Полиморфизм

Сам по себе термин “полиморфизм” не из контекста программирования означает “изменчивость”. В программировании существуют несколько типов полиморфизма. Одни предполагают, что один и тот же код обрабатывает данные разных типов, другие в свою очередь, то, что данные разных типов обрабатывает потенциально разный код, но имеет он один и тот же интерфейс. Таким образом термин “полиморфизм” может быть применен к различных аспектам языка программирования. В контексте объектно-ориентированного программирования в современных языках используется сочетание разных типов полиморфизма, чаще всего это полиморфизм подтипов и ad-hoc полиморфизм.

Для понимания механизма полиморфизма совершенно противопоказано изучать его по строгим определениям. Разберем механизм работы полиморфизма и его практическое применение.

В разделе наследования мы говорили, что классы потомка и предка связаны отношением “is-a” (“является”). То есть экземпляр класса потомка можно использовать там, где ожидается экземпляр класса предка. Сделать это можно по той причине, что *интерфейс* (то есть набор методов, которые может вызвать пользователь нашего экземпляра класса) предка унаследовал потомок, а значит он тоже обладает теми же методами, что и предок. А это в свою очередь значит, что если пользователь нашего класса вызовет метод предка, указав его имя и фактические параметры, будучи в полной уверенности, что он вызывает именно метод предка, наш потомок спокойно сможет этот метод исполнить.

```
class Typewriter: # печатная машинка, предок
    def type(self, text):
        # реализация
    # остальной интерфейс

class Computer(Typewriter): # компьютер, потомок
    # дополнительные методы

def fun(device):
    device.type(“Hello, World!”)

c = Computer()
fun(c)
```

В данном примере функции fun() мы подсунули вместо ожидаемого ей экземпляра предка экземпляр потомка и благодаря отношению “is-a” между предком и потомком этот код сработает и функция fun() ничего не заподозрит. Однако в отличие от языков

со статической типизацией, тот факт, что мы ожидаем экземпляр именно Typewriter, находится только в голове программиста и никак не отражен в коде. Таким образом, нам может подойти любой объект, обладающий методом type(), который принимает один параметр.

Утиная типизация

Если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка. Таким образом, нам не обязательно помечать в коде факт того, что мы ожидаем увидеть утку, а передающий не обязан нам передавать действительно утку. Это должно быть что-то, что “выглядит как утка, плавает как утка и крякает как утка”, т.е. реализует соответствующие методы с соответствующими формальными параметрами.

В языках со статической типизацией важно заранее знать, есть ли такой метод у имеющегося в наличии объекта, поэтому даже если мы не хотим знать конкретный тип этого объекта, он должен реализовывать какой-то нужный нам интерфейс (или наследовать от общего класса-предка). Языки с утиной типизацией (такие как JS) позволяют заранее не определять формальный интерфейс подходящих объектов, а просто вызывать нужные методы объекта.

```
class Typewriter: # печатная машинка
    def type(self, text):
        # реализация
    # остальной интерфейс

class ToyKeyboard: # игрушечная клавиатура
    def type(self, text):
        # реализация
    # остальной интерфейс

def fun(device):
    device.type(“Hello, World!”)

fun(Typewriter())
fun(ToyKeyboard())
```

Несмотря на то, что классы Typewriter и ToyKeyboard в коде никак не связаны, оба могут быть переданы как параметр в функцию fun(), т.к. реализуют метод type() с одним параметром.

Переопределение методов
Утиная типизация это одна форма полиморфизма (параметрический полиморфизм). Другой формой является переопределение методов предка потомками (и способность из методов предка вызывать эти переопределенные методы).

В Python можно переопределить любой метод предка и для этого не требуется специальных синтаксических обозначений. Достаточно в классе-потомке указать метод с тем же именем и списком параметров. Так, можно сказать, что все методы в Python являются виртуальными (т.е. доступными к переопределению).

```
class Typewriter:
    def type(self, text):
        print(f”Typed on a paper: {text}”)
    # остальной интерфейс

class Computer(Typewriter):
    def type(self, text): # переопределение метода
        print(f”Typed on a screen: {text}”)
    # остальной интерфейс

def fun(device):
    device.type(“Hello, World!”)

// напечатает:
// Typed on a paper: Hello, World!
fun(Typewriter())

// напечатает:
// Typed on a screen: Hello, World!
fun(Computer())
```

Таким образом, в зависимости от типа переданного экземпляра менялось поведение функции fun() без изменения ее кода. Строго говоря, здесь срабатывает механизм не виртуальных методов, а все той же утиной типизации, но он позволяет достичь того же эффекта, что и в языках со статической типизацией достигают виртуальные методы.

Зачем это может быть нужно? Очень часто нужно реализовать алгоритм для разных типов, который в общих чертах один и тот же, но реализация некоторых технических деталей различается. Можно, конечно, жестко закодировать все варианты деталей в коде алгоритма при помощи операторов ветвления, но с одной стороны это “размывает” код, отвлекая программиста от реализации непосредственно алгоритма, а с другой – при необходимости добавить новый вариант деталей, придется дописывать еще одну

ветку в if (или switch) по всему алгоритму. И совсем невозможным становится вариант, когда деталей реализации может быть бесконечное количество и пользователи нашего алгоритма должны сами его добавлять. Здесь на помощь приходит полиморфизм.

```
class Tea: # класс чая
    def brew(self): # заварить чай
        pass

class PacketTea(Tea):
    def brew(self):
        # реализация

class HerbalTea(Tea):
    def brew(self):
        # реализация

def make_tea(tea):
    take_cup()     # взять чашку
    pour_water()   # залить водой
    tea.brew()     # заварить чай (с особенностью чая)
    serve_tea()    # подать чай
```

Мы можем определять свои классы чая и определять метод brew(). Таким образом, функции make_tea становится неважно как реализован этот метод, главное, что он выполняет нужную часть алгоритма.

Такие методы можно рассматривать как *черный ящик*. Взяв один черный ящик и подключив к нему проводки в нужные разъемы, мы на выходе получим один результат. Если мы возьмем другой черный ящик с такими же разъемами и подключим к нему те же проводки, можем получить другой результат. Но так как входы и выходы совпадают, черные ящики взаимозаменяемы. Мы не знаем как они устроены, но можем менять поведение всей системы простой заменой черного ящика.

Опять же, строго говоря, благодаря утиной типизации, определение класса Tea можно опустить, однако подробнее мы рассмотрим это в разделе абстрактных методов и классов.

Методы-”зацепки” (hooks)

В предыдущих примерах пользователь виртуальных методов был внешний. Но ничто не запрещает пользоваться виртуальным методом самому предку.

```
class TextEditor:
    def save(self):
        # вызвать диалог для выбора места сохранения файла
        s = get_save_filename()
        self._save_document(s)

    def _save_document(self, path):
        pass # мы сами не умеем сохранять

    # остальной интерфейс

class LocalEditor(TextEditor):
    def _save_document(self, path):
        # как-то сохраняем в локальной файловой системе
        with local_file(path) as f:
            f.write(self.get_contents())

    # остальной интерфейс

class RemoteEditor(TextEditor):
    def _save_document(self, path):
        # как-то сохраняем файл на удаленном компьютере
        with remote_file(path) as f:
            f.write(self.get_contents())

    # остальной интерфейс
```

Здесь алгоритм сохранения документа не меняется. Поэтому не меняется и содержимое метода save(). Меняется только деталь реализации сохранения документа. Обратите внимание, что класс TextEditor сам вообще не умеет сохранять документы. Поэтому для сохранения нам нужно использовать какого-то из его конкретных наследников.

```
local_editor = LocalEditor()
# метод предка вызывает полиморфный _save_document
local_editor.save()

remote_editor = RemoteEditor()
# метод один и тот же, не переопределенный, а ведет себя
# по-разному из-за разной реализации метода-“зацепки”
remote_editor.save()
```

Если рассматривать, что self передается в save() разных типов, до определенной степени это тоже является проявлением утиной типизации, однако так как что save(), что _save_document() связаны с self, и их нельзя вызвать в таком виде на произвольном объекте, не являющемся наследником TextEditor, этот пример больше похож на механизм виртуальных методов.

Дополнение метода предка, а не его полная замена

Часто мы не хотим полностью заменять какой-то метод, так как он уже содержит какую-то реализованную функциональность. А ее мы повторно прописывать не хотим. Для таких случаев есть возможность вызвать метод предка в переопределенном методе для того, чтобы вызвать уже реализованную функциональность. А до и после этого вызова можно дописать свой код.

```
class Parent:
    def do_something_useful(self):
        # реализация

class Child(Parent):
    def do_something_useful(self):
        # действия до
        super().do_something_useful() # вызов метода предка
        # действия после
```

Аналогично можно вызывать метод предка, возвращающего значение:

```
def do_something_useful(self):
    # действия до
    r = super().do_something_useful()
    # действия после
    return r
```

Или если действий после нет, можно сделать так:

```
def do_something_useful(self):
    # действия до
    return super().do_something_useful()
```