



Programming
Languages

Лабораторная работа #28

Объектно-Ориентированное Программирование. Скрытие реализации (инкапсуляция) в языке Python



LEARN. GROW. SUCCEED.

© 2019-2020. Department: <Software of Information Systems and Technologies>

Faculty of Information Technology and Robotics

Belarusian National Technical University

by Viktor Ivanchenko / ivanvikvik@bntu.by / Minsk

ЛАБОРАТОРНАЯ РАБОТА #28

ООП. Соккрытие реализации (инкапсуляция) в языке Python

Цель работы

Изучить механизмы и способы соккрытия реализации в языке Python, а также научиться отделять внутреннее представление объекта (метода, класса, модуля, пакета и т.д.) от внешнего на примере проектирования и реализации ООП-программ с использованием языка Python.

Общее задание

Необходимо произвести рефакторинг программной системы, созданной в предыдущей лабораторной работе, следующим образом:

- объекты системы (бизнес-объекты), созданные на базе соответствующих классов предметной области, должны скрывать свою реализацию и предоставлять только внешний интерфейс для взаимодействия (т.е. необходимо реализовать инкапсуляцию данных и поведения);
- ввести, где это необходимо, контейнерный класс для хранения набора бизнес-объектов, т.е. инкапсулировать объекты в более высокоуровневый объект на уровне программной системы; логика системы должны принимать на вход объекты только данных контейнерных классов;
- объекты соответствующих классов-сущностей должны реализовать инкапсулированный доступ к будущему состоянию объектов через соответствующие свойства: только для чтения, для чтения и записи, только для записи – при необходимости (определяется предметной областью);
- классы-сущности моделируемой системы обязательно должны содержать соответствующий метод `__str__()`, который служит для автоматического строкового представления состояния объекта, на который указывает соответствующая ссылочная переменная;
- классы системы (обычно, функциональные классы) должны содержать собственные атрибуты и статические методы, которые должны быть обоснованы и актуальны для всего класса в целом, а не отдельного экземпляра.



Требования к выполнению задания

- 1) Необходимо модернизировать спроектированную в предыдущей лабораторной работе UML-диаграмму взаимодействия классов и объектов программной системы исходя из текущих дополнений. На базе данной изменённой UML-диаграммы реализовать рабочее приложение с использованием архитектурного шаблона проектирования **MVC**.
- 2) Каждый класс разрабатываемого приложения должен иметь адекватное осмысленное имя (обычно это *имя существительное*). Имена полей и методов должны нести также логический смысл (имя метода, который что-то вычисляет, обычно называют *глаголом*, а поле – именем существительным). Имя класса пишется с большой (заглавной) буквы, а имена методов и переменных – с маленькой (строчной).
- 3) Соответствующие классы должны группироваться по модулям, которые затем подключаются там, где происходит создание объектов классов и их использование.
- 4) При проектировании классов необходимо придерживаться принципа единственной ответственности (**Single Responsibility Principle**), т.е. классы должны проектироваться и реализовываться таким образом, чтобы они были слабо завязаны с другими классами при своей работе – они должны быть самодостаточными.
- 5) При выполнении заданий необходимо по максимуму пытаться разрабатывать универсальный, масштабируемый, легко поддерживаемый и читаемый код.
- 6) В соответствующих компонентах (классах, функциях) бизнес-логики необходимо предусмотреть «защиту от дурака».
- 7) Рекомендуется избегать использования глобальных переменных.
- 8) Там, где это необходимо, необходимо обеспечить грамотную обработку исключительных ситуаций, которые могут произойти при выполнении разработанной программы.
- 9) Все действия, связанные с демонстрацией работы приложения, должны быть размещены в главном модуле программы в функции **main**. При проверки работоспособности приложения необходимо проверить все тестовые случаи.



- 10) Программы должны обязательно быть снабжены комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и её название, версию программы, ФИО разработчика, номер группы и дату разработки. Исходный текст программного кода и демонстрационной программы рекомендуется также снабжать поясняющими краткими комментариями.
- 11) Программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом для взаимодействия с пользователем. Интерфейс программы должен быть на английском языке.
- 12) При разработке программы придерживайтесь соглашений по написанию кода на Python (***Python Code Convention***).

Best of LUCK with it, and remember to HAVE FUN while you're learning :)

Victor Ivanchenko



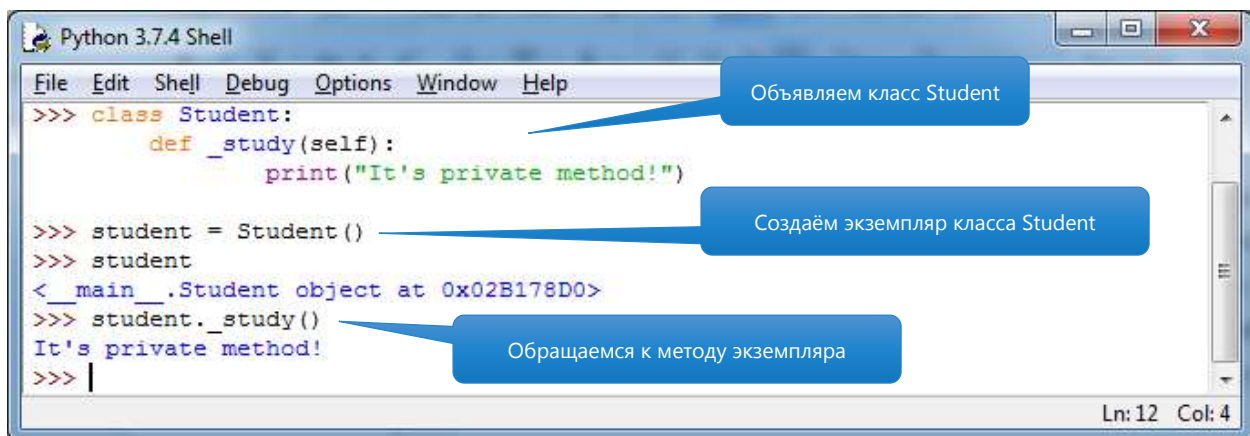
Что нужно запомнить (краткие тезисы)

1. **Инкапсуляция (encapsulation)** – это сокрытие реализации класса и отделение его внутреннего представления от внешнего. В большинстве случаев при использовании ООП и других методологий не рекомендуется использовать прямой доступ к данным и реализации моделируемых объектов соответствующих классов, а общаться (иметь доступ) только через общедоступный интерфейс.
2. **Инкапсуляция** позволяет описать характеристики и реализацию класса поведения объектов таким образом, чтобы они были доступны только в пределах данного класса.
3. **Очень важно!** Все объекты в ООП-системе должны взаимодействовать между собой как **чёрные ящики**. У каждого объекта (компонента) системы должен быть свой личный открытый интерфейс, посредством которого он общается (взаимодействует) с другими объектами системы, а вся его внутренняя реализация скрыта (инкапсулирована).
4. **Главная идея инкапсуляции в программировании** – разработчик программного компонента может легко вносить в него изменения или полностью менять его внутреннюю структуру и логику без изменения интерфейсной части компонента, что не повлечёт к изменению остальных компонентов системы или программных слоёв, которые зависят от данного компонента.
5. **Дополнительные преимущества инкапсуляции (следствие):**
 - ✓ обеспечивает **согласованность, целостность и непротиворечивость данных** внутреннего состояния компонента (объекта или класса в целом);
 - ✓ гораздо проще контролировать корректные значения полей путём полного контроля над входящими и исходящими данными;
 - ✓ не составит труда **изменить способ хранения состояния** (данных) компонента (если информация станет храниться не в памяти, а в долговременном хранилище, такой как файловая система или база данных, то потребуется изменить внутреннюю реализацию ряда методов одного класса, а не вносить изменения во все части системы, где напрямую использовались данные компонента);
 - ✓ облегчает поиск ошибок, а также **программный код легче отлаживать** (для того, чтобы узнать, в какой момент времени и кто изменил состояние



интересующего компонента через предоставляемый общий интерфейсный метод, достаточно добавить отладочную информации в данный интерфейсный метод, посредством которого осуществляется доступ к состоянию этого объекта).

6. Инкапсуляция в языке программирования Python **работает лишь на уровне соглашения между разработчиками**. О том, какие данные должны быть общедоступными, а какие – внутренними (инкапсулированными), решает сам программист.
7. **Одиночное подчёркивание** в начале имени идентификатора (поля или метода) говорит о том, что он не предназначен для использования вне класса, однако в коде идентификатор всё равно доступен извне по этому имени:



```

Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
>>> class Student:
>>>     def _study(self):
>>>         print("It's private method!")
>>>
>>> student = Student()
>>> student
<_main_.Student object at 0x02B178D0>
>>> student._study()
It's private method!
>>>
    
```

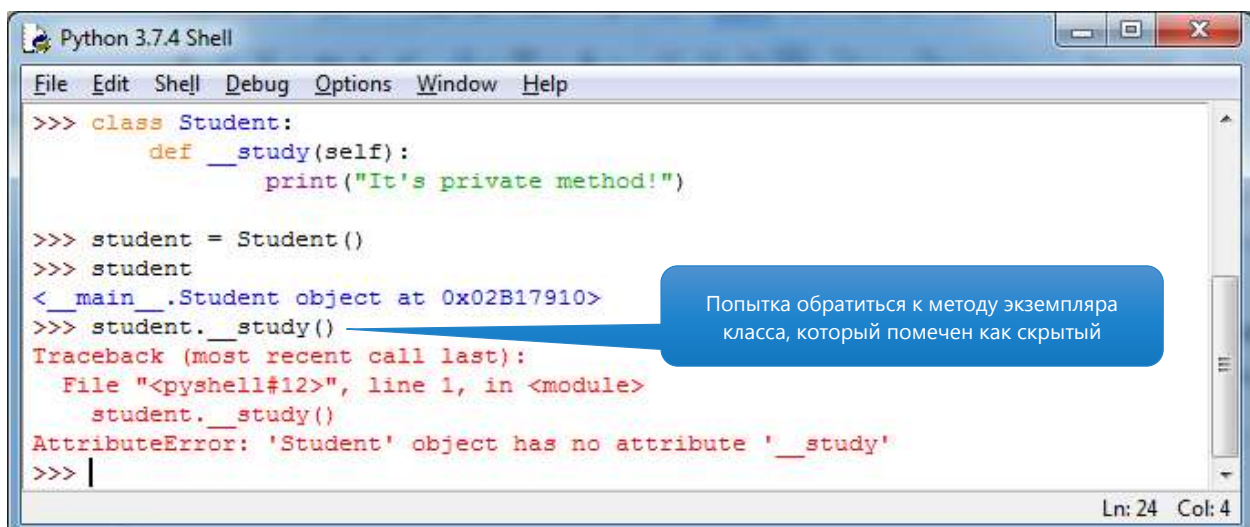
Объявляем класс Student

Создаём экземпляр класса Student

Обращаемся к методу экземпляра

Ln: 12 Col: 4

8. **Двойное подчёркивание** в начале имени идентификатора (поля или метода) даёт большую защиту: в коде напрямую идентификатор становится недоступен по данному имени:



```

Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
>>> class Student:
>>>     def __study(self):
>>>         print("It's private method!")
>>>
>>> student = Student()
>>> student
<_main_.Student object at 0x02B17910>
>>> student.__study()
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    student.__study()
AttributeError: 'Student' object has no attribute '__study'
>>>
    
```

Попытка обратиться к методу экземпляра класса, который помечен как скрытый

Ln: 24 Col: 4



9. **Двойное подчёркивание** в коде полностью **не защищает доступ** к имени идентификатора (поля или метода), т.к. к нему можно всегда обратиться с использованием специального имени через ссылку на соответствующий класс – **`__ClassName__ IdentifierName`**:

```

Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
>>> class Student:
>>>     def __study(self):
>>>         print("It's private method!")
>>>
>>> student = Student()
>>> student
<__main__.Student object at 0x02B17910>
>>> student.__study()
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    student.__study()
AttributeError: 'Student' object has no attribute '__study'
>>> student._Student__study()
It's private method!
>>>
    
```

Попытка обратиться к методу экземпляра класса через специальное имя класса

10. Недостатки инкапсуляции:

- ✓ необходимо больше писать кода;
- ✓ сложность исправлять ошибки сторонних библиотек;
- ✓ снижается общая производительность и скорость работы приложения.

11. В языке Python в классе можно описать несколько типов вычислительных блоков:

- ✓ **динамические методы** (*instance methods*) или **методы экземпляра класса** – обычно вызываются на ссылочных переменных, т.е. на экземплярах класса, но с помощью специального синтаксиса могут быть вызваны и непосредственно на ссылке класса;
- ✓ **статические методы** (*static methods*);
- ✓ **методы класса** (*class methods*) – обычно вызываются на ссылке класса, но могут вызываться и через ссылочную переменную;
- ✓ **обычные функции** (*functions*) – вызываются только на ссылке класса.

12. Динамический метод в коде ничем не помечается и используется в тех случаях, если его реализация на прямую или косвенно связано с состоянием объекта. Согласно ООП парадигме правильный вызов динамических методов в



коде должен происходить на ссылочной переменной, т.е. на самом экземпляре. Это наиболее часто используемый вид методов.

13. Методы экземпляра класса принимают объект класса как первый аргумент, который принято называть ***self*** и который указывает на сам экземпляр. Количество параметров метода не ограничено.
14. С помощью параметра ***self*** можно менять состояние объекта и обращаться к другим его методам и параметрам.
15. С помощью атрибут ***self.__class__*** можно получить доступ к атрибутам класса и возможности менять состояние самого класса. То есть **методы экземпляров класса позволяют менять как состояние определённого объекта, так и класса.**
16. **Методы класса** в коде помечаются декораторами ***@classmethod*** и привязан к самому классу, а не его экземплярам. Они могут менять состояние класса, что отразится на всех объектах этого класса, но не могут менять конкретный объект. Методы класса принимают класс в качестве параметра, который принято обозначать как ***cls***.
17. Не забывайте, что для каждого класса в системе создаётся единственный объект-класса. Как раз в нём и хранятся атрибуты класса и мн.др.
18. Заметьте, что ***self*** и ***cls*** – не обязательные названия и эти параметры можно называть иначе. Это лишь общепринятые обозначения, которым следуют все. Тем не менее они должны находиться первыми в списке параметров.
19. **Статические методы** в коде помечаются декораторами ***@staticmethod*** и описывают общий функционал, который связан с каким-либо программируемым процессом, а не поведением какого-нибудь отдельного объекта или объекта-класса.
20. Статические методы работают как обычные функции, но принадлежат области имён класса. Статический метод не принимает неявных аргументов, к примеру, ***self*** или ***cls***, а следовательно, никак не может поменять состояние объекта или атрибуты класса, в котором он описан. Они работают только с теми данными, которые им передаются явно в качестве аргументов.



Пример выполнения задания

Произведём рефакторинг программной системы «Университет» («*University*»), которая была создана на предыдущей лабораторной работе.

Пункты решения поставленной задачи:

- 1) Инкапсулируем (скроем) состояние (характеристики) будущих объектов классов *Teacher* и *Student*. Доступ к ним предоставим через специальные методы (или свойства), а также, где это необходимо, добавим в них логику, которая предотвращает присвоение противоречивых значений объектам, т.е. защитим объекты от присвоения им неверных данных (см. рис. 1 и 2).

```

import random
from student import Student

class Teacher:
    """class define teacher's information and logic"""

    def __init__(self, name, experience=1):
        self.__name = name
        self.__experience = experience

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

    def get_experience(self):
        return self.__experience

    def set_experience(self, experience):
        if experience >= 0:
            self.__experience = experience

    def __str__(self):
        return (self.__name + "("
                + str(self.__experience) + ")")

    def grade(self, list_of_student):
        MIN_MARK = 4
        MAX_MARK = (100 - self.__experience) // 10
        for st in list_of_student:
            if isinstance(st, Student):
                st.mark = random.randint(MIN_MARK, MAX_MARK)
  
```

Чтобы сделать поля объекта скрытыми, необходимо каждое из полей предварить двумя знаками нижнего подчёркивания!

Динамические методы для чтения и записи атрибута *name* будущих объектов класса

Динамические методы для записи атрибута *experience* будущих объектов класса с проверкой на присвоение непротиворечивых данных

Переопределённый метод базового класса для вывода строкового эквивалента состояния объекта «Преподаватель»

Рисунок 1 – Обновлённый исходный код класса *Teacher*



```

import datetime

class Student:
    """ class defines student's information """

    def __init__(self, name, birthday, mark=0):
        self.__name = name
        self.__mark = mark
        self.set_birthday(birthday)

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

    def get_birthday(self):
        return self.__birthday

    def set_birthday(self, birthday):
        if isinstance(birthday, str):
            lst = birthday.split(sep='.')
            self.__birthday = datetime.date(year=int(lst[0]),
                                             month=int(lst[1]),
                                             day=int(lst[2]))

    def get_mark(self):
        return self.__mark

    def set_mark(self, mark):
        if 0 <= mark <= 10:
            self.__mark = mark

    @property
    def age(self):
        age = datetime.date.today().year - self.__birthday.year

        if (datetime.date.today().month < self.__birthday.month or
            datetime.date.today().day < self.__birthday.day and
            datetime.date.today().month == self.__birthday.month):
            age -= 1

        return age

    def __str__(self):
        return (self.__name +
                "(birthday = " + str(self.__birthday) +
                ", age = " + str(self.age) +
                ", mark = " + str(self.__mark) + ")")
    
```

Чтобы сделать поля объекта скрытыми, необходимо каждое из полей предварить двумя знаками нижнего подчёркивания!

Динамические методы для чтения и записи атрибута *name* будущих объектов класса

Динамический метод для записи даты рождения с небольшой проверкой и преобразованием строки в объект типа *date* из стандартного модуля *datetime*

Динамический метод для записи значения атрибута *mark* будущих объектов класса с проверкой на присвоение непротиворечивых данных

Свойство «только для чтения», которое высчитывает возраст студента

Переопределённый метод базового класса для вывода строкового эквивалента состояния объекта «Студент»

 Рисунок 2 – Обновлённый исходный код класса-сущности *Student*


- 2) Дополнительно введём в класс *Student* новое поле «*birthday*» (*дата рождения*), а вместо поля «*age*» (*возраст*) создадим свойство «только для чтения», которое будет вычислять возраст студента на базе текущей даты и даты рождения студента. Для вычисления возраста студента воспользуемся встроенным типом ***date***, который описан в стандартном модуле ***datetime*** (см. рис. 2).
- 3) Т.к. поведение (действие), которое сосредоточено (описано) в функциональных классах *God* и *Manager* никак не зависит от состояния объектов данных классов, то вынесем данное поведение в статические методы соответствующих классов (см. рис. 3 и 4).
- 4) Для демонстрации использования **атрибутов** самого **класса** в классе *Manager* введём специальную переменную, в которой будет храниться информация о том, сколько раз осуществлялся мониторинг успеваемости студентов, а для того, чтобы можно было прочитать её, опишем в данном классе ещё один статический метод (см. рис. 4).

```

from student import Student

class Manager:
    """ functional class of business logic """

    __inspection_number = 0

    @staticmethod
    def inspection_number():
        return Manager.__inspection_number

    @staticmethod
    def calculate(list_of_student):
        Manager.__inspection_number += 1
        avg = 0
        count = 0
        for st in list_of_student:
            if isinstance(st, Student):
                avg += st.get_mark()
                count += 1

        return float(avg / count)
    
```

Атрибут класса для сохранения количества мониторинга успеваемости студентов

Статический метод для реализации инкапсуляции на уровне класса и доступа к его атрибуту

Статический метод для подсчёта успеваемости студентов. В данном методе также идёт инкрементирования атрибута класса (счётчик проверок)

Рисунок 3 – Обновлённый исходный код функционального класса *Manager*



```

import random
from student import Student

class God:
    """ functional class for working with list of students """

    START_ALPHABET_WITH_UPPER_LETTER = 65
    END_ALPHABET_WITH_UPPER_LETTER = 91

    @staticmethod
    def create(count):
        names = ["Alexander", "Pavel", "Artyom", "Michael",
                 "Olya", "Nastya", "Kirill", "Stas", "Nikita",
                 "Oleg", "Max", "Ilya", "Sergey", "Alexey"]

        list_of_student = []

        for i in range(count + 1):
            name = random.choice(names)

            name += " " + chr(random.randint(
                God.START_ALPHABET_WITH_UPPER_LETTER,
                God.END_ALPHABET_WITH_UPPER_LETTER)) + "."

            age = random.randint(17, 19)

            student = Student(name, age)

            list_of_student.append(student)

        return list_of_student

    @staticmethod
    def convert_to_string(list_of_student):
        string = "List of student:\n"

        for student in list_of_student:
            if isinstance(student, Student):
                string += str(student) + "\n"

        return string

```

Статический метод для создания списка студентов. Данный метод можно теперь вызывать на самом классе без создания объекта (экземпляра) данного класса

Статический метод для формирования строкового представления списка студентов

При вызове строенной функции `str()` на ссылочной переменной `student` будет неявно вызываться метод `__str__()`, который был описан в классе `Student`

Рисунок 4 – Обновлённый исходный код утилитного (дополнительного) класса `God`



- 5) Также в классах *Student* и *Teacher* произвели замену соответствующих методов вывода информации об объекте на специальный **переопределяемый** метод `__str__(self)`, который мы наследуем по умолчанию от базового класса **object**. Данный метод будет вызываться автоматически на ссылке соответствующих объектов классов, когда требуется их строковый эквивалент (см. рис. 1 и 2).
- 6) Общая UML-диаграмма классов программной реализации системы заданной предметной (проблемной) области «University» приведена ниже на рисунке 5.

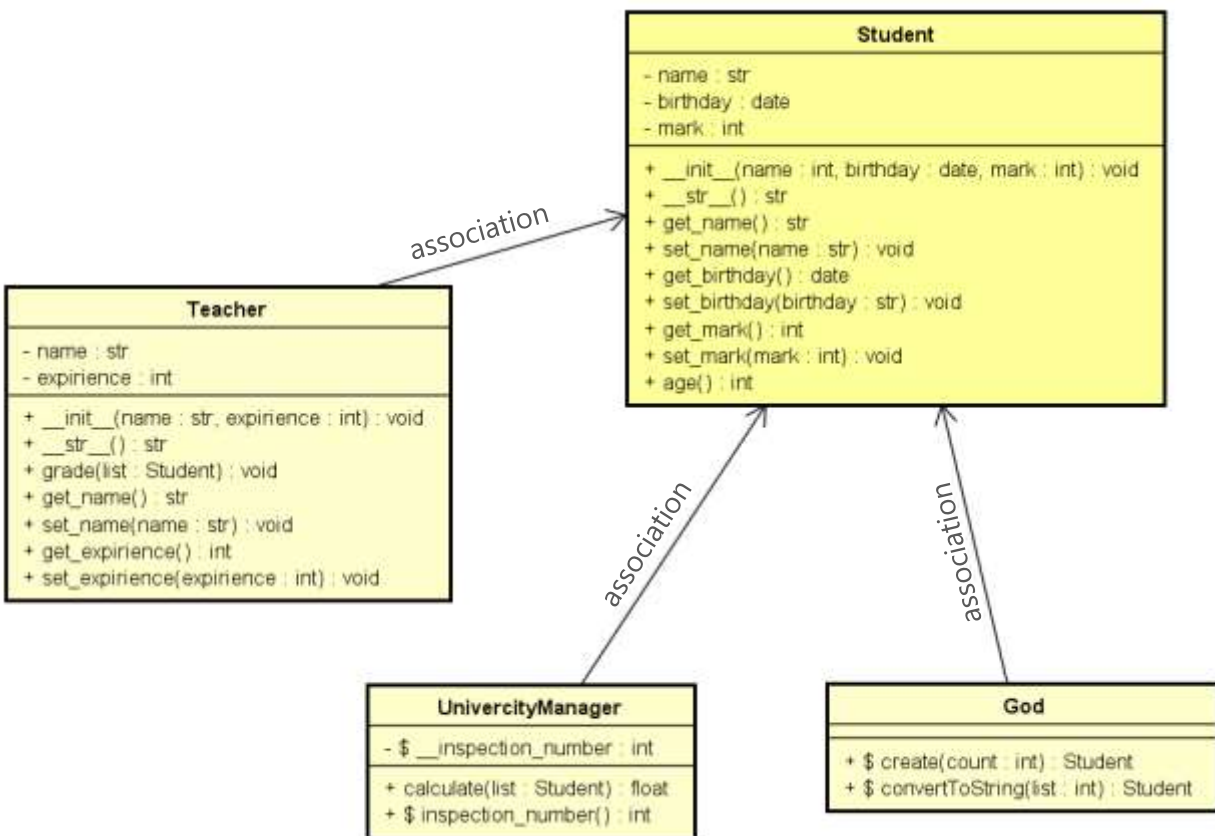


Рисунок 5 – Обновлённая UML-диаграмма классов заданной предметной (проблемной) области «University»

- 7) Для тестирования разработанной модели поведения системы создадим ещё один модуль `main` (см. рис. 6). В нём опишем функцию `main()`, в которой смоделируем поведения нашей системы в целом. Результат работы функции представлен на рисунке 7.



```

from god import God
from teacher import Teacher
from manager import Manager

def main():

    #генерируем список из десяти студентов
    students_list = God.create(10)

    #выводи список студентов на консоль
    print(God.convert_to_string(students_list))

    # создаём объект-преподаватель
    teacher = Teacher("Victor Victorovich")

    # эмулируем процесс выставления оценок за первый семестр
    term_process(teacher, students_list)

    # эмулируем процесс выставления оценок за второй семестр
    term_process(teacher, students_list)

    print("\nNumber of assessment of academic progress: " + str(Manager.inspection_number()))

def term_process(teacher, students_list):
    teacher.grade(students_list)

    print(God.convert_to_string(students_list))

    avg = Manager.calculate(students_list)
    print("Avg: %.1f" % avg)

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

Рисунок 6 – Обновлённый стартовый (тестовый) модуль main



```

Run main (3)
C:\Python34\python.exe "C:/Users/VikVik/PycharmProjects/ProgrammingLanguages/Labs/Lab 13/main.py"

List of student:
Olya P.(birthday = 1998-12-07, age = 17, mark = 0)
Michael R.(birthday = 1998-11-21, age = 17, mark = 0)
Max C.(birthday = 1998-06-15, age = 17, mark = 0)
Ilya Y.(birthday = 1998-12-20, age = 17, mark = 0)
Kirill O.(birthday = 1998-03-06, age = 18, mark = 0)
Olya Z.(birthday = 1997-11-21, age = 18, mark = 0)
Nikita Q.(birthday = 1999-02-27, age = 17, mark = 0)
Nastya F.(birthday = 1997-04-11, age = 19, mark = 0)
Alexey D.(birthday = 1997-05-17, age = 19, mark = 0)
Nastya U.(birthday = 1997-10-19, age = 18, mark = 0)
Oleg X.(birthday = 1998-07-27, age = 17, mark = 0)

List of student:
Olya P.(birthday = 1998-12-07, age = 17, mark = 9)
Michael R.(birthday = 1998-11-21, age = 17, mark = 9)
Max C.(birthday = 1998-06-15, age = 17, mark = 5)
Ilya Y.(birthday = 1998-12-20, age = 17, mark = 8)
Kirill O.(birthday = 1998-03-06, age = 18, mark = 7)
Olya Z.(birthday = 1997-11-21, age = 18, mark = 4)
Nikita Q.(birthday = 1999-02-27, age = 17, mark = 6)
Nastya F.(birthday = 1997-04-11, age = 19, mark = 7)
Alexey D.(birthday = 1997-05-17, age = 19, mark = 6)
Nastya U.(birthday = 1997-10-19, age = 18, mark = 7)
Oleg X.(birthday = 1998-07-27, age = 17, mark = 9)

Avg: 7.0
    Результат работы метода по вычислению успеваемости за первый семестр

List of student:
Olya P.(birthday = 1998-12-07, age = 17, mark = 6)
Michael R.(birthday = 1998-11-21, age = 17, mark = 8)
Max C.(birthday = 1998-06-15, age = 17, mark = 5)
Ilya Y.(birthday = 1998-12-20, age = 17, mark = 9)
Kirill O.(birthday = 1998-03-06, age = 18, mark = 8)
Olya Z.(birthday = 1997-11-21, age = 18, mark = 8)
Nikita Q.(birthday = 1999-02-27, age = 17, mark = 8)
Nastya F.(birthday = 1997-04-11, age = 19, mark = 4)
Alexey D.(birthday = 1997-05-17, age = 19, mark = 8)
Nastya U.(birthday = 1997-10-19, age = 18, mark = 8)
Oleg X.(birthday = 1998-07-27, age = 17, mark = 6)

Avg: 7.1
    Результат работы метода по вычислению успеваемости за второй семестр

Number of assessment of academic progress: 2
    Значение атрибута класса, отвечающее за количество аттестаций
    
```

Рисунок 7 – Результат тестирования работы программной системы «University»



Контрольные вопросы

1. Что такое инкапсуляция и зачем она нужна?
2. Каков главный козырь инкапсуляции?
3. Где Вы в реальной жизни встречаете инкапсуляцию и на что она влияет?
4. Является ли инкапсуляция только прерогативой ООП или где-то она уже Вам встречалась при получении опыта программирования?
5. Какими средствами обычно обеспечивается инкапсуляция в ООП?
6. Как в языке Python реализуется инкапсуляция на уровне синтаксиса языка и на уровне методологии программирования?
7. Как получить доступ к инкапсулированным данным в языке Python?
8. Что такое свойства в языке Python и как их реализовать?
9. Как реализовать свойства «только для чтения» или свойство «только для записи»?
10. Зачем нужны динамические методы и как их реализовать в языке Python?
11. Зачем нужны методы класса и как их реализовать в языке Python?
12. Зачем нужны статические методы и как их реализовать в языке Python?
13. Каким образом можно вызвать в языке Python динамические, статические методы и методы класса?
14. В чём концептуальная разница между разными типами методов и функций, которые можно описать внутри класса? Приведите примеры.
15. Преимущества и недостатки инкапсуляции?

