Парадигмы программирования

Парадигма программирования

Парадигма программирования — это набор идей и понятий, которые определяют стиль написания программ, подход к программированию.

Программа — это инструкция. Когда вы объясняете другу, как к вам доехать, вы, в принципе, программируете.

Объяснение, как к вам добраться, можно построить по-разному. Можно просто назвать адрес, а можно рассказать в деталях, где свернуть, на какую сторону улицы перейти, в какой дом зайти.

- Выходи на Александра Невского, сверни налево и иди до перекрёстка, там перейдёшь дорогу, свернёшь налево и пройдёшь до 38 дома, обойди дом, так как вход со двора, дойди до 2 подъезда слева и набери 2468, поднимись на четвёртый этаж, тридцать третья квартира.
- Адрес: ул. Свободы, д. 38, кв. 33, домофон 2468.

Парадигма программирования

Парадигма программирования — это набор идей и понятий, которые определяют стиль написания программ, подход к программированию.

Программа — это инструкция. Когда вы объясняете другу, как к вам доехать, вы, в принципе, программируете.

Объяснение, как к вам добраться, можно построить по-разному. Можно просто назвать адрес, а можно рассказать в деталях, где свернуть, на какую сторону улицы перейти, в какой дом зайти.

- Выходи на Александра Невского, сверни налево и иди до перекрёстка, там перейдёшь дорогу, свернёшь налево и пройдёшь до 38 дома, обойди дом, так как вход со двора, дойди до 2 подъезда слева и набери 2468, поднимись на четвёртый этаж, тридцать третья квартира.
- Адрес: ул. Свободы, д. 38, кв. 33, домофон 2468.

Императивный стиль

```
1  function onlyOdd(array) {
2    const result = []
3
4    for (const element of array) {
5      if (element % 2 !== 0) {
6        result.push(element)
7      }
8    }
9
10   return result
11 }
```

Императивная — это самая простая и часто используемая парадигма. Её смысл в последовательном выполнении действий.

К императивной парадигме относятся следующие виды программирования:

- процедурное;
- структурное;
- объектно-ориентированное

Декларативный стиль

```
1 function onlyOdd(array) {
2   return array.filter((element) => element % 2 !== 0)
3 }
```

В декларативной парадигме разработчик описывает проблему и ожидаемый результат, но не пишет никаких инструкций. В декларативном программировании отсутствуют переменные, состояние и прочие понятия, которые свойственны императивной парадигме.

К декларативной парадигме относятся функциональное и логическое программирование.

Зачем

- декларативное программирование — это обёртка для императивного. Компьютер не может вот так просто понять, чего хочет программист, поэтому для него нужно написать конкретные инструкции, что и в каком порядке делать.
- декларативное программирование не подходит для задач, для решения которых важно иметь доступ к состоянию программы — например, если нужно проверить, нажата ли кнопка или поставлена ли галочка в чекбокс.
- императивное программирование даёт больше свободы, поэтому его чаще используют в творческих областях, особенно там, где важен порядок выполняемых действий.

Процедурное программирование

```
/*  Как пример мы можем рассмотреть программу,
    которая использует «подпрограммы» (в нашем случае функции),
    меняя состояние памяти (в нашем случае простой массив битов).

    Состояние памяти потом может быть использовано,
    например, для работы с каким-то устройством. */

let memory = [0, 0, 0, 0, 0, 0, 0, 0]

function invertSmallestBit() {
  memory[7] = Number(!memory[7])
  return memory
}

function invertBiggestBit() {
  memory[0] = Number(!memory[0])
  return memory
}

invertSmallestBit()
// [0, 0, 0, 0, 0, 0, 0, 1]

invertBiggestBit()
// [1, 0, 0, 0, 0, 0, 0, 1]

invertSmallestBit()
// [1, 0, 0, 0, 0, 0, 0, 0]
```

Это парадигма, в которой последовательные команды собираются в подпрограммы.

Между собой эти подпрограммы общаются через общую память. Если проводить аналогию с функциями, то они бы общались через глобальные переменные.

Из подпрограмм появилось понятие модулей, но само по себе процедурное программирование не очень удобно в плане переиспользования кода.

Примеры языков: C, Pascal, COBOL, ALGOL, BASIC, Fortran.

Объектно ориентированное программирование

ООП (объектно-ориентированное программирование) — парадигма, в которой сущности в программе представляются в виде объектов.

Каждый объект — экземпляр какого-то класса, некой абстрактной сущности, в которой описано поведение.

ООП характеризуется 4 основными аспектами:

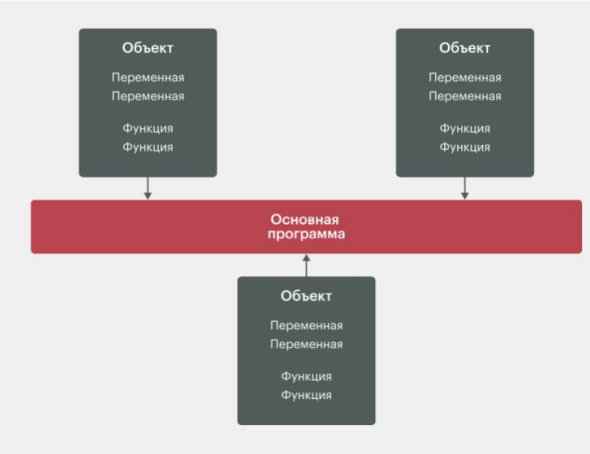
- Абстракцией — выделением таких характеристик объекта, которые достаточно точно описывают его поведение, но не вдаются в детали;
- Инкапсуляцией — размещением данных внутри того объекта, который их использует;
- Полиморфизмом — умением работать с разными типами объектов или данных;
- Наследованием — умением объекта «забирать по наследству» свойства или характеристики от объектов-родителей.

Примеры языков: Java, Python, C++, Ruby, C#, Objective-C, PHP.

ООП

Краеугольное понятие в ООП — объект. Это такой своеобразный контейнер, в котором сложены данные и прописаны действия, которые можно с этими данными совершать.

Логика ООП совершенно: к основной программе подключаются не функции, а объекты, внутри которых уже лежат собственные переменные и функции. Так выстраивается более иерархичная структура. Переменные внутри объектов называются полями, или атрибутами, а функции — методами.



ООП

Каждый объект в ООП строится по определённому классу — абстрактной модели, описывающей, из чего состоит объект и что с ним можно делать.

Например, у нас есть класс «Кошка», обладающий атрибутами «порода», «окрас», «возраст» и методами «мяукать», «мурчать», «умываться», «спать». Присваивая атрибутам определённые значения, можно создавать вполне конкретные объекты.

ООП

Инкапсуляция

Доступ к данным объекта должен контролироваться, чтобы пользователь не мог изменить их в произвольном порядке и что-то поломать. Поэтому для работы с данными программисты пишут методы, которые можно будет использовать вне класса и которые ничего не сломают внутри.

```
class Cat():
  def __init__(self, breed, color, age):
     self._breed = breed
     self._color = color
     self._age = age

  @property
  def breed(self):
    return self._breed

  @property
  def color(self):
    return self._color

  @property
  def age(self):
    return self._age

  @age.setter
  def age(self, new_age):
    if new_age > self._age:
      self._age = new_age
    return self._age
```

Наследование

Классы могут передавать свои атрибуты и методы классам-потомкам.

```
class HomeCat(Cat):
  def __init__(self, breed, color, age, owner, name):
    super().__init__(breed, color, age)
    self._owner = owner
    self._name = name

  @property
  def owner(self):
    return self._owner

  @property
  def name(self):
    return self._name

  def getTreat(self):
    print('Мяу-мяу')
```

Полиморфизм

Этот принцип позволяет применять одни и те же команды к объектам разных классов, даже если они выполняются по-разному.

```
class Cat:
  def sleep(self):
    print('Свернулся в клубок и сладко спит.')

class Parrot:
  def sleep(self):
    print('Сел на жёрдочку и уснул.')
def homeSleep(animal):
  animal.sleep()
```

```
cat = Cat()
parrot = Parrot()
homeSleep(cat) # Свернулся в клубок и сладко спит.
homeSleep(parrot) # Сел на жёрдочку и уснул.
```

Абстракция

При создании класса мы упрощаем его до тех атрибутов и методов, которые нужны именно в этом коде, не пытаясь описать его целиком и отбрасывая всё второстепенное. Например, у всех хищников есть метод «охотиться», поэтому все животные, которые являются хищниками, автоматически будут уметь охотиться.

```
class Predator:
  def hunt(self):
    print('Охотится...')
```

```
class Cat(Predator):
  def __init__(self, name, color):
    super().__init__()
    self._name = name
    self._color = color

  @property
  def name(self):
    return self._name

  @property
  def color(self):
    return self._color
```

ООП

Плюсы ООП

Самый важный плюс — это *удобство моделирования систем*. Когда каждый компонент системы представлен в виде объекта, отношения между этими объектами проще регламентировать и зафиксировать.

Ещё один плюс — это *изученность подхода*. ООП достаточно старо, и о нём написано очень много книг и рекомендаций, а минусы хорошо изучены. Поэтому и велосипедов писать по ходу написания приложения не надо.

Минусы ООП

Один из принципов ООП — это инкапсуляция, из-за которой доступ к данным может быть ограничен. Если мы хотим этими данными поделиться, то может случиться, что доступ к ним хотят получить сразу несколько объектов.

Ещё одна проблема — это *наследование*. Простое наследование не всегда полностью отражает отношения компонентов.

Логическое программирование

В целом это скорее математика, чем программирование. Его суть заключается в том, чтобы, используя математические доказательства и законы логики, решать бизнес-задачи.

Чтобы использовать логическое программирование, необходимо уметь переводить любую задачу на язык математики.

Логическое программирование часто используется для моделирования процессов.

Функциональное программирование

В этой парадигме понятие функции близко к математическому понятию функции. То есть это штука, которая как-то преобразует *входные* данные.

Особенность функции в этой парадигме в том, что она должна быть *чистой*, то есть должна зависеть только от аргументов и не может иметь никаких побочных эффектов.

Побочный эффект — это какое-либо изменение внешней среды.

Если функция меняет глобальную переменную или, например, вызывает метод внешнего объекта, то она меняет внешнюю среду. Это и есть побочный эффект.

Функциональное программирование

```
// Эта функция чистая:

function double(x) {
  return x * 2
}

/*  При одинаковых вызовах она
    всегда возвращает одинаковый результат. */

double(2)
// 4
double(2)
// 4
double(2)
// 4

// Следующая функция нечистая:

let x = 1

function double() {
  x *= 2
  return x
}

/*  Она меняет (или мутирует) переменную x,
    которая находится снаружи области видимости функции. */

double()
// 2
double()
// 4
double()
// 8
```

Функциональное программирование

Плюсы ФП

Мощь функционального программирования проявляется в *параллельных вычислениях*. Так как нет никакого общего состояния или общей памяти, параллелить вычисления можно сколько угодно, никаких негативных последствий это не вызовет.

Кроме этого чистые функции отлично *тестируются*, потому что не требуют сложной настройки теста. Мы прекрасно видим, что функции потребуется для проверки, потому что всё находится в списке аргументов.

Минусы ФП

Первый минус — *потребление памяти*. ФП требует, чтобы не было побочных эффектов. Значит, если мы хотим изменить какой-то объект, нам надо создать свежую копию этого объекта и менять её. Иногда это может приводить к большому количеству данных, которые надо держать в памяти.

Вселенная — это *побочный эффект*. Концепция чистых функций — это замечательно, но ФП в чистейшем его виде просто невозможно, потому что общение с пользователем, сетью, обработка ошибок — либо сами по себе побочные эффекты, либо включают их в себя.

Примеры использования

1. Процедурная парадигма подойдёт для написания кода микрочипа.
Он может иметь слишком специфичный процессор, чтобы разрабатывать для него компилятор сложного объектно-ориентированного языка. Технических возможностей чипа может быть недостаточно для работы программ, написанных на современных языках.

2. Объектно-ориентированная парадигма.
Каждый пользовался программами, написанными с использованием объектно-ориентированной парадигмы. Большинство современных приложений для компьютеров и смартфонов, популярные интернет-сервисы, умный телевизор и роутер написаны на объектно-ориентированном языке.

3. Декларативная парадигма отлично подходит для описания внешнего вида веб-сайтов.
Сейчас для этого используется декларативное программирование на языке CSS (от англ. Cascading Style Sheets — «каскадные таблицы стилей»). С его помощью задают шрифт текста, цвет фона, размер картинки.

4. Функциональная парадигма используется в криптовалютах и блокчейне.
Это области, в которых надёжность выдвигается на первый план, они новые и открыты для экспериментальных технологий. Другие функциональные языки стирают грань между математикой и программированием, позволяя буквально программировать математические доказательства. В основе этого соответствие Карри — Ховарда — математический факт, который гласит, что программа на функциональном языке и доказательство — это одно и то же. Его можно использовать для формальной верификации — строгой автоматической проверки корректности программ.