прокси-сервер. Если ничего не указывать, будет использоваться прокси-сервер, который был задан принятым в конкретной ОС способом. В Unix прокси-серверы задаются в переменных окружения

http_proxy, ftp proxy и т.п., в Windows прокси-серверы записаны в реестре, а в Mac

OS они берутся из конфигурации Internet. Задать прокси-сервер можно следующим образом*:*

```
 Использовать указанный прокси
import urllib.request as urllib2
proxy = urllib2.ProxyHandler({'http': 'http://proxy.xy.z:8080'})
opener = urllib2.build_opener(proxy)
urllib2.install_opener(opener)
urllib2.urlopen('http://www.google.com')
```

Функция *urlretrieve()* позволяет записать заданный URL сетевой объект в файл. Она имеет следующие параметры:

urllib.urlretrieve(url[, filename[, reporthook[, data]]])

Здесь *url* - URL сетевого объекта, *filename* - имя локального файла для помещения объекта, *reporthook* - функция, которая будет вызываться для сообщения о состоянии загрузки, *data* - данные для метода POST (если он используется). Функция возвращает кортеж *(filepath, headers)*, где

filepath - имя локального файла, в который закачан объект, *headers* - результат метода *info()* для объекта, возвращенного *urlopen().*

Для обеспечения интерактивности функция *urllib.urlretrieve()* вызывает время от времени функцию, заданную в *reporthook().* Этой функции передаются три аргумента: количество принятых блоков, размер блока и общий размер принимаемого объекта в байтах (если он неизвестен, этот параметр равен -1).

В следующем примере программа принимает большой файл и, чтобы пользователь не скучал, пишет процент от выполненной загрузки и предполагаемое оставшееся время:

```
FILE = 'boost-1.31.0-9.src.rpm'
URL = 'http://download.fedora.redhat.com/pub/fedora/linux/coreZ3/SRPMS/' +
FILE
def download(url, file):
    import urllib.request, time
    start_t = time.time()
    def progress(bl, blsize, size):
        dldsize = min(bl*blsize, size)
        if size != -1:
            p = float(dldsize) / size
            try:
                elapsed = time.time() - start_t
                est_t = elapsed / p - elapsed
            except:
                est_t = 0
            print("%6.2f %% %6.0f s %6.0f s %6i / %-6i bytes" %    (p*100,
                                              elapsed, est_t, dldsize,
size))
        else:
            print ("%6i / %-6i bytes" % (dldsize, size))

    urllib.request.urlretrieve(URL, FILE, progress)
download(URL, FILE)
```

Эта программа выведет примерно следующее (процент от полного объема закачки, прошедшие секунды, предполагаемое оставшееся время, закачанные байты, полное количество байтов):

```
0.00 %    1    s  0     s  0      / 6952309 bytes
0.12 %    5    s  3941  s  8192   / 6952309 bytes
0.24 %    7    s  3132  s  16384  / 6952309 bytes
0.35 %    10   s  2864  s  24576  / 6952309 bytes
0.47 %    12   s  2631  s  32768  / 6952309 bytes
0.59 %    15   s  2570  s  40960  / 6952309 bytes
0.71 %    18   s  2526  s  49152  / 6952309 bytes
0.82 %    20   s  2441  s  57344  / 6952309 bytes
```

3.4.2. Функции для анализа URL

Согласно документу RFC 2396 URL должен строиться по следующему шаблону:

```
scheme://netloc/path;parameters?query#fragment
```

Где

```
scheme
```

Адресная схема. Например: http, ftp, gopher.

`netloc`

Местонахождение в сети.

`path`

Путь к ресурсу.

`params`

Параметры.

`query`

Строка запроса.

`fragment`

Для работы с url адресами в python3 используется модуль urllib.parse, который содержит методы. Рассмотрим их более детально.

```
urllib.parse.quote(<строка>[, safe='/'][, encoding=None][, errors=None])
```

заменяет все специальные симолы последовательностями *%nn*. Цифры, английские буквы и символы подчеркивания, точки и дефиса не кодируются. Пробелы преобразуются в последовательность %20.

Параметры: **safe** (*str*) – символы, которые преобразовывать нельзя

```
urllib.parse.quote_plus(<строка>[, safe='/'][, encoding=None][, er-
rors=None])
```

функция аналогична функции *quote()*, но пробелы заменяются символом +, а символ / заменяется на *%2F*

Параметры: **safe** (*str*) – символы, которые преобразовывать нельзя

```
urllib.parse.quote_from_bytes(<последовательность байтов>[, safe='/'])
```

функция аналогична функции *quote()*, но в качестве первого параметра принимает последовательность байтов

```
urllib.parse.unquote(<строка>[, encoding='utf-8'][, errors='replace'])
```

заменяет последовательность *%nn* на соответсвующие символы. символ + не заменяется ничем

```
urllib.parse.unquote_plus(<строка>[, encoding='utf-8'][, errors='replace'])
```

заменяет последовательность *%nn* на соответсвующие символы. символ + заменяет пробелом

```
urllib.parse.urlencode(<объект> [, doseq=False][, safe=''][, encod-
ing=None][, errors=None])
```

преобразовывает отдельные составляющие в строку запроса.

Параметры:

- **объект** – словарь или список кортежей (кортеж из 2-х элементов);
- **doseq** (*bool*) – если истина, то можно указать последовательность из нескольких значений во втором параметре кортежа.

```
urlencode({"str": "Строка 2", "var": 20}, encoding="cp1251")
'var=20&str='Ш1%F2•;FO%EE%EA%E0+2'
urlencode([("str", "Строка 2"), ("var", 20)], encoding="cp1251")
'str=%D1%F2%FO%EE%EA%E0+2&var=20'
```

```
urllib.parse.urljoin(<базовый урл>, <относитьльный или абсолютный
урл>[, <разбор якоря>])
```

преобразует относитльный урл в абсолютный

Пример:

```
from urllib.parse import urljoin
print(urljoin('http://admin.ru/f1/f2/test.html', 'file.html'))
```

```
print(urljoin('http://admin.ru/f1/f2/test.html', 'f3/file.html'))
print(urljoin('http://admin.ru/f1/f2/test.html', '/file.html'))
print(urljoin('http://admin.ru/f1/f2/test.html', './file.html'))
print(urljoin('http://admin.ru/f1/f2/test.html', '../file.html'))
```

Результат:

```
http://admin.ru/f1/f2/file.html
http://admin.ru/f1/f2/f3/file.html
http://admin.ru/file.html
http://admin.ru/f1/f2/file.html
http://admin.ru/f1/file.html
```

```
urllib.parse.urlsplit(<url>[, <схема>[, <разбор_якоря>=False]])
```

возвращает *SplitResult* с результатом разбора адреса

```
urlsplit('http://ilnurgi.ru:80/test.php;st?var=5#metka')
SplitResult(scheme='http', netloc='ilnurgi.ru:80', path='/test.php',
query='var=5', fragment='metka')
urllib.parse.urlunparse(<последовательность>)
```

возвращает строку, адрес, собранную из отдельных значений

```
urlunparse(('http', 'ilnurgi.ru:80', '/test.php', '', 'var=5', 'metka'))
'http://ilnurgi.ru:80/test.php?var=5#metka'
urllib.parse.urlunsplit(<последовательность>)
```

возвращает строку, адрес, собранную из отдельных значений

```
urlunsplit(('http', 'ilnurgi.ru:80', '/test.php', '', 'var=5', 'metka'))
'http://ilnurgi.ru:80/test.php?var=5#metka'
```

Результат парсинга адреса представляет собой экзкмпляр класса class *urllib.parse.SplitResult* и содержит следующие поля:

scheme – название протокола

netloc – название домена вместе с номером порта

path – путь

hostname – название домена в нижнем регистре

port – номер порта

params – параметры

query – строка запроса

fragment – якорь

username - имя пользователя

password – пароль

geturl() – возвращает адрес

Таким образом, модуль urllib языка python содержит полный набор функций для разбора url адресов.

Функциональности модулей *urllib* и *urlparse* хватает для большинства задач, которые решают сценарии на Python как web-клиенты. Тем не менее, иногда требуется больше. На этот случай можно использовать модули для работы с протоколом HTTP – *http.client и http.server*(в данной методичке модуль *httplib* не рассматривается).

3.5. XML-RPC сервер

До сих пор высокоуровневые протоколы рассматривались с точки зрения клиента. Не менее просто создавать на Python и их серверные части.

Для иллюстрации того, как разработать программу на Python, реализующую сервер, был выбран протокол XML- RPC. Несмотря на свое название, конечному пользователю необязательно знать XML (об этом языке разметки говорилось на одной из предыдущих лекций), так как он скрыт от него. Сокращение **RPC** (**R** emote **P** rocedure **C** all, вызов удаленной процедуры) объясняет суть дела: с помощью XML-RPC можно вызывать процедуры на удаленном хосте. Причем при помощи XML-RPC можно абстрагироваться от конкретного языка программирования за счет использования общепринятых типов данных (строки, числа, логические значения и т.п.). В языке Python вызов удаленной функции по синтаксису ничем не отличается от вызова обычной функции.

```
import  xmlrpc.client
req = xmlrpc.client.ServerProxy("http://localhost:8000")
try:
    # Вызвать удаленную функцию
    print (req.add(1, 3))
except xmlrpc.client.Error:
    print ("ERROR")
```

А вот как выглядит XML-RPC-сервер (для того чтобы попробовать пример выше, необходимо сначала запустить сервер):

```
from xmlrpc.server import SimpleXMLRPCServer
#  Запустить сервер
srv = SimpleXMLRPCServer(("localhost", 8000))
# Зарегистрировать функцию
srv.register_function(pow)
# И еще одну
srv.register_function(lambda x,y: x+y, 'add')
 #  Обслуживать запросы
srv.serve_forever()
```

С помощью XML-RPC (а этот протокол достаточно "легковесный" среди других подобных протоколов) приложения могут общаться друг с другом на понятном им языке вызова функций с параметрами основных общепринятых типов и такими же возвращаемыми значениями. Преимуществом же Python является удобный синтаксис вызова удаленных функций.

Разумеется, это только пример. При реальном использовании необходимо позаботиться, чтобы XML-RPC сервер отвечал требованиям безопасности. Кроме того, сервер лучше делать многопоточным, чтобы он мог обрабатывать несколько потоков одновременно. Для многопоточности (она будет обсуждаться в отдельной лекции) не нужно многое переделывать достаточно определить свой класс, скажем, *ThreadingXMLRPCServer*, в котором вместо *SocketServer.TCPServer* использовать *socketserver.ThreadingTCPServer*. Это предлагается в качестве упражнения. Наводящий вопрос: где находится определение класса *SimpleXMLRPCServer*?

Раздел 4. СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ

Сериализация (в программировании) – процесс перевода какой-либо структуры данных в последовательность битов. Обратной к операции сериализации является операция десериализации (структуризации) – восстановление начального состояния структуры данных из битовой последовательности.

Сериализация используется для передачи объектов по сети и для сохранения их в файлы. Например, нужно создать распределенное приложение, разные части которого должны обмениваться данными со сложной структурой. В таком случае для типов данных, которые предполагается передавать, пишется код, который осуществляет сериализацию и десериализацию. Объект заполняется нужными данными, затем вызывается код сериализации, в результате получается, например, XML-документ. Результат сериализации передается принимающей стороне по, скажем, электронной почте или HTTP. Приложение-получатель создает объект того же типа и вызывает код десериализации, в результате получая объект с теми же данными, что были в объекте приложения-отправителя. По такой схеме работает, например, сериализация объектов через SOAP в Microsoft .NET.

4.1. Модуль pickle для серилизации объектов

Модуль pickle реализует мощный алгоритм сериализации и десериализации объектов Python. "Pickling" - процесс преобразования объекта Python в поток байтов, а "unpickling" - обратная операция, в результате которой поток байтов преобразуется обратно в Python-объект. Так как поток байтов легко можно записать в файл, модуль pickle широко применяется для сохранения и загрузки сложных объектов в Python.

Не загружайте с помощью модуля pickle файлы из ненадежных источников. Это может привести к необратимым последствиям.

Модуль pickle предоставляет следующие функции для удобства сохранения/загрузки объектов:

pickle.dump(obj, file, protocol=None, *, fix_imports=True) - записывает сериализованный объект в файл. Дополнительный аргумент protocol указывает используемый протокол. По умолчанию равен 3 и именно он рекомендован для использования в Python 3 (несмотря на то, что в Python 3.4 добавили протокол версии 4 с некоторыми оптимизациями). В любом случае, записывать и загружать надо с одним и тем же протоколом.

pickle.dumps(obj, protocol=None, *, fix_imports=True) - возвращает сериализованный объект. Впоследствии вы его можете использовать как угодно.

pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict") - загружает объект из файла.

pickle.loads(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict") - загружает объект из потока байт.

Модуль pickle также определяет несколько исключений:

pickle.PickleError

- pickle.PicklingError - случились проблемы с сериализацией объекта.
- pickle.UnpicklingError - случились проблемы с десериализацией объекта.

Этих функций вполне достаточно для сохранения и загрузки встроенных типов данных.

```
import pickle
data = {
     'a': [1, 2.0, 3, 4+6j],
     'b': ("character string", b"byte string"),
     'c': {None, True, False}
}
with open('data.pickle', 'wb') as f:
     pickle.dump(data, f)
with open('data.pickle', 'rb') as f:
     data_new = pickle.load(f)
print(data_new)
{'c': {False, True, None}, 'a': [1, 2.0, 3, (4+6j)], 'b': ('character
string', b'byte string')}
```

4.2. Серилизация через модуль json

JSON (JavaScript Object Notation) - простой формат обмена данными, основанный на подмножестве синтаксиса JavaScript. Модуль json позволяет кодировать и декодировать данные в удобном формате.

Кодирование основных объектов Python:

```
import json
json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
print(json.dumps("\"foo\bar"))
"\"foo\bar"
print(json.dumps('\u1234'))
"\u1234"
print(json.dumps('\\'))
"\\"
print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
```

Компактное кодирование:

```
import json
json.dumps([1,2,3,{'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

Красивый вывод:

```
import json
print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Декодирование (парсинг) JSON:

```
import json
json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
json.loads('"\\"foo\\bar"')
'"foo\x08ar'
```

Основы.

json.dump(obj, fp, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw) - сериализует obj как форматированный JSON поток в fp.

Если skipkeys = True, то ключи словаря не базового типа (str, unicode, int, long, float, bool, None) будут проигнорированы, вместо того, чтобы вызывать исключение TypeError.

Если ensure_ascii = True, все не-ASCII символы в выводе будут экранированы последовательностями \uXXXX, и результатом будет строка, содержащая только ASCII символы. Если ensure_ascii = False, строки запишутся как есть.

Если check_circular = False, то проверка циклических ссылок будет пропущена, а такие ссылки будут вызывать OverflowError.

Если allow_nan = False, при попытке сериализовать значение с запятой, выходящее за допустимые пределы, будет вызываться ValueError (nan, inf, -inf) в строгом соответствии со спецификацией JSON, вместо того, чтобы использовать эквиваленты из JavaScript (NaN, Infinity, -Infinity).

Если indent является неотрицательным числом, то массивы и объекты в JSON будут выводиться с этим уровнем отступа. Если уровень отступа 0, отрицательный или "", то вместо этого будут просто использоваться новые строки. Значение по умолчанию None отражает наиболее компактное представление. Если indent - строка, то она и будет использоваться в качестве отступа.

Если sort_keys = True, то ключи выводимого словаря будут отсортированы.

json.dumps(obj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw) - сериализует obj в строку JSON-формата.

Аргументы имеют то же значение, что и для dump().

Ключи в парах ключ/значение в JSON всегда являются строками. Когда словарь конвертируется в JSON, все ключи словаря преобразовываются в строки. В результате этого, если словарь сначала преобразовать в JSON, а потом обратно в словарь, то можно не получить словарь, идентичный исходному. Другими словами, loads(dumps(x)) != x, если x имеет нестроковые ключи.

json.load(fp, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw) - десериализует JSON из fp.

object_hook - опциональная функция, которая применяется к результату декодирования объекта (dict). Использоваться будет значение, возвращаемое этой функцией, а не полученный словарь.

object_pairs_hook - опциональная функция, которая применяется к результату декодирования объекта с определенной последовательностью пар ключ/значение. Будет использован результат, возвращаемый функцией, вместо исходного словаря. Если задан так же object_hook, то приоритет отдается object_pairs_hook.

parse_float, если определен, будет вызван для каждого значения JSON с плавающей точкой. По умолчанию, это эквивалентно float(num_str).

parse_int, если определен, будет вызван для строки JSON с числовым значением. По умолчанию эквивалентно int(num_str).

parse_constant, если определен, будет вызван для следующих строк: "-Infinity", "Infinity", "NaN". Может быть использовано для возбуждения исключений при обнаружении ошибочных чисел JSON.

Если не удастся десериализовать JSON, будет возбуждено исключение ValueError.

json.loads(s, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw) - десериализует s (экземпляр str, содержащий документ JSON) в объект Python.

Остальные аргументы аналогичны аргументам в load().

4.2.1. Кодировщики и декодировщики

Класс **json.JSONDecoder**(object_hook=None, parse_float=None, parse_int=None, parse_constant=None, strict=True, object_pairs_hook=None) - простой декодер JSON.

Выполняет следующие преобразования при декодировании:

Таблица 4.1 – Соответствие форматов преобразования JSON в Python

JSON	**Python**
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Он также понимает NaN, Infinity, и -Infinity как соответствующие значения float, которые находятся за пределами спецификации JSON.

Класс **json.JSONEncoder**(skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None)

Расширяемый кодировщик JSON для структур данных Python. Поддерживает следующие объекты и типы данных по умолчанию.

Таблица 4.1 – Соответствие форматов преобразования Python в JSON

Python	**JSON**
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

ЛИТЕРАТУРА

1. Прохоренок Н.А. Python 3 и PyQt. Разработка приложений. – СПб.: БХВ-Петербург, 2012. – 704 с.: ил. – ISBN 978-5-9775-0797-4.

2. Уэс Маккинни. Python и анализ данных; пер. с англ. – М.: ДМК Пресс, 20015. – 482 с. – ISBN 978-5-97060-315-4.

3. Фримен Э., Фримен Э., Сьерра К., Бейтс Б. Паттерны проектирования. – СПб.: Питер, 2011. – 656 с.: ил. – ISBN 978-5-459-00435-9.

4. https://docs.python.org/3.4/ – официальная документация по языку python(Дата обращения на сайт 23.03.2015 15:00).

5. http://citforum.ru/SE/project/pattern/ – информационный портал по технологиям программирования (Дата обращения на сайт 18.04.2015 10:00)

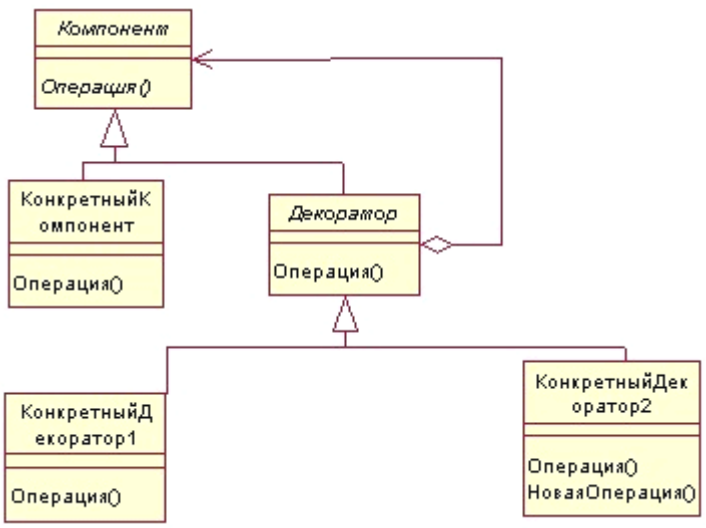
6. . http://pythonworld.ru/ – самоучитель по языку программирования python (Дата обращения на сайт 18.04.2015 17:00).

Краткое описание структурных паттернов проектирования

Адаптер (Adapter) – GoF

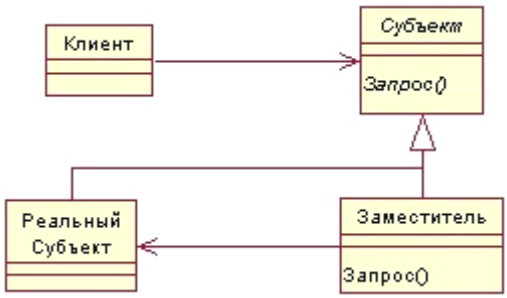
Проблема	Необходимо обеспечить взаимодействие несовместимых интерфейсов или как создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами
Решение	Конвертировать исходный интерфейс компонента к другому виду с помощью промежуточного объекта - адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних обьектов к этому объекту – адаптеру
Пример	Соответствует примеру из описания паттерна "Полиморфизм",

Декоратор (Decorator) или Оболочка (Wrapper) - GoF

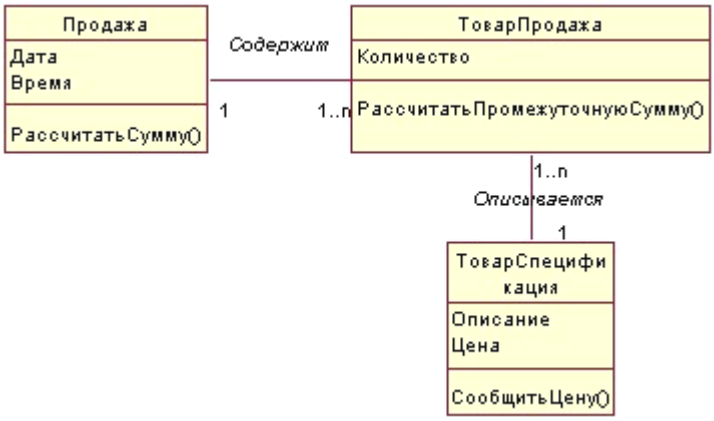
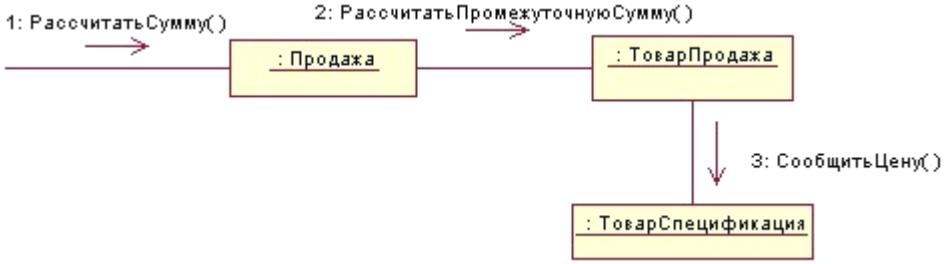
Проблема	Возложить дополнительные обязанности (прозрачные для клиентов) на отдельный объект, а не на класс в целом
Рекомендации	Применение нескольких "Декораторов" к одному "Компоненту" позволяет произвольным образом сочетать обязанности, например, одно свойство можно добавить дважды
Решение	Динамически добавить объекту новые обязанности не прибегая при этом к порождению подклассов (наследованию). "Компонент"определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности, "КонкретныйКомпонент" определяет объект, на который возлагаются дополнительные обязанности, "Декоратор" - хранит ссылку на объект "Компонент" и определяет интерфейс, соответствующий интерфейсу "Компонента". "КонкретныйДекоратор" возлагает дополнительные обязанности на компонент. "Декоратор" переадресует запросы объекту "Компонент" 

Преимущества	Большая гибкость, чем у статического наследования: можно добавлять и удалять обязанности во время выполнения программы в то время как при использовании наследования надо было бы создавать новый класс для каждой дополнительной обязанности. Данный паттерн позволяет избежать перегруженных методами классов на верхних уровнях иерархии - новые обязанности можно добавлять по мере необходимости.
Недостатки	"Декоратор" и его "Компонент" не идентичны, и, кроме того, получается что система состоит из большого числа мелких обьектов, которые похожи друг на друга и различаются только способом взаимосвязи а не классом и не значениями своих внутренних переменных - такая система сложна в изучении и отладке.

Заместитель (Proxy) или Суррогат (Surrogate) - GoF

Проблема	Необходимо управлять доступом к объекту, так чтобы создавать громоздкие объекты "по требованию".
Решение	Создать суррогат громоздкого объекта. "Заместитель" хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту (объект класса "Заместитель" может обращаться к объекту класса "Субъект", если интерфейсы "РеальногоСубъекта" и "Субъекта" одинаковы). Поскольку интерфейс "РеальногоСубъекта" идентичен интерфейсу "Субъекта", так, что "Заместителя" можно подставить вместо "РеальногоСубъекта", контролирует доступ к "РеальномуСубъекту", может отвечать за создание или удаление "РеальногоСубъекта". "Субъект" определяет общий для "РеальногоСубъекта" и "Заместителя" интерфейс, так, что "Заместитель" может быть использован везде, где ожидается "РеальныйСубъект". При необходимости запросы могут быть переадресованы "Заместителем" "РеальномуСубъекту".  "Заместитель" может иметь и другие обязанности, а именно: • удаленный "Заместитель" может отвечать за кодирование запроса и его аргументов и отправку закодированного запроса реальному "Субъекту", • виртуальный "Заместитель" может кэшировать дополнительную информацию о реальном "Субъекте", чтобы отложить его создание, • защищающий "Заместитель" может проверять, имеет ли вызывающий объект необходимые для выполнения запроса права.

Информационный эксперт (Information Expert) - GRASP

Проблема	В системе должна аккумулироваться, рассчитываться и т. п. необходимая информация.
Решение	Назначить обязанность аккумуляции информации, расчета и т. п. некоему классу (информационному эксперту), обладающему необходимой информацией.
Рекомендации	Информационным экспертом может быть не один класс, а несколько.
Пример	Необходимо рассчитать общую сумму продажи. Имеются классы проектирования "Продажа", "ТоварПродажа" (продажа отдельного вида товара в рамках продажи в целом), "ТоварСпецификация" (описание конкретного вида товара). Необходимо распределить обязанности по предоставлению информации и расчету между этими классами. Объект "Продажа" должен передать сообщение "Рассчитать промежуточную сумму" каждому экземпляру класса "ТоварПродажа" (которые, в свою очередь, передают сообщения "СообщитьЦену" объектам "ТоварСпецификация", с целью получения информации о цене экземпляра товара), и, затем, просуммировать полученные результаты. Промежуточную сумму рассчитывает объект "Товар Продажа". Таким образом, все три объекта являются информационными экспертами.  Диаграмма классов проектирования. 
Преимущества	Поддерживает инкапсуляцию, то есть объекты используют свои собственные данные для выполнения поставленных задач.
Недостатки	Если объект, обладающий наиболее полной информацией, например, о продаже (*см. пример* - класс "Продажа"), будет отвечать и за со-

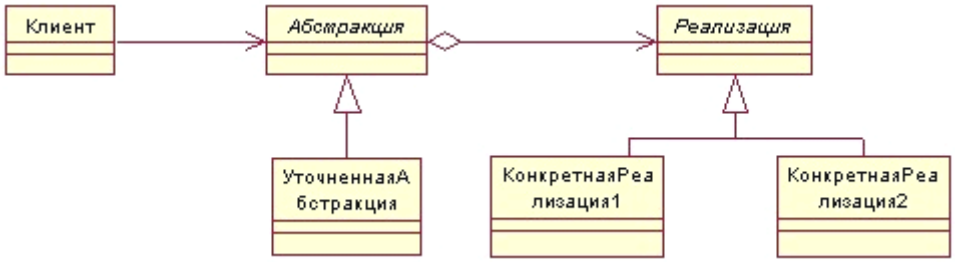
	хранение этой информации в базе данных, то получится, что логика приложения (моделирование продажи) и логика связи с базой данных "помещаются" в один класс (нарушение принципа разделения обязанностей основных объектов системы, и, кроме того, логика связи с базой данных будет дублироваться во многих других классах.

Компоновщик (Composite) - GoF

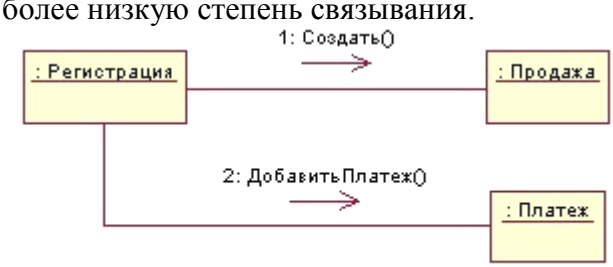
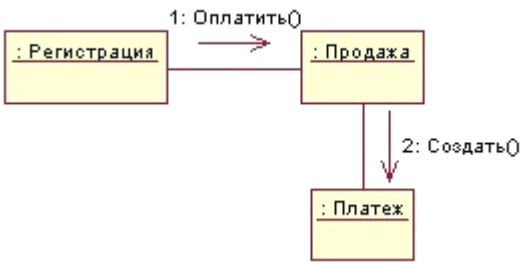
Проблема	Как обрабатывать группу или композицию структур обьектов одновременно?
Решение	Определить классы для композитных и атомарных обьектов таким образом, чтобы они реализовывали один и тот же интерфейс.
Пример	См. паттерн "Стратегия", 3.2.9, необходимо учесть несколько скидок различных видов (зависят от времени, типа покупателя, типом выбранного продукта. Как применять политику ценообразования? Вырабатывается стратегия приоритета скидок, объект "Продажа" не должен обладать информацией о применяемых скидках, но можно было бы применить стратегию расчета скидок. Создается новый класс "РасчетСкидкиАлгоритмКомпозит".

Мост (Bridge), Handle (описатель) или Тело (Body) - GoF

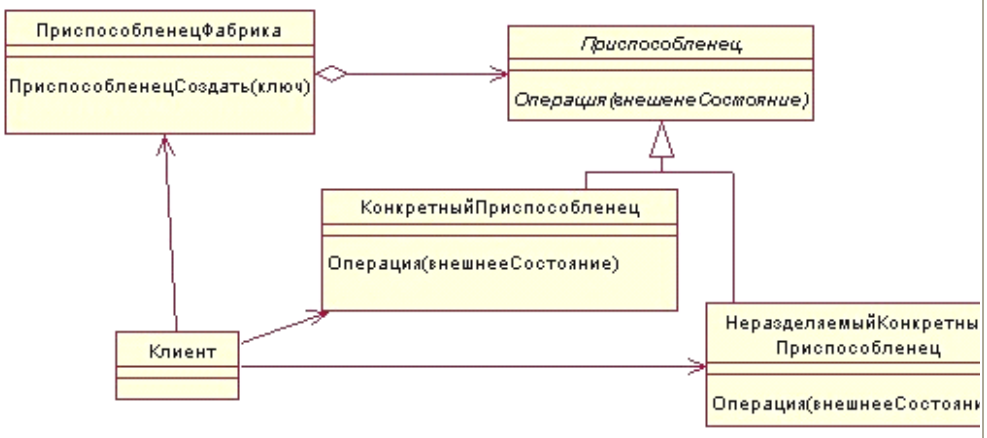
Проблема	Требуется отделить абстракцию от реализации так, чтобы и то и другое можно было изменять независимо. При использовании наследования реализация жестко привязывается к абстракции, что затрудняет независимую модификацию.
Решение	Поместить абстракцию и реализацию в отдельные иерархии классов.
Рекомендации	Можно использовать если, например, реализацию необходимо выполнять во время реализации программы.
Пример	"Абстракция" определяет интерфейс "Абстракции" и хранит ссылку на объект "Реализация", "УточненнаяАбстракция" расширяет интерфейс, определенный "Абстракцией". "Реализация" определяет интерфейс для классов реализации, он не обязан точно соответствовать интерфейсу класса "Абстракция" - оба интерфейса могут быть совершенно различны. Обычно интерфецйс класса "Реализация" предоставляет только примитивные операции, а класс "Абстракция" определяет операции более высокого уровня, базирующиеся на этих примитивных. "КонкретнаяРеализация" содержит конкретную реализацию класса "Реализация". Объект "Абстракция" перенаправляет своему объекту "Реализация" запросы "Клиента".

	
Преимущества	Отделение реализации от интерфейса, то есть, "Реализацию" "Абстракции" можно конфигурировать во время выполнения. Кроме того, следует упомянуть, что разделение классов "Абстракция" и "Реализация" устраняет зависимости от реализации, устанавливаемые на этапе компиляции: чтобы изменить класс "Реализация" вовсе не обязательно перекомпилировать класс "Абстракция".

Низкая связанность (Low Coupling) - GRASP

Проблема	Обеспечить низкую связанность при создании экземпляра класса и связывании его с другим классом.
Решение	Распределить обязанности между объектами так, чтобы степень связанности оставалась низкой.
Пример	Необходимо создать экземпляр класса "Платеж". В предметной области регистрация объекта "Платеж" выполняется объектом "Регистрация" (ведется рестр). Ниже приводятся 2 способа создания экземпляра класса "Платеж". Верхний рисунок - с использованием паттерна "Создатель", нижний - с использованием "Низкая связанность". Последний способ обеспечивает более низкую степень связывания.  

Приспособленец (Flyweight) - GoF

Проблема	Необходимо обеспечить поддержку множества мелких обьектов.
Рекомендации	Приспособленцы моделируют сущности, число которых слишком велико для представления объектами. Имеет смысл использовать данный паттерн если одновременно выполняются следующие условия: • в приложении используется большое число обьектов, из-за этого расходы на хранение высоки, • большую часть состояния обьектов можно вынести вовне, • многие группы обьектов можно заменить относительно небольшим количеством обьектов, поскольку состояния обьектов вынесены вовне.
Решение	Создать разделяемый объект, который можно использовать одновременно в нескольких контекстах, причем, в каждом контексте он выглядит как независимый объект (неотличим от экземпляра, который не разделяется). "Приспособленец" объявляет интерфейс, с помощью которого приспособленцы могут получить внешнее состояние или как-то воздействовать на него, "КонкретныйПриспособленец" реализует интерфейс класса "Приспособленец" и добавляет при необходимости внутреннее состояние. Внутреннее состояние хранится в объекте "КонкретныйПриспособленец", в то время как внешнее состояние хранится или вычисляется "Клиентами" ("Клиент" передает его "Приспособленцу" при вызове операций). Объект класса "КонкретныйПриспособленец" должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть независимым от контекста, "ПриспособленецФабрика" - создает объекты - "Приспособленцы" (или предоставляет существующий экземпляр) и управляет ими. "НеразделяемыйКонкретныйПриспособленец" - не все подклассы "Приспособленца" обязательно должны быть разделяемыми. "Клиент" - хранит ссылки на одного или нескольких "Приспособленцев", вычисляет и хранит внешнее состояние "Приспособленцев". 
Преимущества	Вследствие уменьшения общего числа экземпляров и вынесения состояния экономится память.

Устойчивый к изменениям (Protected Variations) - GRASP

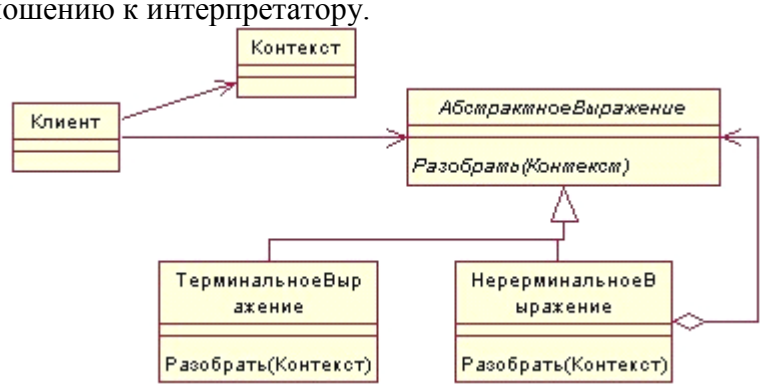
Проблема	Как спроектировать систему так, чтобы изменение одних ее элементов не влияло на другие?
Решение	Идентифицировать точки возможных изменений или неустойчивости и распределить обязанности таким образом, чтобы обеспечить устойчивую работу системы.
Пример	Паттерн проектирования "Полиморфизм", см. 3.2.15 является хорошей иллюстрацией данного метода. В данном случае точкой вариации или неустойчивости являются интерфейсы внешних систем. При добавлении интерфейса "ІНалоговаяСистемаАдаптер" на основе принципа полиморфизма получается, что внутренние объекты смогут взаимодействовать с устойчивым интерфейсом, а детали взаимодействия с внешними системами будут скрыты в конкретных реализациях адаптеров.

Фасад (Facade) - GoF

Проблема	Как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться?
Решение	Определить одну точку взаимодействия с подсистемой - фасадный объект, обеспечивающий общий интерфейс с подсистемой и возложить на него обязанность по взаимодействию с ее компонентами. Фасад - это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам. Фасадный объект обеспечивает реализацию паттерна "Устойчивый к изменениям" с точки зрения защиты от изменений в реализации подсистемы см. п. 3.1.9.

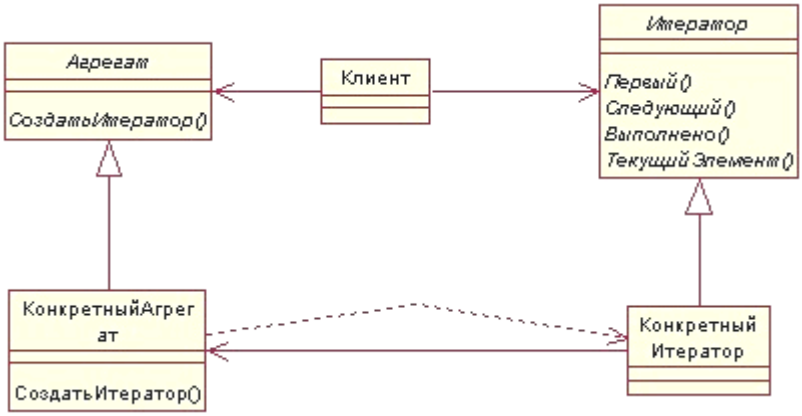
Краткое описание поведенческих паттернов проектирования

Интерпретатор (Interpreter) – GoF

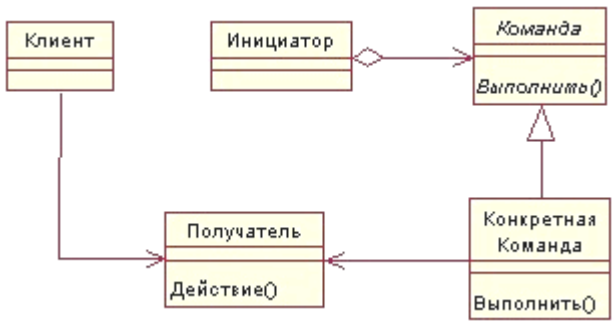
Проблема	Имеется часто встречающаяся, подверженная изменениям задача.
Решение	Создать интерпретатор, который решает данную задачу.
Пример	Задача поиска строк по образцу может быть решена посредством создания интерпретатора, определяющего грамматику языка. "Клиент" строит предложение в виде абстрактного синтаксического дерева, в узлах которого находятся объекты классов "НетерминальноеВыражение" и "ТерминальноеВыражение" (рекурсивное), затем "Клиент" инициализирует контекст и вызывает операцию Разобрать(Контекст). На каждом узле типа "НетерминальноеВыражение" определяется операция Разобрать для каждого подвыражения. Для класса "ТерминальноеВыражение" операция Разобрать определяет базу рекурсии. "АбстрактноеВыражение" определяет абстрактную операцию Разобрать, общую для всех узлов в абстрактном синтаксическом дереве. "Контекст" содержит информацию, глобальную по отношению к интерпретатору. 
Преимущества	Грамматику становится легко расширять и изменять, реализации классов, описывающих узлы абстрактного синтаксического дерева похожи (легко кодируются). Можно легко изменять способ вычисления выражений.
Недостатки	Сопровождение грамматики с большим числом правил затруднительно.

Итератор (Iterator) или Курсор (Cursor) - GoF

Проблема	Составной объект, например, список, должен предоставлять доступ к своим элементам (объектам), не раскрывая их внутреннюю структуру, причем перебирать список требуется по-разному в зависимости от задачи.
Решение	Создается класс "Итератор", коорый определяет интерфейс для доступа и перебора элементов, "КонкретныйИтератор" реализует ин-

	терфейсклассa "Итератор" и следит за текущей позицией при обходе "Агрегата". "Агрегат" определяет интерфейс для создания объекта - итератора. "КонкретныйАгрегат" реализует интерфейс создания итератора и возвращает экземпляр класса "КонкретныйИтератор", "КонкретныйИтератор" отслеживает текущий объект в агрегате и может вычислить следующий объект при переборе. 
Преимущества	Поддерживает различные способы перебора агрегата, одновременно могут быть активны несколько переборов.

Команда (Command), Действие (Action) или Транзакция (Транзакция) - GoF

Проблема	Необходимо послать объекту запрос, не зная о том, выполнение какой операции запрошено и кто будет получателем.
Решение	Инкапсулировать запрос как объект. "Клиент" создает объект "КонкретнаяКоманда", который вызывает операции получателя для выполнения запроса, "Инициатор" отправляет запрос, выполоняя операцию "Команды" Выполнить(). "Команда" объявляет интерфейс для выполнения операции, "КонкретнаяКоманда" определяет связь между объектом "Получатель" и операцией Действие(), и, кроме того, реализует операцию Выполнить() путем вызова соответствующих операций объекта "Получатель". "Клиент" создает экземпляр класса "КонкретнаяКоманда" и устанавливает его получателя, "Инициатор" обращается к команде для выполнения запроса, "Получатель" (любой класс) располагает информацией о способах выполнения операций, необходимых для выполнения запроса. 

Преимущества	Паттерн "Команда" разрывает связь между объектом, инициирующим операции, и объектом, имеющим информацию о том, как ее выполнить, кроме того создается объект "Команда", который можно расширять и манипулировать им как объектом.

Наблюдатель (Observer), Опубликовать - подписаться (Publish - Subscribe) или Delegation Event Model - GoF

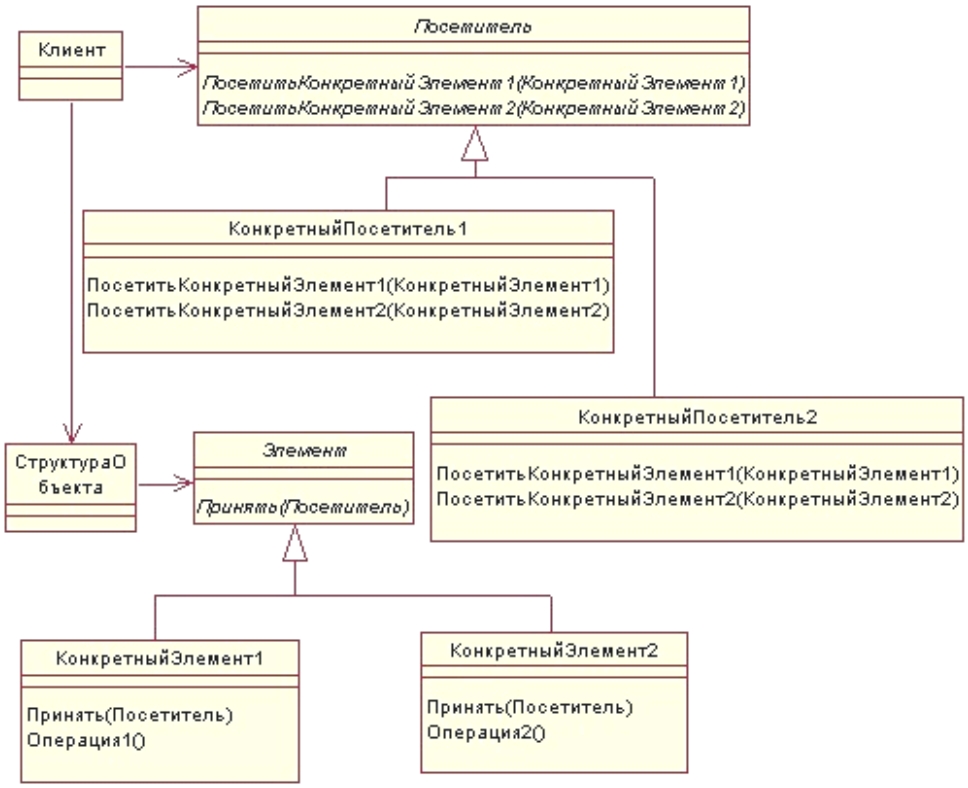
Проблема	Один объект ("Подписчик") должен знать об изменении состояний или некоторых событиях другого объекта. При этом необходимо поддерживать низкий уровень связывания с объектом - "Подписчиком".
Решение	Определить интерфейс "Подписки". Объекты - подписчики реализуют этот интерфейс и динамически регистрируются для получении информации о некотором событии. Затем при реализации условленного события оповещаются все объекты - подписчики.

Не разговаривайте с неизвестными (Don't talk to strangers) - GRASP

Проблема	Обеспечить связь клиентского объекта с непрямыми объектами (то есть известными другим объектам, а не самому клиенту).
Решение	Необходимо избегать проектных решений, предполагающих передачу сообщений с удаленными непрямыми объектами (незнакомцами). Решением может быть частный случай паттерна "Устойчивый к изменениям", см. 3.1.9. Прямым объектом потребуются новые операции.
Преимущества	Обеспечивает устойчивость системы к изменению структуры обьектов.

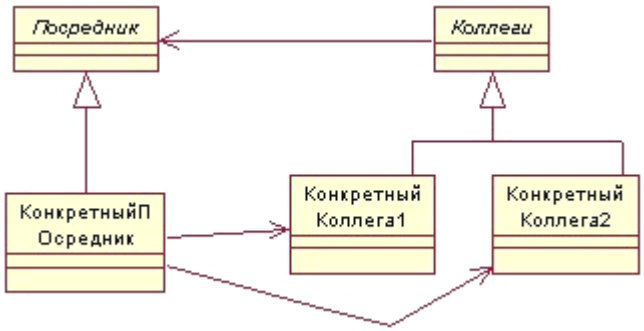
Посетитель (Visitor) - GoF

Проблема	Над каждым объектом некоторой структуры выполняется операция. Определить новую операцию, не изменяя классы обьектов.
Решение	Клиент, использующий данный паттерн, должен создать объект класса "КонкретныйПосетитель", а затем посетить каждый элемент структуры. "Посетитель" объявляет операцию "Посетить" для каждого класса "КонкретныйЭлемент" (имя и сигнатура данной операции идентифицируют класс, элемент которого посещает "Посетитель" - то есть, посетитель может обращаться к элементу напрямую). "КонкретныйПосетитель" реализует все операции, обьявленные в классе "Посетитель". Каждая операция реализует фрагмент алгоритма, определенного для класса соответствующего объекта в структуре. Класс "КонкретныйПосетитель"предоставляет контекст для этого алгоритма и сохраняет его локальное состояние. "Элемент" определяет операцию "Принять", которая принимает "Посетителя" в качестве аргумента, "КонкретныйЭлемент" реализует операцию "Принять", которая принимает "Посетителя" в качестве аргумента. "СтруктураОбьекта" может перечислить свои аргументы и предоставить посетителю

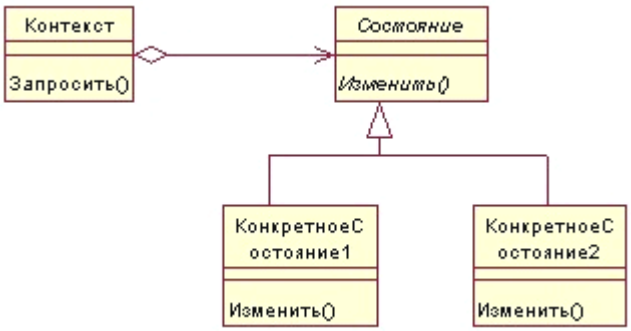
	высокоуровневый интерфейс для посещения своих элементов. 
Рекомендации	Логично использовать, если в структуре присутствуют объекты многих классов с различными интерфейсами, и необходимо выполнить над ними операции, зависящие от конкретных классов, или если классы, устанавливающие структуру обьектов изменяются редко, но новые операции над этой структурой добавляются часто.
Преимущества	Упрощается добавление новых операций, объединяет родственные операции в классе "Посетитель".
Недостатки	Затруднено добавление новых классов "КонкретныйЭлемент", поскольку требуется объявление новой абстрактной операции в классе "Посетитель".

Посредник (Mediator) - GoF

Проблема	Обеспечить взаимодействие множества обьектов, сформировав при этом слабую связанность и избавив объекты от необходимости явно ссылаться друг на друга.
Решение	Создать объект инкапсулирующий способ взаимодействия множества обьектов.
Пример	"Посредник" определяет интерфейс для обмена информацией с объектами "Коллеги", "КонкретныйПосредник" координирует действия обьектов "Коллеги". Каждый класс "Коллеги" знает о своем объекте "Посредник", все "Коллеги" обмениваются информацией только с посредником, при его отсутствии им пришлось бы обмениваться

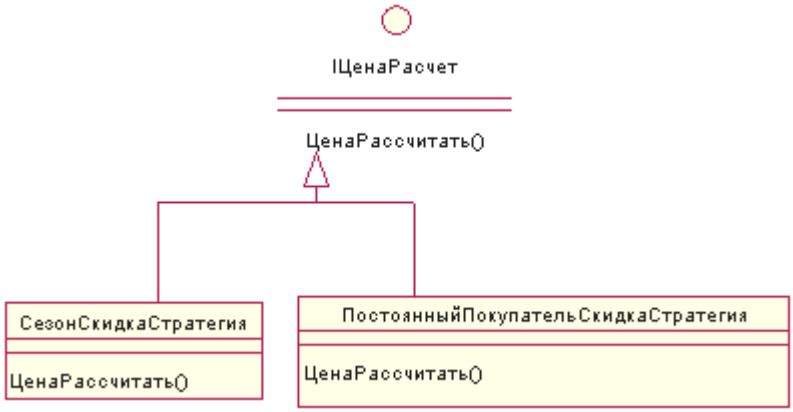
	информацией напрямую. "Коллеги" посылают запросы посреднику и получают запросы от него. "Посредник" реализует кооперативное поведения, пересылая каждый запрос одному или нескольким "Коллегам". 
Преимущества	Устраняется связанность между "Коллегами", централизуется управление.

Состояние (State) - GoF

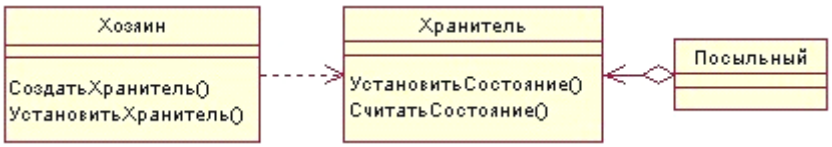
Проблема	Варьировать поведение объекта в зависимости от его внутреннего состояния
Решение	Класс "Контекст" делегирует зависящие от состояния запросы текущему объекту "КонкретноеСостояние" (хранит экземпляр подкласса "КонкретноеСостояние", которым определяется текущее состояние), и определяет интерфейс, представляющий интерес для клиентов. "КонкретноеСостояние" реализует поведение, ассоциированное с неким состоянием объекта "Контекст". "Состояние" определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным экземпляром "Контекста". 
Преимущества	Локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям, переходы между состояниями становятся явными.

Стратегия (Strategy) - GoF

Проблема	Спроектировать изменяемые, но надежные алгоритмы или стратегии.
Решение	Определить для каждого алгоритма или стратегии отдельный класс со стандартным интерфейсом.

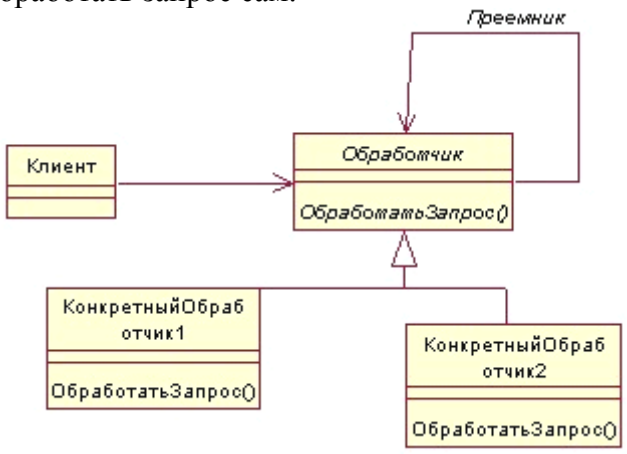
Пример	Обеспечение сложной логики вычисления стоимости товаров с учетом сезонных скидок, скидок постоянным клиентам и т. п. Данная стратегия может изменяться.  Создается несколько классов "Стратегия", каждый из которых содержит один и тот же полиморфный метод "ЦенаРассчитать". В качестве параметров в этот метод передаются данные о продаже. Объект стратегии связывается с контекстным объектом (тем объектом, к которому применяется алгоритм).

Хранитель (Memento) - GoF

Проблема	Необходимо зафиксировать поведение объекта для реализации, например, механизма отката.
Решение	Зафиксировать и вынести (не нарушая инкапсуляции) за пределы объекта его внутреннее состояние так, чтобы впоследствии можно было восстановить в нем объект. "Хранитель" сохраняет внутреннее состояние объекта "Хозяин" и запрещает доступ к себе всем другим объектам кроме "Хозяина", который имеет доступ ко всем данным для восстановления в прежнем состоянии. "Посыльный" может лишь передавать "Хранителя" другим объектам. "Хозяин" создает "Хранителя", содержащего снимок текущего внутреннего состояния и использует "Хранитель" для восстановления внутреннего состояния. "Посыльный" отвечает за сохранение "Хранителя", при этом не производит никаких операций над "Хранителем" и не исследует его внутреннее содержимое. "Посыльный" запрашивает "Хранитель" у "Хозяина", некоторое время держит его у себя, а затем возвращает "Хозяину". 
Преимущества	Не раскрывается информация, которая доступна только "Хозяину", упрощается структура "Хозяина".
Недостатки	С использованием "Хранителей" могут быть связаны значительные издержки, если "Хозяин" должен копировать большой объем инфор-

	мации, или если копирование должно проводиться часто.

Цепочка обязанностей (Chain of Responsibility) - GoF

Проблема	Запрос должен быть обработан несколькими объектами.
Рекомендации	Логично использовать данный паттерн, если имеется более одного объекта, способного обработать запрос и обработчик заранее неизвестен (и должен быть найден автоматически) или если весь набор обьектов, которые способны обработать запрос, должен задаваться автоматически.
Решение	Связать объекты - получатели запроса в цепочку и передать запрос вдоль этой цепочки, пока он не будет обработан. "Обработчик" определяет интерфейс для обработки запросов, и, возможно, реализует связь с преемником, "КонкретныйОбработчик" обрабатывает запрос, за который отвечает, имеет доступ к своему преемнику ("КонкретныйОбработчик" направляет запрос к своему преемнику, если не может обработать запрос сам. 
Преимущества	Ослабляется связанность (объект не обязан "знать", кто именно обработает его запрос).
Недостатки	Нет гарантий, что запрос будет обработан, поскольку он не имеет явного получателя.

Шаблонный метод (Template Method) - GoF

Проблема	Определить алгоритм и реализовать возможность переопределения некоторых шагов алгоритма для подклассов (без изменения общей структуры алгоритма.
Решение	"АбстрактныйКласс" определяет абстрактные Операции(), замещаемые в конкретных подклассах для реализации шагов алгоритма, и реализует ШаблонныйМетод(), определяющий "скелет" алгоритма. "КонкретныйКласс" релизует Операции(), выполняющие шаги алгоритма способом, который зависит от подкласса. "КонкретныйКласс" предполагает, что инвариантные шаги алгоритма будут выполнены в "АбстрактномКлассе".

Высокое зацепление (High Cohesion) - GRASP

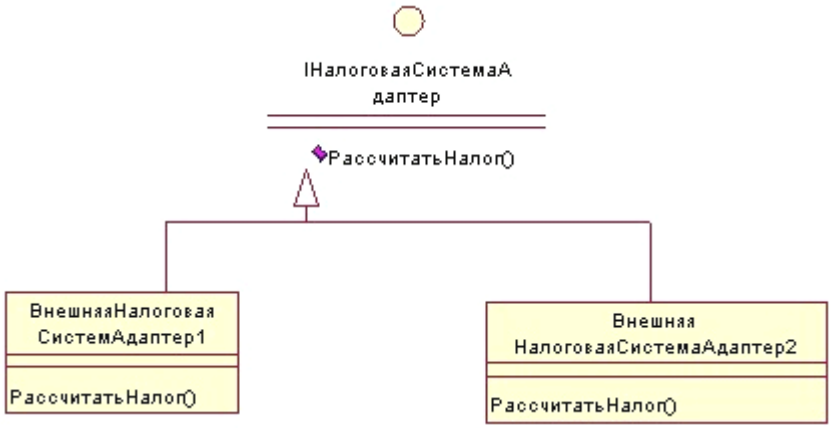
Проблема	Необходимо обеспечить выполнение объектами разнородных функций.
Решение	Обеспечить распределение обязанностей с высоким зацеплением.
Пример	Если в примере для паттерна "Низкая связанность", на класс "Регистрация" возлагать все новые и новые системные функции, связанные с системными операциями, то данный класс будет слишком перегружен и будет обладать низкой степенью зацепления. Второй рисунок для примера Low Coupling обладает более высоким уровнем зацепления и низким уровнем связывания (он является более предпочтительным).
Преимущества	Классы с высокой степенью зацепления просты в поддержке и повторном использовании.
Недостатки	Иногда бывает неоправданно использовать высокое зацепление для распределенных серверных обьектов. В этом случае для обеспечения быстродействия необходимо создать несколько более крупных серверных обьектов со слабым зацеплением.

Контроллер (Controller) - GRASP

Проблема	"Кто" должен отвечать за обработку входных системных событий?
Решение	Обязанности по обработке системных сообщений делегируются специальному классу. Контроллер - это объект, который отвечает за обработку системных событий и не относится к интерфейсу пользователя. Контроллер определяет методы для выполнения системных операций.
Рекомендации	Для различных прецедентов логично использовать разные контроллеры (контроллеры прецедентов) - контроллеры не должны быть перегружены. Внешний контроллер представляет всю систему целиком, его можно использовать, если он будет не слишком перегруженным (то есть, если существует лишь несколько системных событий).

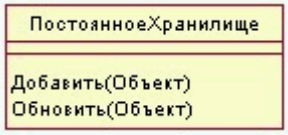
Преимущества	Удобно накапливать информацию о системных событиях (в случае, если системные операции выполняются в некоторой определенной последовательности). Улучшаются условия для повторного использования компонентов (системные события обрабатываются Контроллером а не элементами интерфейса пользователя).
Недостатки	Контроллер может оказаться перегружен.

Полиморфизм (Polymorphism) - GRASP

Проблема	Как обрабатывать альтернативные варианты поведения на основе типа? Как заменять подключаемые компоненты системы?
Решение	Обязанности распределяются для различных вариантов поведения с помощью полиморфных операций для этого класса. Каждая внешняя система имеет свой интерфейс.
Пример	Интеграция разрабатываемой системы с различными внешними системами учета налогов. Используются локальные программные объекты, обеспечивающие адаптацию (Адаптеры), при отправке сообщения к такому объекту выполняется обращение к внешней системе с использованием ее собственного программного интерфейса.  Использование полиморфизма оправдано для адаптации к различным внешним системам.
Преимущества	Впоследствии легко расширять и модернизировать систему.
Недостатки	Не следует злоупотреблять добавлением интерфейсов с применением принципа полиморфизма с целью обеспечения дееспособности системы в неизвестных заранее новых ситуациях.

Искусственный (Pure Fabrication) - GRASP

Проблема	Какой класс должен обеспечивать реализацию паттернов "Высокое зацепление", и "Низкая связанность"?
Решение	Присвоить группу обязанностей с высокой степенью зацепления классу, который не представляет конкретного понятия из предметной области (синтезировать искусственную сущность для обеспече-

	ния высокого зацепления и слабого связывания).
Пример	См. пример паттерна "Информационный эксперт". Какой класс должен сохранять экземпляры класса "Продажа" в реляционной базе данных? Если возложить эту обязанность на класс "Продажа", то будем иметь низкую степень зацепления и высокую степень связывания (поскольку класс "Продажа" должен быть связан с интерфейсом реляционной базы данных. Хранение обьектов в реляционной базе данных - это общая задача, которую придется решать для многих классов. Решением данной проблемы будет создание нового класса "ПостоянноеХранилище", ответственного за сохранение обьектов некоторого вида в базе данных. 
Преимущества	Класс "ПостоянноеХранилище" будет обладать низкой степенью связывания и высокой степенью зацепления.
Недостатки	Данным паттерном не следует злоупотреблять иначе все функции системы превратятся в объекты.

Перенаправление (Indirection) - GRASP

Проблема	Как перераспределить обязанности обьектов, чтобы обеспечить отсутствие прямого связывания?
Решение	Присвоить обязанности по обеспечению связи между службами или компонентами промежуточному объекту.
Пример	См. пример к паттерну "Искусственный" Класс "Хранилище" выступает в роли промежуточного звена между классом "Продажа" и базой данных.

Краткое описание порождающих паттернов проектирования

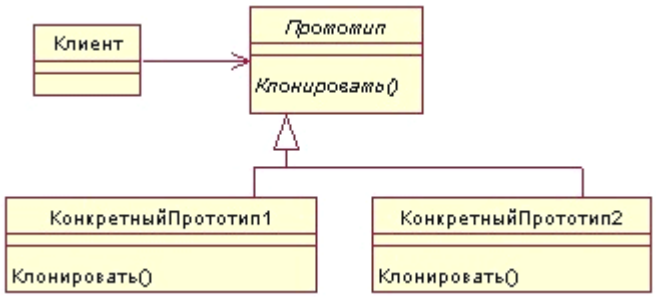
Абстрактная фабрика (Abstract Factory, Factory), др. название Инструментарий (Kit) - GoF

Проблема	Создать семейство взаимосвязанных или взаимозависимых обьектов (не специфицируя их конкретных классов).
Решение	Создать абстрактный класс, в котором объявлен интерфейс для создания конкретных классов.
Пример	Какой класс должен отвечать за создание обьектов - адаптеров при использовании паттерна "Адаптер". Если подобные объекты создаются неким объектом уровня предметной области, то будет нарушен принцип разделения обязанностей.
Преимущества	Изолирует конкретные классы. Поскольку "Абстрактная фабрика" инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Упрощена замена "Абстрактной фабрики", поскольку она используется в приложении только один раз при инстанцировании.
Недостатки	Интерфейс "Абстрактной фабрики" фиксирует набор обьектов, которые можно создать. Расширение "Абстрактной фабрики" для изготовления новых обьектов часто затруднительно.

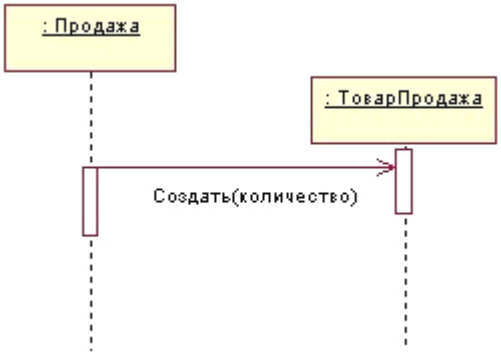
Одиночка (Singleton) - GoF

Проблема	Какой специальный класс должен создавать "Абстрактную фабрику", и как получить к ней доступ? Необходим лишь один экземпляр специального класса, различные объекты должны обращаться к этому экземпляру через единственную точку доступа.
Решение	Создать класс и определить статический метод класса, возвращающий этот единственный объект.
Рекомендации	Разумнее создавать именно статический экземпляр специального класса, а не объявить требуемые методы статическими, поскольку при использовании методов экземпляра можно применить механизм наследования и создавать подклассы. Статические методы в языках программирования не полиморфны и не допускают перекрытия в производных классах. Решение на основе создания экземпляра является более гибким, поскольку впоследствии может потребоваться уже не единственный экземпляр объекта, а несколько.

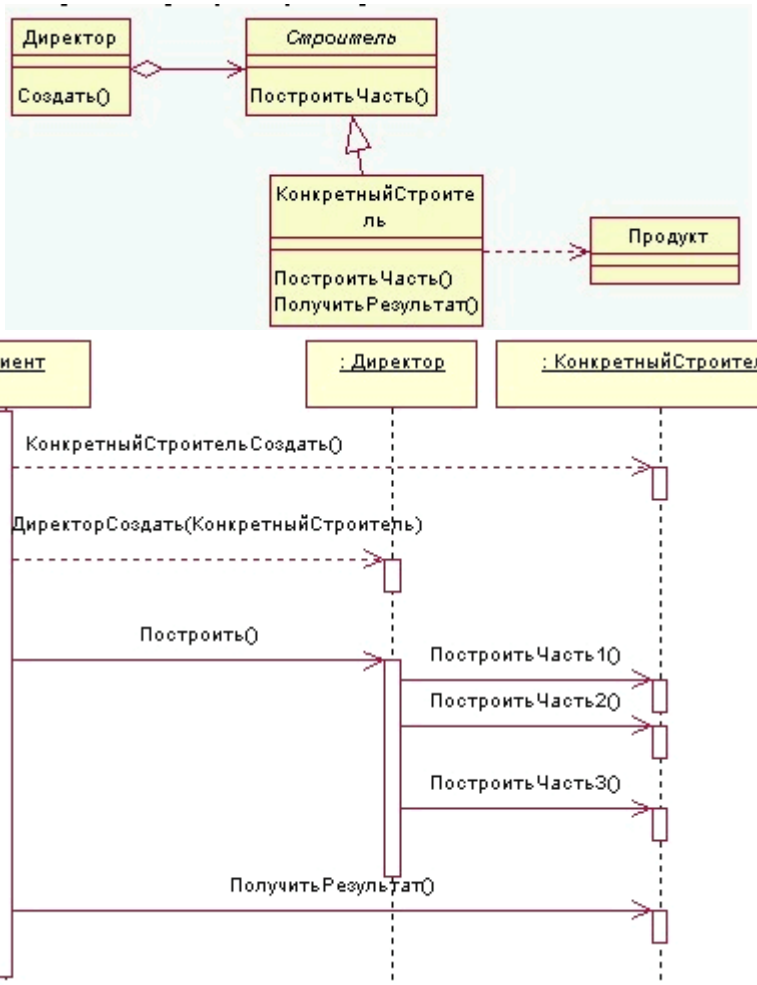
Прототип (Prototype) - GoF

Проблема	Система не должна зависеть от того, как в ней создаются, компонуются и представляются объекты.
Решение	Создавать новые объекты с помощью паттерна - прототипа. "Прототип" объявляет интерфейс для клонирования самого себя. "Клиент" создает новый объект, обращаясь к "Прототипу" с запросом клонировать "Прототип". 

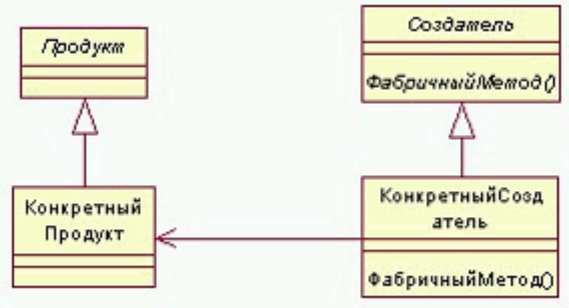
Создатель экземпляров класса (Creator) - GRASP

Проблема	"Кто" должен отвечать за создание экземпляров класса
Решение	Назначить классу B обязанность создавать объекты другого класса A
Рекомендации	Логично использовать паттерн если класс B содержит, агрегирует, активно использует и т.п. объекты класса A.
Пример	См. пример к паттерну "Информационный эксперт" в, необходимо определить, какой объект должен отвечать за создание экземпляра "ТоварПродажа". Логично, чтобы это был объект "Продажа", поскольку он содержит (агрегирует) несколько обьектов "ТоварПродажа". 
Преимущества	Использование этого паттерна не повышает связанности, поскольку созданный класс, как правило, виден только для класса - создателя.
Недостатки	Если процедура создания объекта достаточно сложная (например выполняется на основе некоего внешнего условия), логично использовать паттерн "Абстрактная Фабрика", , то есть, делегировать обязанность создания обьектов специальному классу.

Строитель (Builder) - GoF

Проблема	Отделить конструирование сложного объекта от его представления, так чтобы в результате одного и того же конструирования могли получаться различные представления. Алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой.
Решение	"Клиент" создает объект - распорядитель "Директор" и конфигурирует его объектом - "Строителем". "Директор" уведомляет "Строителя" о том, что нужно построить очередную часть "Продукта". "Строитель" обрабатывает запросы "Директора" и добавляет новые части к "Продукту", затем "Клиент" забирает "Продукт" у "Строителя". 
Преимущества	Объект "Строитель" предоставляет объекту "Директор" абстрактный интерфейс для конструирования "Продукта", за которым может скрыть представление и внутреннюю структуру продукта, и, кроме того, процесс сборки "продукта". Для изменения внутреннего представления "Продукта" достаточно определить новый вид "Строителя". Данный паттерн изолирует код, реализующий создание объекта и его представление.

(Фабричный метод) Factory Method или Виртуальный конструктор (Virtual Constructor) - GoF

Проблема	Определить интерфейс для создания объекта, но оставить подклассам решение о том, какой класс инстанцировать, то есть, делегировать инстанцирование подклассам.
Решение	Абстрактный класс "Создатель" объявляет ФабричныйМетод, возвращающий объект типа "Продукт" (абстрактный класс, определяющий интерфейс обьектов, создаваемых фабричным методом). "Создатель также может определить реализацию по умолчанию ФабричногоМетода, который возвращает "КонкретныйПродукт". "КонкретныйСоздатель" замещает ФабричныйМетод, возвращающий объект "КонкретныйПродукт". "Создатель" "полагается" на свои подклассы в определении ФабричногоМетода, возвращающего объект "КонкретныйПродукт". 
Преимущества	Избавляет проектировщика от необходимости встраивать в код зависящие от приложения классы.
Недостатки	Возникает дополнительный уровень подклассов.

Учебное издание

Артеменко Максим Александрович
Лебеденко Алексей Владимирович

ОСНОВЫ ЗАЩИТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ВВЕДЕНИЕ В PYTHON

Учебное пособие

Редактор - О.В. Дмитриева

Изд. № 128/18. Объем 5 п.л.
РИИЦМ ФГАОУ ВО «Севастопольский государственный университет»