

Оглавление

1	Тестирование и отладка программ	2
1.1	Знакомство с курсом	2
1.2	Структурное программирование	3
1.2.1	Профессионализм в программировании	3
1.2.2	Культура программирования	4
1.2.3	Выбор идентификаторов	5
1.2.4	Структурное программирование	6
1.2.5	Проектирование приложения «сверху вниз»	7
1.3	Тестирование и отладка	12
1.3.1	Зачем нужно тестировать программу	12
1.3.2	Контрактное программирование	14
1.3.3	Модульное тестирование и Test-Driven Development	17
1.3.4	Библиотека doctest	20
1.3.5	Библиотека unittest	22

Неделя 1

Тестирование и отладка программ

1.1 Знакомство с курсом

Структура курса:

- Неделя 1
 - Культура программирования;
 - Проектирование «сверху-вниз»;
 - Программирование по контракту;
 - Тестирование.
- Неделя 2
 - Объектно-ориентированное мышление;
 - UML-диаграммы.
- Неделя 3
 - Паттерны проектирования;
 - «Декоратор», «Адаптер», «Наблюдатель».
- Неделя 4
 - Паттерн «Цепочка обязанностей»;
 - Паттерн «Абстрактная фабрика»;
 - Конфигурация программ.
- Неделя 5
 - Курсовой проект.

1.2 Структурное программирование

1.2.1 Профессионализм в программировании

Профессиональный программист отличается от любителя тем, что он:

- адекватно оценивает:
 - срок создания программного продукта;
 - необходимые для этого ресурсы.
- хорошо планирует:
 - архитектуру программного продукта;
 - последовательность разработки.
- выполняет работу качественно:
 - ПО надёжно функционирует;
 - программный код легко читать и сопровождать.

Классический процесс это каскадная модель разработки ПО (**Waterfall**):

1. Анализ требований к ПО
2. Разработка архитектуры программы
3. Кодирование
4. Тестирование и отладка
5. Инсталляция и поддержка

Проектирование архитектуры программы помогает понять, что именно надо делать и когда. Профессиональному программисту важно:

- Знать парадигмы программирования:
 - Структурная парадигма (движение «сверху-вниз»);
 - Модульная парадигма (разбиение программы на модули);
 - Объектно-ориентированное проектирование.
- Использовать готовые архитектурные решения - паттерны;
- Уметь визуализировать дизайн программы.

Читабельность кода (**readability**) очень важна, ведь она даёт возможность быстро понимать смысл исходного текста, лучше видеть ошибки, увереннее модифицировать код и уменьшить объём внутренней документации.

Важно и качество работы программы. Она должна делать то, что заявлено в техническом задании (ТЗ) без ошибок и быстро.

1.2.2 Культура программирования

Код читается намного больше раз, чем пишется и поэтому критически важна «читабельность» программного кода. Для этого в Python существует универсальный стиль кода **PEP 8**:

1. Внешний вид кода:

- Кодировка для файла исходного текста только юникод т.е. **UTF-8**. А идентификаторы, переменные, функции и комментарии **ASCII**;
- Никогда не смешивать табуляции и пробелы;
- Желательно 4 пробела на один уровень отступа;
- Ограничить максимальную длину строки 79-ю символами (для этого можно использовать уже имеющиеся скобки или обратный слэш). Перенос строки делать строго после бинарного оператора.

2. Пробелы в выражениях и инструкциях

- Окружайте ровно одним пробелом с каждой стороны:
 - операторы присваивания (`=`, `+=`, `-=` и т.п.);
 - операторы сравнения (`==`, `<`, `>`, `!=`, `<=`, `>=`, `in`, `not in`, `is`, `is not`);
 - логические операторы (`and`, `or`, `not`).
- Ставьте пробелы вокруг арифметических операций
- Избегать пробелов:
 - Сразу после или перед скобками `()`, `[]`, `{}`;
 - Перед запятой, точкой с запятой, двоеточием;
 - Перед открывающейся скобкой при вызове функций или скобкой, после которой следует индекс или срез.

3. Пустые строки

- Используйте пустые строки, чтобы отделить друг от друга логические части функции;
- Отделять одной пустой строкой определения методов внутри класса;
- Функции верхнего уровня и определения классов отделять двумя пустыми строчками.

4. Комментарии

- Не объясняйте очевидное и обновляйте комментарии вместе с кодом;

- Блок комментариев обычно объясняет код, идущий *после* блока, и должен иметь тот же отступ, что и код;
- «Встрочные» комментарии отделяйте хотя бы двумя пробелами от инструкции и начинайте их с символа `#` и одного пробела;
- Там, где это возможно, вместо комментариев используйте документ-строки;
- Пишите комментарии на грамотном английском языке. Первое слово с большой буквы и в конце ставьте точку. Предложения отделяйте двумя пробелами.

1.2.3 Выбор идентификаторов

Соглашение об именах:

- Модуль (или пакет) должен называться коротко, записываться маленькими буквами и без подчёркиваний. Например, **`mymodule`**;
- Классы и исключения называются несколькими словами слитно, каждое из которых с большой буквы. Например, **`ClassName`**;
- Функции, переменные и методы записываются несколькими словами с маленькой буквы, через знак подчёркивания. Например, **`function_name`**;
- Иногда функции называют так: **`notDesiredFunctionName`**, но первая буква - маленькая;
- Глобальные константы пишутся заглавными буквами через подчёркивание: **`GLOBAL_CONSTANT`**.

Использование символа подчёркивания:

- В начале — функция для внутренних нужд: **`_internal_function`**;
- В конце — избегание конфликта с зарезервированным словом: **`reserve_`**;
- Два подчёркивания в начале — скрываемая функция или атрибут: **`__hidden`**;
- Два в начале, два в конце — функция с особым использованием (согласно документации). **`__magic_method__`**.

Глобальные переменные:

- Это плохой стиль. Их надо избегать везде, где возможно;
- Поведение функций начинает зависеть от неявно заданных обстоятельств;
- Мешают распараллеливать код.

Рассмотрим пример использования PEP 8:

```
class MyClass:
    """
    Use PEP 8 to be a Master of Code!
    """
    def __init__(self, class_):
        self._internal = class_
    def public_method(self, x: int):
        """
        some document string here
        """
        if x > 0:
            return x + 1
    return x
```

Видим, что здесь соблюдаются все вышеперечисленные рекомендации.
Замечания:

- В случае конфликта стилей, если имя — часть [API](#), то оно должно быть согласовано со стилем кода интерфейса, а не реализации;
- Имена должны содержать только символы ASCII и означать только английские слова;
- Имена должны отражать смысл объекта.

[Краткая версия PEP 8 на русском](#)

[Полная версия PEP 8 на английском языке](#)

1.2.4 Структурное программирование

Структурное программирование — это методология, облегчающая создание больших программ. У начинающих программистов проблема в том, что они сконцентрированы на синтаксисе языка. А методология это то, что помогает приподняться над синтаксисом. Рассмотрим аналогию с водителями:

Начинающий водитель думает о мелких технических моментах:

- Как не врезаться?
- Разрешён ли обгон?
- Можно ли тут останавливаться?

Опытный водитель задаёт глобальные вопросы:

- Какая цель у поездки?
- Каков оптимальный маршрут?
- Как объехать нужные места?

Аналогично, начинающий программист думает о синтаксисе и деталях.

Может быть, этот кусок программы вообще не надо писать.

Опытные программисты думают о безопасности, комфорте и надёжности.

Принципы структурного программирования:

- Откажитесь от использования `goto`;
- Стройте программу из вложенных конструкций: последовательность, ветвление, циклы;
- Оформляйте повторяющиеся фрагменты программы в виде функций;
- Разработка программы ведётся пошагово, методом «сверху вниз».

1.2.5 Проектирование приложения «сверху вниз»

Главная задача разработки сверху вниз состоит в том, чтобы управлять концентрацией внимания программиста. Чтобы не теряться в большом количестве программного кода и разбить исходную задачу на подзадачи.

Рассмотрим проектирование приложения «сверху вниз» на примере создания графического приложения с библиотекой графики Джона Зелли. Импортируем библиотеку `graphics`. Напишем функцию `main` и пропишем краткое рисование окна

```
import graphics as gr #импорт библиотеки

def main():
    window = gr.GraphWin("My Image", 600, 600)
    draw_image(window) # здесь должно быть содержимое
    window.getMouse()

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

Пишем в нём вызов функции `draw_image()`, в которой всё будет отрисовываться. И в этот момент можно пойти по неправильному пути — сразу начать реализовывать эти функции. А подход «сверху вниз» говорит нам выделять подзадачи и отбрасывать их (реализовать позднее). Поскольку функции нет, то интерпретатор не даст нам запустить программу. Внутри функции `draw_image()` не нужно задумываться как устроено окно `window`. Поэтому мы прописываем её заготовку:

```
def draw_image(window):
    pass # TODO
```

И в этом состоянии программа уже может быть запущена и мы увидим окошко. Теперь будем её прописывать из соображения, что пейзаж будет состоять из фона и домика. Мы делегируем выполнение двух подзадач в отдельные функции. Это и называется **декомпозицией**:

```
def draw_image(window):
    house_x, house_y = window.width // 2, \
```

```

        window.height * 2 // 3
    house_width = window.width // 3
    house_height = house_width * 4 // 3

    draw_background(window)
    draw_house(window, house_x, house_y,
               house_width, house_height)

```

В данном случае названия функций говорят сами за себя, но старайтесь везде писать документ строки. Мы параметризовали рисование домика. И чтобы всё снова запускалось, пропишем функции-заглушки:

```

def draw_background(window):
    pass      # TODO
def draw_house(window, house_x, house_y,
               house_width, house_height):
    pass      # TODO

```

Такие функции-заглушки (и документ-строки к ним) надо прописывать в моменты концентрации, до отхода от компьютера. Всё снова будет работать и мы в любой момент можем их заполнить.

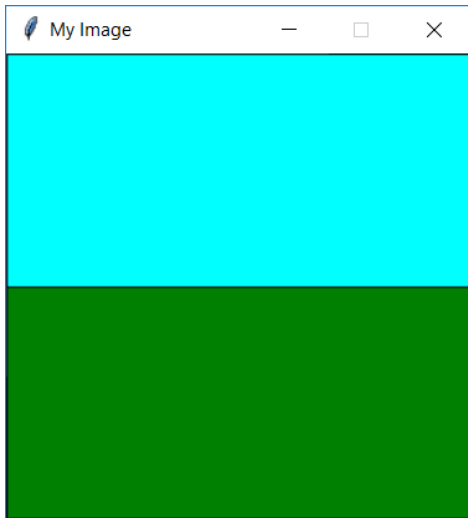
Теперь реализуем функции и посмотрим на результат.

```

def draw_background(window):
    earth = gr.Rectangle(gr.Point(0, window.height // 2),
                        gr.Point(window.width - 1,
                                window.height - 1))

    earth.setFill("green")
    earth.draw(window)
    sci = gr.Rectangle(gr.Point(0, 0),
                      gr.Point(window.width - 1, window.height // 2))
    sci.setFill("cyan")
    sci.draw(window)

```

Запустив программу видим, нарисованный фон, потому что эта функция уже вызвана. При движении снизу вверх мы бы вначале написали эту функцию, а потом бы придумывали, как ее использовать. Аналогия: можно делать велосипед, начиная с деталей, но не продумав, как их собирать. А мы делаем, спускаясь сверху вниз. Сначала вызвали функцию и поставили на неё заглушку, а потом, спустившись вниз, сделали бэкграунд чтобы он отвечал требованию — залить всё окно.

Проделаем ещё одну итерацию проектирования «сверху вниз» на функции рисования домика. Дом будет состоять из трёх частей: цоколь, стены и крыша. Это и будут три подзадачи. В начале пишем:

```
def draw_house(window, x, y, width, height):

    foundation_height = height // 8
    walls_height = height // 2
    walls_width = 7 * width // 8
    roof_height = height - walls_height - \
                  foundation_height

    draw_house_foundation(window, x, y, width,
                          foundation_height)
    draw_house_walls(window, x, y - foundation_height,
                    walls_width, walls_height)
    draw_house_roof(window, x,
                   y - foundation_height - walls_height,
                   width, roof_height)

# обязательно создаём функции-заглушки перед переключением
def draw_house_foundation(window, x, y, width, height):
    pass # TODO

def draw_house_walls(window, x, y, width, height):
    pass # TODO

def draw_house_window(window, x, y, width, height):
    pass # TODO
```

У нас есть три функции с прописанным интерфейсом. Мы можем делегиро-

вать их реализацию трём программистам, чтобы они независимо реализовывали каждый свою функцию. А потом всё сливается в репозиторий и работает. Каждый из них может выполнить работу как хочет, но соблюдая контракт. Итоговая реализация функций независимо друг от друга:

```
def draw_house_foundation(window, x, y, width, height):

    foundation = gr.Rectangle(gr.Point(x - width // 2, y),
                              gr.Point(x + width // 2,
                                         y - height))

    foundation.setFill("brown")
    foundation.draw(window)

def draw_house_walls(window, x, y, width, height):
    walls = gr.Rectangle(gr.Point(x - width // 2, y),
                        gr.Point(x + width // 2,
                                   y - height))

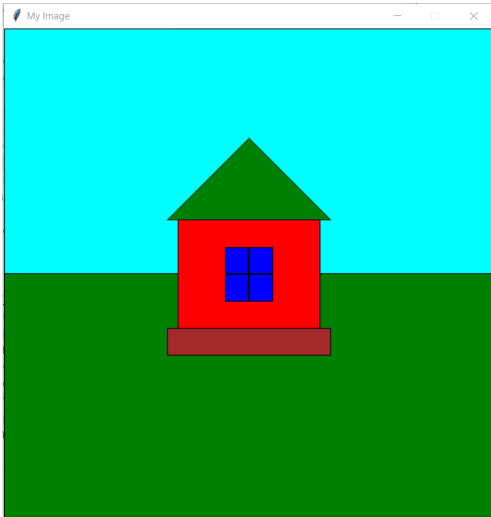
    walls.setFill("red")
    walls.draw(window)
    draw_house_window(window, x, y - height // 4,
                      width // 3, height // 2)

def draw_house_window(window, x, y, width, height):
    glass = gr.Rectangle(gr.Point(x - width // 2, y),
                        gr.Point(x + width // 2,
                                   y - height))

    glass.setFill("blue")
    line1 = gr.Line(gr.Point(x, y),
                    gr.Point(x, y - height))
    line2 = gr.Line(gr.Point(x - width // 2,
                              y - height // 2),
                    gr.Point(x + width // 2,
                              y - height // 2))

    glass.draw(window)
    line1.draw(window)
    line2.draw(window)
    line1.setOutline("black")
    line2.setOutline("black")
    line1.setWidth(2)
    line2.setWidth(2)
```

```
def draw_house_roof(window, x, y, width, height):  
    roof = gr.Polygon(gr.Point(x - width // 2, y),  
                      gr.Point(x + width // 2, y),  
                      gr.Point(x, y - height))  
    roof.setFill("green")  
    roof.draw(window)
```



Таким образом, мы получили готовый домик с окошком и крышей. Структурное программирование прекрасно ложится на групповую работу программистов, когда подзадача делегирована. И она не просто делегирована в отдельную функцию, она еще и передана конкретному программисту, который может рисовать окно как угодно, но не вылезая за x и y . Это очень удобно с точки зрения дальнейшей модификации и работы — если захочется нарисовать ещё три таких домика, можно просто из главной функции сделать пару вызовов с другими параметрами.

Итоговый код программы

1.3 Тестирование и отладка

1.3.1 Зачем нужно тестировать программу

Обратная связь — это данные, которые с выхода поступают на вход. Её типы:

- Положительная обратная связь усиливает сигнал на выходе. В случае разработки это:
 - положительные отзывы конечных пользователей;
 - запросы пользователей на новую функциональность;
 - увеличение объема продаж.
- Отрицательная обратная связь гасит сигнал
 - негативные отзывы конечных пользователей;
 - отсутствие интереса к программному продукту;
 - падение объёма продаж.

К сожалению, отрицательная обратная связь обычно поступает слишком поздно. Тут-то и нужен **тестировщик** — человек, который даёт участникам проекта по разработке ПО отрицательную обратную связь о качестве программного продукта на самой ранней стадии, когда ещё не поздно всё исправить.

Тестирование (Quality Control) — это проверка соответствия между реальным и ожидаемым поведением программы, которая проводится на конечном наборе специально выбранных тестов.

Обязанности тестировщика:

- находить дефекты («баги»);
- вносить описание найденного дефекта в систему отслеживания ошибок (Bug tracking system);
- описывать способы воспроизведения ошибок;
- создавать отчёты о тестировании для каждой версии продукта;
- (дополнительно) читать и исправлять документацию;
- (дополнительно) анализировать и уточнять требования к программе;
- (дополнительно) создавать ПО для автоматизации процесса тестирования.

Главный результат работы тестировщика - повышение качества ПО.
Аспекты качества программы:

1. Функциональность:

- пригодность к использованию;
- правильность выполнения задачи;
- поддержка стандартов;
- защищенность (security).

2. Надёжность:

- низкая частота отказов;
- отказоустойчивость;
- способность к восстановлению.

3. Практичность (user-friendly):

- понятность в использовании;
- управляемость;
- привлекательность.

4. Эффективность:

- время отклика программы;
- объём использования ресурсов ПК.

5. Сопровождаемость;

6. Переносимость.

И для каждого аспекта качества программы есть соответствующий вид тестирования. Например, функциональное тестирование проверяет пригодность к использованию, правильность работы и защищенность программы.

Классификация тестирования по масштабу (как их видит тестировщик):

1. Модульное тестирование (unit testing) — тестирование отдельных операций, методов и функций;
2. Интеграционное тестирование — проверка корректности взаимодействия модулей между собой;
3. Системное тестирование — тестирование на уровне пользовательского интерфейса.

С технической точки зрения эти три вида тестирования похожи друг на друга: если у вас есть какой-то инструмент, например, модульного тестирования, то его иногда можно применить и к системному тестированию.

Крайне желательно иметь тестировщика у себя в команде, когда проект разрастётся, но кто может быть тестировщиком на начальных этапах?

- Программист или Старший программист (Team Leader) не могут быть тестировщиками. Они ходят только «по протоптанному», жалеют собственный код и не сильно придираются к нему;
- Менеджер проекта, технический писатель, эксперт предметной области или представитель заказчика могут, потому что тестировщик должен не знать деталей реализации, воспринимать программу как чёрный ящик и не быть слишком привязанным к ней;
- Методика **test-driven development (TDD)** позволяет программисту быть самому себе эффективным тестировщиком.

Когда ошибка найдена (самостоятельно или с помощью тестировщика), необходимо **отладить** программу:

1. Понять суть ошибки и обнаружить её причину;
2. Локализовать её в исходном тексте программы;
3. Устранить ошибку.

Типичные задачи отладки: узнать текущие значения переменных и выяснить, по какому пути выполнялась программа.

Существуют два пути отладки:

- Использование отладчиков («дебаггеров»);
- Логгирование (вывод отладочных сведений в файл).

1.3.2 Контрактное программирование

Поиск ошибок в программе — это очень дорогостоящее, неприятное и утомительное занятие. Поэтому есть методики, уменьшающие количество ошибок и позволяющие избежать их еще на этапе создания программы. Одна из них — это **проектирование по контракту**. **Design by contract** — это метод проектирования программ, основанный на идее взаимных обязательств и преимуществ взаимодействующих элементов программы. Так же называется «контрактное программирование». Автор — Бертран Мейер.

- Взаимодействующие элементы программы:
 - «клиент» — это вызывающая функция, объект или модуль;
 - «поставщик» — это вызываемая функция, объект или модуль.
- «Контракт» между ними — это взаимные
 - обязательства — то, что требуется каждой стороне соблюсти при взаимодействии;

– преимущества — та выгода, которая получается при соблюдении обязательств другой стороной.

- Архитектор программы определяет **формальные, точные и верифицируемые** спецификации интерфейсов.

Как и в бизнесе, клиент и поставщик действуют в соответствии с определенным контрактом. Архитектор программы должен определить формальные, точные и верифицируемые спецификации интерфейсов для функций и методов.

Содержание контракта:

- Предусловия — обязательства клиента перед вызовом функции-поставщика услуги;
- Постусловия — обязательства функции-поставщика, которые обязаны быть выполнены в итоге её работы;
- Инварианты — условия, которые должны выполняться как при вызове функции-поставщика, так и при окончании его работы.

Предусловия, постусловия, инварианты записываются через **формальные утверждения корректности — assertions**:

- Синтаксис: `assertion, "Сообщение об ошибке!"` ;
- Пример: `assert 0 <= hour <= 23, "Hours should be in range of 0..23"` ;
- Жёсткое падение облегчает проверку выполнения контрактов во время отладки программы;
- Проверка `assert` работает только в режиме отладки (`__debug__ is True`)

Библиотека **PyContracts** позволяет элегантно ввести в Python элементы проектирования по контракту (в том числе и проверку типов).

Способы описания предусловий:

1. Через параметры декоратора:

```
@contract(a='int,>0',
          b='list[N],N>0',
          returns='list[N]')
def my_function(a, b):
    pass
```

2. Через аннотации типов

```
@contract
def my_function(a: 'int,>0',
               b: 'list[N],N>0') -> 'list[N]':
    pass
```

3. Через документ-строки

```
@contract
def my_function(a, b):
    """ Function description.
        :type a: int,>0
        :type b: list[N], N>0
        :rtype: list[N]
    """
    pass
```

Пример контракта для функции умножения матриц:

```
@contract
def matrix_multiply(a, b):
    """ Multiplies two matrices together.

        :param a: The first matrix. 2D array.
        :type a: array[MxN],M>0,N>0

        :param b: The second matrix. 2D array
                   of compatible dimensions.
        :type b: array[NxP],P>0

        :rtype: array[MxP]
    """
    return numpy.dot(a, b)
```

Обратите внимание: можно потребовать не только положительного значения ширины или высоты матрицы, но и того, чтобы у матриц *a* и *b* соответствующие размеры были равны друг другу.

Библиотека PyContracts позволяет еще и отключить все проверки, когда ваша программа будет отлажена и отправлена в релиз. Это делается вызовом функции:

```
contracts.disable_all()
```

Или установкой переменной окружения

```
DISABLE_CONTRACTS
```

Плюсы контрактного программирования:

- улучшает дизайн программы;
- повышает надёжность работы программы;
- повышает читаемость кода, поскольку контракт документирует обязательства функций и объектов;
- увеличивает шанс повторного использования кода;
- актуализирует документацию программного продукта.

1.3.3 Модульное тестирование и Test-Driven Development

Декомпозиция программы на функции, классы и модули позволяет осуществлять модульное или компонентное тестирование (unit testing). Юнит-тесты — это способ проверить работу функции или метода отдельно от всей программы, вместе взятой. Рассмотрим юнит тестинг пузырьковой сортировки:

```
def bubble_sort(A):
    """
    Sort on place with Bubblesort algorithm
    :type A: list
    """
    for bypass in range(1, len(A)):
        for k in range(0, len(A) - bypass):
            if A[k] > A[k+1]:
                A[k], A[k+1] = A[k+1], A[k]

def main():
    A = input("Enter some words:").split()
    bubble_sort(A)
    print("Sorted words:", ' '.join(A))

main()
```

Поскольку тестирование вручную:

1. Займёт время;
2. Придётся делать вручную каждый раз;
3. И при этом основная часть программы может ещё не работать.

Нам будет удобнее проверять работу функции отдельно от основного кода и автоматизировать это тестирование. Напишем юнит тест для сортировки:

```
def test_sort():
    A = [4, 2, 5, 1, 3]
    B = [1, 2, 3, 4, 5]
    bubble_sort(A)
    print("#1:" , "Ok" if A == B else "Fail")

test_sort()
```

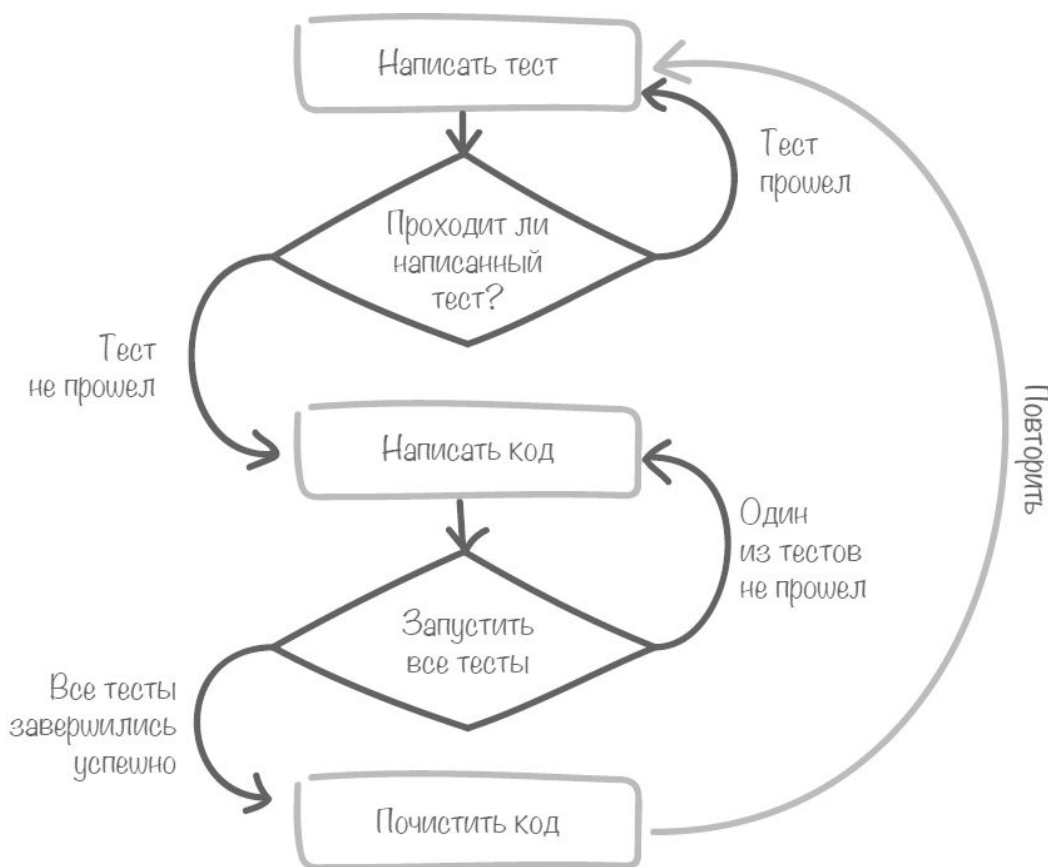
Для большей гарантии стоит добавить несколько различных тестов, которых достаточно для проверки работоспособности функции.

```
def test_sort():
    A = [4, 2, 5, 1, 3]
    B = [1, 2, 3, 4, 5]
    bubble_sort(A)
    print("#1:" , "Ok" if A == B else "Fail")

    A = list(range(40,80)) + list(range(40))
    B = list(range(80))
    bubble_sort(A)
    print("#2:" , "Ok" if A == B else "Fail")

    A = [4, 2, 4, 2, 1]
    B = [1, 2, 2, 4, 4]
    bubble_sort(A)
    print("#3:" , "Ok" if A == B else "Fail")

test_sort()
```



Опережающее тестирование — способ совмещения роли тестировщика и программиста, при котором сначала пишутся unit-тесты, а потом уже пишется код согласно техническому заданию.

Разработка через тестирование или Test-Driven Development (TDD) — это итеративная методика разработки программ, в которой (опережающее) тестирование ставится во главу угла. И оно управляет процессом дизайна программного продукта. Если существующие тесты проходят нормально, значит, в коде нет известных проблем. Создав тест для выявления недостающего функционала, мы чётко выявляем задачу, которую собираемся решить. И вот такими циклами разработки мы двигаем проект вперёд, создавая новую и новую функциональность программы, которая всегда гарантированно покрыта модульными тестами.

Кроме этого, разработка тестов выявляет **дефекты дизайна приложений**:

- Каковы обязанности тестируемой системы?
- Что и когда она должна делать?
- Какой API удобен для того, чтобы тестируемый код выполнял задуманное?
- Что нужно тестируемой системе для выполнения своих обязательств?
- Что мы имеем на выходе?

- Какие есть побочные эффекты работы?
- Как узнать, что система работает правильно?
- Достаточно ли хорошо определена эта правильность?

Преимущества TDD:

1. Эффективное совмещение ролей (тестирование собственного кода);
2. Рефакторинг без риска испортить код;
3. Реже нужно использовать отладчик;
4. Повышает уверенность в качестве программного кода.

Но работы станет больше: вместо одной функции, придётся писать две (саму функцию и её юнит-тест). Зато будет затрачено меньше времени на поиск ошибок.

[Пример разработки через тестирование](#)

1.3.4 Библиотека `doctest`

Разберём быстрый и удобный способ тестирования программы на примере функции проверки корректности скобочной структуры. Допустим, интерфейс уже проработан. С помощью библиотеки **`doctest`** мы можем сделать примеры из документ-строки действующими (выполняться как юнит-тесты). Добавим несколько примеров корректных и некорректных скобочных последовательностей. И оформим это как если бы мы вызывали это в интерпретаторе:

```
import stack

def is_braces_sequence_correct(seq: str) -> bool:
    """
    Check correctness of braces sequence in statement
    >>> is_braces_sequence_correct("() (())")
    True
    >>> is_braces_sequence_correct("() [()]")
    True
    >>> is_braces_sequence_correct("")
    False
    >>> is_braces_sequence_correct("[()")
    False
    >>> is_braces_sequence_correct("[()")
    False
```

```

"""
if __name__ == "__main__":
    import doctest
    doctest.testmod()

```

Подключили библиотеку doctest и вызвали оттуда функцию testmod:

```

Got nothing
*****
File "D:/Google Drive/Документы/Tech/Python/1 неделя (Хирьянов)/исходные коды/braces.py",
Failed example:
    is_braces_sequence_correct("() [()]")
Expected:
    True
Got nothing
*****
File "D:/Google Drive/Документы/Tech/Python/1 неделя (Хирьянов)/исходные коды/braces.py",
Failed example:
    is_braces_sequence_correct("")
Expected:
    False
Got nothing
*****
File "D:/Google Drive/Документы/Tech/Python/1 неделя (Хирьянов)/исходные коды/braces.py",
Failed example:

```

В результате для каждого завалившегося теста мы получили отчёт о выполнении этих комментариев. Пока реализации нет, для всех видим "Expected: ... Got nothing" Допишем реализацию:

```

import stack

def is_braces_sequence_correct(seq: str) -> bool:
    """
    Check correctness of braces sequence in statement
    >>> is_braces_sequence_correct("() (())")
    True
    >>> is_braces_sequence_correct("() [()]")
    True
    >>> is_braces_sequence_correct("")
    False
    >>> is_braces_sequence_correct("[()")
    False
    >>> is_braces_sequence_correct("[()]")
    False
    """

```

```

correspondent = dict(zip("([{", ")]}"))
for brace in seq:
    if brace in "([{":
        stack.push(brace)
        continue
    elif brace in ")]}":
        if stack.is_empty():
            return False
        left = stack.pop()
        if correspondent[left] != brace:
            return False

return stack.is_empty()
if __name__ == "__main__":
    import doctest
    doctest.testmod()

```

При запуске тестов мы увидим только: "Process finished with exit code 0" . Тестирование производится, но мы не видим ошибок и всё работает хорошо. Мы прямо в процессе программирования, еще разрабатывая интерфейс функции и прописывая документ-строку, уже написали тесты простым и естественным способом.

1.3.5 Библиотека unittest

Рассмотрим более сложное тестирование при помощи библиотеки **unittest** на примере функции, вычисляющей числа Фибоначчи.

```

def fib(n: int) -> int:
    """
    Calculates fibonacci number by it's index
    """
    pass

```

Покроем эту функцию модульными тестами в отдельном файле, каждый из которых будет вызовом специального метода.

Для этого создадим тестирующий модуль, который будет содержать тестирующий класс, наследующийся от `unittest.TestCase`. И все методы, начинающиеся со слова `test`, будут тестирующими.

```

import unittest
from fibonacci import fib

class TestFibonacciNumbers(unittest.TestCase):
    def test_zero(self):
        self.assertEqual(fib(0), 0)

```

Простой юнит тест, проверяющий что число Фибоначчи от 0 это действительно 0 (мы так захотели). `assertEqual` проверяет равенство своих параметров. Модуль `unittest` автоматически запустит этот `test_zero` и оформит результат красиво, если тест будет заваливаться.

Другой вариант использования библиотеки `unittest`: если в вашем модуле достаточно много микротестов(подслучаев), и если у вас произойдет заваливание на одном подслучае, то весь этот маленький тест-кейс просто будет выброшен. А можно было бы протестировать их все, чтобы даже было видно, с каким номером это происходило. Это делается при помощи вызова функции `subTest` с указанием некоторого индекса. Проверим первые 1-5 и 10-ое числа Фибоначчи:

```
...
def test_simple(self):
    for n, fib_n in (1, 1), (2, 1), (3, 2), \
                    (4, 3), (5, 5):
        with self.subTest(i=n):
            self.assertEqual(fib(n), fib_n)

def test_positive(self):
    self.assertEqual(fib(10), 55)
```

Когда производится анализ тестовых случаев, мы должны задуматься, а не могут ли нашу функцию вызвать с какими-то некорректными параметрами, например, для отрицательных чисел.

Давайте в этом случае выбрасывать исключение. Библиотека `unittest` позволяет перехватить это исключение. Добавим таким образом тесты на отрицательные и дробные числа.

```
...
def test_negative(self):
    with self.subTest(i=1):
        self.assertRaises(ArithmeticError, fib, -1)
    with self.subTest(i=1):
        self.assertRaises(ArithmeticError, fib, -10)

def test_fractional(self):
    self.assertRaises(ArithmeticError, fib, 2.5)
```

Отличие `assert`-ов в 1 случае (`test_simple`) и во 2 (`test_negative`) в том, что в `test_simple` мы вызываем функцию самостоятельно, а во втором передаём функцию и список её аргументов.

Если сейчас запустить программу, возникнет целый отчёт, где, что и почему упало:

```

=====
FAIL: test_simple (__main__.TestFibonacciNumbers) (i=3)
-----
Traceback (most recent call last):
  File "D:/Google Drive/Документы/Тех/Python/1 неделя (Хирьянов)/исходные коды/fibonacci_test.py", line 13, in test_simple
    self.assertEqual(fib(n), fib_n)
AssertionError: None != 2

=====
FAIL: test_simple (__main__.TestFibonacciNumbers) (i=4)
-----
Traceback (most recent call last):
  File "D:/Google Drive/Документы/Тех/Python/1 неделя (Хирьянов)/исходные коды/fibonacci_test.py", line 13, in test_simple
    self.assertEqual(fib(n), fib_n)
AssertionError: None != 3

=====
FAIL: test_simple (__main__.TestFibonacciNumbers) (i=5)
-----
Traceback (most recent call last):
  File "D:/Google Drive/Документы/Тех/Python/1 неделя (Хирьянов)/исходные коды/fibonacci_test.py", line 13, in test_simple
    self.assertEqual(fib(n), fib_n)
AssertionError: None != 5

=====
FAIL: test_zero (__main__.TestFibonacciNumbers)
-----
Traceback (most recent call last):

```

В результате он сообщает, что пять тестов прошло и десять ошибок найдено. Анализ этих ошибок позволит найти необходимые поправки. Напишем нашу функцию правильно:

```

def fib(n: int) -> int:
    """
    Calculates fibonacci number by it's index
    """
    f = [0, 1] + [0] * (n - 1)
    for i in range(2, n + 1):
        f[i] = f[i - 1] + f[i - 2]
    return f[n]

```

Теперь при запуске тестирующего модуля мы поймаем ошибки `test_negative` и `test_fractional`. Допишем в начале строчки:

```

def fib(n: int) -> int:
    """
    Calculates fibonacci number by it's index
    """
    if not isinstance(n, int) or n < 0:
        raise ArithmeticError
    f = [0, 1] + [0] * (n - 1)
    for i in range(2, n + 1):
        f[i] = f[i - 1] + f[i - 2]

```



```
return f[n]
```

После исправления все тесты проходят и мы получаем короткий отчёт:

```
.....
-----
Ran 5 tests in 0.000s

OK

Process finished with exit code 0
```

Еще пример покрытия функции юнит-тестами

Оглавление

2	Объектно-ориентированное проектирование	2
2.1	Введение в ООП	2
2.1.1	Чем хорошо объектно-ориентированное программирование?	2
2.1.2	Отличие класса от объекта	3
2.1.3	Отличие интерфейса класса от реализации	4
2.2	Парадигмы ООП	6
2.2.1	Инкапсуляция и полиморфизм в Python	6
2.2.2	SOLID принципы ООП	8
2.3	Разработка системы классов	9
2.3.1	Парадигма наследования в Python	9
2.3.2	Абстрактные классы и библиотека abc	10
2.3.3	UML-нотация и диаграммы классов	12
2.4	Рефакторинг кода	14
2.4.1	Объектно-ориентированный рефакторинг программ	14

Неделя 2

Объектно-ориентированное проектирование

2.1 Введение в ООП

2.1.1 Чем хорошо объектно-ориентированное программирование?

Объектно-ориентированный подход:

- Объектно-ориентированный образ мышления;
- Переход от переменных и функций к объектам;
- Взаимодействие между объектами.

Когда вы описываете, например, движение автомобиля по дороге, то оперируете не огромным количеством переменных, которые изменяются по каким-то законам, а вы оперируете целым объектом — автомобилем, который выполняет какие-то действия.

Класс состоит из:

- переменных, которые хранятся и изменяются вместе с классом;
- методов — функций, которые выполняют какие-то действия с классом.

ООП применяется повсеместно прежде всего, благодаря простоте и естественности использования: мы один раз пишем (или импортируем из библиотеки) какой-то класс, который обладает каким-то состоянием и может его изменять. А потом из различных классов и объектов мы строим большое приложение как из кубиков Lego. При использовании объектов класса, мы концентрируемся на том, как их связать друг с другом, а не на том, как они использованы и реализованы внутри.

Другим достоинством ООП является то, что важные компоненты собраны в одном месте и код становится структурированным. Кроме того, инкапсуляция позволяет защищать некоторые данные от несанкционированного доступа, что может уберечь систему от весьма существенных поломок.

Объектно-ориентированный подход позволяет быстро и легко расширять и обновлять существующую систему, не внося в неё существенных изменений.

Отдельным важным достоинством является возможность повторного использования ООП систем: объект, решающий некоторую распространенную задачу, может быть встроен как компонент в большое количество различных систем, которым требуется решений этой задачи. Это сильно упрощает и ускоряет разработку приложений.

Наряду с этим, у ООП есть главный недостаток — слишком сложная система классов. Для полноценного использования средств ООП требуется хорошо понимать его возможности и парадигмы. Очень сложно корректно спроектировать систему классов таким образом, чтобы она работала действительно эффективно. В различных библиотеках содержатся десятки классов, имеющие сотни методов и способные взаимодействовать друг с другом.

И поскольку всё, что есть в Python, является объектами, ООП играет очень важную роль. Целые числа, строки, списки, кортежи и словари — это всё объекты. Даже функции и классы сами по себе являются целыми объектами. Когда вы складываете два числа, вы вызываете метод `add` от целого числа. Когда вы добавляете элемент в список, вы вызываете метод списка `append`.

2.1.2 Отличие класса от объекта

Жизненный путь **объекта**:

1. Создаётся в памяти;
2. Происходит инициализация вызовом метода `init`;
3. Ссылка на объект сохраняется в некоторую переменную;
4. Мы свободно пользуемся объектом, обращаясь к нему по ссылке: вызываем его методы и просматриваем его состояние;
5. Когда на объект нет внешних ссылок, он удаляется.

Удаление происходит в одной из двух ситуаций. Поскольку у каждого объекта есть счётчик внешних ссылок на него, при отсутствии обращений к объекту, он будет удален с использованием механизма `DECREF`.

Если же счётчик не достиг нуля, то `Garbage Collector` освобождает память от объектов, на которые нет внешних ссылок из программы (но могут быть ссылки из других объектов, не связанных с программой).

Определения понятий:

- Объект — это структура, которая хранится в памяти, имеет некоторое состояние, умеет его изменять и взаимодействовать с другими объектами при помощи методов;

- Класс — это описание структуры объекта. В нём описаны переменные и методы объекта, как он будет создаваться в памяти и что необходимо сделать при его удалении;
- Объект - класс, позволяющий хранить информацию о других объектах: как будет создаваться и что сможет сделать некоторый объект. Кроме того, он может самостоятельно создавать объекты других типов.

2.1.3 Отличие интерфейса класса от реализации

В языках Java или C интерфейс это аналог абстрактных базовых классов. Но в Python таких интерфейсов не существует.

Интерфейс (абстракция) в Python— это совокупность всех способов доступа и взаимодействия с некоторым объектом. Интерфейс объекта состоит из публичных методов и переменных, доставшихся классу от его родительских классов и из его собственных публичных полей. Интерфейс будет видеть пользователь объекта, и именно он будет описан в пользовательской документации. Разработка интерфейса:

- Интерфейс должен быть продуманным;
- Прежде всего надо думать о пользователе;
- Все необходимые методы взаимодействия с объектом должны входить в интерфейс;
- Все служебные методы и переменные в интерфейс не входят.

Не забывайте две важные концепции из дзена Python "Красивое лучше, чем уродливое." и "простое лучше, чем сложное." Таким образом интерфейс это оболочка, а внутренности это реализация.

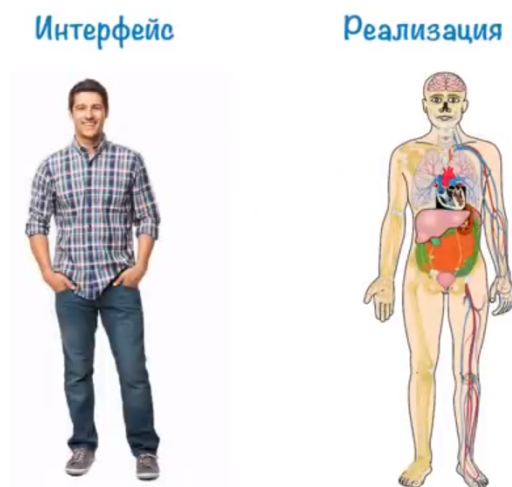


Рис. 2.1: Интерфейс и реализация

Отличия интерфейса и релаксации на примере списка (list):

Метод	Что делает
<code>list.append(x)</code>	Добавляет элемент в конец списка
<code>list.extend(L)</code>	Расширяет список <code>list</code> , добавляя в конец все элементы списка <code>L</code>
<code>list.insert(i, x)</code>	Вставляет на <code>i</code> -ый элемент значение <code>x</code>
<code>list.remove(x)</code>	Удаляет первый элемент в списке, имеющий значение <code>x</code> . <code>ValueError</code> , если такого элемента не существует
<code>list.pop([i])</code>	Удаляет <code>i</code> -ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
<code>list.index(x, [start [, end]])</code>	Возвращает положение первого элемента со значением <code>x</code> (при этом поиск ведется от <code>start</code> до <code>end</code>)
<code>list.count(x)</code>	Возвращает количество элементов со значением <code>x</code>
<code>list.sort([key=функция])</code>	Сортирует список на основе функции
<code>list.reverse()</code>	Разворачивает список
<code>list.copy()</code>	Поверхностная копия списка
<code>list.clear()</code>	Очищает список

Это всё интерфейс. Но реализация списка в Python это **3000** строк на языке C. Там написано, сколько памяти и какого типа следует выделять при создании списка, каким образом в нём хранятся переменные и как получить к ним доступ.

Например, при вызове метода **append()**, в конец списка вставляется новый элемент. Причём, если место заканчивается, под список выделяется больше памяти, и чем больше список, тем больше выделяется. При вставке элемента в середину списка, кроме выделения памяти придётся сдвинуть последующие элементы на 1.

При выталкивании элемента (метод **pop()**), возвращается элемент, лежащий в последней ячейке списка, и его длина уменьшается на 1. Но реального удаления объекта из списка не происходит до тех пор, пока его длина не станет меньше половины размера выделенной памяти. Только тогда данные будут очищены, а память освобождена.

Интерфейс — это совокупность всех публичных переменных и методов класса. Реализация — это внутреннее устройство объекта, которое не видит пользователь.

2.2 Парадигмы ООП

2.2.1 Инкапсуляция и полиморфизм в Python

Парадигмы ООП:

- наследование;
- инкапсуляция;
- полиморфизм.

Наследование позволяет создавать сложные системы классов. Основные понятия в наследовании это класс-родитель и класс-потомок, наследующий все публичные методы и переменные класса-родителя. Это позволяет строить иерархии классов. На этом принципе построены многие паттерны проектирования.



Рис. 2.2: Наследование

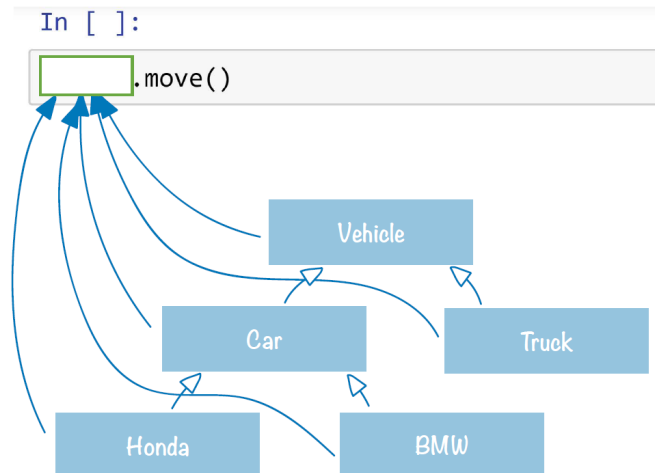


Рис. 2.3: Полиморфизм

Полиморфизм тесно связан с наследованием, интерфейсом и реализацией. Этот принцип позволяет использовать один и тот же код для работы с различными объектами, имеющими одинаковый интерфейс, но обладающими различной его реализацией. Такая ситуация как раз часто возникает при наследовании: мы можем работать с классами-потомками так же, как работали бы с родительским классом.

Пусть у нас есть объект, который мы хотим описать. В процедурном программировании это бы делалось с помощью переменных и функций. В случае ООП мы упаковываем их в единый компонент. Это и есть **инкапсуляция**.

Есть мнение, что в Python нет инкапсуляции, но это не верно. Часто под инкапсуляцией понимают сокрытие данных от пользователя - скрытые методы будут доступны только самому экземпляру класса. Соккрытие данных позволяет, во-первых, построить простой и удобный интерфейс класса и, во-вторых, упростить его реализацию.

Разберём 1 случай: в вашем классе реализован важный для работы системы служебный метод, который бесполезен для пользователя. Тогда этот метод можно скрыть, сделав приватным.

Во 2 случае рассмотрим класс «котик», у которого есть публичные поля возраст и вес. Без проверок входных данных в эти поля можно ввести отрицательные значения, что бессмысленно и может вызвать ошибки. Однако, можно сделать атрибут приватным, а доступ к нему осуществлять с использованием специальных методов — **getter()**(который возвращает значение атрибута) и **setter()**(который устанавливает значение атрибута, при прохождении проверки входных данных). Кроме того, приватные поля не наследуются.

Псевдоскрытие данных в Python устроено так: перед приватными полями следует поставить два нижних подчеркивания (`__private`). Если попытаться прочитать эти поля у экземпляра класса Python выдаст ошибку:

```
class SampleClass:
    def __int__(self):
        self.public = "public"
        self.__private = "private"

    def public_method(self):
        print(f"private data: {self.__private}")

c = SampleClass()
c.public
'public'
c.__private
AttributeError: 'SampleClass' object has no attribute '__private'
c._SampleClass__private
'private'
```

Просто Python автоматически переименовывает приватные поля по правилам: нижнее подчеркивание, название класса, название приватного поля с двумя нижними подчеркиваниями в начале. Посмотреть список всех полей можно командой `dir`.

К трём основным парадигмам ООП часто относят **абстракцию**. Она говорит, что в коде мы описываем только модель объекта, которая обязана содержать ключевые характеристики моделируемого объекта и не должна содержать лишней информации. Но её нельзя полностью отнести к парадигмам ООП, ведь это скорее принцип хорошего кода. В Python нет специальных механизмов, позволяющих реализовать принцип абстракции, и всё ложится на плечи разработчика. Да и само выполнение принципа носит лишь рекомен-

дательный характер: кто знает, какие свойства объекта будут важными для пользователя, а какие нет.

2.2.2 **SOLID** принципы ООП

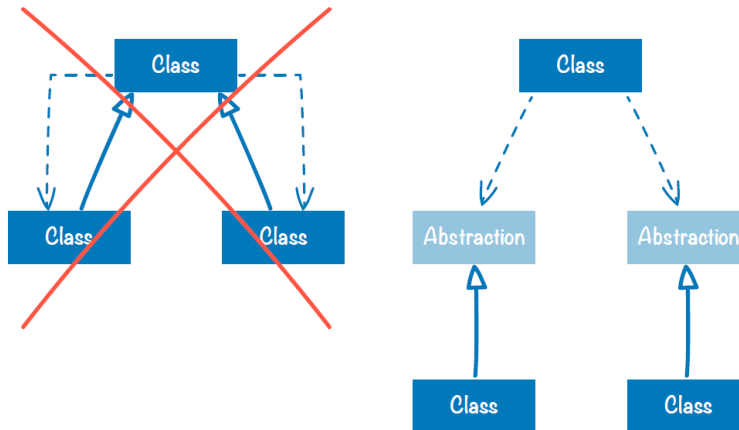
Частые проблемы ООП:

- негибкие системы;
- избыточные зависимости между компонентами;
- ненужная функциональность;
- невозможность повторного использования.

Пять принципов создания качественных систем — **SOLID**:

- **Single responsibility** — у каждого объекта должна быть только одна ответственность и всё его поведение должно быть направлено на обеспечение только этой ответственности;
- **Open/closed** — классы должны быть открыты для расширения (новыми сущностями), но закрыты для изменения;
- **Liskov substitution** (принцип Барбары Лисков) — функции, которые используют базовый тип должны иметь возможность использовать его подтипы не зная об этом;
- **Interface segregation** (принцип разделения интерфейсов) — клиенты не должны зависеть от методов, которые они не используют;
- **Dependency inversion** (принцип инверсии зависимостей) — модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракции. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Разберём подробнее последний принцип. Допустим, что модули верхних уровней напрямую зависят от модулей нижних уровней. Внесём изменение в модуль нижнего уровня. Так как от него зависят модули верхних уровней, их так же придётся менять. Таким образом, придётся переписать часть программы. Если же добавить абстракции, от которых зависят модули обоих уровней, то прямой зависимости нет, и изменения в модулях одного уровня не затронут другой.



2.3 Разработка системы классов

2.3.1 Парадигма наследования в Python

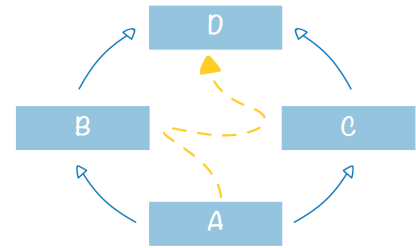
Пусть у нас есть простая иерархия классов: родитель и потомок, который наследует все методы родительского класса (за некоторым исключением). При этом у него могут быть свои методы и атрибуты, отличные от родительского класса. Некоторые из методов родительского класса у потомка могут быть переопределены и выполнять другую задачу.

Мы бы могли реализовать это и без наследования, однако в таком случае пришлось бы переписывать в "потомка" интерфейс "родителя" и полностью его реализовать. Хуже, когда часть интерфейса, которая должна быть общей, изменяется: переписывать нужно одновременно все места, где должен быть общий код. В этом случае и спасает наследование. Потомку передаются публичные методы и переменные родителя, но приватные поля не наследуются.

Инициализатор класса (метод `init`) также наследуется. Однако если его переопределить у класса потомка, при создании объекта метод `init` родительского класса вызван не будет. Чтобы всё-таки вызвать этот метод, нужно явно прописать его вызов в соответствующем методе класса потомка.

Но не во всех случаях использование сложной иерархии классов оправдано. Чем структура сложнее, тем тяжелее поддерживать программу и читать её код. Например, линейная структура, в которой каждый следующий класс наследуется от предыдущего (при условии что система классов не будет расширяться), является избыточной.

Python поддерживает множественное наследование, благодаря которому класс потомок может обладать функциональностью нескольких классов родителей. Но и здесь есть подводные камни, например: "Ромб смерти". А наследуется от B и C. А они в свою очередь от общего предка D, в котором есть некоторый метод `my_method`, переопределённый в B и C по-разному. Тогда непонятно, из какого родительского класса потомку A брать этот метод. С версии 2.3 установлен порядок, по принципу C3-линеаризации: в нашем случае, в A будет реализация метода в первом из классов, где он будет явно определён.



2.3.2 Абстрактные классы и библиотека `abc`

Абстрактные классы не имеют экземпляров, они описывают некоторый интерфейс (например животное - абстрактный класс, от которого наследуются ежик, котик и утка). Причём мы можем общаться со всеми потомками класса используя этот интерфейс. Когда мы объявляем класс абстрактным, мы говорим системе следить, чтобы экземпляров класса не было.

Кроме того, абстрактные методы реализуются не в родительском классе, а в классе-потомке. Но надо следить, чтобы они действительно везде были реализованы. **Абстрактные методы** — это методы, содержащие только определение без реализации.

Виртуальный класс — класс, который при множественном наследовании не включается в классы-потомки, а заменяется ссылкой в них. Но в Python нет явного управления памятью, и о них сложно говорить. Во многих ООП языках класс-наследник является просто копией родительского класса с некоторыми дополнительными методами. При множественном наследовании это может приводить к неочевидным проблемам, например, "ромб смерти". В классах B и C будут содержаться копии метода, определённого в A, и они будут считаться отдельными методами. А в Python такой проблемы нет.

Реализация **виртуальных функций** определяется уже на этапе выполнения. Т.е. в ООП языках мы говорим, что на данном месте будет вызван некоторый метод, но какой именно метод, мы определим хотим определить только когда дойдём до места. В Python все методы считаются виртуальными и такая концепция в нём не рассматривается.

```
from abc import ABC, abstractmethod

class A:
    @abstractmethod
    def do_something(self):
        print("Hi!")
```

```
a = A()
a.do_something()
```

Hi!

Но мы хотели, чтобы нельзя было создать экземпляр абстрактного класса. Абстрактный метод должен быть реализован не в самом методе, а в его потомках. Попробуем это сделать:

```
class A(ABC):
    @abstractmethod
    def do_something(self):
        print("Hi!")
```

```
a = A()
```

TypeError: Can't instantiate abstract class A with abstract method...

Вылетает ошибка: мы не можем создать экземпляр класса, в котором содержится нереализованный абстрактный метод. Создадим класс B, который будет наследоваться от A. Реализуем в B метод `do_something_else()`, который будет что-то печатать. При этом мы не реализовали метод `do_something`.

```
class B(A):
    def do_something_else(self):
        print("Hello")
```

```
b = B()
b.do_something()
```

TypeError: Can't instantiate abstract class B with abstract methods...

У нас вылетает та же ошибка: абстрактный метод не реализован.

```
class B(A):
    def do_something(self):
        print("Hi2!")

    def do_something_else(self):
        print("Hello")
```

```
b = B()
b.do_something()
b.do_something_else()
```

Hi2! Hello!

2.3.3 UML-нотация и диаграммы классов

UML (Unified Modelling Language) — унифицированный язык моделирования. Это графический язык, позволяющий описывать структуру программ, бизнес-процессов и систем. Хотя UML и не язык программирования, по UML-диаграмме возможно построение объектно-ориентированного кода.

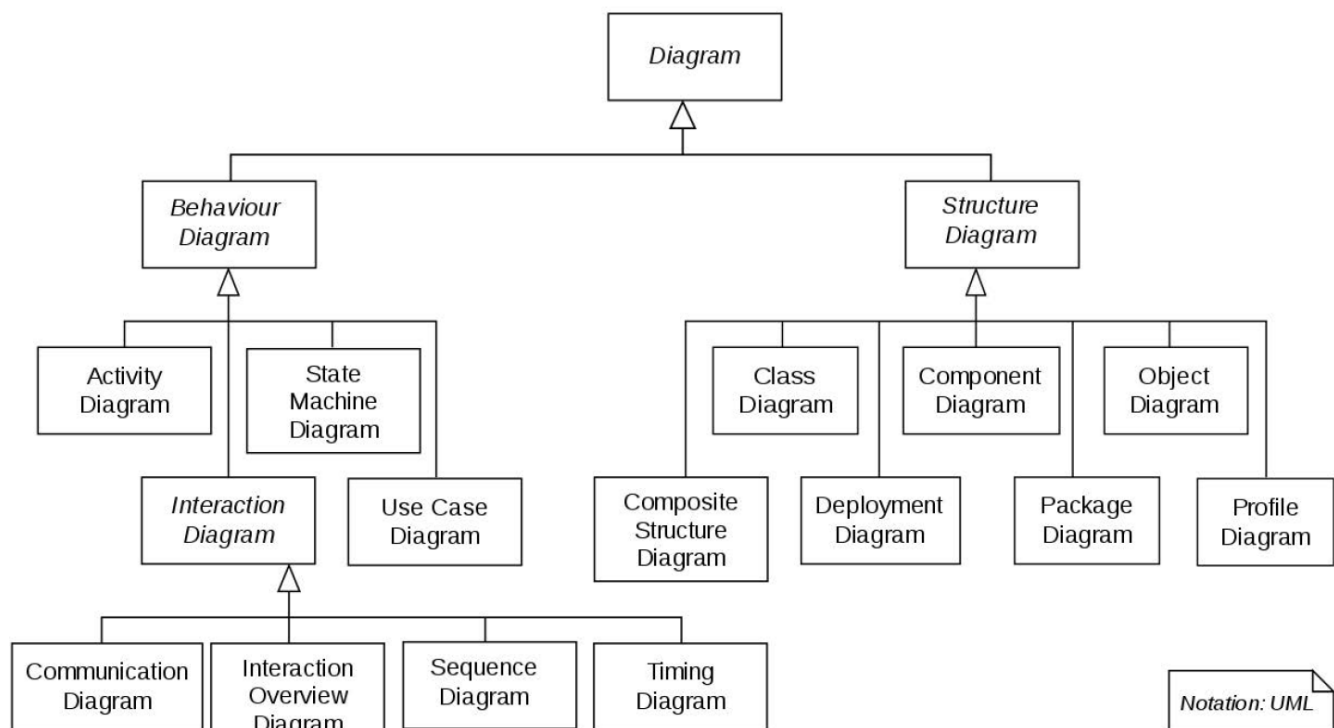
История UML:

- Первые спецификации выпущены в 1996 году;
- Microsoft, IBM, Oracle и HP присоединились к разработке и в 1997 выпущена версия 1.0;
- UML 1.4.1 вошёл в международный стандарт ISO;
- В 2005 выпущена UML 2.0;
- В последней версии стандарта ISO описан UML 2.4.1;
- На данный момент самая новая версия стандарта это 2.5, выпущенная в 2015.

UML-диаграммы бывают двух основных видов:

- **Структурные** — описывают структуру конкретного объекта;
- **Поведенческие** — описывают поведение и взаимодействие между компонентами системы.

Полная структура выглядит так (и она тоже описана на языке UML):



Язык UML состоит из пяти основных элементов:

- **Фигура** может использоваться как контейнер для других типов элементов;
- **Рамка** тоже контейнер;
- **Линия** позволяет соединять или разделять различные объекты. Стрелки тоже линии;
- **Текст**;
- **Значок** — особое обозначение, использующееся в схеме.

Строгих правил построения нет. Главное — это чтобы диаграмма была понятна читающему её пользователю. Чаще всего мы будем встречаться с диаграммой классов. Классы и объекты будем обозначать в виде прямоугольников. Есть две основные цели: показать внешние взаимосвязи (тогда объекты это просто прямоугольники с названием) или показать внутреннюю структуру класса (она прописывается внутри прямоугольника под заголовком. Сначала описываются переменные класса, а потом его методы. Крайне желательно придерживаться обозначений PEP 8. Названия методов заканчивать парой круглых скобок (с существенными аргументами внутри). Кроме того можно отметить тип возвращаемого значения метода или тип переменной. Важно указать, являются ли поля публичными («+» перед названием или приватными («-»)).

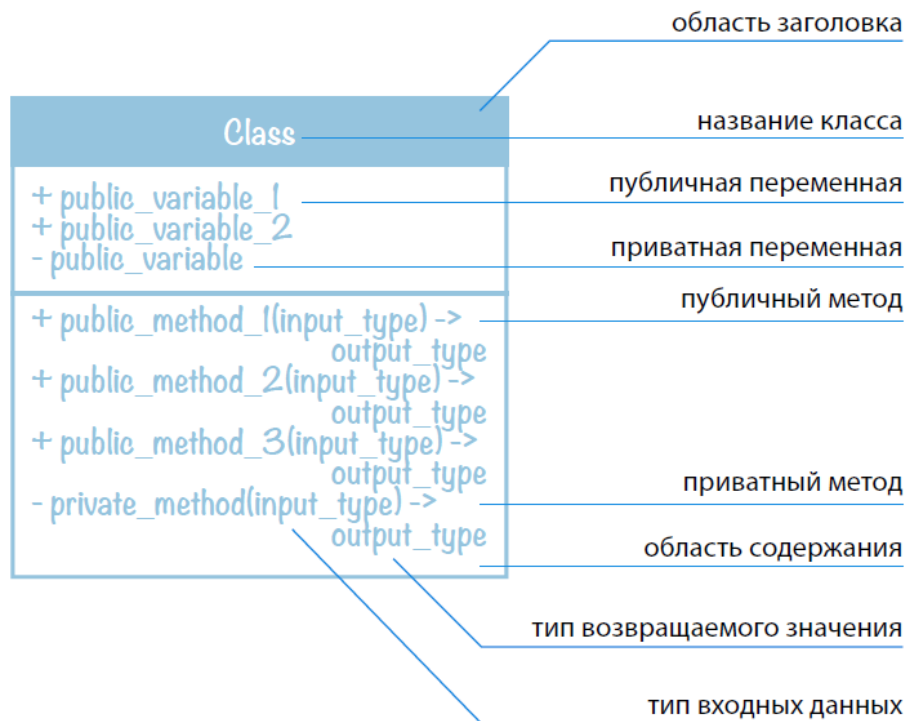








Рис. 2.4: Диаграмма классов

Типы отношений:

-  Ассоциация — простая связь двух объектов. Например, работник и решаемая им задача;
-  Агрегация — отношение между целым и его частью (например, список и его содержимое). Если исчезнет контейнер, содержимое останется. Например, пенал и карандаш;
-  Композиция — более жесткая зависимость, в которой при уничтожении контейнера будет уничтожено и его содержимое. Например, человек и сердце;
-  Наследование (обобщение) — отношение, в котором один из объектов является надтипом (обобщением другого), другой же является его подтипом. Например, млекопитающие и животные;
-  Имплементация (реализация). В абстрактном классе описан интерфейс, который должен быть реализован в наследниках этого класса. Пример: наследники абстрактного класса и он сам;
-  Отношение зависимости показывает, когда изменение одного объекта ведёт к изменению другого. Их следует избегать. Например, система и её модуль.

2.4 Рефакторинг кода

2.4.1 Объектно-ориентированный рефакторинг программ

Рефакторинг (переработка) кода — это процесс преобразования внутренней структуры кода, который призван облегчить его понимание и читабельность.

Оптимизация — это процесс переработки внутренней структуры кода с целью увеличения его производительности. При оптимизации читаемость кода может даже ухудшиться.

Реинжиниринг — это процесс полного переписывания некоторого блока кода. При реинжиниринге может измениться и работа некоторого блока кода. Когда нужен рефакторинг?

- Сложная архитектура системы;
- Использование объектов, затрудняющих понимание системы;
- Упрощение схем наследования;
- Преобразование процедурного кода в объектно-ориентированный.

Признаки, отличающие код, которому срочно нужен рефакторинг. Это так называемый «код с запашком».

- дублирование кода; Решение:
 - выделение повторяющегося кода в функции;
 - если код дублируется в разных подклассах, перенос дублирующегося кода в базовый класс;
 - если же базового класса нет, создать родительский класс для подклассов с дублирующимся кодом.
- длинные методы; Решение:
 - деление методов на логические блоки;
 - выделение общих переменных блоков в качестве аргументов.
- большие классы (слишком большой функционал); Решение:
 - делить на более мелкие фрагменты.
- длинные списки параметров; Решение:
 - выделение параметров в отдельный объект;
 - передача объекта, содержащего часть параметров, в виде параметра.
- «жадные» функции (чрезмерное использование одним классом методов другого); Решение:
 - перенос методов из одного класса в другой.
- избыточные временные переменные; Решение:
 - код, который используется в исключительных случаях вынести в отдельный класс;
 - при необходимости обращаться к объекту этого класса.
- классы данных (класс содержит только поля без методов); Решение:
 - вместо такого класса использовать объект-контейнеры (например, словарь).
- не сгруппированные данные (используется большое количество связанных переменных и функций). Решение:
 - следует выделять такие переменные и функции в класс;
 - переименование переменных для повышения понятности кода.

Оглавление

3	Паттерны проектирования (часть 1)	2
3.1	Достоинства паттернов проектирования и особенности их применения в Python	2
3.1.1	Введение в паттерны проектирования	2
3.1.2	Виды паттернов проектирования и задачи	3
3.2	Паттерн Decorator	6
3.2.1	Задача паттерна Decorator	6
3.2.2	Реализация декоратора класса	8
3.3	Паттерн Adapter	11
3.3.1	Задача паттерна Adapter	11
3.3.2	Реализация адаптера класса	13
3.4	Паттерн Observer	15
3.4.1	Задача паттерна Observer	15
3.4.2	Реализация паттерна Наблюдатель	16

Неделя 3

Паттерны проектирования (часть 1)

3.1 Достоинства паттернов проектирования и особенности их применения в Python

3.1.1 Введение в паттерны проектирования

В ходе работы программистам часто приходится решать похожие задачи. В создании игр это может быть программирование шаблонов поведения противников; в сетевых приложениях — ведение логов или отслеживание состояний клиентских приложений. Использование паттернов (шаблонов) проектирования позволяет использовать качественные решения, проверенные временем, а не изобретать велосипед.

Применяя их, вы избежите многих ошибок, ведь паттерны отлажены поколениями программистов.

Паттернов проектирования очень много. Некоторые из них, например Adapter, используются постоянно. В различных областях, таких как: трёхмерная графика, обработка данных и сетевое взаимодействие, существуют специальные шаблоны, позволяющие эффективно решать задачи, специфичные для данных областей.

Паттерн проектирования (Design Pattern) — повторяемая архитектурная конструкция, применяемая для решения часто встречающихся задач.

Шаблоны проектирования по своей сути очень похожи на кулинарные рецепты: они также описывают некоторый рекомендуемый способ решения какой-то стандартной задачи. Знание паттернов, умение их видеть и применять для решения конкретной задачи - это признак опытного программиста.

Литература:

1. К.Александр — «A Pattern Language: Towns, Buildings, Constructions», 1977 год. Это первое упоминание о паттернах проектирования.
2. Э.Гамм, Р.Хелм, Р.Джонсон, Д.Влиссидес — «Design Patterns: Elements of Reusable Object-Oriented Software», 1995 год. Это первая книга,

где паттерны были применены к ООП. Было описано 23 паттерна проектирования.

Далее в курсе будут рассмотрены самые часто встречающиеся паттерны проектирования.

3.1.2 Виды паттернов проектирования и задачи

Классификация паттернов по уровню абстракции:

- **Низкоуровневые паттерны (идиомы)** — паттерны уровня языка программирования. Например, реализация тернарного оператора или генерация списков на лету (list comprehension);
- **Паттерны проектирования** — более абстрактные и менее привязанные к конкретному языку паттерны, используемые для решения больших задач. Они будут рассмотрены далее;
- **Архитектурные шаблоны** — паттерны, описывающие архитектуру программной системы полностью и не привязанные к языку программирования или решению частных задач. Например, клиент-серверная архитектура или архитектура Model-View-Controller (MVC), позволяющая разделить бизнес логику и её графическое представление.

Примеры архитектурных шаблонов:

- Model-View-Controller (MVC);
- Model-View-Presenter;
- Model-View-View Model;
- Presentation-Abstraction-Control;
- Naked Objects;
- Hierarchial Model-View-Controller;
- View-Interactor-Presenter-Entity-Routing (VIPER).

Виды паттернов проектирования и их назначение:

Структурные шаблоны модифицируют структуру объектов. Могут быть использованы для получения более сложных структур из классов или для реализации альтернативного доступа к объектам.

Основные представители:

- Адаптер (adapter) — взаимодействие несовместимых объектов;
- Мост (bridge) — разделение абстракции и реализации;

- Компоновщик (composite) — агрегирование нескольких объектов в одну структуру;
- Декоратор (decorator) — динамическое создание дополнительного поведения объекта;
- Фасад (facade) — сокрытие сложной структуры за одним объектом, являющимся общей точкой доступа;
- Приспособленец (flyweight) — общий объект, имеющий различные свойства в разных местах программы;
- Заместитель (проху) — контроль доступа к некоторому объекту.

Порождающие паттерны используются при создании различных объектов. Они призваны разделить процесс создания объекта и использования их системой. Например, для реализации способа создания объектов независимо от типов создаваемых объектов, или для сокрытия процесса создания объекта от системы, которая его использует.

- Абстрактная фабрика (abstract factory) — создание семейств взаимосвязанных объектов;
- Строитель (builder) — сокрытие инициализации для сложного объекта;
- Фабричный метод (factory method) — общий интерфейс создания экземпляров подклассов некоторого класса;
- Отложенная инициализация (lazy initialization) — создание объекта только при доступе к нему;
- Пул одиночек (multiton) — повторное использование сложных объектов вместо повторного создания;
- Прототип (object pool) — упрощение создания объекта за счёт клонирования уже имеющегося;
- Одиночка (singleton) — объект, присутствующий в системе в единственном экземпляре.

Поведенческие паттерны описывают способы реализации взаимодействия между объектами различных типов. Самые основные примеры:

- Цепочка обязанностей (chain of responsibility) — обработка данных несколькими объектами;
- Интерпретатор (interpreter) — решение частой незначительно изменяющейся задачи;

- Итератор (iterator) — последовательный доступ к объекту-коллекции;
- Хранитель (memento) — сохранение и восстановление объекта;
- Наблюдатель (observer) — оповещение об изменении некоторого объекта;
- Состояние (state) — изменение поведения в зависимости от состояния;
- Стратегия (strategy) — выбор из нескольких вариантов поведения объекта;
- Посетитель (visitor) — выполнение некоторой операции над группой различных объектов.

Конкурентные паттерны — особые шаблоны, реализующие взаимодействия различных процессов и потоков. Применяются в параллельном программировании.

- Блокировка (lock) — позволяет потоку захватывать общие ресурсы на период выполнения;
- Монитор (monitor) — механизм синхронизации и взаимодействия процессов, обеспечивающий доступ к общим неразделяемым ресурсам;
- Планировщик (scheduler) — позволяет планировать порядок выполнения параллельных процессов с учетом приоритетов и ограничений;
- Активный объект (active object) — позволяет отделять поток выполнения некоторого метода от потока, в котором данный метод был вызван.



Рис. 3.1: Виды паттернов проектирования

3.2 Паттерн Decorator

3.2.1 Задача паттерна Decorator

Декоратор (decorator) — структурный паттерн, который используется для динамического создания дополнительного поведения объекта.

Пусть для некоторого множества задач определен класс, объекты которого эту задачу могут решать. И пусть для некоторых из объектов требуется дополнительная функциональность, реализованная в оригинальном классе. Это большая проблема. Классическим решением этой проблемы было бы создание подкласса, в котором реализована необходимая функциональность. Но что если заранее неизвестно, какая функциональность или какие свойства нужны объекту? Можно попытаться реализовать классы-потомки со всеми возможными комбинациями нужных свойств. Но число комбинаций растёт экспоненциально и становится очень большим.

Декоратор позволяет красиво решить проблему реализации дополнительной функциональности. Разберёмся, как он работает на примере класса «пицца». Свойства: размер, набор ингредиентов и стоимость. Определим абстрактный базовый класс Pizza с методами посчитать стоимость (`get_cost()`), узнать ингредиенты (`get_ingredients()`) и посмотреть размер (`get_size()`). Чем больше пицца и её набор ингредиентов, тем больше её стоимость.

Определим реализацию этого класса — `margarita`, состоящая из теста, томатной пасты и сыра. Все остальные пиццы являются лишь её улучшениями.

Объявим абстрактный класс декоратор, в котором объявим конструктор, принимающий на вход пиццу. Именно наследники этого абстрактного декоратора и будут изменять нашу Маргариту, делая её вкуснее. Добавляемые ингредиенты: мясо, помидоры, перец и оливки. Пицца может быть трёх размеров: большой, средней и маленькой.

Пусть пицца уже частично готова, и на данном шаге нужно добавить в неё мясо. Тогда список ингредиентов будет пополнен, и стоимость пиццы возрастёт на стоимость мяса, а размер не изменится. Если же изменить размер пиццы, то изменится соответствующее свойство пиццы и мультипликатор стоимости.

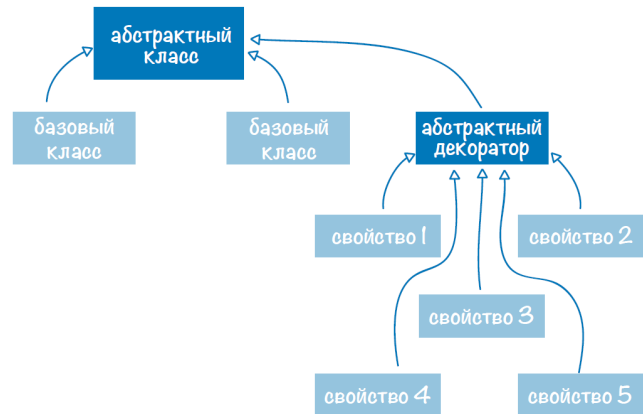


Рис. 3.2: Схема абстрактного декоратора

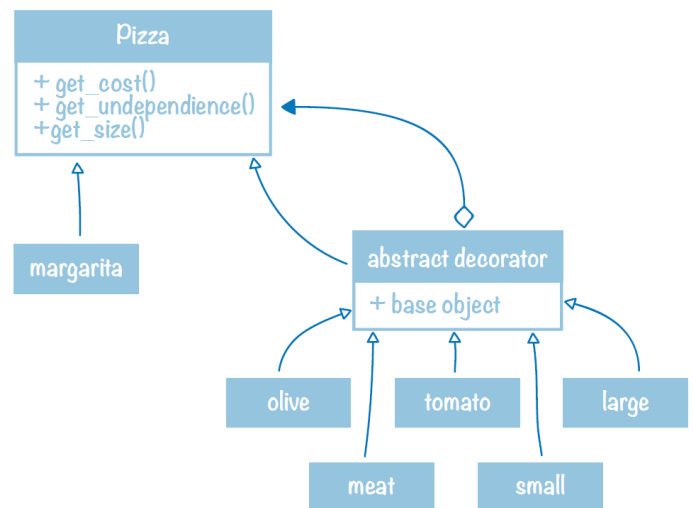


Рис. 3.3: Декоратор на примере пиццы

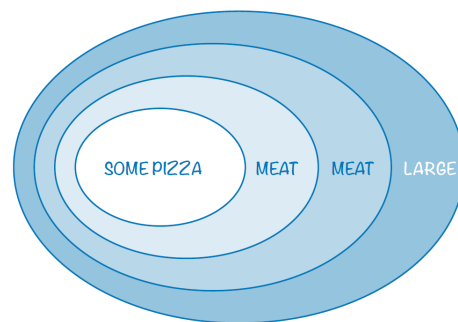


Рис. 3.4: Вложенность декораторов

Использование паттерна декоратор позволяет динамически добавлять объекту функциональность, которой у него до этого не было. Реализуется же он путём создания абстрактного базового класса, абстрактного декоратора для этого класса и их наследников — базовых классов и базовых декорато-

ров. Применение декоратора к объекту заключается в оборачивании нашего объекта в некоторый декоратор с дополнительной функциональностью. К одному объекту может быть применено произвольное число декораторов.

3.2.2 Реализация декоратора класса

Рассмотрим реализацию конкретного примера паттерна «Декоратор». Пусть мы занимаемся разработкой игры. В нашей игре есть различные животные: некоторые живут на суше, некоторые в воде. Есть хищники и травоядные. Есть быстрые и медленные. Определим основной объект животное (animal) и декораторы, которые могут к нему применяться. Напишем абстрактный базовый класс Существо (creature) с методами: кормиться, двигаться и мычать.

```
from abc import ABC, abstractmethod
class Creature(ABC):
    @abstractmethod
    def feed(self):
        pass

    @abstractmethod
    def move(self):
        pass

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

Теперь объявим базовое животное (наследник класса «Существо»), которое будет травоядным (т.е. писать нам "I eat grass"), способным ходить (т.е. выдавать нам "I walk forward" и что-то кричать:

```
class Animal(Creature):
    def feed(self):
        print("I eat grass")

    def move(self):
        print("I walk forward")

    def make_noise(self):
        print("W000!")
```

Для создания иерархии декораторов сначала опишем абстрактный декоратор (он так же наследуется от базового класса) , от которого уже будут наследоваться конкретные реализации, добавляющие нашему объекту допол-

нительную функциональность. Конструктор (`__init__`) должен принимать обязательный аргумент `self` и в качестве дополнительного аргумента наш декорируемый объект. Объявим у декоратора необходимые методы, взятые из декорируемого объекта (пока без изменений): `feed`, `move`, `make_noise`. В этих методах мы будем вызывать соответствующий метод базового объекта.

```
class AbstractDecorator(Creature):
    def __init__(self, obj):
        self.obj = obj

    def feed(self):
        self.obj.feed()

    def move(self):
        self.obj.move()

    def make_noise(self):
        self.obj.make_noise()
```

Теперь создадим несколько декораторов. Первый декоратор — водоплавающее животное. Водоплавающие вместо того, чтобы ходить, будут плавать. И не будут уметь говорить. Переопределим у них методы `move` и `make_noise`.

```
class Swimming(AbstractDecorator):
    def move(self):
        print("I swim")

    def make_noise(self):
        print("...")
```

Следующее свойство, которое мы хотим добавить — хищник, который будет поедать других животных. Создадим класс Хищник и переопределим метод `feed` так, что он будет есть каких-то других животных.

```
class Predator(AbstractDecorator):
    def feed(self):
        print("I eat other animals")
```

Определим третий класс свойств — скорость движения. Класс `Fast` будет классом быстрых животных. Переопределим у него только метод `move`, который кроме обычного перемещения ещё и будет делать это быстро.

```
class Fast(AbstractDecorator):
    def move(self):
        self.obj.move()
        print("Fast!")
```

Создадим базовое животное и выполним все его методы:

```

animal = Animal()
animal.feed()
animal.move()
animal.make_noise()

print()

```

```

I eat grass
I walk forward
W000!

```

Теперь сделаем из нашего животного водоплавающее. Для этого определим, что оно является водоплавающим и в качестве декорируемого объекта укажем наше животное.

```

swimming = Swimming(animal)
swimming.feed()
swimming.move()
swimming.make_noise()

```

```

I eat grass
I swim
...

```

Оно ест траву, плавает и не издаёт звуков. Сделаем из нашего животного хищника:

```

predator = Predator(animal)
predator.feed()
predator.move()
predator.make_noise()

```

```

I eat other animals
I swim
...

```

Теперь наше водоплавающее ест других животных и не издаёт звуков. Сделаем нашего водоплавающего хищника быстрым:

```

fast = Fast(animal)
fast.feed()
fast.move()
fast.make_noise()

```

```

I eat other animals
I swim
Fast!
...

```

Получили быстрого водоплавающего хищника. Рассмотрим такое свойство,

что можно применять несколько одинаковых декораторов к одному объекту. Мы можем сделать наше животное ещё быстрее:

```
faster = Fast(animal)
faster.feed()
faster.move()
faster.make_noise()
```

I eat other animals I swim Fast! Fast!

Таким образом, обложив наше животное декораторами, мы получили очень быстрого водоплавающего хищника.

Теперь поговорим о снятии декораторов. У каждого декоратора есть базовый объект, на котором он строится. Доступ к нему можно получить обращением к полю `base`. Например:

```
faster.base
```

```
<__main__.Fast at ...
```

```
faster.base.base
```

```
<__main__.Predator at ...
```

Для снятия эффекта "Хищник" с животного присвоим базовому(для быстрого) объекта животное, которое было раньше по иерархии декораторов.

```
faster.base.base = faster.base.base.base
faster.feed()
faster.move()
faster.make_noise()
```

I eat grass I swim Fast! Fast!

Видим, что животное так же плавает и быстро бежит, но теперь питается травой. [Исходный код decorator.ipynb](#)

3.3 Паттерн Adapter

3.3.1 Задача паттерна Adapter

Адаптер (adapter, "обёртка") — структурный паттерн, который используется для реализации взаимодействия несовместимых объектов.

Пусть есть объект и система, с которой этот объект должен взаимодействовать. При этом, его интерфейс не может быть напрямую встроен в систему. В таких случаях помогает адаптер. Он позволяет создать объект, который может обеспечить взаимодействие нашего исходного объекта с системой. Пример адаптера в реальной жизни: переходник для розеток разных типов.

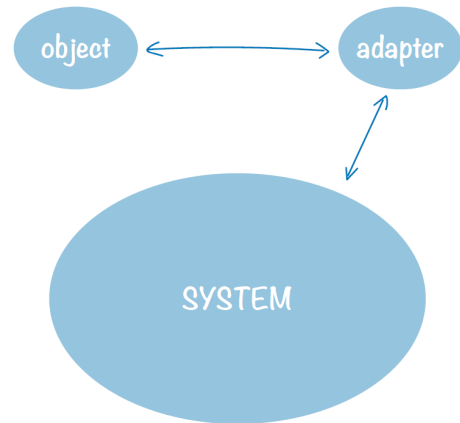


Рис. 3.5: Взаимодействия адаптера

В программировании же может быть, например, такая ситуация: имеется консольная утилита, которая читает данные из файла, обрабатывает их и сохраняет результат в другой файл. И мы пользуемся системой, которая может создавать данные для обработки, но обрабатывать их не умеет. В качестве обработчика ей нужен объект, принимающий данные в виде списка и возвращать результат обработки также в виде списка.

Как видно, интерфейсы консольной утилиты и системы совершенно несовместимы. В данном случае адаптер будет представлять из себя объект с интерфейсом ввода-вывода, который нужен системе, при этом внутри адаптера данные преобразуются в файл, который передаётся на обработку консольной утилите. Результат считывается из выходного файла и передаётся на вход системе. Проблема решена.

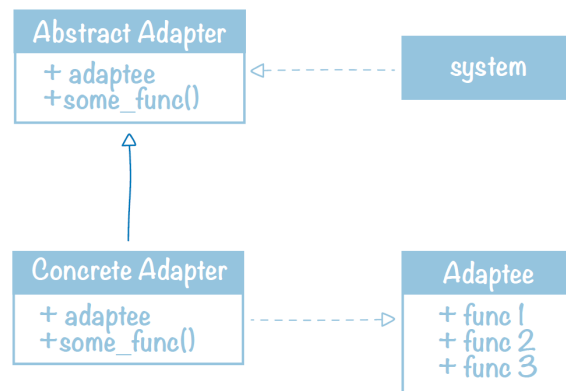


Рис. 3.6: UML-диаграмма адаптера

Подобным образом работают обёртки над некоторыми библиотеками, такими как `fastText` или `Vowpal Wabbit`. В машинном обучении же используется библиотека `Scikit-Learn`, в которой многие объекты по сути являются обёртками над классами из других библиотек. При этом объекты из этой библиотеки имеют понятные названия и стандартизованный интерфейс доступа, что позволяет гораздо удобнее работать с ними, а не с изначальными классами из библиотек.

3.3.2 Реализация адаптера класса

Пусть у нас есть некоторая система, которая берет текст, делает его предварительную обработку, а дальше хочет вывести слова в порядке убывания их частоты. Но собственного обработчика у системы нет. Она принимает в качестве обработчика некоторый объект `TextProcessor` с данным интерфейсом:

```
from System import *
import re
from abc import ABC, abstractmethod

class System:
    def __init__(self, text):
        tmp = re.sub(r'\W', ' ', text.lower())
        tmp = re.sub(r' +', ' ', tmp).strip()
        self.text = tmp

    def get_processed_text(self, processor):
        result = processor.process_text(self.text)
        print(*result, sep = '\n')

class TextProcessor:
    @abstractmethod
    def process_text(self, text):
        pass
```

В качестве обработчика есть некоторый счётчик слов, который по заданному тексту может посчитать в нём слова (`count_words()`), может сказать, сколько раз встретилось конкретное слово (`get_count()`) и может вывести частотный словарь всех встреченных слов (`get_all_words()`).

```
class WordCounter:
    def count_words(self, text):
        self.__words = dict()
        for word in text.split():
            self.__words[word] = self.__words.get(word, 0) + 1

    def get_count(self, word):
        return self.__words.get(word, 0)

    def get_all_words(self):
        return self.__words.copy()
```

Создадим объект системы и передадим в него некоторый текст.

```
system = System(text)
system.text
```

Design Patterns: Elements of Reusable Object-Oriented Software is a software engineering book describing software design patterns...

Это отрывок из статьи в Википедии, про книгу Elements of reusable object-oriented software, о которой было рассказано ранее. Создадим наш обработчик и передадим в него текст.

```
counter = WordCounter()
system.get_processed_text(counter)
```

AttributeError: 'WordCounter' object has no attribute 'process_text'

Эта ошибка появляется потому, что интерфейсы обработчика и системы совершенно несовместимы. Напишем адаптер, который позволит системе использовать наш обработчик. Объявим класс WordCounterAdapter, который будет наследоваться от нашего абстрактного обработчика TextProcessor. Объявим конструктор __init__, принимающий в качестве дополнительного аргумента объект, который мы хотим адаптировать, и сохраняющий этот объект в некоторую переменную класса. Метод process_text() будет принимать на вход текст и возвращать слова в порядке убывания частоты их вхождений. Чтобы вывести список слов используем метод get_all_words(), нашего счётчика. Для получения списка из словаря всех слов воспользуемся методом keys(). И вернём наш отсортированный массив слов с помощью return sorted() сортируем список слов по ключу. В данном случае ключ — это количество встреч конкретного слова в частотном словаре. Для этого воспользуемся лямбда-функцией, которая выдаёт количество встреч слова в нашем счётчике (используем метод get_count() от адаптируемого объекта). reverse=True для сортировки в порядке убывания.

```
class WordCounterAdapter(TextProcessor):
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def process_text(self, text):
        self.adaptee.count_words(text)
        words = self.adaptee.get_all_words().keys()
        return sorted(words, key=lambda
                        x: self.adaptee.get_count(x),
                        reverse=True)
```

Объявим адаптер, который будет экземпляром класса WordCounterAdapter и принимать в качестве адаптируемого объекта объявленный ранее счётчик. Обработаем текст с использованием адаптера и системы, а в качестве обработчика для get_processed_text передадим наш адаптер:

```
adapter = WordCounterAdapter(counter)
system.get_processed_text(adapter)
```

the
and
software
design
of ...

Видим список слов, причём их порядок примерно совпадает с частотой встречи в английском языке, так что результат похож на правду.

[Исходный код адаптера](#)

3.4 Паттерн Observer

3.4.1 Задача паттерна Observer

Наблюдатель (observer) часто применяется в самых разных задачах: от веб программирования и gamedevelopment-а до обработки сложных физических экспериментов. Это поведенческий шаблон, оповещающий об изменении некоторого объекта.

Пусть в некоторой системе есть наблюдаемый объект, который со временем изменяет своё состояние, и объект-наблюдатель, который отслеживает состояние наблюдаемого объекта и который хочет своевременно узнавать об изменениях в наблюдаемом объекте. Самый простой способ это сделать — напрямую спрашивать у наблюдаемого объекта, произошли ли с ним какие-то изменения. Но тогда с какой частотой наблюдатель должен запрашивать эти изменения? 1, 100, 1000 раз в секунду? И каждый раз при запросе наблюдаемый объект вместо того, чтобы продолжать делать свою задачу, должен отвечать наблюдателю на вопрос о своём состоянии. Такое можно использовать, если наблюдатель один. Но если за обновлениями следят тысячи объектов, нужен иной подход: научить наблюдаемый объект самостоятельно ставить в известность наблюдателей при возникновении изменений.

Именно для этого и применяется данный паттерн. Он позволяет от pull-системы отслеживания изменений перейти к push-системе. На примере социальной сети Вконтакте разберёмся в устройстве паттерна. В ней есть сообщества, в которых периодически публикуются новости, и подписчики, которые при появлении новых записей хотели бы выидеть их в разделе новостей. Таким образом, мы имеем отношение «один — много» (или single to multiple). Реализуем абстрактные интерфейсы группы и её подписчика.

Группа может рассылать подписчикам новости. У каждой группы для этого существуют методы: подписать пользователя на обновления (`subscribe()`), отписать его (`unsubscribe()`) и рассылка подписчикам сообщений об изменении состояния (`notify()`). У абстрактного интерфейса подписчика определяем метод `update()`, описывающий действия, которые будут выполнены при получении сообщений.

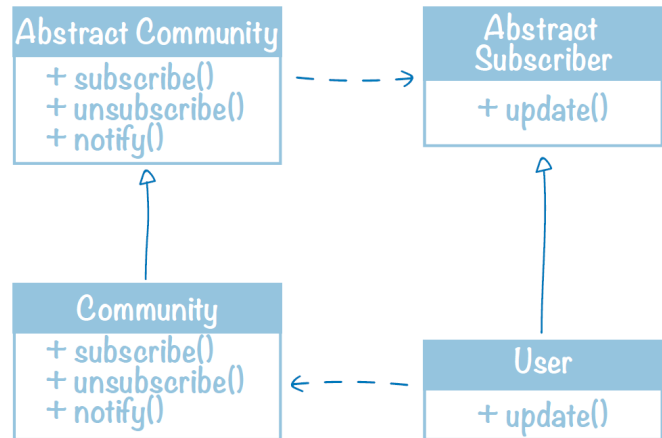


Рис. 3.7: UML-диаграмма наблюдателя

В конкретных реализациях подписчиков эти действия определяются по-разному. Метод «подписать пользователя» добавляет пользователя во множество возможных подписчиков, а метод «отписать» удаляет его из этого множества. Метод `Request()`, который оповещает пользователей об изменениях, вызывает у конкретных пользователей метод `update`.

Использование такой структуры позволяет минимизировать нагрузку на наблюдаемый объект за счёт того, что вместо постоянных запросов об изменениях от пользователей, он сам сообщает о том, что с ним происходит, когда изменения появляются.

Паттерн наблюдатель используется, когда существуют два объекта: наблюдающий и наблюдаемый. Он предназначен для организации системы оповещений наблюдающего объекта, об изменении состояния наблюдаемого объекта. Вместо pull-системы (когда наблюдатель самостоятельно запрашивает наблюдаемый объект об изменениях), используется push-система (наблюдаемый объект оповещает наблюдателя об изменениях). Оповещение происходит путём вызова метода `update()` у всех подписчиков.

3.4.2 Реализация паттерна Наблюдатель

Пусть у нас есть менеджер уведомлений, который может рассылать уведомления его подписчикам. И есть сами подписчики двух типов: `printer`, который печатает сообщение менеджера и `notifier`, который уведомляет о том, что сообщение пришло.

Сначала создадим класс `notifier manager`. В инициализаторе объявим пустой список подписчиков `__subscribers`. Для того, чтобы подписчик мог подписаться на уведомления менеджера, добавим методы `subscribe` (добавляет подписчика в список) и `unsubscribe` (удаляет подписчика из списка). Важной частью наблюдателя является отправка уведомлений, и для этого объявим у менеджера метод `notify()`, который отправит уведомление всем подписчикам (т.е. вызовет у них метод `update()` с сообщением в качестве параметра).


```

from abc import ABC, abstractmethod

class NotificationManager:
    def __init__(self):
        self.__subscribers = set()

    def subscribe(self, subscriber):
        self.__subscribers.add(subscriber)

    def unsubscribe(self, subscriber):
        self.__subscribers.remove(subscriber)

    def notify(self, message):
        for subscriber in self.__subscribers:
            subscriber.update(message)

```

Теперь объявим абстрактного наблюдателя AbstractObserver, который наследуется от абстрактного базового класса. Дадим ему имя в инициализаторе и объявим у него метод update (пока абстрактный и пустой)

```

class AbstractObserver(ABC):
    @abstractmethod
    def update(self, message):
        pass

```

Теперь объявим две конкретные реализации: notifier (просто печатает, что сообщение пришло) и printer (печатает и сам текст сообщения). В них нам нужно по-разному реализовать метод update.

```

class MessageNotifier(AbstractObserver):
    def __init__(self, name):
        self.__name = name

    def update(self, message):
        print(f'{self.__name} recieved!')

class MessagePrinter(AbstractObserver):
    def __init__(self, name):
        self.__name = name

    def update(self, message):
        print(f'{self.__name} recieved: {message}')

```

Создадим один notifier и два printer-а. Далее объявим менеджер уведомлений и подпишем на него наших наблюдателей. И отправим им уведомление методом notify().

```
notifier1 = MessageNotifier("Notifier1")
printer1 = MessagePrinter("Printer1")
printer2 = MessagePrinter("Printer2")

manager = NotificationManager()
manager.subscribe(notifier1)
manager.subscribe(printer1)
manager.subscribe(printer2)

manager.notify("Hi!")
```

```
Notifier1 recieved message!
Printer1 recieved message: Hi!
Printer2 recieved message: Hi!
```

Всё работает, как мы и хотели: notifier получил сообщение, а принтеры его ещё и распечатали.

[Исходный код наблюдателя](#)

Оглавление

4	Паттерны проектирования (часть 2)	2
4.1	Паттерн Chain of responsibility	2
4.1.1	Задача паттерна Chain of Responsibility	2
4.1.2	Краткая реализация паттерна Chain of Responsibility	5
4.1.3	Практическая реализация паттерна Chain of Responsibility	7
4.2	Паттерн Abstract Factory	11
4.2.1	Задача паттерна Abstract Factory	11
4.2.2	Краткая реализация паттерна Abstract Factory	12
4.2.3	Практическая реализация паттерна Abstract Factory	16
4.3	Конфигурирование через YAML	20
4.3.1	Язык YAML. Назначение и структура. PyYAML	20
4.3.2	Использование YAML для конфигурирования паттерна	23

Неделя 4

Паттерны проектирования (часть 2)

4.1 Паттерн Chain of responsibility

4.1.1 Задача паттерна Chain of Responsibility

Рассмотрим ещё один поведенческий шаблон — Chain of Responsibility (цепочка обязанностей). Представьте, что ваш автомобиль только что сломался и вам необходимо сдать его в ремонт (и можно починить не только саму поломку, но и поменять масло и так далее).

Цепочка со стороны пользователя будет выглядеть так:

- Случилась поломка автомобиля;
- Посещаете автосервис;
- Даёте задание:
 1. Починить двигатель;
 2. Поменять масло;
 3. Залить антифриз;
 4. Поменять резину;
 5. И другие задания по ситуации.
- Ждёте выполнения задания и звоните через день.

Примерно так выглядит спроектированная цепочка событий с точки зрения пользователя. А именно, некоторому классу задаётся список заданий, после чего производится запуск сразу всех действий.

С точки зрения сервиса это выглядит так: работник передаёт задание в первый отдел, и если её может решить первый отдел, то задача переходит к нему. Иначе ко второму отделу и так далее, пока не дойдёт до того отдела, который может выполнить задачу с этой машиной. Когда первая задача выполнена, работник запускает по этой же цепочке следующую задачу и так, пока не будет выполнено всё.

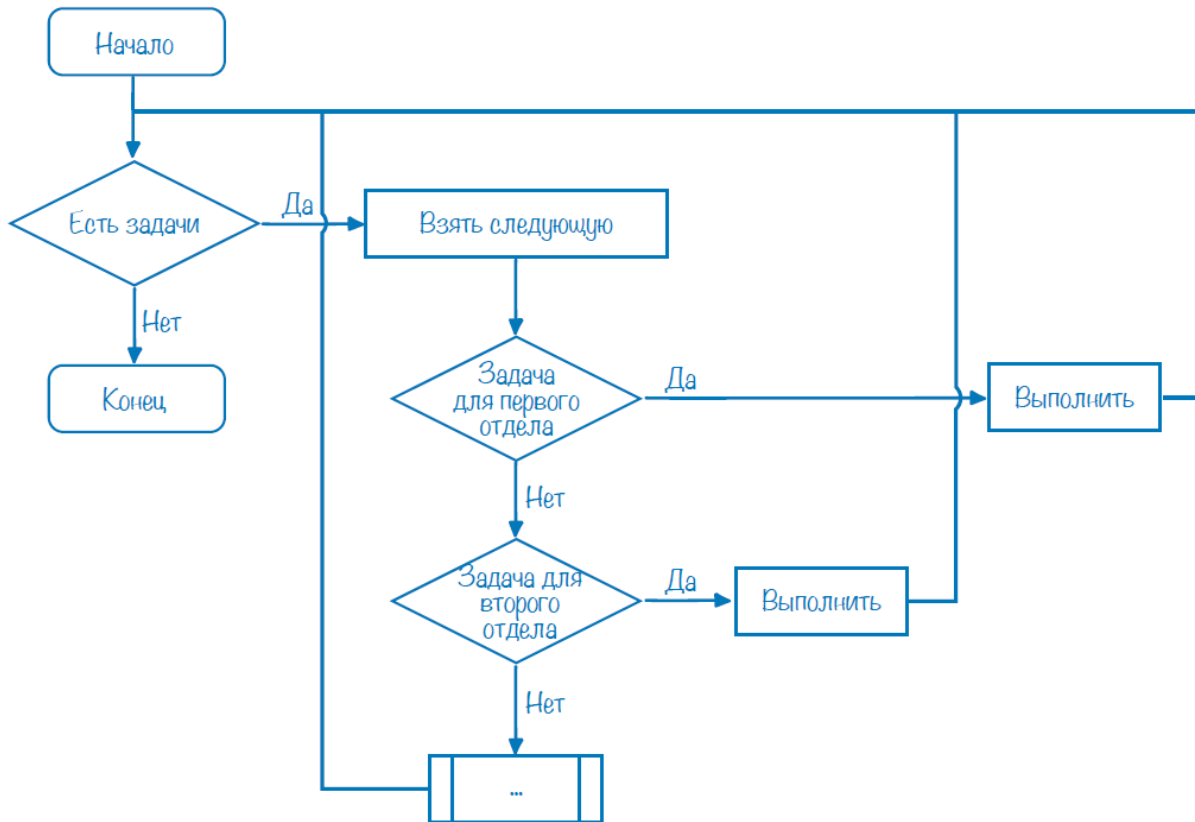


Рис. 4.1: Механизм поведенческого шаблона Chain of responsibility

Таков принцип поведенческого шаблона Chain of responsibility: объекту передаётся одно или же сразу несколько заданий, выполнение которых запускается по цепочке.

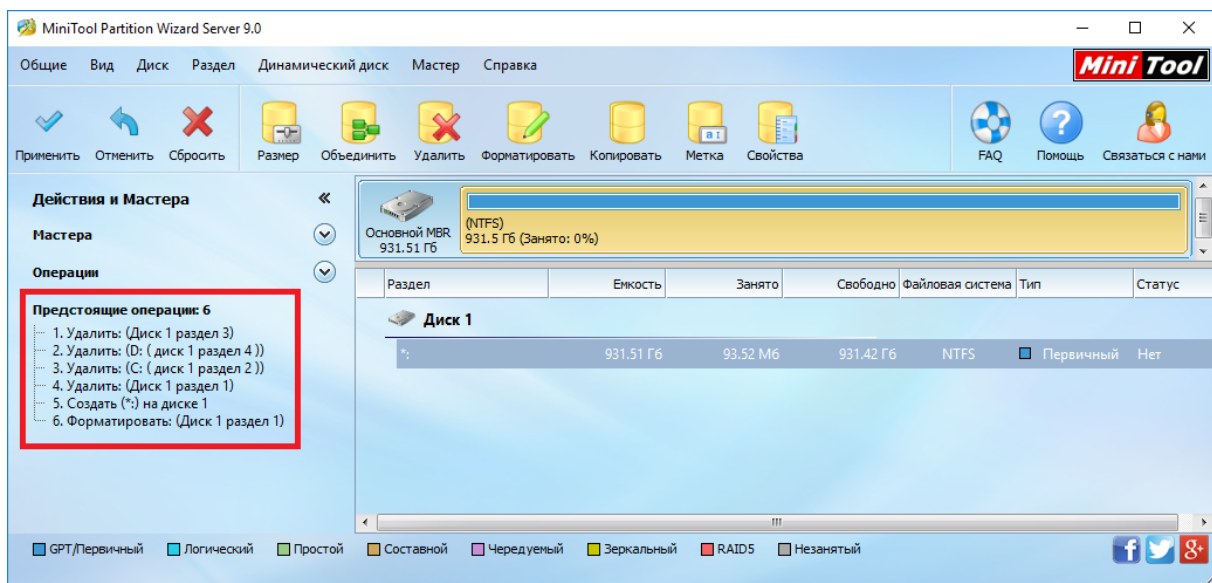
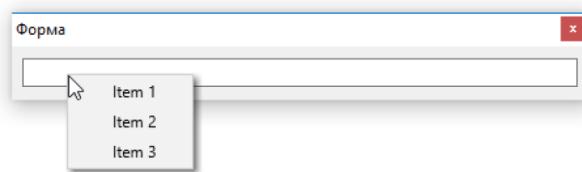


Рис. 4.2: Список операций с диском

Этот шаблон используется в самых разных случаях. Например, программы работы с жёсткими дисками, где вы задаёте, что вам сначала удалить одни разделы, потом создать другие, затем отформатировать, после чего все эти задания по очереди начинают выполняться.

Большинство графических программ применяют этот подход при обработке сообщений. Например, вы запрашиваете контекстное меню у кнопки. Она смотрит, есть ли у неё своё контекстное меню или нет, если есть — выдаёт, если нет — то передаёт сообщение о том, что необходимо показать контекстное меню, своему родителю, форме. Та в свою очередь тоже изучает и смотрит, может ли она показать. Если нет — то передаёт дальше. Может получиться, что запрос на показ контекстного меню может передаться непосредственно операционной системе, в результате чего вы получите стандартное контекстное меню.

Встроенное всплывающее меню



Системное всплывающее меню

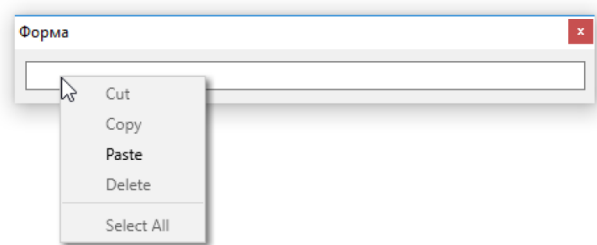


Рис. 4.3: Контекстное меню

Условия применения Chain of Responsibility:

- Присутствуют типизированные сообщения;
- Все сообщения должны быть обработаны хотя бы один раз;
- Работа с сообщением: делай сам или передай другому.

Таким способом можно реализовать, например, работу web-сервера, на который приходит огромное количество сообщений от пользователей, которые после передаются, в некоторую цепочку обработки событий. Одни исполнители записывают в базу данных, другие позволяют скачать файл, а третьи выдают html-страницу текста.

4.1.2 Краткая реализация паттерна Chain of Responsibility

Рассмотрим реализацию цепочки обязанностей на примере квестов в компьютерной игре. Опишем класс игрока, у которого есть имя, опыт и множества сданных и полученных квестов.

```
class Character:
    def __init__(self):
        self.name = "Nagibator"
        self.xp = 0
        self.passed_quests = set()
        self.taken_quests = set()
```

Объявим несколько квестов: поговорить с , поохотиться на крыс и принести доски из сарая. У каждого квеста будет название и опыт за его выполнение. При получении квеста мы проверяем, не был ли он нами получен или сдан. Если он был сдан, то взять его мы не можем. Если же он был получен, то квест можно сдать т.е. написать "квест сдан удаляем из списка полученных, добавляем в список выполненных и получаем опыт.

```
def add_quest_speak(char):
    quest_name = "Поговорить"
    xp = 100
    if quest_name not in (char.passed_quests | char.
        taken_quests):
        print(f"Квест получен: \"{quest_name}\"")
        char.taken_quests.add(quest_name)
    elif quest_name in char.taken_quests:
        print(f"Квест сдан: \"{quest_name}\"")
        char.passed_quests.add(quest_name)
        char.taken_quests.remove(quest_name)
        char.xp += xp

def add_quest_hunt(char):
    quest_name = "Охота "
    xp = 300
    if quest_name not in (char.passed_quests | char.
        taken_quests):
        print(f"Квест получен: \"{quest_name}\"")
        char.taken_quests.add(quest_name)
    elif quest_name in char.taken_quests:
        print(f"Квест сдан: \"{quest_name}\"")
        char.passed_quests.add(quest_name)
        char.taken_quests.remove(quest_name)
        char.xp += xp
```

```
def add_quest_carry(char):
    quest_name = "Принести доски"
    xp = 200
    if quest_name not in (char.passed_quests | char.
        taken_quests):
        print(f"Квест получен: \"{quest_name}\"")
        char.taken_quests.add(quest_name)
    elif quest_name in char.taken_quests:
        print(f"Квест сдан: \"{quest_name}\"")
        char.passed_quests.add(quest_name)
        char.taken_quests.remove(quest_name)
        char.xp += xp
```

Напишем QuestGiver, который будет давать квесты из списка. В этот список можно добавлять квесты (app_quests()). Так же можно дать персонажу квесты (handle_quests()), а для этого мы проходимся по списку квестов и выполняем их персонажем.

```
class QuestGiver:
    def __init__(self):
        self.quests = []

    def add_quest(self, quest):
        self.quests.append(quest)

    def handle_quests(self, character):
        for quest in self.quests:
            quest(character)
```

Для проверки создадим список квестов, QuestGiver (в который добавим квесты из списка) и персонажа, которому передадим все квесты.

```
all_quests = [add_quest_speak, add_quest_hunt,
    add_quest_carry]
quest_giver = QuestGiver()

for quest in all_quests:
    quest_giver.add_quest(quest)

player = Character()
quest_giver.handle_quests(player)
```

```
Квест получен: "Поговорить"
Квест получен: "Охота"
Квест получен: "Принести доски"
```



```
print("Получено: ", player.taken_quests)
print("Сдано: ", player.passed_quests)
```

```
Получено: 'Поговорить' , 'Принести доски' , 'Охота'
Сдано: set()
```

Персонаж получил все квесты, но ни одного не сдал. Изменим список полученных квестов. Пусть на данный момент будут получены квесты "принести доски" и "поговорить".

```
player.taken_quests = {'Принести доски' , 'Поговорить'}
quest_giver.handle_quests(player)
```

```
Квест сдан: "Принести доски"
Квест получен: "Охота"
Квест сдан: "Поговорить"
```

Уже полученные квесты оказались сданы, и мы получили квест, которого раньше не было.

```
quest_giver.handle_quests(player)
```

```
Квест сдан: "Охота"
Полученный на прошлой итерации квест оказался сдан, а остальные два уже
были получены и сданы:
```

```
print("Получено: ", player.taken_quests)
print("Сдано: ", player.passed_quests)
```

```
Получено: set()
Сдано: 'Поговорить', 'Принести доски', 'Охота'
```

Мы реализовали самую простую цепочку выполнения задач. Это пока не цепочка обязанностей, потому что не хватает ключевой особенности: передачи задачи следующему обработчику, если текущий не может её выполнить.

4.1.3 Практическая реализация паттерна Chain of Responsibility

Персонажа оставим без изменений. И создадим список константных названий квестов:

```
QUEST_SPEAK, QUEST_HUNT, QUEST_CARRY = "QSPEAK", "QHUNT", "
    QCARRY"

class Character:
    def __init__(self):
        self.name = "Nagibator"
```

```

self.xp = 0
self.passed_quests = set()
self.taken_quests = set()

```

Чтобы цепочка обязанностей работала корректно, опишем класс события, которое будет происходить:

```

class Event:
    def __init__(self, kind):
        self.kind = kind

```

Опишем нулевой обработчик (нулевое звено цепочки), которое будет передавать событие на обработку следующему обработчику, если таковой имеется.

```

class NullHandler:
    def __init__(self, successor=None):
        # передаём следующее звено
        self.__successor = successor
    def handle(self, char, event): # обработчик
        if self.__successor is not None: #даём следующему
            self.__successor.handle(char, event)

```

Изменим код квестов из предыдущего параграфа. Теперь все квесты это классы, унаследованные от NullHandler. Переопределим метод handle: если происходит событие, означающее квест поговорить, то мы выполняем. Иначе передаём на обработку следующему звену цепочки.

```

class HandleQSpeak(NullHandler):

    def handle(self, char, event):
        if event.kind == QUEST_SPEAK:
            xp = 100
            quest_name = "Поговорить сфермером "
            if event.kind not in
                (char.passed_quests | char.taken_quests):
                print(f"Квест получен: \"{quest_name}\"")
                char.taken_quests.add(event.kind)
            elif event.kind in char.taken_quests:
                print(f"Квест сдан: \"{quest_name}\"")
                char.passed_quests.add(event.kind)
                char.taken_quests.remove(event.kind)
                char.xp += xp
        else:
            print("Передаю обработкудальше ")
            super().handle(char, event)

```

```

class HandleQHunt(NullHandler):
    def handle(self, char, event):
        if event.kind == QUEST_HUNT:
            xp = 300
            quest_name = "Охота накрыс "
            if event.kind not in
                (char.passed_requests | char.taken_requests):
                print(f"Квест получен: \"{quest_name}\"")
                char.taken_requests.add(event.kind)
            elif event.kind in char.taken_requests:
                print(f"Квест сдан: \"{quest_name}\"")
                char.passed_requests.add(event.kind)
                char.taken_requests.remove(event.kind)
                char.xp += xp
        else:
            print("Передаю обработку дальше ")
            super().handle(char, event)

```

```

class HandleQCarry(NullHandler):

    def handle(self, char, event):
        if event.kind == QUEST_CARRY:
            xp = 200
            quest_name = "Принести дроваизсарая "
            if event.kind not in
                (char.passed_requests | char.taken_requests):
                print(f"Квест получен: \"{quest_name}\"")
                char.taken_requests.add(event.kind)
            elif event.kind in char.taken_requests:
                print(f"Квест сдан: \"{quest_name}\"")
                char.passed_requests.add(event.kind)
                char.taken_requests.remove(event.kind)
                char.xp += xp
        else:
            print("Передаю обработку дальше ")
            super().handle(char, event)

```

Изменим QuestGiver, чтобы тот мог работать с цепочкой обязанностей. Объявим ему цепочку, с которой он будет работать.

```

class QuestGiver:

    def __init__(self):

```

```

        self.handlers = HandleQSpeak(HandleQHunt(
            HandleQCarry(NullHandler())))
        self.events = [] # изначально пустой список событий

    def add_event(self, event): # добавляем события
        self.events.append(event)

    # передаём событие цепочке обязанностей
    def handle_requests(self, char):
        for event in self.events:
            self.handlers.handle(char, event)

```

Объявим список всех возможных событий events и наш questgiver, который может реагировать на все события из списка.

```

events = [Event(QUEST_CARRY), Event(QUEST_HUNT), Event(
    QUEST_SPEAK)]

quest_giver = QuestGiver()

for event in events:
    quest_giver.add_event(event)

```

Проверим работу цепочки обязанностей на примере, аналогичном предыдущему:

```

player = Character()

quest_giver.handle_requests(player)
print()
player.taken_requests = {QUEST_CARRY, QUEST_SPEAK}
quest_giver.handle_requests(player)
print()
quest_giver.handle_requests(player)

```

```

Передаю обработку дальше
Передаю обработку дальше
Квест получен: "Принести дрова из сарая"
Передаю обработку дальше
Квест получен: "Охота на крыс"
Квест получен: "Поговорить с фермером"

```

```

Передаю обработку дальше
Передаю обработку дальше
Квест сдан: "Принести дрова из сарая"
Передаю обработку дальше

```

Квест получен: "Охота на крыс"

Квест сдан: "Поговорить с фермером"

Передаю обработку дальше

Передаю обработку дальше

Передаю обработку дальше

Квест сдан: "Охота на крыс"

Видно, что цепочка обязанностей работает, и квесты, обработка которых невозможна на данном этапе, передаются по ней дальше.

4.2 Паттерн Abstract Factory

4.2.1 Задача паттерна Abstract Factory

Для понимания паттерна абстрактной фабрики сначала рассмотрим пример из жизни. Представьте, что вы хотите построить дом. Самое главное это постройка каркаса, где будут располагаться двери и окна, высота потолка и другие важные вопросы. А с дизайном и материалами определитесь уже в процессе строительства.

Суть абстрактной фабрики: для программы не имеет значение, как создаются компоненты. Необходима лишь «фабрика», производящая компоненты, умеющие взаимодействовать друг с другом.

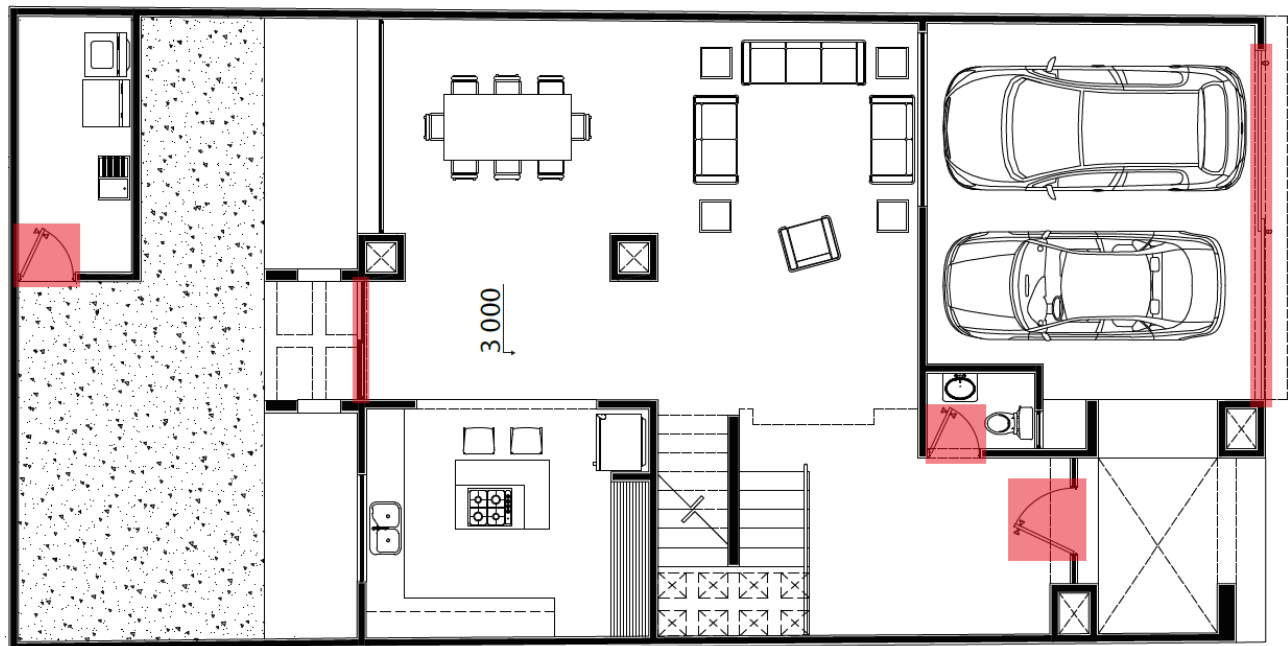


Рис. 4.4

Теперь разберём, как это выглядит в программировании. Предположим, что нам нужна функция создания диалогового окна.

Для этого мы должны уметь:

1. Создавать окно;
2. Создавать кнопки ("OK", "Cancel" и другие);
3. Выводить текстовую информацию ("Are you sure?");
4. (дополнительно) создавать чекбоксы и прочие визуальные эффекты.

Так вот для функции создания диалогового окна нет никакой необходимости знать, как создаются визуальные компоненты внутри него. Функции необходим лишь класс, который умеет как фабрика производить кнопки, текстовые сообщения, чекбоксы и прочее. Сами элементы уже можно производить различным образом и в разных стилях (например, Windows 10, OS X, Linux). Естественно, одна фабрика будет производить элементы только одного типа, а уже другая фабрика какого-то определённого другого типа.

Поэтому фабрика, с которой мы работаем во время написания функции (на этапе проектирования приложения), это некая абстрактная фабрика — класс с описанием всех создаваемых объектов без реализации. Но когда из приложения будет запускаться функция создания диалогового окна (этап выполнения), ей будет передаваться уже реальная фабрика — класс с реализацией создания всех компонент.

Далее мы разберём реализацию шаблонов абстрактной фабрики для формирования текстового отчёта в форматах html и markdown. Причём сначала реализуем её классическим способом, а затем произведём модификацию в духе Python.

4.2.2 Краткая реализация паттерна Abstract Factory

Шаблон абстрактная фабрика позволяет создавать сложные системы взаимодействующих классов, и при помощи конкретных фабрик, специализирует создание определённой структуры для конкретных объектов. На примере реализации классов персонажа в игре рассмотрим реализацию данного шаблона. Создадим абстрактную фабрику с абстрактными методами:

```
from abc import ABC, abstractmethod

class HeroFactory(ABC):
    @abstractmethod # создаёт героя с заданным именем
    def create_hero(self, name):
        pass

    @abstractmethod # создаёт оружие
    def create_weapon(self):
        pass
```

```

@abstractmethod # создаёт заклинание
def create_spell(self):
    pass

```

Герой будет одного из нескольких классов: воин, маг или убийца. Причём каждому классу даётся свой конкретный предмет экипировки и своё заклинание. Опишем фабрику воинов:

```

class WarriorFactory(HeroFactory):
    def create_hero(self, name):
        return Warrior(name) # создаём война с заданным именем

    def create_weapon(self):
        return Claymore() # оружие война - клеймор

    def create_spell(self):
        return Power() # заклинание война - сила

class Warrior: # класс воинов
    def __init__(self, name):
        self.name = name
        self.weapon = None
        self.armor = None
        self.spell = None

    def add_weapon(self, weapon):
        self.weapon = weapon

    def add_spell(self, spell):
        self.spell = spell

    def hit(self): # удар оружием
        print(f"W. {self.name} uses {self.weapon.hit()}")
        self.weapon.hit()

    def cast(self): # использование заклинания
        print(f"W. {self.name} casts {self.spell.cast()}")
        self.spell.cast()

class Claymore:
    def hit(self):
        return "Claymore"

```

```
class Power:
    def cast(self):
        return "Power"
```

Аналогично описываются фабрики Мага и Убийцы:

```
class MageFactory(HeroFactory):
    def create_hero(self, name):
        return Mage(name)

    def create_weapon(self):
        return Staff()

    def create_spell(self):
        return Fireball()

class Mage:
    def __init__(self, name):
        self.name = name
        self.weapon = None
        self.armor = None
        self.spell = None

    def add_weapon(self, weapon):
        self.weapon = weapon

    def add_spell(self, spell):
        self.spell = spell

    def hit(self):
        print(f"M. {self.name} uses {self.weapon.hit()}")
        self.weapon.hit()

    def cast(self):
        print(f"M. {self.name} casts {self.spell.cast()}")
        self.spell.cast()

class Staff:
    def hit(self):
        return "Staff"

class Fireball:
    def cast(self):
        return "Fireball"
```



```

class AssassinFactory(HeroFactory):
    def create_hero(self, name):
        return Assassin(name)

    def create_weapon(self):
        return Dagger()

    def create_spell(self):
        return Invisibility()

class Assassin:
    def __init__(self, name):
        self.name = name
        self.weapon = None
        self.armor = None
        self.spell = None

    def add_weapon(self, weapon):
        self.weapon = weapon

    def add_spell(self, spell):
        self.spell = spell

    def hit(self):
        print(f"A. {self.name} uses {self.weapon.hit()}")
        self.weapon.hit()

    def cast(self):
        print(f"A. {self.name} casts {self.spell.cast()}")

class Dagger:
    def hit(self):
        return "Dagger"

class Invisibility:
    def cast(self):
        return "Invisibility"

```

Абстрактные фабрики описаны. Теперь с их помощью определим функцию, создающую персонажа. На вход она будет принимать конкретную фабрику, которая создаёт персонажа нужного класса с соответствующей экипировкой:

```
def create_hero(factory):
    hero = factory.create_hero("Nagibator")
    weapon = factory.create_weapon()
    ability = factory.create_spell()

    hero.add_weapon(weapon)
    hero.add_spell(ability)

    return hero
```

Создадим убийцу :

```
factory = AssassinFactory()
player = create_hero(factory)
player.cast()
player.hit()
```

A. Nagibator casts Invisibility

A. Nagibator uses Dagger

Убийца успешно создан, и может бить оружием и кастовать. Создадим мага:

```
factory = MageFactory()
player = create_hero(factory)
player.cast()
player.hit()
```

M. Nagibator casts Fireball

M. Nagibator uses Staff

Таким образом мы написали простейшую реализацию абстрактной фабрики, создающую систему взаимосвязанных классов из персонажа, оружия и заклинания. Но этот код можно сделать короче, используя конструкции класс-метод, в Python.

[Полная реализация Абстрактной фабрики](#)

4.2.3 Практическая реализация паттерна Abstract Factory

Предыдущий код получился очень громоздким и с повторениями. Изменим его с помощью питоновского механизма "класс-метод". Избавимся от абстрактных классов. Вместо абстрактных методов будем использовать класс-методы. Они позволяют пользоваться некоторыми функциями, специфичными для некоторого класса. Вместо self будет передаваться класс, методы которого мы будем использовать.

```
class HeroFactory:
    @classmethod
    def create_hero(Class, name):
```

```

        return Class.Hero(name)

    @classmethod
    def create_weapon(Class):
        return Class.Weapon()

    @classmethod
    def create_spell(Class):
        return Class.Spell()

```

Создадим фабрику война с использованием механизма класс-метод:

```

class WarriorFactory(HeroFactory):
    class Hero: # подкласс героя для данного класса
        def __init__(self, name):
            self.name = name
            self.weapon = None
            self.armor = None
            self.spell = None

        def add_weapon(self, weapon):
            self.weapon = weapon

        def add_spell(self, spell):
            self.spell = spell

        def hit(self):
            print(f"Warrior hits with {self.weapon.hit()}")
            self.weapon.hit()

        def cast(self):
            print(f"Warrior casts {self.spell.cast()}")
            self.spell.cast()

    class Weapon: # подкласс оружия
        def hit(self):
            return "Claymore"

    class Spell: # подкласс заклинания
        def cast(self):
            return "Power"

```

Аналогично с двумя другими фабриками:

```

class MageFactory(HeroFactory):
    class Hero:
        def __init__(self, name):
            self.name = name
            self.weapon = None
            self.armor = None
            self.spell = None

        def add_weapon(self, weapon):
            self.weapon = weapon

        def add_spell(self, spell):
            self.spell = spell

        def hit(self):
            print(f"Mage hits with {self.weapon.hit()}")
            self.weapon.hit()

        def cast(self):
            print(f"Mage casts {self.spell.cast()}")
            self.spell.cast()

    class Weapon:
        def hit(self):
            return "Staff"

    class Spell:
        def cast(self):
            return "Fireball"

class AssassinFactory(HeroFactory):
    class Hero:
        def __init__(self, name):
            self.name = name
            self.weapon = None
            self.armor = None
            self.spell = None

        def add_weapon(self, weapon):
            self.weapon = weapon

```

```

def add_spell(self, spell):
    self.spell = spell

def hit(self):
    print(f"Assassin hits with {self.weapon.hit()}")
    self.weapon.hit()

def cast(self):
    print(f"Assassin casts {self.spell.cast()}")

class Weapon:
    def hit(self):
        return "Dagger"

class Spell:
    def cast(self):
        return "Invisibility"

```

Создание героя методом `create_hero()` остаётся таким же:

```

def create_hero(factory):
    hero = factory.create_hero("Nagibator")

    weapon = factory.create_weapon()
    spell = factory.create_spell()

    hero.add_weapon(weapon)
    hero.add_spell(spell)

    return hero

```

Попробуем создать персонажей различных классов, передавая различные фабрики:

```

player = create_hero(AssassinFactory)
player.cast()
player.hit()

```

Assassin casts Invisibility

Assassin hits with Dagger

Создали убийцу, который становится невидимым и бьёт кинжалом.

```

player = create_hero(MageFactory)
player.cast()
player.hit()

```

```
Mage casts Fireball
Mage hits with Staff
```

Таким образом мы сделали громоздкий код простым и читаемым, избавившись от повторений с помощью механизма "класс-метод".

[Исходный код сокращённой абстрактной фабрики](#)

4.3 Конфигурирование через YAML

4.3.1 Язык YAML. Назначение и структура. PyYAML

В какой-то момент программисты сталкиваются с тем, что может понадобиться написать конфигуратор к программе. Например, создать текстовый файл с основными параметрами программы.

Основные способы конфигурирования программ:

- **xml-файл** — файл разметки тэгами;

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <anc>
    <data>10</data>
    <name>none</name>
  </anc>
  <option1>sample
text
</option1>
  <option2>smample text
</option2>
  <option3>
    <element>
      <el1>
        <name>data</name>
        <obj><data>20.0</data></obj>
      </el1>
    </element>
    <element>
      <el2>
        <name>data</name>
        <obj><data>10</data>
          <name>none</name></obj>
      </el2>
    </element>
  </option3>
</root>
```

- **ini-файл**, где всё делится на секции ключ — значение;

```
[category1]
option1=value1
option2=value2
[category2]
option1=value1
option2=value2
```

- **json-файл** — код на языке javascript;

```
{
  "option1": "sample\ntext\n",
  "option2": "smaple text\n",
  "option3": [
    {
      "el1": {
        "obj": {
          "data": 20.0
        },
        "name": "data"
      }
    },
    {
      "el2": {
        "obj": {
          "data": 10,
          "name": "none"
        },
        "name": "data"
      }
    }
  ],
  "anc": {
    "data": 10,
    "name": "none"
  }
}
```

- **YAML-файл.**

```
# блочные литералы
option1: | # далее следует текст с учётом переносов
  sample
  text
option2: > # в тексте далее переносы учитываться не будут
  smaple
  text

anc: &anc # определяем якорь
  data: 10 # сопоставление имени и значения
  name: none

option3: # далее идёт список элементов
- el1:
  name: data
  obj: {data: !!float 20} # явное указание типа
- el2: {name: data, obj: *anc} # используем якорь
```

Изначально YAML разрабатывался как замена xml и его расшифровка была **Yet Another Markup Language**. Цели создания YAML:

- быть легко понятным человеку;
- поддерживать структуры данных, родные для разных языков;
- быть переносимым между языками программирования;
- использовать цельную модель данных для поддержки обычного инструментария;
- поддерживать потоковую обработку;
- быть выразительным и расширяемым;
- быть лёгким в реализации и использовании.

Со временем он развивался и стал расшифровываться как **YAML Ain't Markup Language** поскольку он перестал быть просто языком разметки. И теперь стандарт YAML покрывает JSON, то есть фактически любой файл формата JSON является файлом формата YAML.

В YAML можно перечислять последовательности, делать сопоставление имени и значения (записывать словарь), использовать блочные литералы, занимающие больше одной строки и использовать подстановки: в одном месте файла пометить якорь, а в другом используем на него ссылку (это фактически использование переменных). Файл YAML имеет древовидную структуру данных и форматируется он таким же образом, как и в Python (при помощи пробелов). В YAML позволяет явным образом указывать тип хранимой информации. Например, в каком-то месте написано "20". И вы хотите подчеркнуть, что это не число 20, а текст "20". И этот формат позволяет вам так сделать. Поддержка YAML есть в большинстве современных языков программирования. В Python для этого используется модуль PyYAML.

4.3.2 Использование YAML для конфигурирования паттерна

Сконфигурируем абстрактную фабрику, которую писали ранее. Импортируем YAML и опишем конфигурацию создания героя. Factory покажет, каким классом будет персонаж с именем name.

```
import yaml
hero_yaml = '''
--- !Character
factory:
  !factory assassin
name:
  7NaGiBaToR7
'''
```

Теперь адаптируем код под работу с yaml-файлами. После [кода написанного ранее](#), создадим конструктор фабрик, который по текстовому описанию (т.е. по загрузчику loader и node yaml-файла) делает фабрику. Загрузчик позволяет выгрузить из yaml-файла какие-то данные, а node - ячейка, которую будем обрабатывать.

```
def factory_constructor(loader, node):
    data = loader.construct_scalar(node)
    if data == "mage":
        return MageFactory
    elif data == "warrior":
        return WarriorFactory
    else:
        return AssassinFactory
```

Методом загрузчика construct_scalar(node) мы выгрузим данные. Scalar потому что в node хранится единственное строковое значение, а не несколько разных значений. И в зависимости от содержания переменной data вернём нужный объект. В данном случае мы считаем, что yaml записан корректно.

Теперь поменяем создание персонажа для работы с yaml-файлами. Для этого создадим класс Character, который наследуется от yaml.YAMLObject и будет обрабатывать файл.

```
class Character(yaml.YAMLObject):
    yaml_tag = "!Character" # тэг, на который надо смотреть

    # теперь стала методом класса.
    # а factory стал атрибутом класса
    def create_hero(self):
        hero = self.factory.create_hero(self.name)
```

```

    weapon = self.factory.create_weapon()
    spell = self.factory.create_spell()

    hero.add_weapon(weapon)
    hero.add_spell(spell)

    return hero

```

Попробуем загрузить yaml-файл и создать героя при помощи конфигурации. Нашему загрузчику loader добавим новый конструктор, создающий фабрики из данных с тэгом factory. Затем создадим героя hero, по данным загруженным из yaml-файла с помощью метода create_hero() у yaml.load(hero_yaml).

```

loader = yaml.Loader
loader.add_constructor("!factory", factory_constructor)
hero = yaml.load(hero_yaml).create_hero()
hero.hit()      # пробуем ударить
hero.cast()     # пробуем ударить

```

Assassin 7NaGiBaToR7 hits with Dagger

Assassin 7NaGiBaToR7 casts Invisibility

В итоге мы получили убийцу из yaml-файла. Таким образом мы можем конфигурировать абстрактные фабрики с помощью yaml-файлов.

[Продвинутый пример на использование YAML](#)