Проектирование «сверху вниз»

1 Понятие о структурном программировании

Структурное программирование - парадигма разработки программ с помощью представления их в виде иерархической структуры блоков. Сюда входит три пункта:

1. Любая программа состоит из трех типов конструкций:
   - последовательное исполнение;
   - ветвление;
   - цикл.
2. Логически целостные фрагменты программы оформляются в виде подпрограмм. В тексте основной программы вставляется инструкция вызова подпрограммы. После окончания подпрограммы исполнение продолжается с инструкции, следующей за командой вызова подпрограммы.
3. Разработка программы ведется поэтапно методом “сверху вниз”.

Первый пункт скорее важен не тем, что в нем есть, а тем, чего в нем нет: в нем нет оператора безусловного перехода goto. Именно это отличает структурное программирование от процедурного. Благодаря пункту два в языках высокого уровня появились новые синтаксические конструкции: функции и процедуры.

Пункт три самый важный и является сутью парадигмы структурного программирования.

2 Проектирование “сверху вниз”

Как было сказано выше, сама суть структурного программирования лежит в подходе к проектированию программы. Для понимания этого приведем процесс решения конкретной задачи во времени.

Условие. Пусть в некоторой стране зарплата сотрудника дорожной патрульной службы состоит из штрафов, накладываемых на водителей за превышение скорости в 60 км/ч, штраф напрямую зависит от номера автомобиля, а рабочий день заканчивается с приездом начальника, при этом начальника штрафовать нельзя. Требуется посчитать зарплату сотрудника за день.

Входные данные подаются построчно, в каждой строке - скорость (целое число) и номер автомобиля (6 символов - 1 буква, 3 цифры и еще 2 буквы).

Штраф для автомобильных номеров зависит от количества совпадающих цифр: три совпадают - 1000 у.е., две любые цифры совпадают - 500 у.е., все цифры разные - 100 у.е.

Примечание: все комментарии в примерах будут на русском языке, что нарушает PEP8. Это сделано для простоты восприятия примера.

Шаг №1. Предположим, что все уже написано за нас:

```
def main():
    salary = calculate_salary()
    print(salary)


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

Шаг №2. Но это не будет работать, так как функция calculate_salary отсутствует. Исправим это:

```
 def main():
    salary = calculate_salary()
    print(salary)


def calculate_salary():
    """
    Функция считает зарплату сотрудника ДПС,
    считывая исходные данные с клавиатуры.

    :returns: Зарплата сотрудника

    .. todo:: Реализовать функцию
    """
    pass


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

Теперь код будет успешно выполняться, но сейчас функция ничего не делает. В документации к функции явно указано, что ее надо реализовать. Такая функция называется заглушкой и широко используется в практике программирования.

Шаг №3. Займемся реализацией функции. Предположим, что у нас написаны все необходимые функции, и напишем calculate_salary так:

```
def calculate_salary():
    """
```

```
    Функция считает зарплату сотрудника ДПС,
    считывая исходные данные с клавиатуры.

    :returns: Зарплата сотрудника
    """
    sum_of_fines = 0
    speed_of_car, number_of_car = read_data()
    while not detect_chief(number_of_car):
        if speed_of_car > 60:
            sum_of_fines += calculate_fine(number_of_car)
        speed_of_car, number_of_car = read_data()
    return sum_of_fines
```

Заметим, что в этом решении делегировано в отдельные функции всё, что только можно было делегировать. Разве что проверка превышения скорости осталась без собственной функции, но она уж и так слишком тривиальна.

Заметим, что в групповой разработке ПО старший программист на этом завершил бы свою работу. Ему осталось только сформулировать для каждой из функций её заглушку и документирующий комментарий, раздать работу по написанию мелких функций младшим программистам и идти пить чай.

Шаг №4. На данном этапе пропустим написание функций-заглушек и сразу приведём окончательное решение.

```
def main():
    salary = calculate_salary()
    print(salary)


def calculate_salary():
    """
    Функция считает зарплату сотрудника ДПС,
    считывая исходные данные с клавиатуры.

    :returns: зарплата сотрудника
    """
    sum_of_fines = 0
    speed_of_car, number_of_car = read_data()
    while not detect_chief(number_of_car):
        if speed_of_car > 60:
            sum_of_fines += calculate_fine(number_of_car)
        speed_of_car, number_of_car = read_data()
    return sum_of_fines


def read_data():
```

```
    """
    Считывает следующую строку данных.

    :returns: tuple(int, str) - скорость, номер машины.
    """
    data = input().split()
    return int(data[0]), data[1]


def detect_chief(number_of_car):
    """
    Проверяет, принадлежит ли данный номер начальнику.

    :param number_of_car: номер автомобиля
    :returns: True, если номер принадлежит начальнику, иначе False
    """
    return number_of_car == "A999AA"


def calculate_fine(number_of_car):
    """
    Считает штраф для автомобиля с конкретным номером.

    :param number_of_car: номер автомобиля
    :returns: Целое число, размер штрафа
    """
    if is_super_number(number_of_car):
        return 1000
    elif is_good_number(number_of_car):
        return 500
    else:
        return 100


def is_super_number(number_of_car):
    """
    Проверяет, является ли номер «крутым» (совпадение трёх цифр)

    :param number_of_car: номер автомобиля
    :returns: True, если номер «крутой», иначе False
    """
    return number_of_car[1] == number_of_car[2] == number_of_car[3]


def is_good_number(number_of_car):
```

```
    """
    Проверяет, является ли номер «хорошим» (совпадение двух цифр)

    :param number_of_car: номер автомобиля
    :returns: True, если номер «хороший», иначе False
    """
    return number_of_car[1] == number_of_car[2] or \
           number_of_car[1] == number_of_car[3] or \
           number_of_car[2] == number_of_car[3]


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

3 Выводы

Обычно перед использованием функции сначала реализуют ее. Мы же сейчас делали все наоборот. Сначала использовали функцию, потом писали для нее заглушку и только вконце дописывали реализацию. В этом и заключается процесс проектирования “сверху вниз”.