Объектно-ориентированный подход к проектированию программ

Д.Н. Лавров
2018

Объектная модель - концептуальная основа моделирования
Основные понятия:

- объекты и атрибуты
- целое и часть
- классы и экземпляры

- Объект - осязаемая реальность (tangible entity) - предмет или явление, имеющие четко определяемое поведение

- Класс - множество объектов, связанных общностью структуры и поведения. Класс инкапсулирует (объединяет) в себе данные (атрибуты) и поведение (операции)

К числу свойств относятся присущие объекту или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Эти свойства принято называть ***атрибутами*** *класса*

Наследование – это такое отношение между классами, когда один класс частично или полностью повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Наследование устанавливает между классами иерархию "общее-частное".

Связи наследования также называют обобщениями (generalization) и изображают в виде больших белых стрелок от класса-потомка к классу-предку

Наследование

- Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства.
- Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов.
- По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт:

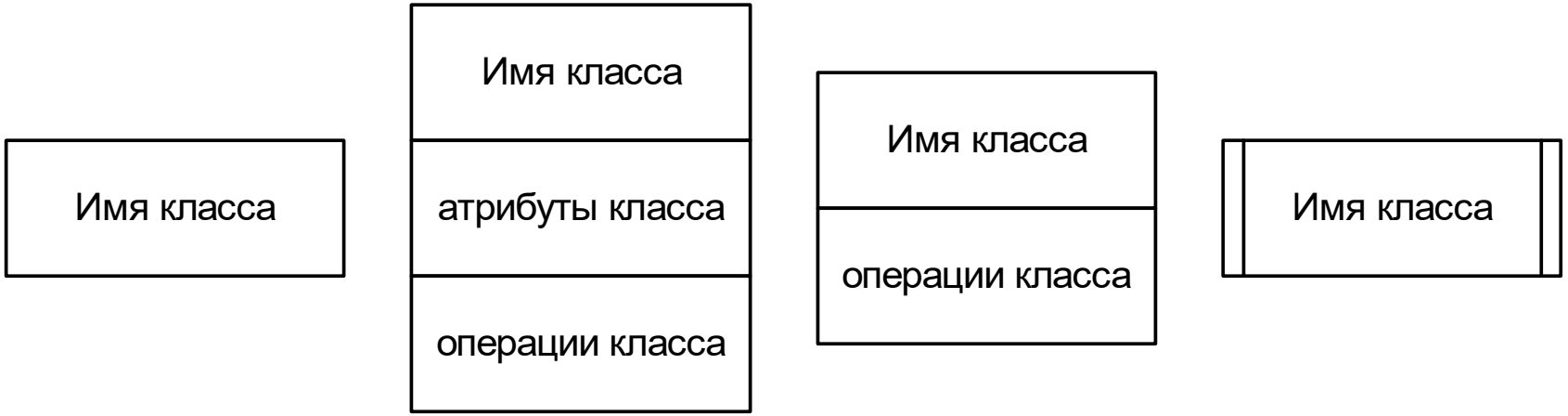
Введение в язык UML

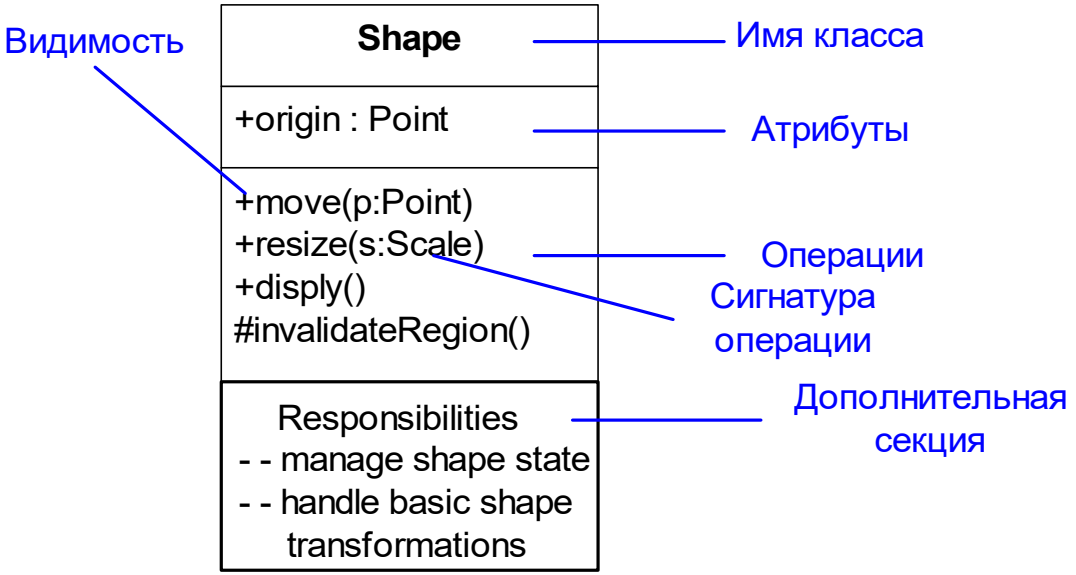
Диаграмма классов языка UML 2

Диаграмма классов — основная логическая модель системы

- *Диаграмма классов (class diagram)* — диаграмма, предназначенная для представления модели статической структуры системы в терминологии классов объектно-ориентированного подхода
- Диаграмма классов представляет собой граф, вершинами или узлами которого являются элементы типа “классификатор”, которые связаны различными типами структурных отношений
- *Классификатор (classifier)* – специальное понятие, предназначенное для классификации экземпляров, которые имеют общие характеристики

Варианты графического изображения класса на диаграмме классов





Примеры записи атрибутов

- + имяСотрудника : String {readOnly}
- ~ датаРождения : Data {readOnly}
- # /возрастСотрудника : Integer
- + номерТелефона : Integer [1..*] {unique}
- – заработнаяПлата : Currency = 500.00

Операции класса

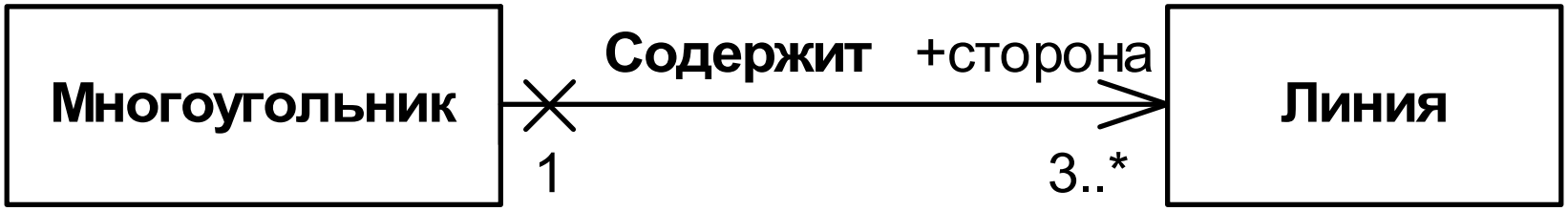
- *Операция (operation)* класса служит для представления отдельной характеристики поведения, которая является общей для всех объектов данного класса
- Общий формат записи отдельной операции класса следующий (БНФ):
- <*операция*>::=[<*видимость*>] <*имя операции*> ‘(‘ [<*список параметров*>] ‘)’ [‘:’ [<*тип возвращаемого результата*>] ‘{‘ <*свойство операции*> [‘,’ <*свойство операции*>]* ‘}’]
- Где:
- <*видимость*> ::= ‘+’ | ‘-‘ | ‘#’ | ‘~’
- <*имя операции*> (operation name) представляет собой строку текста, которая используется в качестве идентификатора соответствующей операции и поэтому должна быть уникальной для каждой операции данного класса

Примеры записи операций:

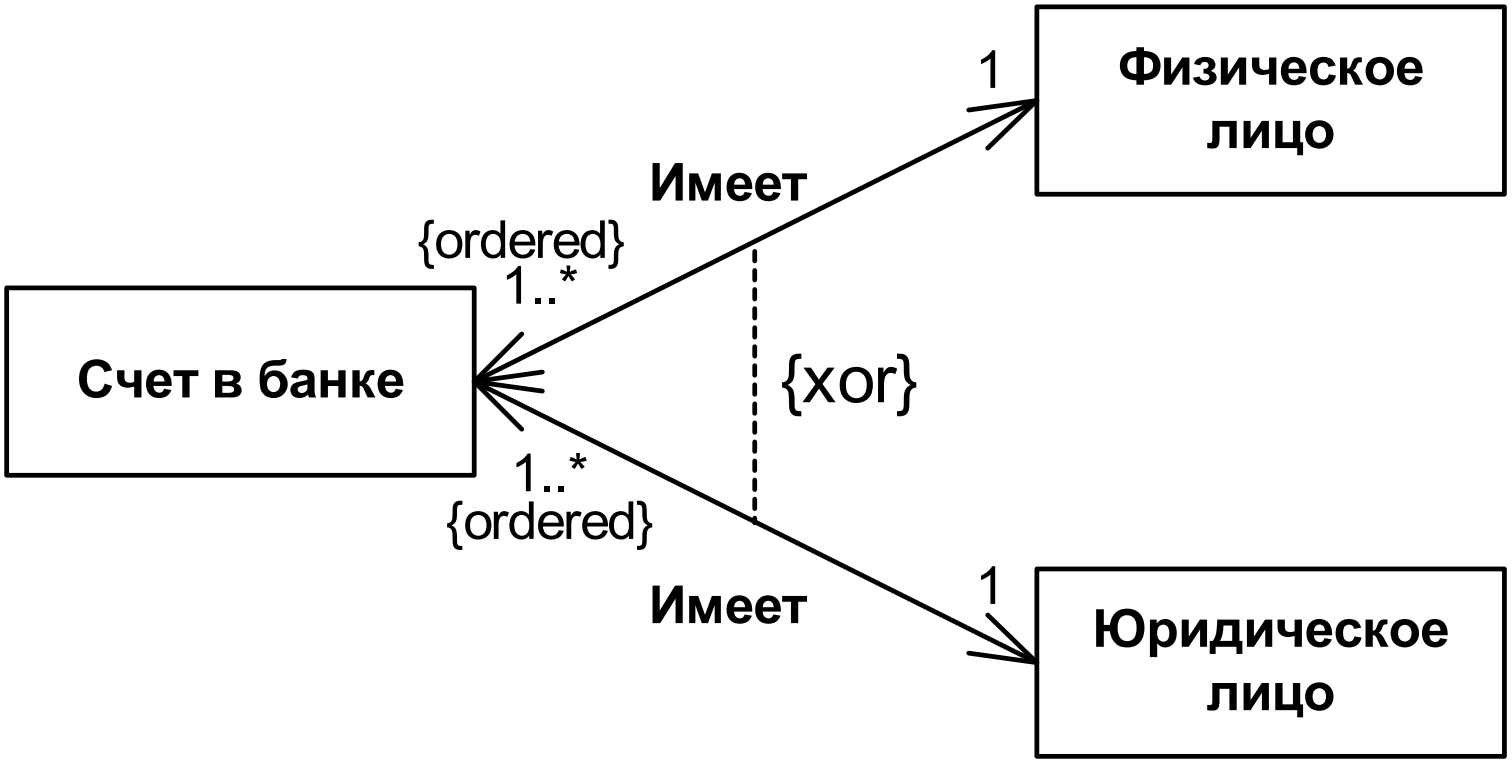
- +добавить(**in** номерТелефона : Integer [*] {unique})
- –изменить(**in** заработнаяПлата : Currency)
- +создать() : Boolean
- toString(return : String)
- toString( ) : String

Отношения на диаграмме классов

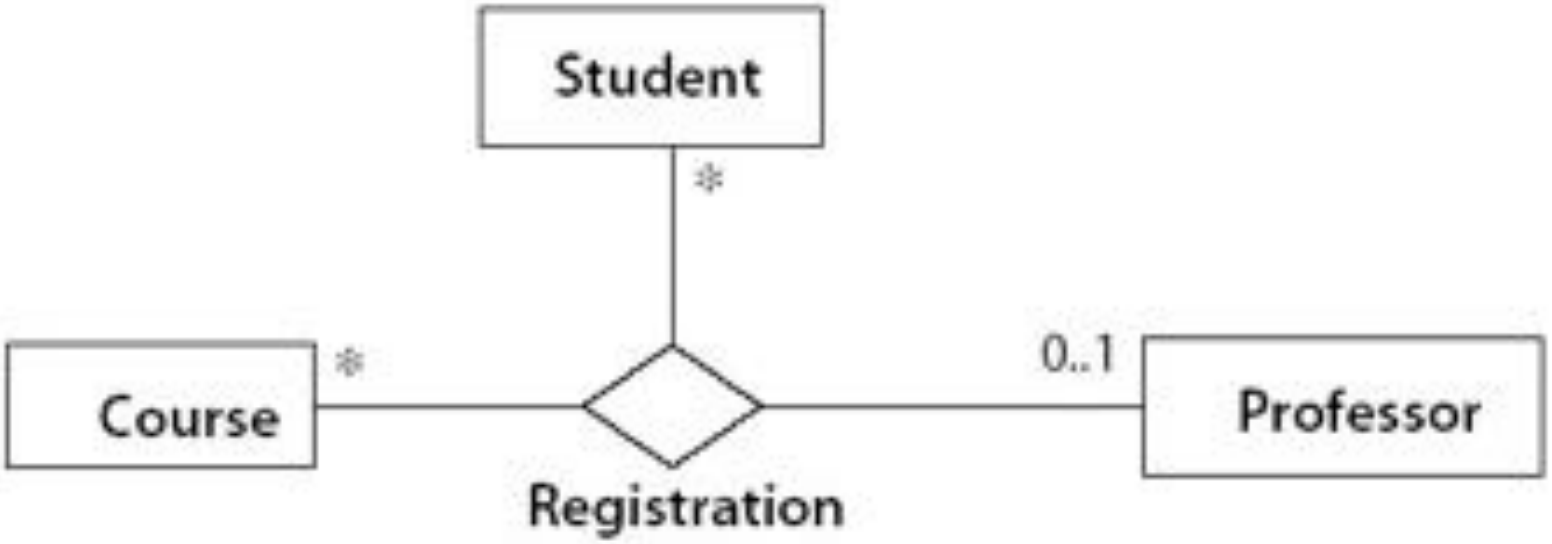
Ассоциация с навигацией и эквивалентное ему представление класса с атрибутом



Исключающая ассоциация между тремя классами

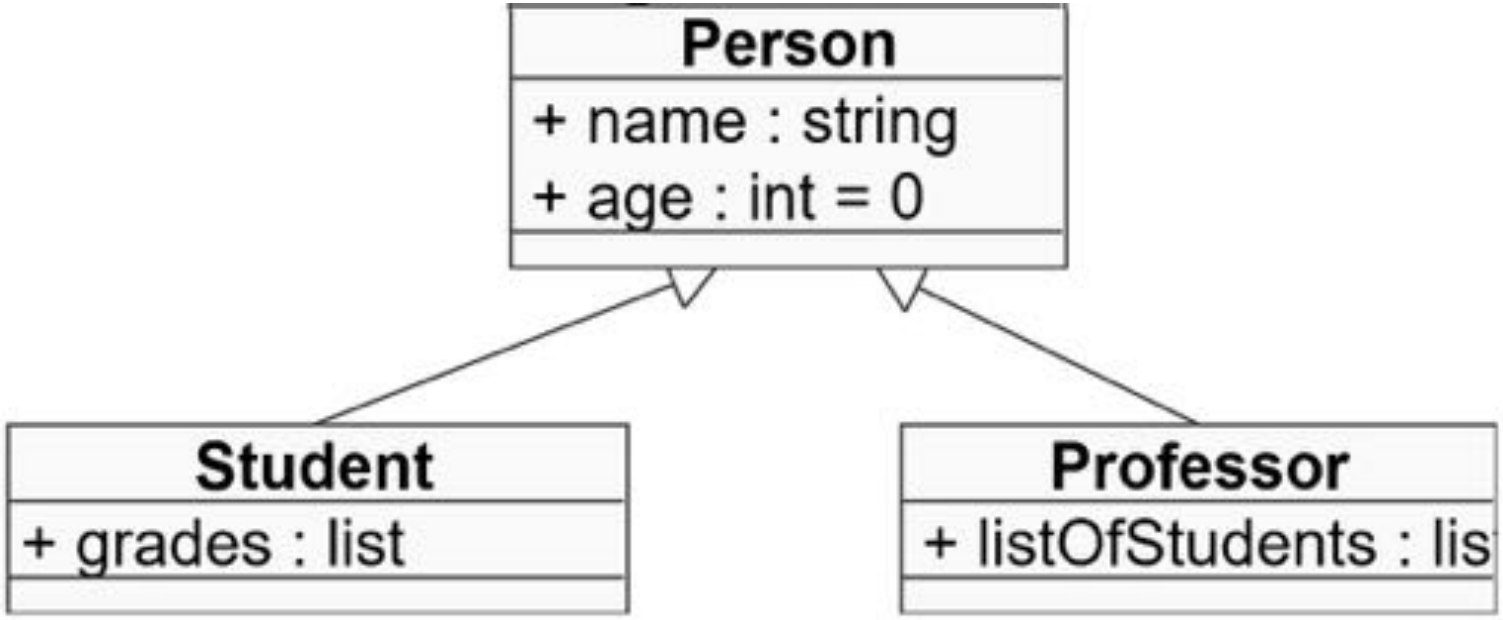


Пример тернарной ассоциации

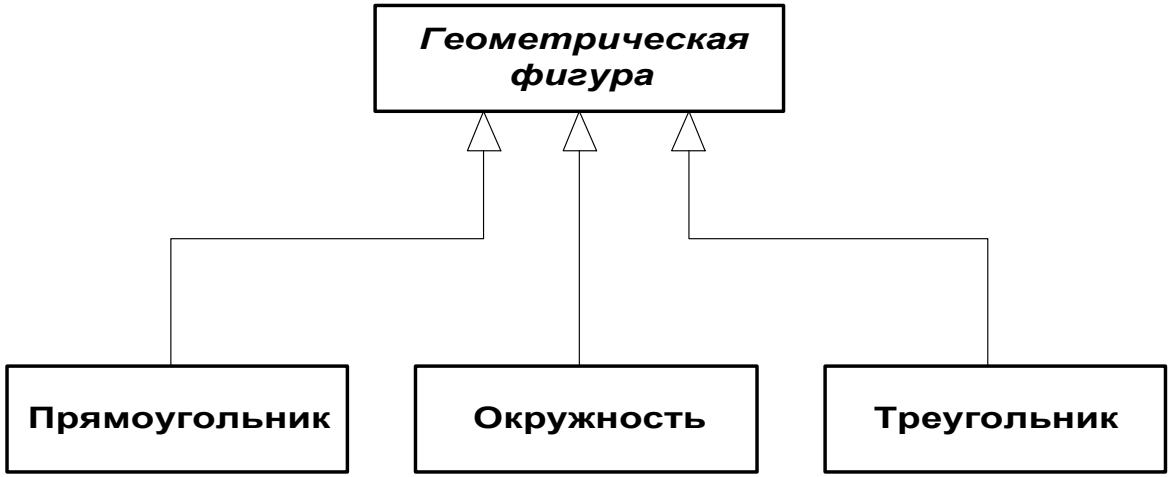


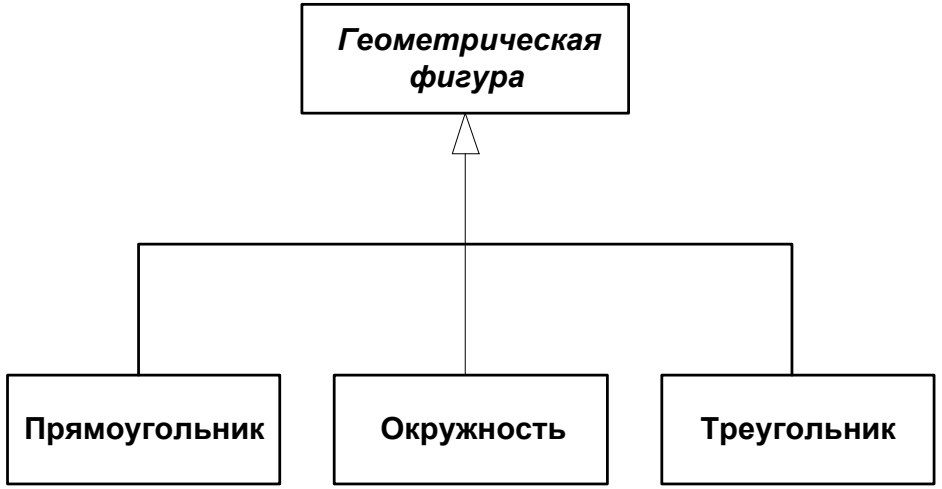
Обобщение *(generalization)*

- – таксономическое отношение между более общим классификатором (родителем или предком) и более специальным классификатором (дочерним или потомком)



Примеры отношения обобщения

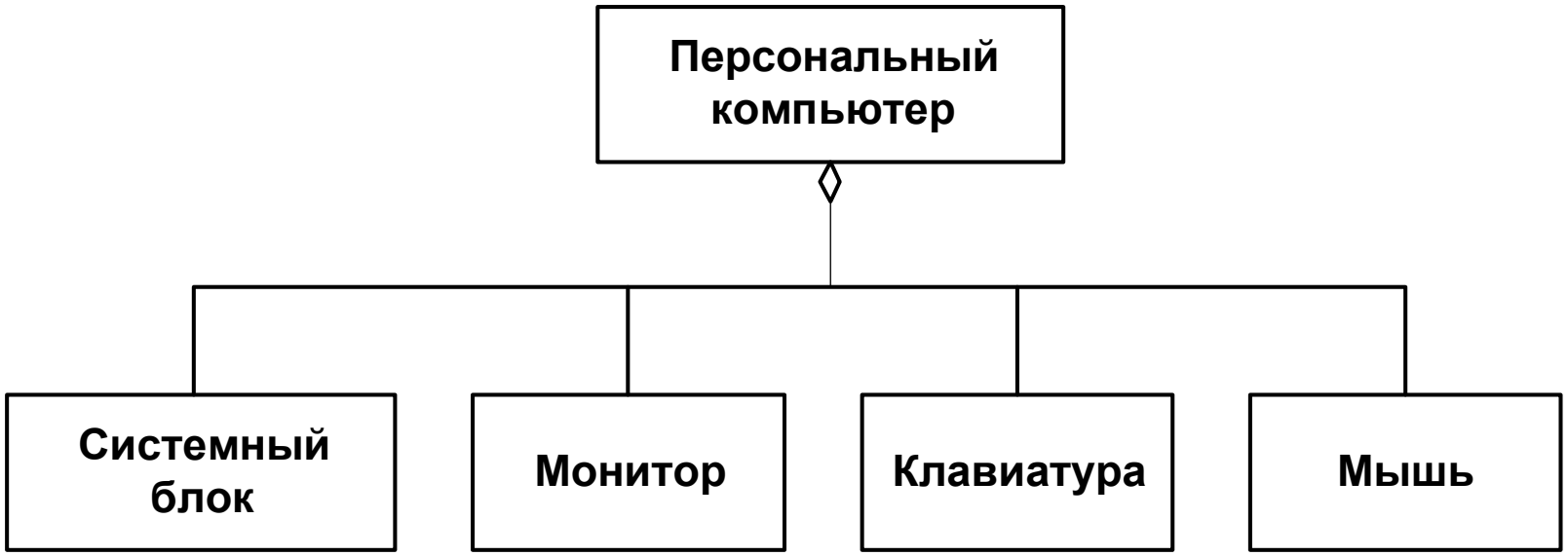




Агрегация *(aggregation)*

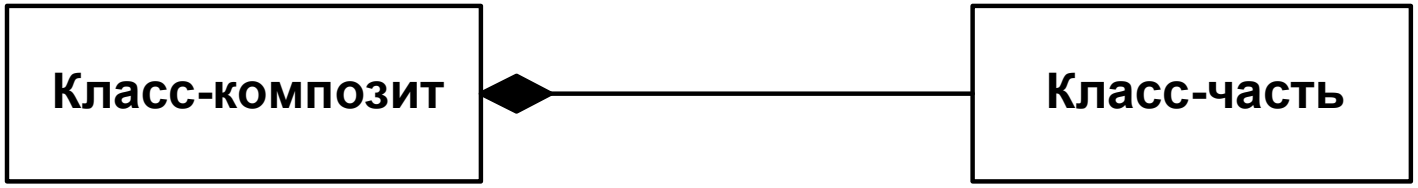
- – направленное отношение между двумя классами, предназначенное для представления ситуации, когда один из классов представляет собой некоторую сущность, которая включает в себя в качестве составных частей другие сущности

Пример отношения агрегации

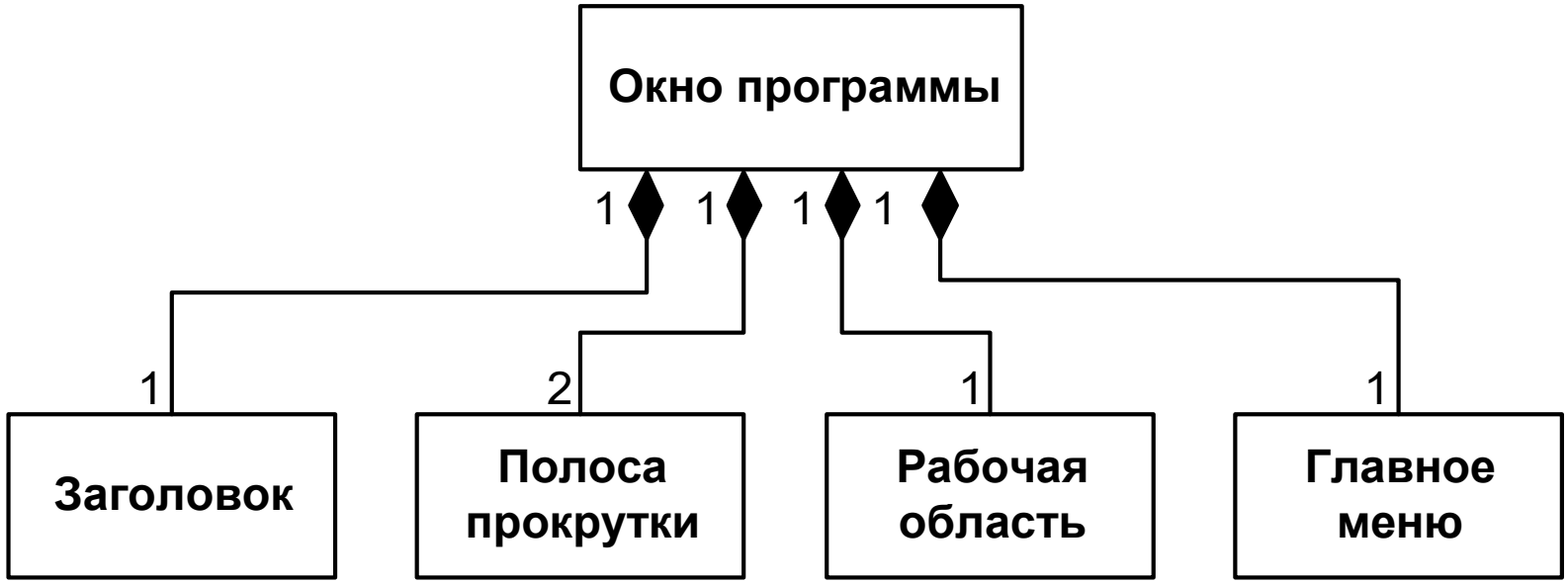


Композиция *(composition)*

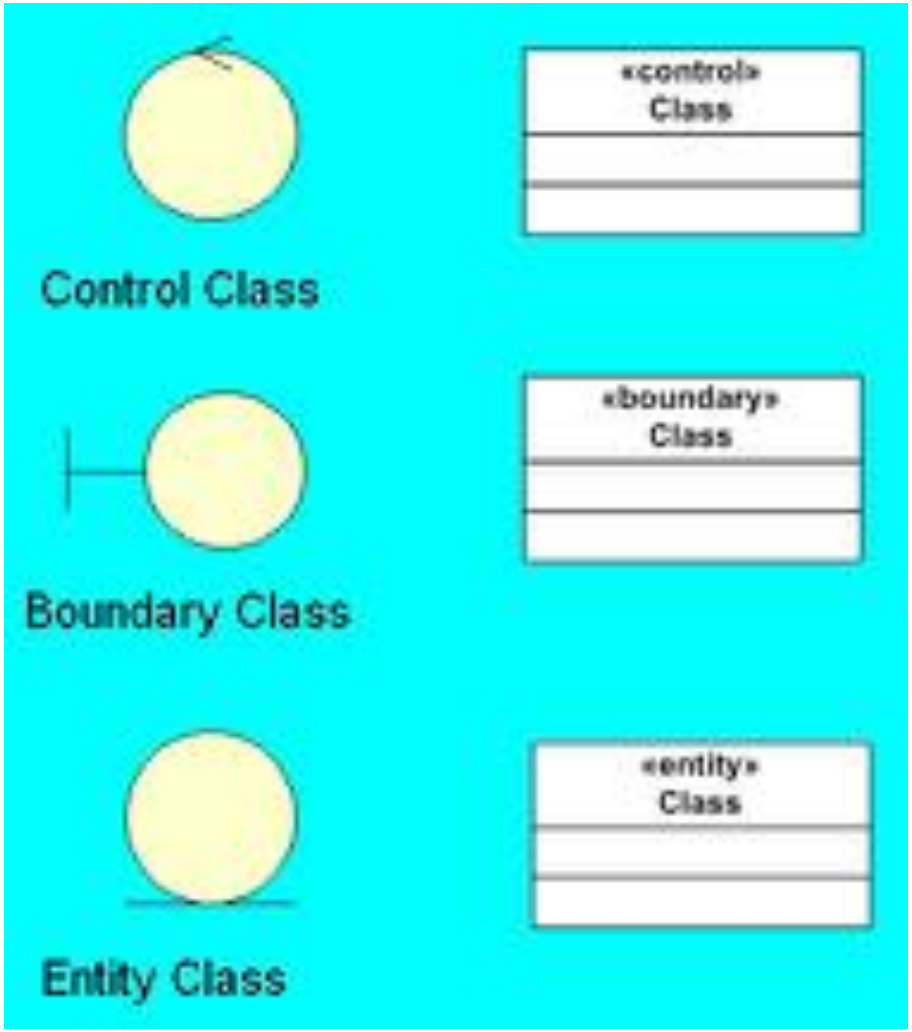
- или *композитная агрегация* предназначена для спецификации более сильной формы отношения "часть-целое", при которой с уничтожением объекта класса-контейнера уничтожаются и все объекты, являющимися его составными частями.



Пример отношения композиции

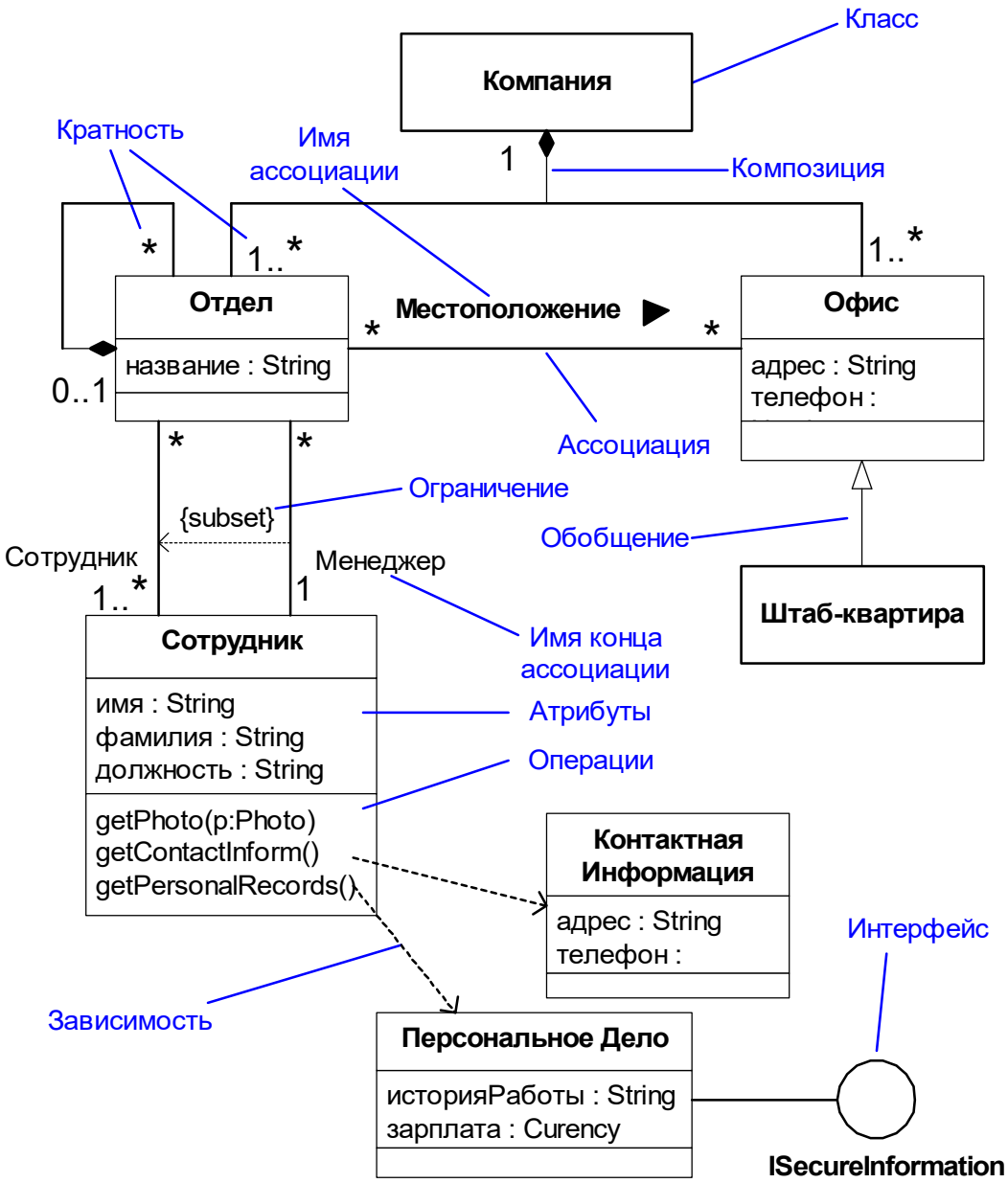


Классы в UML



- Управляющий класс отвечает за координацию действий других классов. Этому классу посылают мало сообщений, а он рассылает много сообщений
- Граничный класс располагается на границе системы с внешней средой.
- Класс-сущность содержит информацию, которая хранится постоянно и не уничтожается с выключением системы

Основные обозначения на диаграмме классов



GRASP

- **GRASP** - General Responsibility Assignment Software Patterns (основные шаблоны распределения обязанностей в программном обеспечении). Это методический подход к объектному проектированию. Эти шаблоны называют также шаблонами распределения обязанностей. Под "шаблоном" в данном контексте, да и в любом контексте, связанном с разработкой ПО, понимается именованная пара "проблема/решение", содержащая рекомендации для применения в различных конкретных ситуациях. Шаблоны GRASP относятся к этапу проектирования и отвечают за взаимосвязь объектов в системе. GRASP состоит из 5 основных и 4 дополнительных шаблонов.

Основные шаблоны

- Information Expert
- Creator
- Controller
- Low Coupling
- High Cohesion

Information Expert

- **Проблема**. Формулируется в виде вопроса: "Каков наиболее общий принцип распределения между объектами при объектно-ориентированном анализе и проектировании?".

- **Решение**. Назначайте обязанность информационному эксперту – классу, у которого имеется информация для выполнения, требуемая для выполнения объязанности.

Другие названия

- «Хранение обязанностей вместе с данными»
- «Кто знает, тот и выполняет»
- «Оживление» Animation
- «Сделай сам» Do it myself
- «Размещаете службы вместе с атрибутами»

Преимущества

- Шаблон поддерживает инкапсуляцию. Для выполнения требуемых задач объекты используют только собственные данные.
- Нужное поведение системы обеспечивается несколькими классами, содержащими требуемую информацию. Это приводит к определениям классов, которые гораздо проще понимать и поддерживать.
- Поддерживается шаблон High Cohesion

Пример

- Из жизни: Работа команды
- В программировании: Решение квадратного уравнения

Creator

- **Проблема.** Кто должен отвечать за создание нового экземпляра некоторого класса? Создание объектов является одним из наиболее стандартных видов деятельности.
- **Решение**. Назначить классу B обязанность по созданию класса A, если выполняется одно из следующих условий:
  - Класс B агрегирует A
  - Класс B содержит объекты A
  - Класс B записывает объекты A
  - Класс B активно использует объекты A
  - Класс B обладает данными инициализации, которые будут использоваться при создании A

  При этом говорят, что класс B является создателем (creator) A

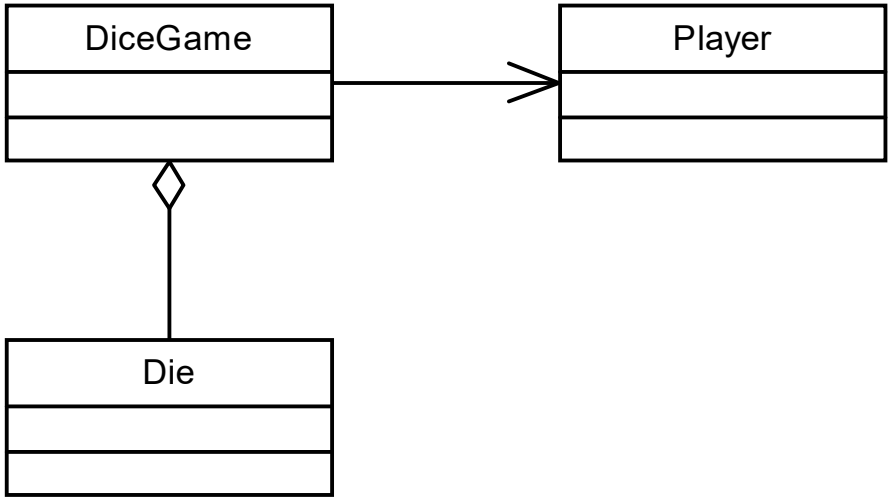
  Если выполняется несколько условий, то предпочтение отдается агрегированию и содержанию.

Преимущества

- Поддержка шаблона Low Coupling
- Способствует снижению затрат на сопровождение
- Способствует повторному использованию кода
- **Обсуждение**. Шаблон определяет, что хорошими кандидатами на создателя являются внешний контейнер и/или класс регистратор.
- **Другие названия и аналогичные принципы.** Других названий нет, но связанными шаблонами являются Low Coupling и Expert.

Пример

- Кто в "игре в кости" должен создавать сами кости?
- Наиболее подходящий кандидат – это класс DiceGame, так как именно он агрегирует кости.



Controller

- **Проблема.** Кто должен отвечать за обработку системных событий? Системное событие – это событие, генерируемое внешним исполнителем. Системная операция – это действие, выполняемое на системное событие. Контроллер – это объект, не относящийся к интерфейсу пользователя и отвечающий за обработку системных операций. Контроллер определяет методы для выполнения системных операций.
- **Решение.** Передача обязанностей по обработке системных сообщений классу удовлетворяющему одному из следующих условий:
  - Класс представляет всю систему в целом (Внешний контроллер)
  - Класс представляет всю организацию (Внешний контроллер)
  - Класс представляет активный объект из реального мира, который может участвовать в решении задачи, например человек (Контроллер роли).
  - Класс представляет искусственный обработчик всех системных событий некоторого прецедента (Контроллер прецедента)

Пример

- "Игра в кости"
- В более сложных случаях при выборе из нескольких равноправных кандидатов, необходимо проверять их с помощью других, ранее изученных шаблонов. Наиболее часто употребимые шаблоны Low Coupling и High Cohesion.

Преимущества

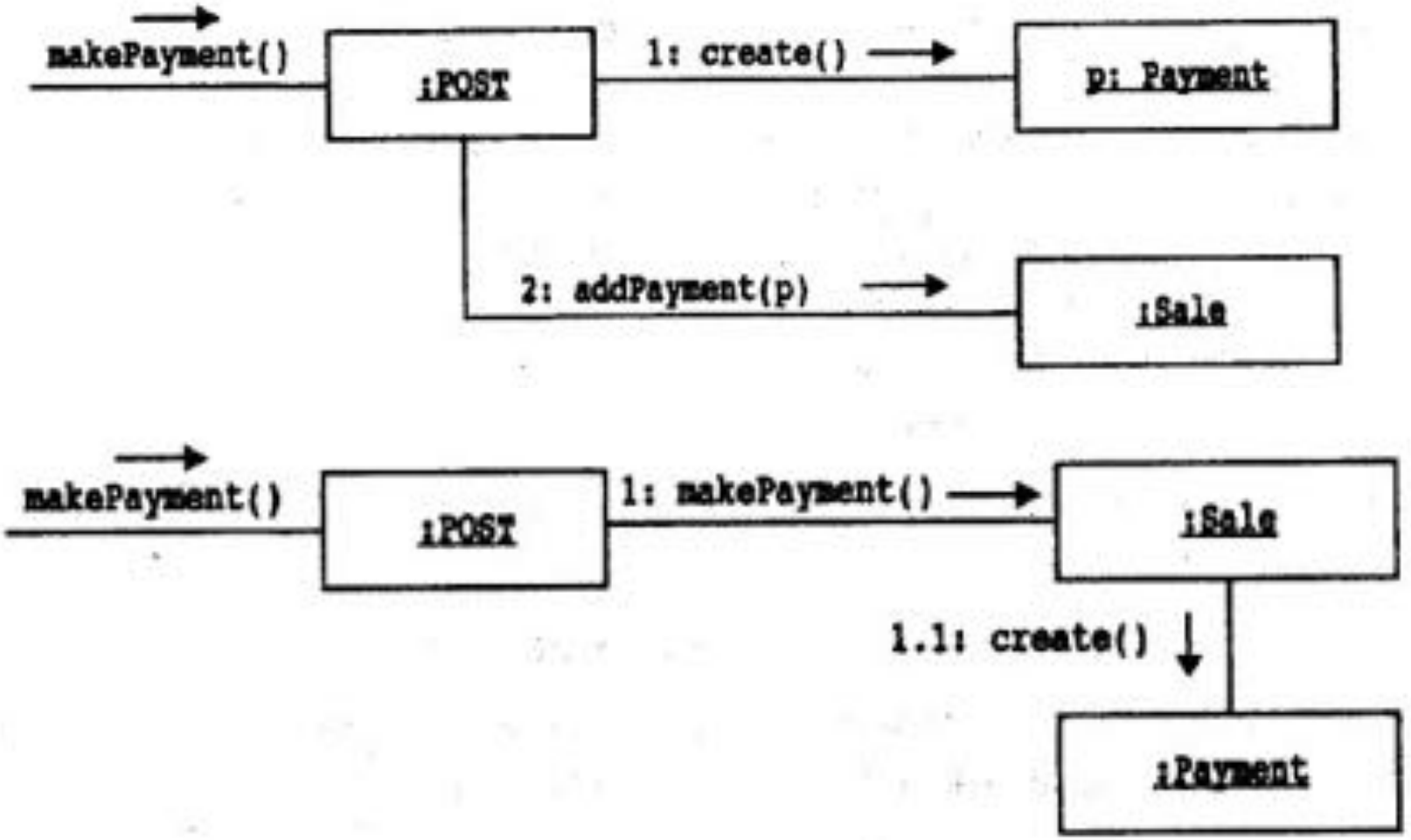
- Улучшения условий повторного использования компонентов. Обязанности контроллера могут быть технически реализованы в объектах интерфейса, однако в этом случае программный код и логические решения, относящиеся к процессам предметной области, будут жестко привязаны с элементами интерфейса (с окнами, кнопками и т. п.), что не правильно.
- Контроль состояния прецедента. Иногда необходимо удостовериться, что системные операции выполняются в определенной последовательности. Для этого лучше иметь отдельный контроллер прецедента.

Low Coupling и High Cohesion

- **Решение Low Coupling**: распределить обязанности так, чтобы степень связности оставалась низкой
- **Преимущества:**
  - Изменения компонентов мало сказываются на других объектах
  - Принципы работы компонент можно понять не изучая другие объекты
  - Улучшает повторное использование

- **Решение High Cohesion:** Распределяйте обязанности, поддерживающие высокую степень зацепления.
- **Преимущества.**
  - Повышаются ясность и простота проектных решений
  - Упрощаются поддержка и доработка
  - Обеспечивается, часто, слабое связывание
  - Улучшается повторное использование

Пример


makePayment()
:POST
1: create()
p: Payment
2: addPayment(p)
:Sale
makePayment()
:POST
1: makePayment()
:Sale
1.1: create()
:Payment

Обсуждение

- Эти два шаблона тесно связанны применение одного из них, автоматически ослабляет в том или ином виде проблему второго.
- Об этих двух принципах необходимо помнить в течении всего процесса проектирования и применять их при оценке эффективности каждого проектного решения.
- Наследование усиливает связность между классами, но зато можно получить наследованное поведение.
- Меры связности не существует.
- Полное отсутствие связности, крайняя ситуация, не нужная никому на практике.
- Аналогия в реальном мире. Человек выполняющий большое число не связных между собой задач, работает не эффективно. Это не умение менеджеров распределять обязанности между подчиненными.

Дополнительные шаблоны

- Pure Fabrication
- Indirection
- Polymorphism
- Protected Variations

Pure Fabrication

- **Проблема**. Кто должен обеспечить реализацию шаблонов Low Coupling и High Cohesion, если использование существующих объектов, связанных с понятиями предметной области, не эффективно.
- **Решение**. Присвоить группу обязанностей с высокой степенью зацепления искусственному классу, не представляющему конкретного понятия предметной области, т.е. синтезировать искусственную сущность для поддержки сильного зацепления, слабого связывания и повторного использования.

Indirection

- **Проблема**. В ряде случаев необходимо связать объекты не связанные напрямую. Кто должен обеспечить отсутствие прямого связывания? Как снизить уровень связывания согласно шаблону Low Coupling и обеспечить повторное использование.
- **Решение**. Присвоить обязанности промежуточному объекту для обеспечения связи между компонентами или службами, которые не связанны друг с другом на прямую. При таком подходе связи перенаправляются между другими компонентами или службами.

Пример

- Предположим следующее система торговли должна обрабатывать платежи по пластиковой карточке с использованием модемного соединения. Операционная система поддерживает вызов низкоуровневых функций API для решения этой задачи. Класс CreditAuthorizationService отвечает за взаимодействие с модемом.
- Если он напрямую использует вызовы низкоуровневых функций API, то он сильно связан с конкретным интерфейсом данной ОС. Если класс необходимо перенести на другую платформу, его понадобиться модифицировать. Поэтому прямое взамодействие нежелательно.
- Решение состоит в создании промежуточного класса Modem, который и берет на себя интерфейсную часть модема, а класс авторизации общается с модемом через этот класс.

- **Преимущества**. Слабое связывание.
- **Связанные шаблоны**. Low Coupling, Mediator, различные частные случаи Pure Fabrication.
- Шаблон Publish-Subscribe или Observer из GoF является одной из реализаций этого шаблона.

Polymorphism

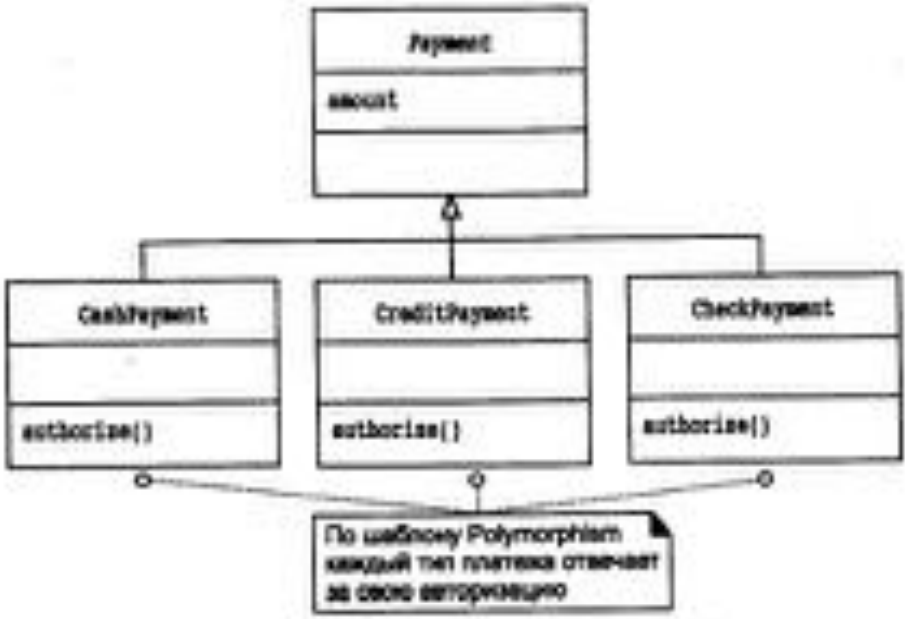
- **Проблема**. Как обрабатывать альтернативные варианты поведения на основе типа? Как создавать подключаемые программные компоненты? Как обеспечить расширяемость приложения?
- Альтернативные варианты поведения на основе типа. Условная передача управления (if-then-else или switch-case) при добавлении новых вариантов поведения требует модификации логики условных операторов везде где они применяются. Это усложняет код программы и ее расширяемость.
- Подключаемые программные модули. Если рассматривать компоненты с точки зрения отношения клиент/сервер, то как можно заменить один серверный компонент другим, не затрагивая при этом клиентские компоненты?

Polymorphism

- **Решение**. Если поведение объектов одного типа (класса) может изменятся, обязанности распределяются для различных вариантов поведения с помощью полиморфных операций для этого класса.
- **Замечание**. Обратите внимание на то, что в Java происходит динамическое связывание, то есть все методы при наследовании являются виртуальными по умолчанию и больше никак иначе.
- **Следствие**. Используйте не проверку типа объекта, а условную (полиморфную) логику для реализации альтернативных вариантов поведения на основе типа

Пример

1. Ферма.
2. Система розничной торговли. В системе возможны различные виды платежей. В каждом из них различные процедуры авторизации.



- **Преимущества использования**. С помощью этого шаблона можно легко расширять систему, добавляя в нее новые вариации.
- **Аналоги**. Do it Myself ("Сделай сам"), Choosing Message ("Выбор сообщения"), Don`t Ask "What Kind?" ("Не спрашивай "Что это?")
- **Обсуждение**. При использовании шаблона Полиморфизм (как и шаблона Эксперт) необходимо действовать в духе "Сделай сам". Авторизация платежа делается не внешним участником, а самим платежом.
- Если Expert важнейший тактический шаблон, то Ploymorphism – важнейший стратегический.
- Рассматривая объекты с точки зрения клиент / сервер, можно сказать, что клиентские объекты практически не требуют модификаций при появлении нового типа серверного объекта, если этот новый серверный объект поддерживает полиморфные операции.

Protected Variations

- Основная проблема, решаемая шаблоном Protected Variations: как спроектировать объекты, подсистемы и систему, чтобы изменение этих элементов не оказывало нежелательного влияния на другие элементы? необходимо идентифицировать точки возможных вариаций или неустойчивости; распределить обязанности таким образом, чтобы обеспечить устойчивый интерфейс.
- Шаблон PV описывает ключевой принцип, на основе которого реализуются механизмы и шаблоны программирования и проектирования с целью обеспечения гибкости и защиты системы от влияния изменений внешних систем.
- Инкапсуляция данных, интерфейсы, полиморфизм, перенаправление - все эти принципы реализуются в рамках шаблона PV.

Аналоги

- Проектирование на основе данных
- Поиск служб
- Проектирование на основе интерпретаторов
- Рефлексивное проектирование или проектирование на метауровне
- Унифицированный доступ

Каким бы ни был способ реализации, неизменным остается базовый принцип шаблона: обеспечить устойчивость интерфейса.

Объектно-ориентированное программирование в Python
Часть 1. Введение

Основные свойства ООП – полиморфизм, наследование, инкапсуляция.

Полиморфизм: в разных объектах одна и та же операция может выполнять различные функции.

Инкапсуляция: можно скрыть ненужные внутренние подробности работы объекта от окружающего мира.

Наследование: можно создавать специализированные классы на основе базовых.

Создание классов:

Класс – это пользовательский тип.

Для создания классов предусмотрен оператор **class**.

В терминологии Питона члены класса называются **атрибутами**, функции класса – **методами.**

```
* class ИМЯКЛАССА:
    ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
....
    def ИМЯМЕТОДА(self, ...):
self.ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ ...
        * ....
```

В Питоне класс не является чем-то статическим после определения, поэтому добавить атрибуты можно и после:

Пример:

```
class A:
    pass
def myMethod(self, x):
    return x ** 2
A.m1 = "My IQ is:"
A.m2 = myMethod
b = A()
print b.m1 + ' ' + str(b.m2(9))
```

Различать нужно атрибуты класса и объекта

Пример

```
class MyClass:   # Определяем пустой класс
    pass


MyClass.x =50 # Создаем атрибут объекта класса
c1, c2 = MyClass(), MyClass()    # Создаем два экземпляра класса
c1.y = 10 # Создаем атрибут экземпляра класса
c2.y = 20 # Создаем атрибут экземпляра класса
print(c1.x,  c1.y) # Выведет: 50 10
print(c2.x, c2.y)  # Выведет: 50 20
```

Для создания экземпляра класса, достаточно вызвать класс по имени и задать параметры конструктора.

Делается это с помощью оператора _init_.

Первым параметром, как и у любого другого метода, у **__init__** является **self**, на место которого подставляется объект в момент его создания.

```
class YesInit:
    def __init__ (self,one,two):
        self.fname = one
        self.sname = two

obj1 = YesInit(“Chris",“Rock")
print(obj1.fname + ' ' + obj1.sname)
    >>
Chris Rock
```

Однако создание объекта класса оператором _init_ предполагает передачу аргументов.

Если аргументы не переданы, то происходит ошибка. Поэтому можно присваивать параметры значениям по умолчанию.

```
class YesInit:
    def __init__(self,one="noname",two="nonametoo"):
        self.fname = one
        self.sname = two
Obj1 = YesInit()
print(obj.one + ” ” + obj.two)
    >>
noname nonametoo
```

Специальные методы вызываются при создании экземпляра класса и при удалении класса («деструктор» - вызывается сборщиком мусора). Следующий класс имеет конструктор и «деструктор» :

```
class Line:
    def __init__(self, p1, p2):
        self.line = (p1, p2)
    def __del__(self):
        print('Удаляется линия %s - %s' % self.line)
l = Line((0.0, 1.0), (0.0, 2.0))
del(l)
Удаляется линия (0.0, 1.0) - (0.0, 2.0)
```

Наследование

Простое наследование:

Класс, от которого произошло наследование, называется базовым или родительским. Классы, которые произошли от базового, называются потомками, наследниками или производными классами.

Множественное наследование:

При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости.

#Множественное наследование – потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках.

Python поддерживает как одиночное наследование, так и множественное, позволяющее классу быть производным от любого количества базовых классов.

```
class Par1(object):
# наследуем один базовый класс - object
        def name1 (self): return 'Par1'
 class Par2 (object):
        def name2 (self): return 'Par2'
class Child (Par1, Par2):
# создадим класс, наследующий Par1, Par2 (и,
 опосредованно, object)
        pass
x = Child()
print x.name1() + ‘  ’+ x.name2()
# экземпляру Child доступны методы из Par1 и Par2
```

- Типичная проблема, возникающая при проектировании ООП,состоит в следующем. Объект некоторого типа требуется передавать в качестве аргумента в различные функции.
- Разным функциям нужны разные свойства и методы этого объекта.
- При этом хотелось бы все эти функции сделать полиморфными, то есть способными принимать объекты разных типов.

В Питоне используется концепция, называемая Duck Typing:

” Если ЭТО ходит как утка, и крякает,как утка - значит это утка”.

Т.е., если у объекта есть все нужные функции, свойства и методы, то он подходит в качестве аргумента.

Например, в функцию

```
def f(x):
  return x.get_value_()
```

Можно передавать объект любого типа, лишь бы у него был метод get_value().

Ещё одна проблема, возникающая в связи с множественным наследованием - не всегда очевидно, в каком порядке будут просматриваться родительские классы в поисках нужного свойства или метода. В Питоне для упрощения этой проблемы у каждого класса есть свойство __mro__(method resolution order):

```
>>>class A(object): pass
...
>>>class B(object): pass
...     x = 0
...
>>>class C(A,B):
...     z = 3
...
>>>C.__mro__
(<class 'C'>,<class 'A'>,<class 'B'>,<type object>)
```

В Python есть метод super(), который обычно применяется к объектам. Его главная задача это возможность использования в классе потомке, методов класса-родителя.

```
class Child(Parent):
 def __init__(self):
       super(Child, self).__init__(self) #   но можно и без параметров super().__init__()
```

Пример:

```
class A(object):
   def __init__(self):
                print('конструктор класса A')
 # Потомок класса A
 class B(A):
     def __init__(self):
        print('конструктор класса B')
        super(B,self).__init__()
```

#Смысл примера заключается в том, что Python не запустит родительский конструктор, поскольку мы его переопределили в классе B... Поэтому методом super() мы явно вызываем родительский конструктор.

Описание параметров super()

- **super**([type[, object-or-type]])
- **Параметры**
- **type** - Тип, от которого следует начать поиск объекта-посредника. Ранее атрибут был обязателен.

  obj-or-type - Если не указан, возвращается несвязанный объект-посредник. Если атрибут является объектом, то будет получен посредник для получения метода объекта, для которого isinstance(obj, type) возвращает True. Если атрибут является типом, то будет получен посредник для получения метод класса, для которого issubclass(subtype, type) возвращает True.

Свойства super():

```
class C(B, A):
    def __init__(self):
        # something
        super(C,self).__init__()
```

По сути, объект класса super запоминает аргументы переданные ему в момент инициализации и при вызове любого метода (super().__init__(self) в примере выше) проходит по списку линеаризации класса второго аргумента (self.__class__.__mro__), пытаясь вызвать этот метод по очереди для всех классов, следующих за классом в первом аргументе (класс C), передавая в качестве параметра первый аргумент (self). Т.е. для нашего случая:
self.__class__.__mro__ = [C, B, A, P1, P2, ...]
super(C, self).__init__() => B.__init__(self)
super(B, self).__init__() => A.__init__(self)
super(A, self).__init__() => P1.__init__(self)

Изменяя атрибут __class__, можно перемещать объект вверх или вниз по иерархии наследования (впрочем, как и к любому другому типу)

```
>>>c = child()
 >>c.val = 10
 >>c.who()
'child'
>>> c.__class__ = parent
>>> c.who()
'parent'
>>> c.val
10
```

Полиморфизм

В ООП программировании этим термином обозначают возможность использования одного и того же имени операции или метода к объектам разных классов, при этом действия, совершаемые с объектами, могут существенно различаться.

Два разных класса могут содержать метод (например total) , однако инструкции в методах могут предусматривать совершенно разные операции.

```
class T1:
    n=10
    def total(self,N):
        self.total = int(self.n) + int(N)

class T2:
    def total(self,s):
        self.total = len(str(s))
```

Переопределение методов

Использование полиморфизма при наследовании классов позволяет переопределять методы суперклассов их подклассами. Например, может возникнуть ситуация, когда все подклассы реализуют определенный метод из суперкласса, и лишь один подкласс должен иметь его другую реализацию. В таком случае метод переопределяется в подклассе.

В компилируемых языках программирования полиморфизм достигается за счёт создания виртуальных методов, которые в отличие от невиртуальных можно перегрузить в потомке. В Питоне все методы являются виртуальными, что является естественным следствием разрешения доступа на этапе исполнения.

```
class Animal:
      def __init__(self,name): # Общий конструктор
            self.nick = name

      def voice(self): # Абстрактный метод
            raise NotImplemented("Не реализовано")


class Cat(Animal):
      def __init__(self,name):
            super().__init__(name)

     # Реализация абстрактного метода
      def voice(self):
            return "Мяу-мяу"

class Dog(Animal):
      def __init__(self, name):
            self.nick = name

      def voice(self):
            return "Гав-гав"
```

```
class Pig(Animal):
      def __init__(self, name):
            self.nick = name

      def voice(self):
            return "Хрю-хрю"

class Zoo:
      def __init__(self,list):
            self.animals=list

      def printAllVoices(self):
          for a in self.animals:
               print(a.nick+": "+a.voice())

# Пример полиморфной функции
     def getVoice(self,animal):
           return animal.voice()

z=Zoo([Cat("Мурзик"), Dog("Шарик"), \
      Pig("Борька"), Cat("Васька")])
z.printAllVoices()

print(z.getVoice(z.animals[1]))
```

Инкапсуляция и доступ к свойствам

Инкапсуляция — свойство языка программирования, позволяющее объединить и защитить данные и код в объекте и скрыть реализацию объекта от пользователя. При этом пользователю предоставляется только спецификация (интерфейс) объекта.

Пользователь может взаимодействовать с объектом только через этот интерфейс.

Пользователь не может использовать закрытые данные и методы.

Одиночное подчеркивание в начале имени атрибута говорит о том, что метод не предназначен для использования вне методов класса (или вне функций и классов модуля), однако, атрибут все-таки доступен по этому имени. Два подчеркивания в начале имени дают несколько большую защиту: атрибут перестает быть доступен по этому имени. Последнее используется достаточно редко.

Есть существенное отличие между такими атрибутами и private-элементами класса в таких языках как C++ или Java: атрибут остается доступным, но под именем вида _ИмяКласса__ИмяАтрибута, а при каждом обращении Python будет модифицировать имя в зависимости от того, через экземпляр какого класса происходит обращение к атрибуту. Таким образом, родительский и дочерний классы могут иметь атрибут с именем, например, «__f», но не будут мешать друг другу.

Пример:

```
class Parent(object):
     def __init__(self):
         self.__f = 2
    def get(self):
         return self.__f

 class Child(Parent):
     def __init__(self):
         self.__f = 1
         Parent.__init__(self)
     def cget(self):
         return self.__f

 c = Child()
 print(c.get())  # 2
 print(c.cget()) # 1
 print(c.__dict__) # {'_child__f': 1, '_parent__f': 2}
```

на самом деле у объекта "c" два разных атрибута

Источники

1. Ларман К. Применение UML 2.0 и шаблонов проектирования
2. Марк Лутц. Изучаем Питон. 3-е издание.
3. Россум Г. , Дрейк Ф.Л.Дж., Откидач Д.С. Язык программирования Python
4. Николай Вяххи. Лекция Объектно-ориентированное программирование в Python. URL: http://vyahhi.spbsu.ru/teaching/sites/default/files/2010-10-26_OOP_v2.pptx