## Парадигмы программирования

### Введение

**Парадигма программирования** — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Своим современным значением в научно-технической области термин «парадигма» обязан, по-видимому, Томасу Куну и его книге «[Структура научных революций](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BD%D0%B0%D1%83%D1%87%D0%BD%D1%8B%D1%85_%D1%80%D0%B5%D0%B2%D0%BE%D0%BB%D1%8E%D1%86%D0%B8%D0%B9)». Кун называл парадигмами устоявшиеся системы научных взглядов, в рамках которых ведутся исследования. Согласно Куну, в процессе развития научной дисциплины может произойти замена одной парадигмы на другую (как, например, геоцентрическая небесная механика Птолемея сменилась гелиоцентрической системой Коперника), при этом старая парадигма ещё продолжает некоторое время существовать и даже развиваться благодаря тому, что многие её сторонники оказываются по тем или иным причинам неспособны перестроиться для работы в другой парадигме.

Термин «парадигма программирования» впервые применил в 1978 году Роберт Флойд - лауреат премии Тьюринга.

Флойд отмечает, что в программировании можно наблюдать явление, подобное парадигмам Куна, но, в отличие от них, парадигмы программирования не являются взаимоисключающими:

Если прогресс искусства программирования в целом требует постоянного изобретения и усовершенствования парадигм, то совершенствование искусства отдельного программиста требует, чтобы он расширял свой репертуар парадигм.

Таким образом, по мнению Роберта Флойда, в отличие от парадигм в научном мире, описанных Куном, парадигмы программирования могут сочетаться, обогащая инструментарий программиста.

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.

Первой парадигмой, с которой знакомят современные ВУЗы в первую очередь - ООП. Она легка в освоении, за счет проецирования разработки на повседневную жизнь. Но ведь *список парадигм* не ограничивается объектно-ориентированным. Он намного больше: 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).

### Объектно-ориентированное программирование

Представляет программу как набор объектов и их взаимодействий.

#### Основные понятия

* *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.

### Аспектно-ориентированное программирование

Существующие парадигмы программирования — процедурное, модульное, объектно-ориентированное программирование (ООП) и предметно-ориентированное проектирование — предоставляют определённые способы для разделения и выделения функциональности: функции, модули, классы, но некоторую функциональность с помощью предложенных методов невозможно выделить в отдельные сущности. Такую функциональность называют сквозной (от англ. scattered — разбросанный или англ. tangled — переплетённый), так как её реализация распределена по различным модулям программы. Сквозная функциональность приводит к рассредоточенному и запутанному коду, сложному для понимания и сопровождения.

Ведение лога и обработка исключений — типичные примеры сквозной функциональности. Другие примеры: трассировка; аутентификация и проверка прав доступа; контрактное программирование (в частности, проверка пред- и постусловий). Для программы, написанной в парадигме ООП, любая функциональность, по которой не была проведена декомпозиция, является сквозной.

Все языки АОП предоставляют средства для выделения сквозной функциональности в отдельную сущность. Так как *AspectJ* является родоначальником этого направления, используемые в этом расширении концепции распространились на большинство языков АОП.

#### Основные понятия

* **Аспект** (англ. aspect) — модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определённых некоторым срезом.
* **Совет** (англ. advice) — средство оформления кода, которое должно быть вызвано из точки соединения. Совет может быть выполнен до, после или вместо точки соединения.
* **Точка соединения** (англ. join point) — точка в выполняемой программе, где следует применить совет. Многие реализации АОП позволяют использовать вызовы методов и обращения к полям объекта в качестве точек соединения.
* **Срез** (англ. pointcut) — набор точек соединения. Срез определяет, подходит ли данная точка соединения к данному совету. Самые удобные реализации АОП используют для определения срезов синтаксис основного языка (например, в AspectJ применяются Java-сигнатуры) и позволяют их повторное использование с помощью переименования и комбинирования.
* **Внедрение** (англ. introduction, введение) — изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код. Обычно реализуется с помощью некоторого *метаобъектного протокола* (англ. metaobject protocol, MOP).

### 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.

Примеры языков: Rust, Perl, диалекты Lisp: Clojure, Common Lisp, Scheme.

### Программирование 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.

Примеры языков: ECMAScript, C++, C#, Java, Brainfuck, Scala.

### Программирование 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.

#### Основные понятия

* **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).

Примеры языков: APL, FP, J.

### Строгое программирование:

В строгом языке программирования могут быть определены только строгие функции, то есть функции, которые придерживаются строго модели вычислений.

Строгая модель вычислений означает, что аргументы всегда вычисляются полностью до применения функции к ним. Нестрогая модель вычислений означает, что аргументы не вычисляются до тех пор, пока их значение не используется в теле функции. В ряде языков булевы выражения имеют нестрогий порядок вычисления, называемый в русской литературе «вычислениями по короткой схеме», где вычисления прекращаются, как только результат становится однозначно предсказуем — например, значение «истина» в операции дизъюнкции, «ложь» в операции конъюнкции, и так далее. Операторы ветвления зачастую также имеют ленивую семантику вычислений, то есть возвращают результат всего оператора, как только однозначная ветвь его породит.

### Нестрогое программирование:

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.

Примеры языков: Аналитик, Miranda.

### Обобщенное 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.

### Стек-ориентированное программирование

Стек-ориентированная парадигма программирования использует для передачи параметров модель *стека*.

Стек-ориентированный язык программирования оперирует одним или несколькими стеками и обычно использует префиксную или постфиксную нотацию вместо инфиксной, обычной для других языков. Две основные операции, которые выполняются над данными в стеке — pop (удалить верхний элемент и вернуть его) и push (добавить элемент в верх стека). Иногда стек-ориентированные языки предоставляют и более сложные операции, например, dup (скопировать верхний элемент стека и добавить его в верх стека), swap (поменять местами два верхних элемента стека), roll (циклически переставить элементы в заданной части стека) и drop (удалить верхний элемент стека, не возвращая его).

### Скалярное программирование

Низкоуровневая парадигма, диктующая отсутствие в языке матричных операций. Каждая операция применяется к отдельным скалярным величинам, но не ко всему массиву. Таким образом, программист должен организовать обработку массива как последовательность скалярных операций.

### Эзотерическое программирование

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

Примеры языков: Brainfuck, Cat.

### Заключение

Важно заметить, что универсальной парадигмы, или просто самой хорошей парадигмы не существует. Также крайне сложно сказать возможно ли это вообще. Все парадигмы имеют свои области применения, свои плюсы и минусы. Даже то же ООП, которое в современном программировании занимает лидирующие позиции, не обделено своими минусами. (Статья приложения "почему ООП провалилось"). Но вместе с тем, оно заняло свою нишу. Причин тому много, но одна из них кроется в то, что программирование — это искусство оперирования абстрактной информацией. Это очень сложное искусство — поскольку человеческий мозг вообще не приспособлен для оперирования абстракциями. А ООП перекладывает абстракции на вполне осязаемые сущности (в общем случае). Сейчас программист, непонимающий основ ООП, - либо плохой специалист, либо специалист, занявший свою нишу.

### Приложение

Почему ООП провалилось (<http://blogerator.ru/page/oop_why-objects-have-failed>)

Абсурдопедия ([http://absurdopedia.wikia.com/wiki/ООП](http://absurdopedia.wikia.com/wiki/%D0%9E%D0%9E%D0%9F))