**Методологии программирования**

**Методология программирования** – совокупность методов, применяемых на различных стадиях [жизненного цикла](https://ru.wikipedia.org/wiki/%D0%96%D0%B8%D0%B7%D0%BD%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%86%D0%B8%D0%BA%D0%BB_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) [программного обеспечения](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) и имеющих общий философский подход.

Каждая [методология](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F) характеризуется своим:

-философским подходом или основными принципами. Эти принципы, от которых зависит эффективность всей методологии, обычно можно кратко сформулировать и легко объяснить;

-согласованным множеством моделей методов, которые реализуют данную методологию;

-концепциями (понятиями), позволяющими более точно определить методы.

В частном случае, когда методология применяется на стадии программирования (конструирования), её обычно называют [парадигмой программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)

Можно проследить три пути возникновения методологий.

Во-первых, они могут являться выражением практического опыта.

Во-вторых, методологии могут происходить от одной из четырёх моделей алгоритма: абстрактная [машина Тьюринга](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%B0_%D0%A2%D1%8C%D1%8E%D1%80%D0%B8%D0%BD%D0%B3%D0%B0) (императивное программирование), рекурсивные функции Гильберта и Аккермана (структурное программирование), лямбда-исчисление Чёрча (функциональное программирование), нормальные алгорифмы Маркова (логическое программирование).

В-третьих, методологии можно объяснить через отображение одной из трёх структур языка моделирования на структуру языка программирования. Составными частями могут быть структура данных, структура управления и логика. Каждое из девяти отображений определяет либо методологию, либо достаточно серьёзный метод программирования. Например, отображение логика-логика лежит в основе логического программирования.

**Классификация:**

**По ядрам**

При подходе к методологии, как имеющей ядро ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) core), соответствующее способу описания алгоритма, и дополнительные особенности, можно выделить следующие пять основных ядер методологий:

**1. Методология** [**императивного программирования**](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

Императивное программирование – это [парадигма программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), которая, в отличие от [декларативного программирования](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), описывает процесс вычисления в виде [инструкций](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), изменяющих состояние данных. Императивная программа очень похожа на приказы, выражаемые повелительным наклонением в [естественных языках](https://ru.wikipedia.org/wiki/%D0%95%D1%81%D1%82%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA), то есть это последовательность команд, которые должен выполнить [компьютер](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80).

В отличие от [функционального программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) (относящегося к декларативной парадигме), при императивном подходе интенсивно используется [присваивание](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D1%81%D0%B2%D0%B0%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), что увеличивает сложность моделей вычислений и делает императивные программы подверженными специфическим ошибкам, не встречающимся при функциональном программировании.

Первыми императивными языками были [машинные коды](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) – родной язык программирования для компьютера. В этих языках инструкции были крайне просты, что снижало нагрузку на компьютеры, однако затрудняло написание крупных программ. В [1954 году](https://ru.wikipedia.org/wiki/1954_%D0%B3%D0%BE%D0%B4) появился первый «человеческий» [высокоуровневый язык программирования](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) – [FORTRAN](https://ru.wikipedia.org/wiki/FORTRAN), разработанный [Джоном Бэкусом](https://ru.wikipedia.org/wiki/%D0%94%D0%B6%D0%BE%D0%BD_%D0%91%D1%8D%D0%BA%D1%83%D1%81) в [IBM](https://ru.wikipedia.org/wiki/International_Business_Machines). FORTRAN является [компилируемым языком программирования](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%22%20%5Co%20%22%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D1%8B%D0%B9%20%D1%8F%D0%B7%D1%8B%D0%BA%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), позволяющим использовать именованные переменные, составные выражения, подпрограммы и многие другие элементы, распространённые в императивных языках. В конце 1950-х годов с целью упростить выражение математических алгоритмов был разработан [ALGOL](https://ru.wikipedia.org/wiki/ALGOL); в дальнейшем он послужил базой для написания [операционных систем](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) для некоторых моделей компьютеров. [COBOL](https://ru.wikipedia.org/wiki/COBOL) ([1960](https://ru.wikipedia.org/wiki/1960_%D0%B3%D0%BE%D0%B4)) и [BASIC](https://ru.wikipedia.org/wiki/BASIC) ([1964](https://ru.wikipedia.org/wiki/1964_%D0%B3%D0%BE%D0%B4)) являлись первыми попытками сделать программирование более похожим на обычный английский язык. В [1970-х годах](https://ru.wikipedia.org/wiki/1970-%D0%B5_%D0%B3%D0%BE%D0%B4%D1%8B) [Никлаус Вирт](https://ru.wikipedia.org/wiki/%D0%9D%D0%B8%D0%BA%D0%BB%D0%B0%D1%83%D1%81_%D0%92%D0%B8%D1%80%D1%82) разработал язык [Pascal](https://ru.wikipedia.org/wiki/Pascal). Язык[C](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) был создан [Денисом Ритчи](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BD%D0%B8%D1%81_%D0%A0%D0%B8%D1%82%D1%87%D0%B8). Команда разработчиков из [Honeywell](https://ru.wikipedia.org/wiki/Honeywell) начала разработку языка [Ada](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) в [1978 году](https://ru.wikipedia.org/wiki/1978_%D0%B3%D0%BE%D0%B4), и через четыре года опубликовала требования для его работы. Спецификация увидела свет в [1983 году](https://ru.wikipedia.org/wiki/1983_%D0%B3%D0%BE%D0%B4) и была обновлена в [1995](https://ru.wikipedia.org/wiki/1995_%D0%B3%D0%BE%D0%B4) и [2005](https://ru.wikipedia.org/wiki/2005_%D0%B3%D0%BE%D0%B4)–[2006 годах](https://ru.wikipedia.org/wiki/2006_%D0%B3%D0%BE%D0%B4).

В [1980-х](https://ru.wikipedia.org/wiki/1980-%D0%B5_%D0%B3%D0%BE%D0%B4%D1%8B) резко возрос интерес к [объектно-ориентированному программированию](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%25D) (ООП). [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk)-80, впервые разработанный [Аланом Кэем](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D0%B9%2C_%D0%90%D0%BB%D0%B0%D0%BD_%D0%9A%D1%83%D1%80%D1%82%D0%B8%D1%81) в [1969 году](https://ru.wikipedia.org/wiki/1969_%D0%B3%D0%BE%D0%B4), был обновлён в [1980](https://ru.wikipedia.org/wiki/1980_%D0%B3%D0%BE%D0%B4) исследовательским центром[Xerox PARC](https://ru.wikipedia.org/wiki/Xerox_PARC). По образу и подобию языка [Simula](https://ru.wikipedia.org/wiki/Simula) (предположительно, первого в мире ООП-языка, разработанного ещё в [1960-х](https://ru.wikipedia.org/wiki/1960-%D0%B5_%D0%B3%D0%BE%D0%B4%D1%8B)) [Бьерн Страуструп](https://ru.wikipedia.org/wiki/%D0%91%D1%8C%D0%B5%D1%80%D0%BD_%D0%A1%D1%82%D1%80%D0%B0%D1%83%D1%81%D1%82%D1%80%D1%83%D0%BF) разработал [C++](https://ru.wikipedia.org/wiki/C%2B%2B), основанный на [C](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29). C++ был впервые реализован в [1985 году](https://ru.wikipedia.org/wiki/1985_%D0%B3%D0%BE%D0%B4). В [1987 году](https://ru.wikipedia.org/wiki/1987_%D0%B3%D0%BE%D0%B4) [Ларри Уолл](https://ru.wikipedia.org/wiki/%D0%A3%D0%BE%D0%BB%D0%BB%2C_%D0%9B%D0%B0%D1%80%D1%80%D0%B8) выпустил язык [Perl](https://ru.wikipedia.org/wiki/Perl); [Python](https://ru.wikipedia.org/wiki/Python) был выпущен в [1990-м](https://ru.wikipedia.org/wiki/1990_%D0%B3%D0%BE%D0%B4) [Гвидо ван Россумом](https://ru.wikipedia.org/wiki/%D0%92%D0%B0%D0%BD_%D0%A0%D0%BE%D1%81%D1%81%D1%83%D0%BC%2C_%D0%93%D0%B2%D0%B8%D0%B4%D0%BE); в [1994](https://ru.wikipedia.org/wiki/1994_%D0%B3%D0%BE%D0%B4) [Расмус Лердорф](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D1%80%D0%B4%D0%BE%D1%80%D1%84%2C_%D0%A0%D0%B0%D1%81%D0%BC%D1%83%D1%81) разработал [PHP](https://ru.wikipedia.org/wiki/PHP); [Java](https://ru.wikipedia.org/wiki/Java) был разработан в [Sun Microsystems](https://ru.wikipedia.org/wiki/Sun_Microsystems) в 1994; [Ruby](https://ru.wikipedia.org/wiki/Ruby) был выпущен в [1995](https://ru.wikipedia.org/wiki/1995_%D0%B3%D0%BE%D0%B4); [C#](https://ru.wikipedia.org/wiki/C_Sharp) был выпущен в 2002 вместе с программной платформой (фреймворком) [.NET Framework](https://ru.wikipedia.org/wiki/.NET_Framework), объединяющей многие языки.

**2. Методология** [**ООП**](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%25D)

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Необходимо обратить внимание на следующие важные части этого определения:

1) объектно-ориентированное программирование использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы;

2) каждый объект является экземпляром определенного класса;

3) классы образуют иерархии.

Программа считается объектно-ориентированной, только если выполнены все три указанных требования. В частности, программирование, не использующее наследование, называется не объектно-ориентированным, а программированием с помощью абстрактных типов данных.

[Лука Карделли](https://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%B0%D1%80%D0%B4%D0%B5%D0%BB%D0%BB%D0%B8,_%D0%9B%D1%83%D0%BA%D0%B0&action=edit&redlink=1) и Мартин Абади построили теоретическое обоснование ООП и классификацию на основе этого обоснования. Они отмечают, что выделенные ими понятия и категории вместе встречаются далеко не во всех ОО-языках, большинство языков поддерживают лишь подмножества теории, а порой и своеобразные отклонения от неё.

Ключевые понятия:

-[квантификация переменных типа](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B0#.D0.A1.D0.B2.D1.8F.D0.B7.D1.8B.D0.B2.D0.B0.D0.BD.D0.B8.D0.B5_.D0.B8_.D0.BA.D0.B2.D0.B0.D0.BD.D1.82.D0.B8.D1.84.D0.B8.D0.BA.D0.B0.D1.86.D0.B8.D1.8F_.D0.BF.D0.B5.D1.80.D0.B5.D0.BC.D0.B5.D0.BD.D0.BD.D1.8B.D1.85_.D1.82.D0.B8.D0.BF.D0.B0) ([универсальная](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC), [экзистенциальная](https://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D0%B7%D0%B8%D1%81%D1%82%D0%B5%D0%BD%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF), [ограниченная](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%B3%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BA%D0%B2%D0%B0%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F&action=edit&redlink=1));

-[подтипизация](https://ru.wikipedia.org/w/index.php?title=%D0%92%D1%8B%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D0%BE%D0%B2_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85&action=edit&redlink=1" \o "Выделение подтипов данных (страница отсутствует)) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) subtyping – отношения «супертип-подтип»);

-включение ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) subsumption – см. [принцип подстановки Барбары Лисков](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B8_%D0%91%D0%B0%D1%80%D0%B1%D0%B0%D1%80%D1%8B_%D0%9B%D0%B8%D1%81%D0%BA%D0%BE%D0%B2)) – частный случай [подтипизации](https://ru.wikipedia.org/w/index.php?title=%D0%92%D1%8B%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D0%BE%D0%B2_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85&action=edit&redlink=1) – см. [подтипизация на записях](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29#.D0.9F.D0.BE.D0.B4.D1.82.D0.B8.D0.BF.D0.B8.D0.B7.D0.B0.D1.86.D0.B8.D1.8F_.D0.BD.D0.B0_.D0.B7.D0.B0.D0.BF.D0.B8.D1.81.D1.8F.D1.85);

-[объект](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29);

-объектный тип (фактически, специальная форма «[записи](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BF%D0%B8%D1%81%D1%8C_%28%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85%29)», в число полей которой входят [функции первого класса](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0));

-[класс](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29).

Класс описывает абстрактное поведение. Объектные типы строятся на основе класса посредством добавления различных приватных полей и методов. Объект (то есть значение объектного типа, традиционно называемое «экземпляром класса») порождается [конструктором](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80_%28%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE) на основе начальных параметров.

Традиционно перечисляемые основные принципы ООП не аппелируют к данному теоретическому обоснованию, а являются устоявшимися в сообществе догмами (что приводит к сильному разбросу вариантов изложения в разных источниках). По большей части они относятся к языкам – потомкам [Алгола](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D0%BB) и [Симулы](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D1%83%D0%BB%D0%B0); в меньшей степени – к потомкам [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk) (в частности, часто упоминаемый принцип [сокрытия](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D1%80%D1%8B%D1%82%D0%B8%D0%B5_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) в потомках [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk) семантически недоступен и идеологически считается несущественным). В значительно большей степени теоретически обоснованные концепции ООП поддерживают ОО-языки, развившиеся на поприще [функционального программирования](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5): [OCaml](https://ru.wikipedia.org/wiki/OCaml), диалекты [Haskell](https://ru.wikipedia.org/wiki/Haskell) (O’Haskell, Mondrian), [successor ML](https://ru.wikipedia.org/w/index.php?title=Successor_ML&action=edit&redlink=1). Более того, основные идеи объектного моделирования в этом случае не требуют прямой поддержки со стороны языка, а могут быть сравнительно легко эмулированы.

Наиболее заметные отличия в проявлении [показателей качества](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%87%D0%B5%D1%81%D1%82%D0%B2%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) между языками разных видов:

В [мейнстримных](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B9%D0%BD%D1%81%D1%82%D1%80%D0%B8%D0%BC) языках декларируемые принципы нацелены на повышение изначально низкого для [императивного программирования](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) коэффициента [повторного использования кода](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0). В [полиморфно типизированных](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2) применение концепций ООП, напротив, означает очевидное его снижение из-за перехода от [параметрического полиморфизма](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC) к [ad hoc полиморфизму](https://ru.wikipedia.org/wiki/Ad_hoc_%D0%BF%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC). В [динамически типизированных](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) языках ([Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk%22%20%5Co%20%22Smalltalk), [Python](https://ru.wikipedia.org/wiki/Python), [Ruby](https://ru.wikipedia.org/wiki/Ruby)) эти принципы используются для логической организации программы, и их влияние на коэффициент [повторного использования](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0) трудно спрогнозировать – он сильно зависит от дисциплины программиста. Например, в [CLOS](https://ru.wikipedia.org/wiki/CLOS) [мультиметоды](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BC%D0%B5%D1%82%D0%BE%D0%B4) одновременно являются [функциями первого класса](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0), что позволяет рассматривать их одновременно и как [связанно квантифицированные](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%B2%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BA%D0%B2%D0%B0%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F&action=edit&redlink=1), и как [обобщённые](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BE%D0%B1%D1%89%D1%91%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) ([истинно полиморфные](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC)).

Традиционные ОО-языки используют [номинативную типизацию](https://ru.wikipedia.org/w/index.php?title=%D0%9D%D0%BE%D0%BC%D0%B8%D0%BD%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F&action=edit&redlink=1), то есть допустимость соиспользования объектов разных классов только при условии явного указания родственных отношений между классами. Для [полиморфно типизированных](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2) языков характерна [структурная типизация](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F&action=edit&redlink=1), то есть согласование классов между собой тем же механизмом, что и согласование числа 5 с типом [int](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BB%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF). [Динамически типизированные](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) языки также занимают здесь промежуточную позицию.

Обобщённое обоснование [динамической диспетчеризации](https://ru.wikipedia.org/w/index.php?title=%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%B4%D0%B8%D1%81%D0%BF%D0%B5%D1%82%D1%87%D0%B5%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F&action=edit&redlink=1) (включая [множественную](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D0%B4%D0%B8%D1%81%D0%BF%D0%B5%D1%82%D1%87%D0%B5%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F)) в середине [1990-х годов](https://ru.wikipedia.org/wiki/1990-%D0%B5_%D0%B3%D0%BE%D0%B4%D1%8B) построил Джузеппе Кастанья

ООП возникло в результате развития идеологии [процедурного программирования](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), где данные и подпрограммы (процедуры, функции) их обработки формально не связаны. Для дальнейшего развития объектно-ориентированного программирования часто большое значение имеют понятия события (так называемое [событийно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D0%B9%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%25D)) и компонента ([компонентное программирование](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), КОП).

Взаимодействие объектов происходит посредством [сообщений](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%25D#.D0.A1.D0.BE.D0.BE.D0.B1.D1.89.D0.B5.D0.BD.D0.B8.D0.B5). Результатом дальнейшего развития ООП, по-видимому, будет [агентно-ориентированое программирование](https://ru.wikipedia.org/wiki/%D0%90%D0%B3%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B4%D1%85%D0%BE%D0%B4), где агенты – независимые части кода на уровне выполнения. Взаимодействие агентов происходит посредством изменения среды, в которой они находятся.

Языковые конструкции, конструктивно не относящиеся непосредственно к объектам, но сопутствующие им для их безопасной ([исключительные ситуации](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B9), проверки) и эффективной работы, инкапсулируются от них в аспекты (в [аспектно-ориентированном программировании](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D0%BF%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%25D)). [Субъектно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%25D) расширяет понятие объекта посредством обеспечения более унифицированного и независимого взаимодействия объектов. Может являться переходной стадией между ООП и агентным программированием в части самостоятельного их взаимодействия.

Первым языком программирования, в котором были предложены основные понятия, впоследствии сложившиеся в парадигму, была [Симула](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D1%83%D0%BB%D0%B0), но термин «объектная ориентированность» не использовался в контексте использования этого языка. В момент его появления в [1967 году](https://ru.wikipedia.org/wiki/1967_%D0%B3%D0%BE%D0%B4) в нём были предложены революционные идеи: объекты, классы, [виртуальные методы](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4) и др., однако это всё не было воспринято современниками как нечто грандиозное. Фактически, Симула была «Алголом с классами», упрощающим выражение в [процедурном программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) многих сложных концепций. Понятие класса в Симуле может быть полностью определено через композицию конструкций Алгола (то есть класс в Симуле – это нечто сложное, описываемое посредством примитивов).

Взгляд на программирование «под новым углом» (отличным от процедурного) предложили [Алан Кэй](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D0%B9%2C_%D0%90%D0%BB%D0%B0%D0%BD_%D0%9A%D1%83%D1%80%D1%82%D0%B8%D1%81) и [Дэн Ингаллс](https://ru.wikipedia.org/w/index.php?title=%D0%98%D0%BD%D0%B3%D0%B0%D0%BB%D0%BB%D1%81,_%D0%94%D1%8D%D0%BD&action=edit&redlink=1) в языке [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk). Здесь понятие класса стало основообразующей идеей для всех остальных конструкций языка (то есть класс в Смолтоке является примитивом, посредством которого описаны более сложные конструкции). Именно он стал первым широко распространённым [объектно-ориентированным языком программирования](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%25).

В настоящее время количество прикладных языков программирования ([список языков](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%25#.D0.A1.D0.BF.D0.B8.D1.81.D0.BE.D0.BA_.D1.8F.D0.B7.D1.8B.D0.BA.D0.BE.D0.B2)), реализующих объектно-ориентированную парадигму, является наибольшим по отношению к другим парадигмам. Наиболее распространённые в промышленности языки (С++, Delphi, C#, Java и др.) воплощают объектную модель Симулы. Примерами языков, опирающихся на модель Смолтока, являются Python, Ruby.

В центре ООП находится понятие объекта. Объект – это сущность, которой можно посылать сообщения и которая может на них реагировать, используя свои данные. Объект – это экземпляр класса. Данные объекта скрыты от остальной программы.

[Сокрытие](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D1%80%D1%8B%D1%82%D0%B8%D0%B5_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) данных называется [инкапсуляцией](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29).

Наличие инкапсуляции достаточно для объектности языка программирования, но ещё не означает его объектной ориентированности – для этого требуется наличие [наследования](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29).

Но даже наличие инкапсуляции и наследования не делает язык программирования в полной мере объектным с точки зрения ООП. Основные преимущества ООП проявляются только в том случае, когда в языке программирования реализован [полиморфизм подтипов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D0%BE%D0%B2) – возможность единообразно обрабатывать объекты с различной реализацией при условии наличия общего интерфейса.

ООП имеет уже более чем сорокалетнюю историю, но, несмотря на это, до сих пор не существует чёткого общепринятого определения данной технологии. Основные принципы, заложенные в первые объектные языки и системы, подверглись существенному изменению (или искажению) и дополнению при многочисленных реализациях последующего времени. Кроме того, примерно с середины 1980-х годов термин «объектно-ориентированный» стал [модным](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B0), в результате с ним произошло то же самое, что несколько раньше с термином «структурный» (ставшим модным после распространения технологии [структурного программирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) – его стали искусственно «прикреплять» к любым новым разработкам, чтобы обеспечить им привлекательность. [Бьёрн Страуструп](https://ru.wikipedia.org/wiki/%D0%91%D1%8C%D1%91%D1%80%D0%BD_%D0%A1%D1%82%D1%80%D0%B0%D1%83%D1%81%D1%82%D1%80%D1%83%D0%BF) в 1988 году писал, что обоснование «объектной ориентированности» чего-либо, в большинстве случаев, сводится к ложному [силлогизму](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BB%D0%BB%D0%BE%D0%B3%D0%B8%D0%B7%D0%BC): «X – это хорошо. Объектная ориентированность – это хорошо.Следовательно, X является объектно-ориентированным».

Тимоти Бадд пишет: *Роджер Кинг аргументированно настаивал, что его кот является объектно-ориентированным. Кроме прочих своих достоинств, кот демонстрирует характерное поведение, реагирует на сообщения, наделён унаследованными реакциями и управляет своим, вполне независимым, внутренним состоянием.*

По мнению [Алана Кея](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B0%D0%BD_%D0%9A%D1%8D%D0%B9), создателя языка [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk), которого считают одним из «отцов-основателей» ООП, объектно-ориентированный подход заключается в следующем наборе основных принципов (цитируется по вышеупомянутой книге Т. Бадда).

Всё является объектом.

Вычисления осуществляются путём взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщение – это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.

Каждый объект имеет независимую память, которая состоит из других объектов.

Каждый объект является представителем класса, который выражает общие свойства объектов (таких, как целые числа или списки).

В классе задаётся поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.

Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определённого класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

Таким образом, программа представляет собой набор объектов, имеющих состояние и поведение. Объекты взаимодействуют посредством сообщений. Естественным образом выстраивается иерархия объектов: программа в целом – это объект, для выполнения своих функций она обращается к входящим в неё объектам, которые, в свою очередь, выполняют запрошенное путём обращения к другим объектам программы. Естественно, чтобы избежать бесконечной рекурсии в обращениях, на каком-то этапе объект трансформирует обращённое к нему сообщение в сообщения к стандартным системным объектам, предоставляемым языком и средой программирования.

Устойчивость и управляемость системы обеспечивается за счёт чёткого разделения ответственности объектов (за каждое действие отвечает определённый объект), однозначного определения интерфейсов межобъектного взаимодействия и полной изолированности внутренней структуры объекта от внешней среды (инкапсуляции).

Определить ООП можно и многими другими способами.

-Концепции.

Появление в ООП отдельного понятия класса закономерно вытекает из желания иметь множество объектов со сходным поведением. Класс в ООП – это в чистом виде [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), создаваемый программистом. С этой точки зрения объекты являются значениями данного абстрактного типа, а определение класса задаёт внутреннюю структуру значений и набор операций, которые над этими значениями могут быть выполнены. Желательность иерархии классов (а значит, наследования) вытекает из требований к повторному использованию кода – если несколько классов имеют сходное поведение, нет смысла дублировать их описание, лучше выделить общую часть в общий родительский класс, а в описании самих этих классов оставить только различающиеся элементы.

Необходимость совместного использования объектов разных классов, способных обрабатывать однотипные сообщения, требует поддержки [полиморфизма](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29) – возможности записывать разные объекты в переменные одного и того же типа. В таких условиях объект, отправляя сообщение, может не знать в точности, к какому классу относится адресат, и одни и те же сообщения, отправленные переменным одного типа, содержащим объекты разных классов, вызовут различную реакцию.

Отдельного пояснения требует понятие обмена [сообщениями](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D0%B5). Первоначально (например, в том же [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk)) взаимодействие объектов представлялось как «настоящий» обмен сообщениями, то есть пересылка от одного объекта другому специального объекта-сообщения. Такая модель является чрезвычайно общей. Она прекрасно подходит, например, для описания параллельных вычислений с помощью активных объектов, каждый из которых имеет собственный поток исполнения и работает одновременно с прочими. Такие объекты могут вести себя как отдельные, абсолютно автономные вычислительные единицы. Посылка сообщений естественным образом решает вопрос обработки сообщений объектами, присвоенными полиморфным переменным – независимо от того, как объявляется переменная, сообщение обрабатывает код класса, к которому относится присвоенный переменной объект. Данный подход реализован в языках программирования [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk), [Ruby](https://ru.wikipedia.org/wiki/Ruby),[Objective-C](https://ru.wikipedia.org/wiki/Objective-C), [Python](https://ru.wikipedia.org/wiki/Python).

Однако общность механизма обмена сообщениями имеет и другую сторону – «полноценная» передача сообщений требует дополнительных накладных расходов, что не всегда приемлемо. Поэтому во многих современных объектно-ориентированных языках программирования используется концепция «отправка сообщения как вызов метода» – объекты имеют доступные извне методы, вызовами которых и обеспечивается взаимодействие объектов. Данный подход реализован в огромном количестве языков программирования, в том числе [C++](https://ru.wikipedia.org/wiki/C%2B%2B), [Object Pascal](https://ru.wikipedia.org/wiki/Object_Pascal), [Java](https://ru.wikipedia.org/wiki/Java), [Oberon-2](https://ru.wikipedia.org/wiki/Oberon-2). Однако, это приводит к тому, что сообщения уже не являются самостоятельными объектами, и, как следствие, не имеют атрибутов, что сужает возможности программирования. Некоторые языки используют гибридное представление, демонстрируя преимущества одновременно обоих подходов – например, [CLOS](https://ru.wikipedia.org/wiki/CLOS), [Python](https://ru.wikipedia.org/wiki/Python).

Концепция виртуальных методов, поддерживаемая этими и другими современными языками, появилась как средство обеспечить выполнение нужных методов при использовании полиморфных переменных, то есть, по сути, как попытка расширить возможности вызова методов для реализации части функциональности, обеспечиваемой механизмом обработки сообщений.

Как уже говорилось выше, в современных объектно-ориентированных языках программирования каждый объект является значением, относящимся к определённому [классу](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29). Класс представляет собой объявленный программистом составной [тип данных](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), имеющий в составе:

Параметры объекта (конечно, не все, а только необходимые в программе), задающие его состояние (свойства объекта предметной области). Иногда поля данных объекта называют свойствами объекта, из-за чего возможна путаница. Физически поля представляют собой значения (переменные, константы), объявленные как принадлежащие классу.

Процедуры и функции, связанные с классом. Они определяют действия, которые можно выполнять над объектом такого типа, и которые сам объект может выполнять.

Классы могут наследоваться друг от друга. Класс-потомок получает все поля и методы класса-родителя, но может дополнять их собственными либо переопределять уже имеющиеся. Большинство языков программирования поддерживает только единичное наследование (класс может иметь только один класс-родитель), лишь в некоторых допускается множественное наследование – порождение класса от двух или более классов-родителей. Множественное наследование создаёт целый ряд проблем, как логических, так и чисто реализационных, поэтому в полном объёме его поддержка не распространена. Вместо этого в 1990-е годы появилось и стало активно вводиться в объектно-ориентированные языки понятие [интерфейса](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%28%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80). Интерфейс – это класс без полей и без реализации, включающий только заголовки методов. Если некий класс наследует (или, как говорят, реализует) интерфейс, он должен реализовать все входящие в него методы. Использование интерфейсов предоставляет относительно дешёвую альтернативу множественному наследованию.

Взаимодействие объектов в абсолютном большинстве случаев обеспечивается вызовом ими методов друг друга.

Инкапсуляция обеспечивается следующими средствами:

-Контроль доступа

Поскольку методы класса могут быть как чисто внутренними, обеспечивающими логику функционирования объекта, так и внешними, с помощью которых взаимодействуют объекты, необходимо обеспечить скрытость первых при доступности извне вторых. Для этого в языки вводятся специальные синтаксические конструкции, явно задающие область видимости каждого члена класса. Традиционно это модификаторы public, protected и private, обозначающие, соответственно, открытые члены класса, члены класса, доступные только из классов-потомков, и скрытые, доступные только внутри класса. Конкретная номенклатура модификаторов и их точный смысл различаются в разных языках.

-Методы доступа

Поля класса в общем случае не должны быть доступны извне, поскольку такой доступ позволил бы произвольным образом менять внутреннее состояние объектов. Поэтому поля обычно объявляются скрытыми (либо язык в принципе не позволяет обращаться к полям класса извне), а для доступа к находящимся в полях данным используются специальные методы, называемые методами доступа. Такие методы либо возвращают значение того или иного поля, либо производят запись в это поле нового значения. При записи метод доступа может проконтролировать допустимость записываемого значения и, при необходимости, произвести другие манипуляции с данными объекта, чтобы они остались корректными (внутренне согласованными). Методы доступа называют ещё аксессорами (от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) access – доступ), а по отдельности – [геттерами](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D1%82%D1%82%D0%B5%D1%80_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) get – чтение) и [сеттерами](https://ru.wikipedia.org/wiki/Setter) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) set – запись).

[Свойства объекта](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29)

Псевдополя, доступные для чтения и/или записи. Свойства внешне выглядят как поля и используются аналогично доступным полям (с некоторыми исключениями), однако фактически при обращении к ним происходит вызов методов доступа. Таким образом, свойства можно рассматривать как «умные» поля данных, сопровождающие доступ к внутренним данным объекта какими-либо дополнительными действиями (например, когда изменение координаты объекта сопровождается его перерисовкой на новом месте). Свойства, по сути, не более чем [синтаксический сахар](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D1%81%D0%B0%D1%85%D0%B0%D1%80), поскольку никаких новых возможностей они не добавляют, а лишь скрывают вызов методов доступа. Конкретная языковая реализация свойств может быть разной. Например, в [C#](https://ru.wikipedia.org/wiki/C_Sharp) объявление свойства непосредственно содержит код методов доступа, который вызывается только при работе со свойствами, то есть не требует отдельных методов доступа, доступных для непосредственного вызова. В Delphi объявление свойства содержит лишь имена методов доступа, которые должны вызываться при обращении к полю. Сами методы доступа представляют собой обычные методы с некоторыми дополнительными требованиями к сигнатуре.

Полиморфизм реализуется путём введения в язык правил, согласно которым переменной типа «класс» может быть присвоен объект любого класса-потомка её класса.

ООП ориентировано на разработку крупных программных комплексов, разрабатываемых командой программистов (возможно, достаточно большой). Проектирование системы в целом, создание отдельных компонентов и их объединение в конечный продукт при этом часто выполняется разными людьми, и нет ни одного специалиста, который знал бы о проекте всё.

Объектно-ориентированное проектирование ориентируется на описание структуры проектируемой системы (приоритетно по отношению к описанию её поведения, в отличие от [функционального программирования](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), то есть, фактически, в ответе на два основных вопроса:

Из каких частей состоит система;

В чём состоит ответственность каждой из ее частей.

Выделение частей производится таким образом, чтобы каждая имела минимальный по объёму и точно определённый набор выполняемых функций (обязанностей), и при этом взаимодействовала с другими частями как можно меньше.

Дальнейшее уточнение приводит к выделению более мелких фрагментов описания. По мере детализации описания и определения ответственности выявляются данные, которые необходимо хранить, наличие близких по поведению агентов, которые становятся кандидатами на реализацию в виде классов с общими предками. После выделения компонентов и определения интерфейсов между ними реализация каждого компонента может проводиться практически независимо от остальных (разумеется, при соблюдении соответствующей технологической дисциплины).

Большое значение имеет правильное построение иерархии классов. Одна из известных проблем больших систем, построенных по ООП-технологии – так называемая проблема хрупкости базового класса. Она состоит в том, что на поздних этапах разработки, когда иерархия классов построена и на её основе разработано большое количество кода, оказывается трудно или даже невозможно внести какие-либо изменения в код базовых классов иерархии (от которых порождены все или многие работающие в системе классы). Даже если вносимые изменения не затронут интерфейс базового класса, изменение его поведения может непредсказуемым образом отразиться на классах-потомках. В случае крупной системы разработчик базового класса просто не в состоянии предугадать последствия изменений, он даже не знает о том, как именно базовый класс используется и от каких особенностей его поведения зависит корректность работы классов-потомков.

**Различные ООП-методологии**

**Компонентное программирование** – следующий этап развития ООП; прототип- и класс-ориентированное программирование – разные подходы к созданию программы, которые могут комбинироваться, имеющие свои преимущества и недостатки.

[Компонентно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%25D) – это своеобразная «надстройка» над ООП, набор правил и ограничений, направленных на построение крупных развивающихся программных систем с большим временем жизни. Программная система в этой методологии представляет собой набор компонентов с хорошо определёнными интерфейсами. Изменения в существующую систему вносятся путём создания новых компонентов в дополнение или в качестве замены ранее существующих. При создании новых компонентов на основе ранее созданных запрещено использование наследования реализации – новый компонент может наследовать лишь интерфейсы базового. Таким образом компонентное программирование обходит проблему хрупкости базового класса.

[**Прототипное программирование**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), сохранив часть черт ООП, отказалось от базовых понятий – класса и наследования.

Прототип – это объект-образец, по образу и подобию которого создаются другие объекты. Объекты-копии могут сохранять связь с родительским объектом, автоматически наследуя изменения в прототипе; эта особенность определяется в рамках конкретного [языка](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F).

Вместо механизма описания классов и порождения экземпляров язык предоставляет механизм создания объекта (путём задания набора полей и методов, которые объект должен иметь) и механизм клонирования объектов.

Каждый вновь созданный объект является «экземпляром без класса». Каждый объект может стать прототипом – быть использован для создания нового объекта с помощью операции клонирования. После клонирования новый объект может быть изменён, в частности, дополнен новыми полями и методами.

Клонированный объект либо становится полной копией прототипа, хранящей все значения его полей и дублирующей его методы, либо сохраняет ссылку на прототип, не включая в себя клонированных полей и методов до тех пор, пока они не будут изменены. В последнем случае среда исполнения обеспечивает механизм делегирования – если при обращении к объекту он сам не содержит нужного метода или поля данных, вызов передаётся прототипу, от него, при необходимости – дальше по цепочке.

**Класс-ориентированное** программирование – это программирование, сфокусированное на данных, причем данные и поведение неразрывно связаны между собой. Вместе данные и поведение представляют собой класс. Соответственно в языках, основанных на понятии «класс», все объекты разделены на два основных типа – классы и экземпляры. Класс определяет структуру и функциональность (поведение), одинаковую для всех экземпляров данного класса. Экземпляр является носителем данных – то есть обладает состоянием, меняющимся в соответствии с поведением, заданным классом. В класс-ориентированных языках новый экземпляр создаётся через вызов конструктора класса (возможно, с набором параметров). Получившийся экземпляр имеет структуру и поведение, жёстко заданные его классом.

[Гради Буч](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D0%B4%D0%B8_%D0%91%D1%83%D1%87) указывает на следующие причины, приводящие к снижению производительности программ из-за использования объектно-ориентированных средств:

Динамическое связывание методов

Обеспечение полиморфного поведения объектов приводит к необходимости связывать методы, вызываемые программой (то есть определять, какой конкретно метод будет вызываться) не на этапе компиляции, а в процессе исполнения программы, на что тратится дополнительное время. При этом реально динамическое связывание требуется не более чем для 20 % вызовов, но некоторые ООП-языки используют его постоянно.

Значительная глубина абстракции

ООП-разработка часто приводит к созданию «многослойных» приложений, где выполнение объектом требуемого действия сводится к множеству обращений к объектам более низкого уровня. В таком приложении происходит очень много вызовов методов и возвратов из методов, что, естественно, сказывается на производительности.

Наследование «размывает» код

Код, относящийся к «конечным» классам иерархии наследования, которые обычно и используются программой непосредственно, находится не только в самих этих классах, но и в их классах-предках. Относящиеся к одному классу методы фактически описываются в разных классах. Это приводит к двум неприятным моментам:

Снижается скорость трансляции, так как компоновщику приходится подгружать описания всех классов иерархии.

Снижается производительность программы в системе со страничной памятью – поскольку методы одного класса физически находятся в разных местах кода, далеко друг от друга, при работе фрагментов программы, активно обращающихся к унаследованным методам, система вынуждена производить частые переключения страниц.

Инкапсуляция снижает скорость доступа к данным

Запрет на прямой доступ к полям класса извне приводит к необходимости создания и использования методов доступа. И написание, и компиляция, и исполнение методов доступа сопряжены с дополнительными расходами.

Динамическое создание и уничтожение объектов

Динамически создаваемые объекты, как правило, размещаются в [куче](https://ru.wikipedia.org/wiki/%D0%9A%D1%83%D1%87%D0%B0_%28%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%29), что менее эффективно, чем размещение их на стеке и, тем более, статическое выделение памяти под них на этапе компиляции.

Несмотря на отмеченные недостатки, Буч утверждает, что выгоды от использования ООП более весомы. Кроме того, повышение производительности за счёт лучшей организации ООП-кода, по его словам, в некоторых случаях компенсирует дополнительные накладные расходы на организацию функционирования программы. Можно также заметить, что многие эффекты снижения производительности могут сглаживаться или даже полностью устраняться за счёт качественной оптимизации кода компилятором. Например, упомянутое выше снижение скорости доступа к полям класса из-за использования методов доступа устраняется, если компилятор вместо вызова метода доступа использует инлайн-подстановку (современные компиляторы делают это вполне уверенно).

Критика ООП

Несмотря на отдельные критические замечания в адрес ООП, в настоящее время именно эта парадигма используется в подавляющем большинстве промышленных проектов. Однако, нельзя считать, что ООП является наилучшей из методик программирования во всех случаях.

Критические высказывания в адрес ООП:

Было показано отсутствие значимой разницы в продуктивности разработки программного обеспечения между ООП и процедурным подходом.

[Кристофер Дэйт](https://ru.wikipedia.org/wiki/%D0%94%D1%8D%D0%B9%D1%82%2C_%D0%9A%D1%80%D0%B8%D1%81%D1%82%D0%BE%D1%84%D0%B5%D1%80) указывает на невозможность сравнения ООП и других технологий во многом из-за отсутствия строгого и общепризнанного определения ООП.

[Александр Степанов](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BF%D0%B0%D0%BD%D0%BE%D0%B2%2C_%D0%90%D0%BB%D0%B5%D0%BA%D1%81%D0%B0%D0%BD%D0%B4%D1%80_%D0%90%D0%BB%D0%B5%D0%BA%D1%81%D0%B0%D0%BD%D0%B4%D1%80%D0%BE%D0%B2%D0%B8%D1%87_%28%D1%83%D1%87%D1%91%D0%BD%D1%8B%D0%B9%29) в одном из своих интервью указывал, что ООП «методологически неправильно» и что «…ООП практически такая же мистификация, как и искусственный интеллект…».

[Фредерик Брукс](https://ru.wikipedia.org/wiki/%D0%91%D1%80%D1%83%D0%BA%D1%81%2C_%D0%A4%D1%80%D0%B5%D0%B4%D0%B5%D1%80%D0%B8%D0%BA) указывает, что наиболее сложной частью создания программного обеспечения является «…спецификация, дизайн и тестирование концептуальных конструкций, а отнюдь не работа по выражению этих концептуальных конструкций…». ООП (наряду с такими технологиями как [искусственный интеллект](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BA%D1%83%D1%81%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D0%BB%D0%BB%D0%B5%D0%BA%D1%82), [верификация программ](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B5%D1%80%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC&action=edit&redlink=1), [автоматическое программирование](https://ru.wikipedia.org/w/index.php?title=%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1), [графическое программирование](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), [экспертные системы](https://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D1%81%D0%BF%D0%B5%D1%80%D1%82%D0%BD%D1%8B%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) и др.), по его мнению, не является «серебряной пулей», которая могла бы на порядок величины снизить сложность разработки программных систем. Согласно Бруксу, «…ООП позволяет сократить только привнесённую сложность в выражение дизайна. Дизайн остаётся сложным по своей природе…».

[Эдсгер Дейкстра](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B0%2C_%D0%AD%D0%B4%D1%81%D0%B3%D0%B5%D1%80_%D0%92%D0%B8%D0%B1%D0%B5) указывал: «…то, о чём общество в большинстве случаев просит – это [змеиное масло](https://ru.wikipedia.org/w/index.php?title=%D0%97%D0%BC%D0%B5%D0%B8%D0%BD%D0%BE%D0%B5_%D0%BC%D0%B0%D1%81%D0%BB%D0%BE&action=edit&redlink=1). Естественно, „змеиное масло“ имеет очень впечатляющие названия, иначе будет очень трудно что-то продать: „Структурный анализ и Дизайн“, „Программная инженерия“, „Модели зрелости“, „Управляющие информационные системы“ (Management Information Systems), „Интегрированные среды поддержки проектов“, „Объектная ориентированность“, „[Реинжиниринг бизнес-процессов](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B8%D0%BD%D0%B6%D0%B8%D0%BD%D0%B8%D1%80%D0%B8%D0%BD%D0%B3_%D0%B1%D0%B8%D0%B7%D0%BD%D0%B5%D1%81-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D0%B2%22%20%5Co%20%22%D0%A0%D0%B5%D0%B8%D0%BD%D0%B6%D0%B8%D0%BD%D0%B8%D1%80%D0%B8%D0%BD%D0%B3%20%D0%B1%D0%B8%D0%B7%D0%BD%D0%B5%D1%81-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D0%B2)“…».

[Никлаус Вирт](https://ru.wikipedia.org/wiki/%D0%9D%D0%B8%D0%BA%D0%BB%D0%B0%D1%83%D1%81_%D0%92%D0%B8%D1%80%D1%82) считает, что ООП – не более чем тривиальная надстройка над структурным программированием, и преувеличение её значимости, выражающееся, в том числе, во включении в языки программирования всё новых модных «объектно-ориентированных» средств, вредит качеству разрабатываемого программного обеспечения.

[Патрик Киллелиа](https://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%B8%D0%BB%D0%BB%D0%B5%D0%BB%D0%B8%D0%B0,_%D0%9F%D0%B0%D1%82%D1%80%D0%B8%D0%BA&action=edit&redlink=1) в своей книге «Тюнинг веб-сервера» писал: «…ООП предоставляет вам множество способов замедлить работу ваших программ…».

Известная обзорная статья проблем современного ООП-программирования перечисляет некоторые типичные проблемы ООП

В программистском фольклоре получила широкое распространение критика объектно-ориентированного подхода в сравнении с [функциональным подходом](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) с использованием метафоры «Королевства Существительных» из эссе Стива Йегги.

Если попытаться классифицировать критические высказывания в адрес ООП, можно выделить несколько аспектов критики данного подхода к программированию.

Критика рекламы ООП

Критикуется явно высказываемое или подразумеваемое в работах некоторых пропагандистов ООП, а также в рекламных материалах «объектно-ориентированных» средств разработки представление об объектном программировании как о некоем всемогущем подходе, который магическим образом устраняет сложность программирования. Как замечали многие, в том числе упомянутые выше Брукс и Дейкстра, «серебряной пули не существует» – независимо от того, какой парадигмы программирования придерживается разработчик, создание нетривиальной сложной программной системы всегда сопряжено со значительными затратами интеллектуальных ресурсов и времени. Из наиболее квалифицированных специалистов в области ООП никто, как правило, не отрицает справедливость критики этого типа.

Оспаривание эффективности разработки методами ООП

Критики оспаривают тезис о том, что разработка объектно-ориентированных программ требует меньше ресурсов или приводит к созданию более качественного ПО. Проводится сравнение затрат на разработку разными методами, на основании которого делается вывод об отсутствии у ООП преимуществ в данном направлении. Учитывая крайнюю сложность объективного сравнения различных разработок, подобные сопоставления, как минимум, спорны. С другой стороны получается что ровно так же спорны и утверждения об эффективности ООП.

Производительность объектно-ориентированных программ

Указывается на то, что целый ряд «врождённых особенностей» ООП-технологии делает построенные на её основе программы технически менее эффективными, по сравнению с аналогичными необъектными программами. Не отрицая действительно имеющихся дополнительных накладных расходов на организацию работы ООП-программ (см. раздел «Производительность» выше), нужно, однако, отметить, что значение снижения производительности часто преувеличивается критиками. В современных условиях, когда технические возможности компьютеров чрезвычайно велики и постоянно растут, для большинства прикладных программ техническая эффективность оказывается менее существенна, чем функциональность, скорость разработки и сопровождаемость. Лишь для некоторого, очень ограниченного класса программ (ПО встроенных систем, драйверы устройств, низкоуровневая часть системного ПО, научное ПО) производительность остаётся критическим фактором.

Критика отдельных технологических решений в ООП-языках и библиотеках

Эта критика многочисленна, но затрагивает она не ООП как таковое, а приемлемость и применимость в конкретных случаях тех или иных реализаций её механизмов. Одним из излюбленных объектов критики является язык C++, входящий в число наиболее распространённых промышленных ООП-языков.

**Объектно-ориентированные языки**

Многие современные языки специально созданы для облегчения объектно-ориентированного программирования. Однако следует отметить, что можно применять техники ООП и для не-объектно-ориентированного языка и наоборот, применение объектно-ориентированного языка вовсе не означает, что код автоматически становится объектно-ориентированным.

Как правило, объектно-ориентированный язык (ООЯ) содержит следующий набор элементов:

Объявление классов с полями (данными – членами класса) и методами (функциями – членами класса).

Механизм расширения класса (наследования) – порождение нового класса от существующего с автоматическим включением всех особенностей реализации класса-предка в состав класса-потомка. Большинство ООЯ поддерживают только единичное наследование.

Полиморфные переменные и параметры функций (методов), позволяющие присваивать одной и той же переменной экземпляры различных классов.

Полиморфное поведение экземпляров классов за счёт использования виртуальных методов. В некоторых ООЯ все методы классов являются виртуальными.

Некоторые языки добавляют к указанному минимальному набору те или иные дополнительные средства. В их числе:

Конструкторы, деструкторы, финализаторы;

Свойства (аксессоры);

Индексаторы;

Средства управления видимостью компонентов классов (интерфейсы или модификаторы доступа, такие как public, private, protected, feature и др.).

Одни языки отвечают принципам ООП в полной мере – в них все основные элементы являются объектами, имеющими состояние и связанные методы. Примеры подобных языков – [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk), [Eiffel](https://ru.wikipedia.org/wiki/Eiffel). Существуют гибридные языки, совмещающие объектную подсистему в целостном виде с подсистемами других парадигм как «два и более языка в одном», позволяющие совмещать в одной программе объектные модели с иными, и размывающие грань между объектно-ориентированной и другими парадигмами за счёт нестандартных возможностей, балансирующих между ООП и другими парадигмами (таких как [множественная диспетчеризация](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D0%B4%D0%B8%D1%81%D0%BF%D0%B5%D1%82%D1%87%D0%B5%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F), параметрические классы, возможность манипулировать методами классов как самостоятельными объектами, и др.). Примеры таких языков: [CLOS](https://ru.wikipedia.org/wiki/CLOS), [Dylan](https://ru.wikipedia.org/wiki/Dylan_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29), [OCaml](https://ru.wikipedia.org/wiki/OCaml), [Python](https://ru.wikipedia.org/wiki/Python), [Ruby](https://ru.wikipedia.org/wiki/Ruby), [Objective-C](https://ru.wikipedia.org/wiki/Objective-C). Однако, наиболее распространены языки, включающие средства эмуляции объектной модели поверх более традиционной императивной семантики. [Алан Кэй](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D0%B9%2C_%D0%90%D0%BB%D0%B0%D0%BD_%D0%9A%D1%83%D1%80%D1%82%D0%B8%D1%81)назвал такие языки «склеиванием возможностей» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) agglutination of features) в противовес «чистоте стиля» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) crystalization of style) языков, воплощающих некую парадигму непосредственно. Примеры таких языков – [Симула](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D1%83%D0%BB%D0%B0), [C++](https://ru.wikipedia.org/wiki/C%2B%2B), [Visual Basic](https://ru.wikipedia.org/wiki/Visual_Basic),[Delphi](https://ru.wikipedia.org/wiki/Delphi_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29), [Модула](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D0%B0), [Модула-2](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D0%B0-2), [Java](https://ru.wikipedia.org/wiki/Java), [C#](https://ru.wikipedia.org/wiki/C_Sharp), [PHP](https://ru.wikipedia.org/wiki/PHP).

**3.Методология** [**функционального программирования**](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

 Раздел [дискретной математики](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BA%D1%80%D0%B5%D1%82%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) и [парадигма программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в которой процесс [вычисления](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5) трактуется как вычисление значений [функций](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%28%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29) в математическом понимании последних (в отличие от [функций](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) как подпрограмм в [процедурном программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Противопоставляется парадигме [императивного программирования](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), которая описывает процесс вычислений как последовательное изменение [состояний](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5) (в значении, подобном таковому в [теории автоматов](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%BE%D0%B2)). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как [список](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29).

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](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), где одной из базовых концепций является [переменная](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), хранящая своё значение и позволяющая менять его по мере выполнения [алгоритма](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC)).

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](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B1%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%8D%D1%84%D1%84%D0%B5%D0%BA%D1%82_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) 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](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%BC%D0%BE%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (что обеспечивают функции без побочных эффектов – чистые функции).

[Лямбда-исчисление](https://ru.wikipedia.org/wiki/%D0%9B%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5) являются основой для функционального программирования, многие функциональные [языки](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) можно рассматривать как «надстройку» над ними[.](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-lambda-1)

Наиболее известными [языками функционального программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) являются:

-[Лисп](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BF) – ([Джон Маккарти](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%BA%D0%BA%D0%B0%D1%80%D1%82%D0%B8%2C_%D0%94%D0%B6%D0%BE%D0%BD), [1958](https://ru.wikipedia.org/wiki/1958)) и множество его диалектов, наиболее современные из которых:

-[Scheme](https://ru.wikipedia.org/wiki/Scheme%22%20%5Co%20%22Scheme)

-[Clojure](https://ru.wikipedia.org/wiki/Clojure%22%20%5Co%20%22Clojure)

-[Common Lisp](https://ru.wikipedia.org/wiki/Common_Lisp%22%20%5Co%20%22Common%20Lisp)

-[Erlang](https://ru.wikipedia.org/wiki/Erlang%22%20%5Co%20%22Erlang) – ([Joe Armstrong](https://ru.wikipedia.org/w/index.php?title=Joe_Armstrong&action=edit&redlink=1" \o "Joe Armstrong (страница отсутствует)), [1986](https://ru.wikipedia.org/wiki/1986)) функциональный язык с поддержкой процессов.

-[APL](https://ru.wikipedia.org/wiki/APL_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) – предшественник современных научных вычислительных сред, таких как [MATLAB](https://ru.wikipedia.org/wiki/MATLAB).

-[ML](https://ru.wikipedia.org/wiki/ML) ([Робин Милнер](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BB%D0%BD%D0%B5%D1%80%2C_%D0%A0%D0%BE%D0%B1%D0%B8%D0%BD), [1979](https://ru.wikipedia.org/wiki/1979), из ныне используемых диалектов известны [Standard ML](https://ru.wikipedia.org/wiki/SML) и [Objective CAML](https://ru.wikipedia.org/wiki/OCaml)).

-[F#](https://ru.wikipedia.org/wiki/F_Sharp) – функциональный язык семейства ML для платформы [.NET](https://ru.wikipedia.org/wiki/.NET_Framework)

-[Scala](https://ru.wikipedia.org/wiki/Scala_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29%22%20%5Co%20%22Scala%20%28%D1%8F%D0%B7%D1%8B%D0%BA%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29)

-[Miranda](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D1%80%D0%B0%D0%BD%D0%B4%D0%B0_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29%22%20%5Co%20%22%D0%9C%D0%B8%D1%80%D0%B0%D0%BD%D0%B4%D0%B0%20%28%D1%8F%D0%B7%D1%8B%D0%BA%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) ([Дэвид Тёрнер](https://ru.wikipedia.org/wiki/%D0%A2%D1%91%D1%80%D0%BD%D0%B5%D1%80%2C_%D0%94%D1%8D%D0%B2%D0%B8%D0%B4), [1985](https://ru.wikipedia.org/wiki/1985), который впоследствии дал развитие языку Haskell).

-[Nemerle](https://ru.wikipedia.org/wiki/Nemerle%22%20%5Co%20%22Nemerle) – гибридный функционально/императивный язык.

-[XSLT](https://ru.wikipedia.org/wiki/XSLT) и [XQuery](https://ru.wikipedia.org/wiki/XQuery)

-[Haskell](https://ru.wikipedia.org/wiki/Haskell%22%20%5Co%20%22Haskell) – [чистый](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) функциональный. Назван в честь [Хаскелла Карри](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D0%B8%2C_%D0%A5%D0%B0%D1%81%D0%BA%D0%B5%D0%BB%D0%BB).

Ещё не полностью функциональные изначальные версии и [Лиспа](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BF), и [APL](https://ru.wikipedia.org/wiki/%D0%90%D0%9F%D0%9B_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) внесли особый вклад в создание и развитие функционального программирования. Более поздние версии Lisp, такие как [Scheme](https://ru.wikipedia.org/wiki/Scheme), а также различные варианты APL поддерживали все свойства и концепции функционального языка.

Как правило, интерес к функциональным языкам программирования, особенно чисто функциональным, был скорее научный, нежели коммерческий. Однако, такие примечательные языки как [Erlang](https://ru.wikipedia.org/wiki/Erlang), [OCaml](https://ru.wikipedia.org/wiki/OCaml), [Haskell](https://ru.wikipedia.org/wiki/Haskell), Scheme (после 1986) а также специфические [R](https://ru.wikipedia.org/wiki/R_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) (статистика), [Wolfram](https://ru.wikipedia.org/wiki/Wolfram_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) (символьная математика), [J](https://ru.wikipedia.org/wiki/J_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) и K (финансовый анализ), и [XSLT](https://ru.wikipedia.org/wiki/XSLT) ([XML](https://ru.wikipedia.org/wiki/XML)) находили применение в индустрии коммерческого программирования. Такие широко распространенные декларативные языки как [SQL](https://ru.wikipedia.org/wiki/SQL) и[Lex](https://ru.wikipedia.org/wiki/Lex)/[Yacc](https://ru.wikipedia.org/wiki/Yacc%22%20%5Co%20%22Yacc) содержат некоторые элементы функционального программирования, например, они остерегаются использовать переменные. [Языки](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B0%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) работы с электронными таблицами также можно рассматривать как функциональные, потому что в ячейках электронных таблиц задаётся массив функций, как правило зависящих лишь от других ячеек, а при желании смоделировать переменные приходится прибегать к возможностям императивного языка макросов.

[Лямбда-исчисление](https://ru.wikipedia.org/wiki/%D0%9B%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5) стало теоретической базой для описания и вычисления функций. Являясь математической [абстракцией](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D0%B8%D1%8F), а не [языком программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), оно составило базис почти всех языков функционального программирования на сегодняшний день. Сходное теоретическое понятие, [комбинаторная логика](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B1%D0%B8%D0%BD%D0%B0%D1%82%D0%BE%D1%80%D0%BD%D0%B0%D1%8F_%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0), является более абстрактным, нежели λ-исчисления и было создано раньше. Эта логика используется в некоторых эзотерических языках, например в [Unlambda](https://ru.wikipedia.org/wiki/Unlambda). И λ-исчисление, и комбинаторная логика были разработаны для более ясного и точного описания принципов и основ математики.

Первым функциональным языком был [Лисп](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BF), созданный [Джоном Маккарти](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%BA%D0%BA%D0%B0%D1%80%D1%82%D0%B8%2C_%D0%94%D0%B6%D0%BE%D0%BD) в период его работы в [Массачусетском технологическом институте](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B0%D1%87%D1%83%D1%81%D0%B5%D1%82%D1%81%D0%BA%D0%B8%D0%B9_%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%81%D1%82%D0%B8%D1%82%D1%83%D1%82) в конце пятидесятых и реализованный, первоначально, для [IBM 700/7000](https://en.wikipedia.org/wiki/IBM_700/7000_series). В Лиспе впервые введено множество понятий функционального языка, хотя при этом в языке применяется не только парадигма функционального программирования. Дальнейшим развитием Лиспа стали такие языки как [Scheme](https://ru.wikipedia.org/wiki/Scheme) и [Dylan](https://ru.wikipedia.org/wiki/Dylan_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29).

Язык обработки информации ([Information Processing Language](https://en.wikipedia.org/wiki/Information_Processing_Language%22%20%5Co%20%22en%3AInformation%20Processing%20Language) , IPL) иногда определяется как самый первый машинный функциональный язык.Это язык [ассемблерного](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0) типа для работы со списком символов. В нём было понятие «генератора», который использовал функцию в качестве аргумента, а также, поскольку это язык ассемблерного уровня, он может позиционироваться как язык, имеющий функции высшего порядка. Однако, в целом IPL акцентирован на использование императивных понятий.

[Кеннет Е. Айверсон](https://ru.wikipedia.org/wiki/%D0%90%D0%B9%D0%B2%D0%B5%D1%80%D1%81%D0%BE%D0%BD%2C_%D0%9A%D0%B5%D0%BD%D0%BD%D0%B5%D1%82) разработал язык [APL](https://ru.wikipedia.org/wiki/APL_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) в начале шестидесятых, документировав его в своей книге A Programming Language ([ISBN 978-0-471-43014-8](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F%3A%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/9780471430148)). APL оказал значительное влияние на язык [FP](https://en.wikipedia.org/wiki/FP_%28programming_language%29) , созданный [Джоном Бэкусом](https://ru.wikipedia.org/wiki/%D0%91%D1%8D%D0%BA%D1%83%D1%81%2C_%D0%94%D0%B6%D0%BE%D0%BD). В начале девяностых Айверсон и [Роджер Хуэй](https://en.wikipedia.org/wiki/Roger_Hui) создали преемника APL – язык программирования [J](https://ru.wikipedia.org/wiki/J_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29). В середине девяностых [Артур Витни](https://en.wikipedia.org/wiki/Arthur_Whitney) , ранее работавший с Айверсоном, создал язык [K](https://ru.wikipedia.org/wiki/K_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29), который впоследствии использовался в финансовой индустрии на коммерческой основе.

В семидесятых в [университете Эдинбурга](https://ru.wikipedia.org/wiki/%D0%AD%D0%B4%D0%B8%D0%BD%D0%B1%D1%83%D1%80%D0%B3%D1%81%D0%BA%D0%B8%D0%B9_%D1%83%D0%BD%D0%B8%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%82%D0%B5%D1%82) [Робин Милнер](https://ru.wikipedia.org/wiki/%D0%A0%D0%BE%D0%B1%D0%B8%D0%BD_%D0%9C%D0%B8%D0%BB%D0%BD%D0%B5%D1%80) создал язык [ML](https://ru.wikipedia.org/wiki/ML), а [Дэвид Тернер](https://ru.wikipedia.org/wiki/%D0%A2%D1%91%D1%80%D0%BD%D0%B5%D1%80%2C_%D0%94%D1%8D%D0%B2%D0%B8%D0%B4) начинал разработку языка [SASL](https://ru.wikipedia.org/wiki/SASL_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) в [университете Сент-Эндрюса](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BD%D1%82-%D0%AD%D0%BD%D0%B4%D1%80%D1%8E%D1%81%D1%81%D0%BA%D0%B8%D0%B9_%D1%83%D0%BD%D0%B8%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%82%D0%B5%D1%82) и, впоследствии, язык [Miranda](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D1%80%D0%B0%D0%BD%D0%B4%D0%B0_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) в университете города Кент. В конечном итоге на основе ML были созданы несколько языков, среди которых наиболее известные [Objective Caml](https://ru.wikipedia.org/wiki/OCaml) и [Standard ML](https://ru.wikipedia.org/wiki/SML). Также в семидесятых осуществлялась разработка языка программирования, построенного по принципу Scheme (реализация не только функциональной парадигмы), получившего описание в известной работе «Lambda Papers», а также в книге восемьдесят пятого года «[Structure and Interpretation of Computer Programs](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B8_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%86%D0%B8%D1%8F_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D1%8B%D1%85_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%22%20%5Co%20%22%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0%20%D0%B8%20%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%86%D0%B8%D1%8F%20%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D1%8B%D1%85%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC)», в которой принципы функционального программирования были донесены до более широкой аудитории.

В восьмидесятых [Пер Мартин-Лёф](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%80%D1%82%D0%B8%D0%BD-%D0%9B%D1%91%D1%84%2C_%D0%9F%D0%B5%D1%80) создал [интуиционистскую теорию типов](https://ru.wikipedia.org/w/index.php?title=%D0%98%D0%BD%D1%82%D1%83%D0%B8%D1%86%D0%B8%D0%BE%D0%BD%D0%B8%D1%81%D1%82%D1%81%D0%BA%D0%B0%D1%8F_%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2&action=edit&redlink=1) (также называемую конструктивной). В этой теории функциональное программирование получило конструктивное доказательство того, что ранее было известно как зависимый тип. Это дало мощный толчок к развитию диалогового доказательства теорем и к последующему созданию множества функциональных языков.

[Haskell](https://ru.wikipedia.org/wiki/Haskell) был создан в конце восьмидесятых в попытке соединить множество идей, полученных в ходе исследования функционального программирования.

Основной особенностью функционального программирования, определяющей как преимущества, так и недостатки данной парадигмы, является то, что в ней реализуется модель вычислений без состояний. Если императивная программа на любом этапе исполнения имеет состояние, то есть совокупность значений всех переменных, и производит побочные эффекты, то чисто функциональная программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. То, что в императивных языках делается путём присваивания значений переменным, в функциональных достигается путём передачи выражений в параметры функций. Непосредственным следствием становится то, что чисто функциональная программа не может изменять уже имеющиеся у неё данные, а может лишь порождать новые путём копирования и/или расширения старых. Следствием того же является отказ от циклов в пользу рекурсии.

Сильные стороны

Повышение надёжности кода

Привлекательная сторона вычислений без состояний – повышение надёжности кода за счёт чёткой структуризации и отсутствия необходимости отслеживания побочных эффектов. Любая функция работает только с локальными данными и работает с ними всегда одинаково, независимо от того, где, как и при каких обстоятельствах она вызывается. Невозможность мутации данных при пользовании ими в разных местах программы исключает появление труднообнаруживаемых ошибок (таких, например, как случайное присваивание неверного значения глобальной переменной в императивной программе).

Удобство организации модульного тестирования

Поскольку функция в функциональном программировании не может порождать побочные эффекты, менять объекты нельзя как внутри области видимости, так и снаружи (в отличие от императивных программ, где одна функция может установить какую-нибудь внешнюю переменную, считываемую второй функцией). Единственным эффектом от вычисления функции является возвращаемый ей результат, и единственный фактор, оказывающий влияние на результат – это значения аргументов.

Таким образом, имеется возможность протестировать каждую функцию в программе, просто вычислив её от различных наборов значений аргументов. При этом можно не беспокоиться ни о вызове функций в правильном порядке, ни о правильном формировании внешнего состояния. Если любая функция в программе проходит модульные тесты, то можно быть уверенным в качестве всей программы. В императивных программах проверка возвращаемого значения функции недостаточна: функция может модифицировать внешнее состояние, которое тоже нужно проверять, чего не нужно делать в функциональных программах.

Возможности оптимизации при компиляции.

Традиционно упоминаемой положительной особенностью функционального программирования является то, что оно позволяет описывать программу в так называемом «декларативном» виде, когда жесткая последовательность выполнения многих операций, необходимых для вычисления результата, в явном виде не задаётся, а формируется автоматически в процессе вычисления функций. Это обстоятельство, а также отсутствие состояний даёт возможность применять к функциональным программам достаточно сложные методы автоматической оптимизации.

Возможности параллелизма

Ещё одним преимуществом функциональных программ является то, что они предоставляют широчайшие возможности для [автоматического распараллеливания](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D1%80%D0%B0%D1%81%D0%BF%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) вычислений. Поскольку отсутствие побочных эффектов гарантировано, в любом вызове функции всегда допустимо параллельное вычисление двух различных параметров – порядок их вычисления не может оказать влияния на результат вызова.

Недостатки функционального программирования вытекают из тех же самых его особенностей. Отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный [сборщик мусора](https://ru.wikipedia.org/wiki/%D0%A1%D0%B1%D0%BE%D1%80%D0%BA%D0%B0_%D0%BC%D1%83%D1%81%D0%BE%D1%80%D0%B0_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29). Нестрогая модель вычислений приводит к непредсказуемому порядку вызова функций, что создает проблемы при вводе-выводе, где порядок выполнения операций важен. Кроме того, очевидно, функции ввода в своем естественном виде (например, getchar из стандартной библиотеки языка [C](https://ru.wikipedia.org/wiki/C_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29)) не являются чистыми, поскольку способны возвращать различные значения для одних и тех же аргументов, и для устранения этого требуются определенные ухищрения.

Для преодоления недостатков функциональных программ уже первые языки функционального программирования включали не только чисто функциональные средства, но и механизмы императивного программирования (присваивание, цикл, «неявный PROGN» были уже в Лиспе). Использование таких средств позволяет решить некоторые практические проблемы, но означает отход от идей (и преимуществ) функционального программирования и написание императивных программ на функциональных языках. В чистых функциональных языках эти проблемы решаются другими средствами, например, в языке [Haskell](https://ru.wikipedia.org/wiki/Haskell) ввод-вывод реализован при помощи [монад](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%BD%D0%B0%D0%B4%D0%B0_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) – нетривиальной концепции, позаимствованной из теории категорий.

**4. Методология** [**логического программирования**](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

Логическое программирование – [парадигма программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), основанная на автоматическом доказательстве теорем, а также раздел [дискретной математики](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BA%D1%80%D0%B5%D1%82%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0), изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате [математической логики](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0) с использованием математических принципов резолюций.

Самым известным языком логического программирования является [Prolog](https://ru.wikipedia.org/wiki/Prolog).

Первым языком логического программирования был язык [Planner](https://ru.wikipedia.org/wiki/Planner) (см. обзор Шапиро (Ehud Shapiro) [1989]), в котором была заложена возможность автоматического вывода результата из данных и заданных правил перебора вариантов (совокупность которых называлась планом). Planner использовался для того, чтобы понизить требования к вычислительным ресурсам (с помощью бэктрекинга – [поиска с возвратом](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D1%81_%D0%B2%D0%BE%D0%B7%D0%B2%D1%80%D0%B0%D1%82%D0%BE%D0%BC)) и обеспечить возможность вывода фактов, без активного использования [стека](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA). Затем был разработан язык [Prolog](https://ru.wikipedia.org/wiki/Prolog), который не требовал плана перебора вариантов и был, в этом смысле, упрощением языка [Planner](https://ru.wikipedia.org/wiki/Planner).

От языка Planner также произошли логические языки программирования [QA-4](https://ru.wikipedia.org/w/index.php?title=QA-4&action=edit&redlink=1), [Popler](https://ru.wikipedia.org/w/index.php?title=Popler&action=edit&redlink=1), [Conniver](https://ru.wikipedia.org/w/index.php?title=Conniver&action=edit&redlink=1) и [QLISP](https://ru.wikipedia.org/w/index.php?title=QLISP_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)&action=edit&redlink=1). Языки программирования [Mercury](https://ru.wikipedia.org/wiki/Mercury_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29), [Visual Prolog](https://ru.wikipedia.org/wiki/Visual_Prolog), [Oz](https://ru.wikipedia.org/wiki/Oz_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) и [Fril](https://ru.wikipedia.org/w/index.php?title=Fril&action=edit&redlink=1) произошли уже от языка [Prolog](https://ru.wikipedia.org/wiki/Prolog). На базе языка[Planner](https://ru.wikipedia.org/wiki/Planner) было разработано также несколько альтернативных языков логического программирования, не основанных на методе поиска с возвратами), например, [Ether](https://ru.wikipedia.org/w/index.php?title=Scientific_Community_Metaphor&action=edit&redlink=1) (см. обзор Шапиро [1989]).

**5. Методология** [**программирования в ограничениях**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B2_%D0%BE%D0%B3%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%B5%D0%BD%D0%B8%D1%8F%D1%85)

Программирование в ограничениях (или программирование ограничениями) является [парадигмой программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в которой отношения между переменными указаны в форме ограничений. Ограничения отличаются от общих примитивов языков [императивного программирования](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) тем, что они определяют не последовательность шагов для исполнения, а свойства искомого решения. Это делает программирование в ограничениях формой [декларативного программирования](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). Ограничения, которые используются в программировании в ограничениях, бывают различных видов: те, которые используются в[задачах удовлетворения ограничений](https://ru.wikipedia.org/wiki/%D0%A3%D0%B4%D0%BE%D0%B2%D0%BB%D0%B5%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BE%D0%B3%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%B5%D0%BD%D0%B8%D0%B9) (например, «А или В истинно»), те, которые решаются [симплекс-алгоритмом](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%BF%D0%BB%D0%B5%D0%BA%D1%81-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4) (например, «x ≤ 5») и другие. Ограничения, как правило, встроены в язык программирования или осуществляются через отдельные программные библиотеки.

Программирование в ограничениях тесно связано с [теорией удовлетворения ограничений](https://ru.wikipedia.org/wiki/%D0%A3%D0%B4%D0%BE%D0%B2%D0%BB%D0%B5%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BE%D0%B3%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%B5%D0%BD%D0%B8%D0%B9), которая предлагает удобный аппарат и простую формальную схему для представления и решения комбинаторных задач искусственного интеллекта.

Программирование в ограничениях началось с логического программирования с ограничениями (ЛПО), которое является вкладыванием ограничений в [логическое программирование](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). Появление этого варианта логического программирования связано с именами Jaffar и Lassez, которые расширили в 1987 году определенный класс ограничений, которые были введены в Prolog II. Первыми реализациями логического программирование в ограничениях были Пролог III, CLP (R) и CHIP. Некоторые интерпретаторы логического программирования в ограничениях существуют и сегодня, например GNU Prolog.

Помимо логического программирования, ограничения могут быть смешаны с [функциональным программированием](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), [переписыванием термов](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BF%D0%B8%D1%81%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и [императивным языком](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F). Языки программирования с встроенной поддержкой ограничений включают [Oz](https://ru.wikipedia.org/wiki/Oz_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) (функциональное программирование) и Kaleidoscope (императивное программирование). Главным образом, ограничения осуществляются в императивных языках через инструментальные средства для решения задач с ограничениями, которые являются отдельными библиотеками для существующих императивных языков.

Временные ограничения параллельного программирования (КТС) и недетерминированные временные ограничения параллельного программирования (NTCC) являются вариантами программирования в ограничениях, которые могут иметь дело со временем.

CHIP был первым языком ЛПО, использующим распространение ограничений. Другими примерами систем ЛПО могут служить библиотеки поддержки ограничений ILOG и COSYTEC, а также языки логического программирования в ограничениях Prolog III, Prolog IV, CLP(R), ECLiPSe, CIAO, CLP(FD). Используя эти системы, многие сложные прикладные задачи были успешно решены с помощью технологий программирования в ограничениях. В числе примеров упомянем проверку электронных схем, календарное планирование, распределение ресурсов, составление расписаний, управляющие системы, графические интерфейсы, а также множество комбинаторных задач.

Можно заметить, что эти методологии находятся на шкале от навигационных (пошаговое управление исполнением) до спецификационных (определение требований к результату).

**По топологической специфике**

Специфика (топологическая специфика) – способ выбора методов для уточнения ядра методологии. Критерием качества той или иной топологии может могут являться общие затраты на разработку ПО. В свою очередь, затраты на разработку зависят среди прочего от ключевых языковых абстракций: абстракции данных, управления и модульности. Например, в императивной методологии можно придерживаться методов [структурного программирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), что даёт более выгодную топологию с точки зрения языковых абстракций. Результатом является методология [структурного программирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

**По специфике реализации**

В соответствии с архитектурой аппаратного обеспечения, реализация может быть централизованной или параллельной. Например, методология (императивного) параллельного программирования, методология логического параллельного программирования.

Кроме того, методология может быть гибридной. Например, наиболее часта смесь функционального и логического программирования.

Проводятся исследования и по [унификации методологий программирования](https://en.wikipedia.org/wiki/Unifying_Theories_of_Programming)

[Языки программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) могут хорошо поддерживать те или иные методологии, но это не означает, что некоторый язык вообще нельзя использовать с несвойственной ему методологией, а только то, что потребуется затратить больше усилий и ресурсов.

Методологии программирования различаются по общим затратам на решения задач с разными характеристиками (научные расчёты, финансовые задачи, [системы реального времени](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%80%D0%B5%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B2%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%B8) и т. п.). Масштаб задач и эффективность создаваемого программного обеспечения также являются важными факторами при выборе методологии программирования.

Алистэр Коуберн (Alistair Cockburn) изучает методологии разработки ПО с начала 90-х, когда компания IBM дала ему задание написать работу на эту тему. При этом его подход существенно отличается от подхода большинства других методологов. Его теории основаны не только на личном опыте, но и на постоянных исследованиях других проектов и процессов. Более того, он не боится менять свои воззрения в результате своих находок в этой области. Именно поэтому Алистэр Коуберн остается моим любимым методологом.

Первой рекомендацией Коуберна по ведению проектов стала книга Surviving Object-Oriented Projects, и по моему мнению, она остается лучшей работой по управлению итеративным проектом и по сей день. Совсем недавно он написал еще одну книгу, которая посвящена обзору гибких методологий разработки программного обеспечения. В ней он рассматривает базовые принципы, лежащие в основе всех этих методологий.

После этого он продолжал исследовать гибкие методологии, результатом чего стало появление семейства методологий Crystal. Коуберн называет это "семейством", так как убежден, что разным проектам нужны разные методологии. Он вводит следующую градацию проектов: по одной оси откладывается количество занятых в проекте людей, по другой - критичность ошибок. Каждая из методологий "семейства" предназначена для определенной ячейки получившийся сетки. Таким образом, проект, в котором занято 40 человек, и на котором компания может позволить себе потерять некоторую сумму, будет работать по другой методологии, нежели проект для 6 разработчиков, от которого зависит существование компании.

Как и ХР, методологии семейства Crystal тоже ориентированы на человека, однако несколько по-другому. Алистэр считает, что людям сложно использовать в работе процесс, который требует от них высокой дисциплины. В результате своих исследований он выяснил, какой объем дисциплины является для действенной методологии минимальным. При этом производительность методологии сознательно приносилась в жертву простоте ее использования. Пусть Crystal уступает ХР в производительности, зато ей сможет пользоваться большее количество людей, полагает Алистэр.

Особое значение Алистэр придает пересмотрам процесса в конце каждой итерации, благодаря которым процесс можно усовершенствовать в ходе работы. При итеративной разработке все проблемы, связанные с используемым процессом, обнаруживаются рано, и их можно успеть исправить. Поэтому так важно контролировать процесс разработки и адаптировать его к конкретной ситуации.

В феврале 2011 года Алистэр объявил, что они с Джимом Хайсмитом решили объединить свои методологии. Однако пока непонятно не только то, что будет собой представлять их методология, но даже то, как она будет называться.

**Модульное программирование** (modular programming) – способ разработки программ, при котором программа разбивается на относительно независимые составные части (программные модули). При этом каждый модуль может разрабатываться, программироваться, транслироваться и тестироваться независимо от других. Внутреннее строение модуля для функционирования всей программы, как правило, значения не имеет. При модификации алгоритма, реализуемого модулем, структура программы не должна меняться.

Цель модульного программирования

Приступая к разработке каждой программы ПС, следует иметь в виду, что она, как правило, является большой системой, поэтому мы должны принять меры для ее упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями. А сам такой метод разработки программ называют модульным программированием. Программный модуль – это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен с другими модулями программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю. Таким образом, программный модуль может рассматриваться и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании (т.е. как средство накопления и многократного использования программистских знаний).

Модульное программирование является воплощением в процессе разработки программ обоих общих методов борьбы со сложностью: и обеспечение независимости компонент системы, и использование иерархических структур. Для воплощения первого метода формулируются определенные требования, которым должен удовлетворять программный модуль, т.е. выявляются основные характеристики “хорошего” программного модуля. Для воплощения второго метода используют древовидные модульные структуры программ (включая деревья со сросшимися ветвями).

**Основные характеристики программного модуля**

Не всякий программный модуль способствует упрощению программы. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Так, Хольт предложил следующие два общих таких критерия:

• хороший модуль снаружи проще, чем внутри;

• хороший модуль проще использовать, чем построить.

Майерс предлагает для оценки приемлемости программного модуля использовать более конструктивные его характеристики:

• размер модуля,

• прочность модуля,

• сцепление с другими модулями,

• рутинность модуля (независимость от предыстории обращений к нему).

Размер модуля измеряется числом содержащихся в нем операторов или строк. Модуль не должен быть слишком маленьким или слишком большим. Маленькие модули приводят к громоздкой модульной структуре программы и могут не окупать накладных расходов, связанных с их оформлением. Большие модули неудобны для изучения и изменений, они могут существенно увеличить суммарное время повторных трансляций программы при отладке программы. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.

Прочность модуля - это мера его внутренних связей. Чем выше прочность модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести. Для оценки степени прочности модуля Майерс предлагает упорядоченный по степени прочности набор из семи классов модулей. Самой слабой степенью прочности обладает модуль, прочный по совпадению. Это такой модуль, между элементами которого нет осмысленных связей. Такой модуль может быть выделен, например, при обнаружении в разных местах программы повторения одной и той же последовательности операторов, которая и оформляется в отдельный модуль. Необходимость изменения этой последовательности в одном из контекстов может привести к изменению этого модуля, что может сделать его использование в других контекстах ошибочным. Такой класс программных модулей не рекомендуется для использования. Вообще говоря, предложенная Майерсом упорядоченность по степени прочности классов модулей не бесспорна. Однако, это не очень существенно, так как только два высших по прочности класса модулей рекомендуются для использования. Эти классы мы и рассмотрим подробнее.

Функционально прочный модуль - это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется для использования.

Информационно прочный модуль - это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Такой класс следует рассматривать как класс программных модулей с высшей степенью прочности. Информационно прочный модуль может реализовывать, например, абстрактный тип данных.

В модульных языках программирования как минимум имеются средства для задания функционально прочных модулей (например, модуль типа FUNCTION в языке ФОРТРАН). Средства же для задания информационно прочных модулей в ранних языках программирования отсутствовали. Эти средства появились только в более поздних языках. Так в языке программирования Ада средством задания информационно прочного модуля является пакет.

Сцепление модуля - это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления Майерс предлагает упорядоченный набор из шести видов сцепления модулей. Худшим видом сцепления модулей является сцепление по содержимому. Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле). Такое сцепление модулей недопустимо. Не рекомендуется использовать также сцепление по общей области - это такое сцепление модулей, когда несколько модулей используют одну и ту же область памяти. Такой вид сцепления модулей реализуется, например, при программировании на языке ФОРТРАН с использованием блоков COMMON. Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является параметрическое сцепление (сцепление по данным по Майерсу) - это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

Рутинность модуля - это его независимость от предыстории обращений к нему. Модуль будем называть рутинным, если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль будем называть зависящим от предыстории, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, изменяемого в результате предыдущих обращений к нему. Майерс не рекомендует использовать зависящие от предыстории (непредсказуемые) модули, так как они провоцируют появление в программах хитрых (неуловимых) ошибок. Однако такая рекомендация является неконструктивной, так как во многих случаях именно зависящий от предыстории модуль является лучшей реализаций информационно прочного модуля. Поэтому более приемлема следующая (более осторожная) рекомендация:

• всегда следует использовать рутинный модуль, если это не приводит к плохим (не рекомендуемым) сцеплениям модулей;

• зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;

• в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение (эффект выполнения) данного модуля при разных последующих обращениях к нему.

В связи с последней рекомендацией может быть полезным определение внешнего представления (ориентированного на информирование человека) состояний зависящего от предыстории модуля. В этом случае эффект выполнения каждой функции (операции), реализуемой этим модулем, следует описывать в терминах этого внешнего представления, что существенно упростит прогнозирование поведения данного модуля.

**Методы разработки структуры программы**

Как уже отмечалось выше, в качестве модульной структуры программы принято использовать древовидную структуру, включая деревья со сросшимися ветвями. В узлах такого дерева размещаются программные модули, а направленные дуги (стрелки) показывают статическую подчиненность модулей, т.е. каждая дуга показывает, что в тексте модуля, из которого она исходит, имеется ссылка на модуль, в который она входит. Другими словами, каждый модуль может обращаться к подчиненным ему модулям, т.е. выражается через эти модули. При этом модульная структура программы, в конечном счете, должна включать и совокупность спецификаций модулей, образующих эту программу. Спецификация программного модуля содержит

• синтаксическую спецификацию его входов, позволяющую построить на используемом языке программирования синтаксически правильное обращение к нему (к любому его входу),

• функциональную спецификацию модуля (описание семантики функций, выполняемых этим модулем по каждому из его входов).

Функциональная спецификация модуля строится так же, как и функциональная спецификация ПС.

В процессе разработки программы ее модульная структура может по-разному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Поэтому можно говорить о разных методах разработки структуры программы. Обычно в литературе обсуждаются два метода: метод восходящей разработки и метод нисходящей разработки.

Метод восходящей разработки заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модулей самого нижнего уровня (листья дерева модульной структуры программы), в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в принципе в таком же (восходящем) порядке, в каком велось их программирование. Такой порядок разработки программы на первый взгляд кажется вполне естественным: каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули. Однако, современная технология не рекомендует такой порядок разработки программы. Во-первых, для программирования какого-либо модуля совсем не требуется наличия текстов используемых им модулей - для этого достаточно, чтобы каждый используемый модуль был лишь специфицирован (в объеме, позволяющем построить правильное обращение к нему), а для тестирования его возможно (и даже, как мы покажем ниже, полезно) используемые модули заменять их имитаторами (заглушками). Во-вторых, каждая программа в какой-то степени подчиняется некоторым внутренним для нее, но глобальным для ее модулей соображениям (принципам реализации, предположениям, структурам данных и т.п.), что определяет ее концептуальную целостность и формируется в процессе ее разработки. При восходящей разработке эта глобальная информация для модулей нижних уровней еще не ясна в полном объеме, поэтому очень часто приходится их перепрограммировать, когда при программировании других модулей производится существенное уточнение этой глобальной информации (например, изменяется глобальная структура данных). В-третьих, при восходящем тестировании для каждого модуля (кроме головного) приходится создавать ведущую программу (модуль), которая должна подготовить для тестируемого модуля необходимое состояние информационной среды и произвести требуемое обращение к нему. Это приводит к большому объему “отладочного” программирования и в то же время не дает никакой гарантии, что тестирование модулей производилось именно в тех условиях, в которых они будут выполняться в рабочей программе.

Метод нисходящей разработки заключается в следующем. Как и в предыдущем методе сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модуля самого верхнего уровня (головного), переходя к программированию какого-либо другого модуля только в том случае, если уже запрограммирован модуль, который к нему обращается. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (нисходящем) порядке. При этом первым тестируется головной модуль программы, который представляет всю тестируемую программу и поэтому тестируется при “естественном” состоянии информационной среды, при котором начинает выполняться эта программа. При этом те модули, к которым может обращаться головной, заменяются их имитаторами (так называемыми заглушками). Каждый имитатор модуля представляется весьма простым программным фрагментом, который, в основном, сигнализирует о самом факте обращения к имитируемому модулю, производит необходимую для правильной работы программы обработку значений его входных параметров (иногда с их распечаткой) и выдает, если это необходимо, заранее запасенный подходящий результат. После завершения тестирования и отладки головного и любого последующего модуля производится переход к тестированию одного из модулей, которые в данный момент представлены имитаторами, если таковые имеются. Для этого имитатор выбранного для тестирования модуля заменяется самим этим модулем и, кроме того, добавляются имитаторы тех модулей, к которым может обращаться выбранный для тестирования модуль. При этом каждый такой модуль будет тестироваться при “естественных” состояниях информационной среды, возникающих к моменту обращения к этому модулю при выполнении тестируемой программы. Таким образом, большой объем “отладочного” программирования при восходящем тестировании заменяется программированием достаточно простых имитаторов используемых в программе модулей. Кроме того, имитаторы удобно использовать для того, чтобы подыгрывать процессу подбора тестов путем задания нужных результатов, выдаваемых имитаторами. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т.е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Некоторым недостатком нисходящей разработки, приводящим к определенным затруднениям при ее применении, является необходимость абстрагироваться от базовых возможностей используемого языка программирования, выдумывая абстрактные операции, которые позже нужно будет реализовать с помощью выделенных в программе модулей. Однако способность к таким абстракциям представляется необходимым условием разработки больших программных средств, поэтому ее нужно развивать.

Особенностью рассмотренных методов восходящей и нисходящей разработок (которые мы будем называть классическими) является требование, чтобы модульная структура программы была разработана до начала программирования (кодирования) модулей. Это требование находится в полном соответствии с водопадным подходом к разработке ПС, так как разработка модульной структуры программы и ее кодирование производятся на разных этапах разработки ПС: первая завершает этап конструирования ПС, а второе - открывает этап кодирования. Однако эти методы вызывают ряд возражений: представляется сомнительным, чтобы до программирования модулей можно было разработать структуру программы достаточно точно и содержательно. На самом деле это делать не обязательно, если несколько модернизировать водопадный подход. Ниже предлагаются конструктивный и архитектурный подходы к разработке программ, в которых модульная структура формируется в процессе программирования (кодирования) модулей.



Рис. 1. Первый шаг формирования модульной структуры программы при конструктивном подходе

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модулей. Разработка программы при конструктивном подходе начинается с программирования головного модуля, исходя из спецификации программы в целом. При этом спецификация программы принимается в качестве спецификации ее головного модуля, который полностью берет на себя ответственность за выполнение функций программы. В процессе программирования головного модуля, в случае, если эта программа достаточно большая, выделяются подзадачи (внутренние функции), в терминах которых программируется головной модуль. Это означает, что для каждой выделяемой подзадачи (функции) создается спецификация реализующего ее фрагмента программы, который в дальнейшем может быть представлен некоторым поддеревом модулей. Важно заметить, что здесь также ответственность за выполнение выделенной функции несет головной (может быть, и единственный) модуль этого поддерева, так что спецификация выделенной функции является одновременно и спецификацией головного модуля этого поддерева. В головном модуле программы для обращения к выделенной функции строится обращение к головному модулю указанного поддерева в соответствии с созданной его спецификацией. Таким образом, на первом шаге разработки программы (при программировании ее головного модуля) формируется верхняя начальная часть дерева, например, такая, которая показана на рис. 1.



Рис. 2. Второй шаг формирования модульной структуры программы при конструктивном подходе

Аналогичные действия производятся при программировании любого другого модуля, который выбирается из текущего состояния дерева программы из числа специфицированных, но пока еще не запрограммированных модулей. В результате этого производится очередное доформирование дерева программы, например, такое, которое показано на рис. 2.

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Но при этом ставится существенно другая цель разработки: повышение уровня используемого языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяются типичные функции, каждая из которых может использоваться при решении разных задач в этой области, и специфицируются, а затем и программируются отдельные

программные модули, выполняющие эти функции. Так как процесс выделения таких функций связан с накоплением и обобщением опыта решения задач в заданной предметной области, то обычно сначала выделяются и реализуются отдельными модулями более простые функции, а затем постепенно появляются модули, использующие ранее выделенные функции. Такой набор модулей создается в расчете на то, что при разработке той или иной программы заданной предметной области в рамках конструктивного подхода могут оказаться приемлемыми некоторые из этих модулей. Это позволяет существенно сократить трудозатраты на разработку конкретной программы путем подключения к ней заранее заготовленных и проверенных на практике модульных структур нижнего уровня. Так как такие структуры могут многократно использоваться в разных конкретных программах, то архитектурный подход может рассматриваться как путь борьбы с дублированием в программировании. В связи с этим программные модули, создаваемые в рамках архитектурного подхода, обычно параметризуются для того, чтобы усилить применимость таких модулей путем настройки их на параметры.



Рис. 3. Классификация методов разработки структуры программ.

В классическом методе нисходящей разработки рекомендуется сначала все модули разрабатываемой программы запрограммировать, а уж затем начинать нисходящее их тестирование, что опять-таки находится в полном соответствии с водопадным подходом. Однако такой порядок разработки не представляется достаточно обоснованным: тестирование и отладка модулей может привести к изменению спецификации подчиненных модулей и даже к изменению самой модульной структуры программы, так что в этом случае программирование некоторых модулей может оказаться бесполезно проделанной работой. Нам представляется более рациональным другой порядок разработки программы, известный в литературе как метод нисходящей реализации, что представляет некоторую модификацию водопадного подхода. В этом методе каждый запрограммированный модуль начинают сразу же тестировать до перехода к программированию другого модуля.

Все эти методы имеют еще различные разновидности в зависимости от того, в какой последовательности обходятся узлы (модули) древовидной структуры программы в процессе ее разработки. Это можно делать, например, по слоям (разрабатывая все модули одного уровня, прежде чем переходить к следующему уровню). При нисходящей разработке дерево можно обходить также в лексикографическом порядке (сверху вниз, слева направо). Возможны и другие варианты обхода дерева. Так, при конструктивной реализации для обхода дерева программы целесообразно следовать идеям Фуксмана, которые он использовал в предложенном им методе вертикального слоения. Сущность такого обхода заключается в следующем. В рамках конструктивного подхода сначала реализуются только те модули, которые необходимы для самого простейшего варианта программы, которая может нормально выполняться только для весьма ограниченного множества наборов входных данных, но для таких данных эта задача будет решаться до конца. Вместо других модулей, на которые в такой программе имеются ссылки, в эту программу вставляются лишь их имитаторы, обеспечивающие, в основном, сигнализацию о выходе за пределы этого частного случая. Затем к этой программе добавляются реализации некоторых других модулей (в частности, вместо некоторых из имеющихся имитаторов), обеспечивающих нормальное выполнение для некоторых других наборов входных данных. И этот процесс продолжается поэтапно до полной реализации требуемой программы. Таким образом, обход дерева программы производится с целью кратчайшим путем реализовать тот или иной вариант (сначала самый простейший) нормально действующей программы. В связи с этим такая разновидность конструктивной реализации получила название метода целенаправленной конструктивной реализации. Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика. Поэтому этот метод является весьма привлекательным.

Подводя итог сказанному, на рис. 3 представлена общая классификация рассмотренных методов разработки структуры программы.

**Контроль структуры программы**

Для контроля структуры программы можно использовать три метода:

• статический контроль,

• смежный контроль,

• сквозной контроль.

Статический контроль состоит в оценке структуры программы, насколько хорошо программа разбита на модули с учетом значений рассмотренных выше основных характеристик модуля.

Смежный контроль сверху - это контроль со стороны разработчиков архитектуры и внешнего описания ПС. Смежный контроль снизу - это контроль спецификации модулей со стороны разработчиков этих модулей.

Сквозной контроль - это мысленное прокручивание (проверка) структуры программы при выполнении заранее разработанных тестов. Является видом динамического контроля так же, как и ручная имитация функциональной спецификации или архитектуры ПС.

Следует заметить, что указанный контроль структуры программы производится в рамках водопадного подхода разработки ПС, т.е. при классическом подходе. При конструктивном и архитектурном подходах контроль структуры программы осуществляется в процессе программирования (кодирования) модулей в подходящие моменты времени.