Продолжаем классы

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Среда, 28 сентября 2016 года

Где использовать классы

Инвариант — свойство, которое должно всегда выполняться.
Мнемоническое правило для некоторых случаев: если некоторые переменные удовлетворяют свойствам ниже, то перед вами наверняка объект.

- *Логически относятся к одной сущности*: если хранятся в разных массивах, то всегда с одинаковым индексом.
- *Связаны инвариантом*: если меняется одно свойство, то другое может стать некорректным.
- *Требуют обработки вместе*: нет разумных функций, принимающий только одну переменную из группы.

Где ещё использовать классы

В так называемых *паттернах* (будут потом в курсе). Удобно использовать для описания небольших самостоятельных кусочков программы, из которых можно что-то собирать. Например:

- Команды или операции, которые можно применить или отменить.
- Обработчки событий, которые могут реагировать на событие по-разному:
  - Проигнорировать.
  - Передать другому обработчику.
  - Записать в лог.
- Физические объекты: кнопки на экране (реагируют на нажатия, находятся в определённой точке экрана), игроки, окна на экране (содержат кнопки).

Это всего лишь парадигма!

Здесь идея «объектов» позволяет абстрагироваться от деталей реализации, причём с большей мощью, чем простые функции.

Объекты и классы — это просто **ещё один способ структуризации**!

Можно переписать любой код в объектно-ориентированном стиле без объектов вообще.

Наследование

- Иногда классы имеют общую (или похожую) функциональность. Например, кнопки и текстовые поля в окошке: прямоугольники, знают свой размер, могут себя нарисовать.
- Хранить общий код можно в функциях. Но с данными так не получится.
- Более изящное решение: наследование. Заводим общего предка (суперкласс) с общим кодом.
- Подклассы могут переопределять поведение суперкласса или расширять его.

Принцип подстановки Барбары Лисков: если где-то использовался суперкласс, то в том же месте можно без изменений безболезненно использовать и любой подкласс.

Пример-1

Суперкласс (родитель): «объект в окне приложения»:

- `width()/height()`
- `try_change_size(new_width, new_height) -> bool`
- `paint()`

Подклассы:

- *Кнопка без надписи*: рисует бордюр по периметру, реагирует на нажатия кнопки.
- *Текстовое поле*: не позволяет сильно уменьшить высоту.

Пример нарушения принципа: мы требуем вызывать метод `prepare_paint()` перед каждым `paint()`.
Иногда вместо наследования лучше подходит композиция (один объект хранится как поле другого). Например, если объекты сильно разные или если происходит нарушение принципа.

Пример-2

Суперкласс: «банковский аккаунт»:

- `balance()`
- `withdraw(amount)`
- `deposit(amount)`

Пример-2

Суперкласс: «банковский аккаунт»:

- `balance()`
- `withdraw(amount)`
- `deposit(amount)`

Подкласс: «дебетовый счёт», не разрешает уходить в отрицательный баланс.
Есть ли проблемы?

Пример-2

Суперкласс: «банковский аккаунт»:

- `balance()`
- `withdraw(amount)`
- `deposit(amount)`

Подкласс: «дебетовый счёт», не разрешает уходить в отрицательный баланс.
Есть ли проблемы?
Зависит от формулировки:

1. Если суперкласс гарантировал, что всегда можно снять деньги, то нарушается принцип Лисков.
2. Если суперкласс не давал никаких гарантий про операции, то проблем нет.

Пара слов про интерфейсы

Суперкласс в примере с банком — это не столько «общие функции», сколько «общий интерфейс» с какой-то реализацией. Соглашения обычно просто пишутся в комментариях.

Также есть чистые интерфейсы, в которых запрещены реализации. А зачем?

Пара слов про интерфейсы

Суперкласс в примере с банком — это не столько «общие функции», сколько «общий интерфейс» с какой-то реализацией. Соглашения обычно просто пишутся в комментариях.

Также есть чистые интерфейсы, в которых запрещены реализации. А зачем? Чтобы нельзя было забыть что-то реализовать и остаться со странной реализацией «по умолчанию».

В статически типизированных языках чистые интерфейсы полезны. В Python есть утиная типизация и они полезны разве что для программистов, читающих код.

Пара слов про интерфейсы

Суперкласс в примере с банком — это не столько «общие функции», сколько «общий интерфейс» с какой-то реализацией. Соглашения обычно просто пишутся в комментариях.

Также есть чистые интерфейсы, в которых запрещены реализации. А зачем? Чтобы нельзя было забыть что-то реализовать и остаться со странной реализацией «по умолчанию».

В статически типизированных языках чистые интерфейсы полезны. В Python есть утиная типизация и они полезны разве что для программистов, читающих код.

Есть ли теперь проблемы?

Пара слов про интерфейсы

Суперкласс в примере с банком — это не столько «общие функции», сколько «общий интерфейс» с какой-то реализацией. Соглашения обычно просто пишутся в комментариях.

Также есть чистые интерфейсы, в которых запрещены реализации. А зачем? Чтобы нельзя было забыть что-то реализовать и остаться со странной реализацией «по умолчанию».

В статически типизированных языках чистые интерфейсы полезны. В Python есть утиная типизация и они полезны разве что для программистов, читающих код.

Есть ли теперь проблемы? Да: всё ещё нет гарантий, что мы что-то забыли, но теперь программа просто упадёт.

Наследование в Python

```
class A:
    def __init__(self, foo):
        self.foo = foo
    def inc_foo(self):
        self.foo += 1
    def __str__(self):
        return "A({})".format(self.foo)

class B(A):
    # __init__ is not overwritten
    def inc_foo(self):  # override
        self.foo += 2  # problem?
    def __str__(self):  # override
        return "B({})".format(super(B, self).__str__())
```

Что произошло

- Класс `B` — наследник класса `A`.
- Конструктор остался старый.
- Методы `inc_foo` и `__str__` мы *перезаписали*
- В методе `__str__` вызываем старую реализацию.

Что произошло

- Класс `B` — наследник класса `A`.
- Конструктор остался старый.
- Методы `inc_foo` и `__str__` мы *перезаписали*
- В методе `__str__` вызываем старую реализацию.
- Метод `inc_foo` нарушил принцип подстановки Лисков.

Стреляем себе в ногу

Python — язык динамический. Смотрим демонстрацию.

Стреляем себе в ногу

Python — язык динамический. Смотрим демонстрацию.
Процесс поиска метода/переменной `x.foo`:

- В самом объекте `x`: `x.foo`
- В классе объекта `x`: `type(x).foo`
- Во всех предках класса `x`.

Прочие плюшки

- `Class.__name__`
- `x.__dict__` или `vars(x)`
- `type(x)`
- `isinstance(x, A)`
- `issubclass(A, B)`
- Есть множественное наследование.
- Есть даже ромбовидное наследование.
- При множественном наследовании есть специальный алгоритм «линеаризации», которые упорядочивает всех предков для поиска функций.