Парадигмы программирования

Я думаю начать стоит с определения. Паради́гма программи́рования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Важно отметить, что парадигма программирования не определяется однозначно языком программирования; практически все современные языки программирования в той или иной мере допускают использование различных парадигм (мультипарадигмальное программирование). Так, на языке Си, который не является объектно-ориентированным, можно работать в соответствии с принципами объектно-ориентированного программирования, хотя это и сопряжено с определёнными сложностями; функциональное программирование можно применять при работе на любом императивном языке, в котором имеются функции.

Во втором семестре первого курса мы столкнулись лицом к лицу с объектно-ориентированным программированием. Первым языком, в контексте которого это случилось, был С++. Парадигма была легка в освоении, за счет проецирования разработки на повседневную жизнь. Вместе с этой парадигмой мы жили всю нашу сознательную университетскую жизнь. Все помним три принципа ООП: инкапсуляция, полиморфизм, наследование. Надеюсь даже помним их определения. Но ведь множество парадигм не ограничивается ООП. Их список довольно обширен: **автоматное программирование, аспектно-ориентированное, визуальное, вычисления с откатами, декларативное, императивное, конкатенативное, логическое, матричное, метапрограммирование, мультипарадигма, на уровне значений, на уровне функций, строгое, нестрогое, обмен сообщениями, обобщённое, объектно-ориентированное, параллельное, потоковое, правила переписывания, предметно-ориентированное, прототипное, процедурное, рефлексивное, скалярное, стек-ориентированное, структурное, табличное, функциональное, эзотерическое** (более 30 штук). Я же остановлюсь на самых звучных.

**Объектно-ориентированное программирование:**

Представляет программу как набор объектов и их взаимодействий. Основными понятиями ООП:

**объект** — элементарная сущность, описываемая определенными свойствами (хранящимися в виде атрибутов объекта) и поведением (реализованным в виде методов);

**класс** описывает структуру свойств и поведения одного типа объектов. Каждый объект программы является экземпляром некоторого класса;

классы могут **наследовать атрибуты и методы** их родительских классов, в то же время добавляя свои собственные. Иерархия классов позволяет моделировать сущности решаемой задачи на нескольких уровнях детализации и в дальнейшем использовать класс, отвечающий уровню детализации, необходимому для решения конкретной подзадачи.

**инкапсуляция** подразумевает, что некоторые детали реализации класса скрыты от взаимодействующих с ним объектов. У каждого класса есть интерфейс, описывающий взаимодействие объектов этого класса с прочими объектами, и реализация, описывающая то, как это взаимодействие отражается на объекте этого класса.

**Императивное программирование:**

Исторически сложилось так, что подавляющее большинство вычислительной техники, которую мы программируем имеет состояние и программируется инструкциями, поэтому первые языки программирования в основном были чисто императивными, т.е. не поддерживали никаких парадигм кроме императивной. Это были машинные коды, языки ассемблера и ранние высокоуровневые языки, вроде Fortran.

**Ключевые моменты:**

В этой парадигме вычисления описываются в виде инструкций, шаг за шагом изменяющих состояние программы.

В низкоуровневых языках (таких как язык ассемблера) состоянием могут быть память, регистры и флаги, а инструкциями — те команды, что поддерживает целевой процессор.

В более высокоуровневых (таких как Си) состояние — это только память, инструкции могут быть сложнее и вызывать выделение и освобождение памяти в процессе своей работы.

В совсем высокоуровневых (таких как Python, если на нем программировать императивно) состояние ограничивается лишь переменными, а команды могут представлять собой комплексные операции, которые на ассемблере занимали бы сотни строк.

Языки поддерживающие данную парадигму:

**Как основную:**

— Языки ассемблера

— Fortran

— Algol

— Cobol

— Pascal

— C

— C++

— Ada

**Как вспомогательную:**

— Python

— Ruby

— Java

— C#

— PHP

— Haskell (через монады)

Стоит заметить, что большая часть современных языков в той или иной степени поддерживает императивное программирование. Даже на чистом функциональном языке Haskell можно писать императивно.

**Процедурное программирование:**

Опять-же возрастающая сложность программного обеспечения заставила программистов искать другие способы описывать вычисления.

Собственно еще раз были введены дополнительные понятия, которые позволили по-новому взглянуть на программирование.

Этим понятием на этот раз была процедура.

В результате возникла новая методология написания программ, которая приветствуется и по сей день — исходная задача разбивается на меньшие (с помощью процедур) и это происходит до тех пор, пока решение всех конкретных процедур не окажется тривиальным.

Ключевые моменты:

Процедура — самостоятельный участок кода, который можно выполнить как одну инструкцию.

В современном программировании процедура может иметь несколько точек выхода (return в C-подобных языках), несколько точек входа (с помощью yield в Python или статических локальных переменных в C++), иметь аргументы, возвращать значение как результат своего выполнения, быть перегруженной по количеству или типу параметров и много чего еще.

**Функциональное программирование:**

парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний (в значении, подобном таковому в теории автоматов). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции могут опираться не только на аргументы, но и на состояние внешних по отношению к функции переменных, а также иметь побочные эффекты и менять состояние внешних переменных. Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить разные данные на выходе из-за влияния на функцию состояния переменных. А в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат: выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (см. ниже Чистые функции).

λ-исчисления являются основой для функционального программирования, многие функциональные языки можно рассматривать как «надстройку» над ними.

**Эзотерическое программирование:**

Языки, поддерживающие данную парадигму создаются не с серьезными намерениями, а в качестве шутки или вызову самому себе. Примеры: Brainfuck, Cat.

Кратко о **вычислениях с откатами:**

Языки с поддержкой этой парадигмы поддерживают откат в явном виде к предыдущему состоянию. Пример такого языка: Prolog.

Кратко о **табличном программировании:**

Разновидность объектно-ориентированного программирования, в котором классы представляются в виде таблиц. Столбцы содержат свойства классов, а строки - экземпляры объектов.

**Метапрограммирование**

Метапрограммирование предусматривает написание программ, которые работают с другими программами в качестве данных. Язык обрабатывающей программы называется метаязыком, язык обрабатываемой — объектным языком.

Простейшим примером метапрограммирования является любой компилятор, преобразующий код, написанный на языке высокого уровня, в низкоуровневый машинный язык или ассемблер. Очевидно, что большинство языков, поддерживающих работу со строками, могут использоваться для непосредственной генерации кода для других языков. Тем не менее, термин “метапрограммирование” обычно подразумевает, что в качестве метаязыка и объектного языка выступает один и тот же язык, и более того, такое его использование предусмотрено дизайном языка. (Rust, Perl, диалекты Lisp: Clojure, Common Lisp, Scheme)

**На уровне значений**

Программирование на уровне значений (также известное как модель фон Неймана) представляет программу в виде последовательности значений, преобразующихся друг в друга. Выполнение программы начинается с исходных данных, которые комбинируются и преобразуются в другие значения до тех пор, пока не получаются нужные результаты. Новые значения конструируются из имеющихся при помощи предопределенного набора операций.

Эта парадигма сосредотачивается на изучении типов данных, т.е. значений и элементарных операций над ними, их структуры и свойств. Обычно элементарные операции образуют алгебру над пространством значений.

Большинство современных языков используют понятия типов данных, переменных и операторов присваивания, и следовательно, реализуют эту парадигму.

Программирование на уровне значений является противоположностью программирования на уровне функций.

(ECMAScript, C++, C#, Java, Brainfuck, Scala…)

**На уровне функций**

Программирование на уровне функций (другое русское название — комбинаторное программирование) предполагает, что программа строится из элементарных функций, комбинируемых при помощи функционалов (функциональных форм).

Эта парадигма не использует понятия переменной или операции присваивания, а вместо этого сосредотачивается на изучении элементарных функций и функциональных форм.

Программирование на уровне функций использует следующую иерархию сущностей в пределах языка программирования:

• атомы — единицы данных, с которыми оперируют функции. Данные появляются только на входе и выходе программы, и нигде внутри. Атомы могут быть скалярами или множествами других атомов.

• функции — инструменты, преобразующие атомы в другие атомы. Язык задает начальный набор функций, а программист может определять новые, используя функциональные формы. Сама программа тоже является функцией.

• функциональные формы — инструменты, преобразующие функции в другие функции. Язык задает начальный набор функциональных форм, и либо позволяет создание новых форм (FFP), либо нет (FP). Таким образом, язык задает алгебру функциональных форм над пространством функций.

Программирование на уровне функций является противоположностью программирования на уровне значений и разновидностью функционального программирования (с ограничением на то, как создаются новые функции).

(APL, FP, J)

**Нестрогая**

Нестрогий язык программирования позволяет пользователю определять нестрогие функции и, как следствие, использовать ленивые вычисления.

В большинстве нестрогих языков «нестрогость» также применяется к конструкторам данных. Это позволяет управлять бесконечными структурами данных (к примеру, списком всех простых чисел) точно так же, как и обычными конечным. Это облегчает использование очень больших, но конечных структур, к примеру таких, как полное дерево игры в шахматы.

Нестрогость имеет некоторые недостатки, которые помешали её повсеместному использованию: \* из-за неопределённости касательно того, будут ли и когда будут вычислены выражения, нестрогие языки должны быть чисто функциональными для удобного использования; *большинство распространённых архитектур оптимизированы для строгих языков, т. о. лучшие компиляторы для нестрогих языков обычно производят код хуже, чем лучше компиляторы строгих;* пространственную сложность (space complexity) нестрогих программ сложно понять или предсказать.

Термины «энергичные языки программирования» (eager) и «ленивые языки программирования» (lazy) часто используются как синонимы «строгие языки программирования» и «нестрогие языки программирования» соответственно.

Во многих строгих языках некоторые положительные стороны нестрогих функций могут быть достигнуты с помощью макросов. (Аналитик, Miranda…)

**Строгая**

В строгом языке программирования могут быть определены только строгие функции.

В денотационной семантике языков программирования функция f является строгой если f (\_|\_) = *|*. Сущность «\_|\_» обозначает выражение, которое не возвращает нормального значения по причине выполнения бесконечного цикла или прекращения из-за ошибки (к примеру, деления на ноль).

**Обобщенная**

Обобщённое программирование состоит в написании алгоритмов в терминах абстрактных типов данных; когда алгоритм используется для конкретных типов данных, создается экземпляр этого алгоритма с типами данных, переданными в качестве параметров. Такой стиль программирования позволяет использовать универсальный код для похожих заданий, имеющих дело с разными типами данных, и таким образом уменьшить дублирование кода.

Обобщённое программирование широко используется для реализации универсальных контейнеров и алгоритмов. Так, стандартная библиотека шаблонов STL в С++ предоставляет набор контейнеров (динамический массив, связный список, очередь, множество и т.д.) и алгоритмов, применимых к этим или пользовательским контейнерам.