## Стили программирования

Стили программирования можно сопоставить со стилями одежды. Классический стиль приемлем в любой ординарной обстановке, хотя иногда может быть и функционально неудобным, и не очень подходящим в других отношениях.

Поскольку стиль программирования — неформализуемое понятие очень высокого уровня, строгого определения дать невозможно. Поэтому охарактеризуем его и другие, тесно взаимосвязанные с ним, понятия следующим образом.

Под стилем программирования понимается внутренне согласованная совокупность программ, обладающих общими фундаментальными особенностями, как логическими, так и алгоритмическими, и базовых концепций, связанных с этими программами.

Стиль программирования реализуется через методологии программирования, заключающиеся в совокупности соглашений о том, какие базовые концепции языков программирования и какие их сочетания считаются приемлемыми или неприемлемыми для данного стиля. Методология включает в себя модель вычислителя для данного стиля.

Перейдем теперь к конкретным рассмотрениям. Поскольку теории стилей в настоящий момент просто нет, рассмотрим практически сложившиеся основные стили построения программ, несколько упорядочив и отцензурировав их список в соответствии с концепциями:

* Программирование от состояний;
* Структурное программирование;
* Сентенциальное программирование;
* Программирование от событий;
* Программирование от процессов и приоритетов;
* Функциональное программирование;
* Объектно-ориентированное программирование;
* Программирование от переиспользования;
* Программирование от шаблонов.

Это перечисление не претендует на теоретическую полноту и даже на полную обоснованность. Здесь мы стремимся сосредоточиться на классификации методов и отвлечься от конкретных частностей выражения.

*Паради́гма программи́рования* — это совокупность идей и понятий, определяющих стиль написания [компьютерных программ](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) (подход к программированию). В статье Дэниела Боброва парадигма определяется как «стиль программирования как описания намерений программиста».

Часто в статьях, когда пишут про парадигмы программирования, в скобочках пишут словосочетание «стиль программирования».

<http://acm.mipt.ru/twiki/bin/view/Method/StylesSamples>.

<http://www.nsc.ru/win/elbib/data/show_page.dhtml?77+1333>.

Так что многие не выделяют отличий между этими понятиями. Однако источников, описывающих парадигмы или стили достаточно много. Так что иногда их классификация может быть различна в разных источниках.

Почти всегда для языка есть вариант писать под разные стили.   
Тот же LISP является функциональным ЯП, но также обладает чертами императивности, также позволяет реализовать ООП.

Однако некоторые языки, можно сказать, относят к некоторым стилям, полагаясь на то, что в основном на этих языках работают именно на этих конкретных стилях.

К примеру, к структурному стилю относят Fortran, Pascal, C. К функциональному Lisp и Haskell, к ООП Java, C++, Python.

### Программирование от состояний

Это — пожалуй, самый старый стиль программирования. Он соответствует теоретическому понятию конечного автомата. На этот стиль программирования наталкивает само устройство существующих вычислительных машин, которое представляют собой гигантские конечные автоматы.

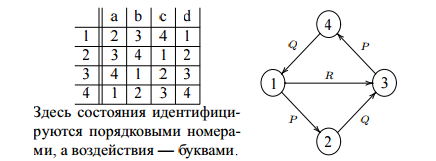
Суть программирования от состояний можно охарактеризовать следующим образом.

Определяются:

* множество так называемых состояний, которые может принимать конечный автомат;
* переходы между состояниями, которые осуществляются под внешним воздействием (например, под воздействием перерабатываемых данных).

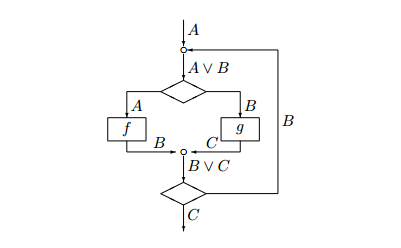
Программа, написанная в таком стиле, является *перечнем команд*, фиксирующих переходы между состояниями. Если же говорить в более ‘теоретических’ терминах, то для каждой возможной пары (состояние, внешнее воздействие) указывается очередное состояние.

Описание такой программы может быть произведено разными способами. Многие из них хорошо изучены теоретически и поддерживаются развитыми методиками программирования. Современные методики программирования от состояний базируются на таблицах состояний, подобных таблице состояний конечного автомата. Эти таблицы часто также представляются в виде графов, что особенно удобно, когда не все возможные переходы между состояниями реализуемы.



Заметим, что естественно рассматривать таблицы состояний и переходов как недетерминированные, поскольку после выполнения действия вполне может оказаться истинно сразу несколько условий, соответствующих выходящим ребрам.

Исторически первой моделью программирования от состояний, использованной и на практике, и для теоретических исследований, явилось представление программы в виде *блок-схемы*, узлы которой представляют собой состояния.



Заметим, что, операции в программировании от состояний глобальны, а условия локальны. Проверка условия не изменяет состояния всей системы (ни одного из ее параметров или характеристик), она лишь ведет нас в то или иное состояние самой программы.

Важно подчеркнуть, что для стиля программирования от состояний характеристическим качеством является не конкретное представление автомата, а то, что программист заставляет себя думать о разрабатываемой или исследуемой программе, как о таком автомате. Во многих случаях эта позиция оправдана и приносит существенные плоды (например, когда требуется перерабатывать потоки данных). Но далеко не всякая обработка соответствует автоматному мышлению. В частности, когда сложность алгоритма превышает определенные пределы и обозримость автомата не может быть достигнута, данный стиль мышления и, как следствие, стиль программирования становится неприемлемым.

### Структурное программирование

В теории схем программ было замечено, что некоторые случаи блок-схем легче поддаются анализу. Поэтому естественно было выделить такой класс блок-схем, что и сделали итальянские ученые С. Бем и К. Джакопини в 1966 г. Они доказали, что любую блок-схему можно преобразовать к структурированному виду, использовав несколько дополнительных булевых переменных. Э. Дейкстра подчеркнул, что программы в таком виде, как правило, являются легче понимаемыми и модифицируемыми, так как каждый блок имеет один вход и один выход. Эти наблюдения послужили основой стиля программирования, который, начиная с 70-х гг. XX века, занимает практически монопольное положение в преподавании и исключительно широко используется в практике программирования. Этот стиль называется **структурным программированием**.

Структурное программирование основано главным образом на теоретическом аппарате теории рекурсивных функций. Программа рассматривается как частично-рекурсивный оператор над библиотечными подпрограммами и исходными операциями. Структурное программирование базируется также на теории доказательств, прежде всего на естественном выводе. Структура программы соответствует структуре простейшего математического рассуждения, не использующего сложных лемм и абстрактных понятий.

Структурное программирование естественно возникает во многих классах задач, прежде всего в таких, где задача естественно расщепляется на подзадачи, а информация — на достаточно независимые структуры данных. Для подобных задач — это, безусловно, лучший стиль программирования.

### Сентенциальное программирование

Сентенциальное программирование отличается следующими *особенностями:*

* Память представляется как единая большая структура данных. Состояние программы (поле зрения) задается как выражение либо совокупность выражений. В некоторых из вариантов сентенциального программирования эти выражения могут содержать переменные, а в других — нет.
* В поле зрения по некоторым правилам может быть выделена активная часть, т. е. рассматриваемая в данный момент.
* На каждом шаге выполнения программы глобальные свойства активной части памяти подвергаются анализу. Выполняется не тот оператор, который стоит следующим в программе либо на который явно передано управление, а тот, который соответствует условию, выполненному для данного состояния внутреннего мира программы.
* Каждое действие программы состоит в замене одного выражения на другое: в традиционной для таких языков записи, левой его части на правую.
* Выполняется тот оператор, для которого активная часть выражения получается из левой части оператора некоторой подстановкой значений переменных, их конкретизацией либо унификацией. Полученные при конкретизации (унификации) значения переменных используются для конкретизации правой части (а при унификации — и всего поля памяти). Активная часть выражения заменяется на правую часть правила.

При сентенциальном стиле образ мышления программиста подобен выводу предложения по грамматике (отсюда и наименование стиля). Этот стиль удобен, если перерабатываемые данные естественно представлять в виде сложной структуры единиц, которые достаточно глобально распознаются и достаточно регулярно, но глобально, меняются. В отличие от структурного программирования, когда структурируются и локализуются прежде всего действия, здесь делается упор на локальности активной (преобразуемой) части данных, а действия глобальны.

### Программирование от событий

Есть довольно обширный круг задач, которые естественно описывать как совокупность реакций на события, возникающие в среде выполнения программы. Вообще говоря, так можно трактовать любую программу, обрабатывающую данные: поступление очередного данного — это внешнее событие, требующее реакции, которая, как минимум, должна быть связана с вводом этого данного. Понятно, что такая трактовка далеко не всегда продуктивна. Но она оправдана, например, когда есть много событий, порядок которых не определяет логику обработки, когда реакция на каждое событие автономна, т. е. не зависит от реакции на другие события.

Общая характеристика подобных ситуаций сводится к трем условиям, которые можно считать определяющими для целесообразного применения стиля программирования от событий, или *событийно-ориентированного стиля программирования*:

* процессы генерации событий отделены от процессов их обработки;
* процессы отработки разных реакций не зависят друг от друга;
* можно определить единый механизм установления контакта между событием и реакцией на него, никак не связанный с обработкой.

Стиль событийного программирования — это создание для каждого события собственной процедуры-обработчика. Порядок, в котором обработчики описываются в программе, не имеет никакого значения. Более того, они могут, а во многих случаях и должны быть приписаны к разным структурным единицам программы, в рамках которых только и осмыслена реакция на событие. Как следствие, продуктивно разбивать реакцию на событие на части, за которые отвечают такие структурные единицы, и иметь несколько реакций (разных) структурных единиц на одно событие.

До сих пор мы ничего не говорили о том, какие события возможны при программировании в событийно-ориентированном стиле. Исторически этот стиль как определенный “художественный прием” сформировался в области разработки операционных систем, где естественно связывать понятие события с прерываниями. *Прерывание* — это сигнал от одного из устройств (может быть, и от самого процессора), который говорит о том, что произошло нечто, на что нужно обратить внимание. Когда происходит прерывание, операционная система распознает, какая причина его вызвала, и далее формирует событие как информационный объект, вызывающий реакцию программной системы. Возможны разные способы реагирования на события, в том числе и передача его для обработки той программе, при выполнении которой возникло прерывание, породившее это событие. Из потребности такой обработки, собственно говоря, и сформировался событийно-ориентированный стиль программирования.

### Программирование от приоритетов

При программировании в стиле событий может возникнуть потребность упорядочивания выполнения нескольких конкурирующих между собой реакций на события. Достаточно универсальным средством удовлетворения этой потребности является приписывание реакциям приоритетов: сначала выполняются те реакции, которые имеют больший приоритет.

Этот прием, как выявилось на практике, пригоден для решения достаточно широкого круга задач. Он стал основой для формирования самостоятельного стиля **программирования от приоритетов**.

В программировании от приоритетов, как и в сентенциальном программировании, порядок расположения операторов в программе имеет малое значение, зато имеет значение его приоритет, т. е. некоторое значение, принадлежащее в самом общем случае частично-упорядоченному множеству и почти всегда рассматриваемое как элементарное. После завершения очередного оператора среди оставшихся выбирается оператор с максимальным приоритетом. Если таких операторов с равными или несравнимыми приоритетами несколько, то, вообще говоря, в идеале надо было бы выбирать один из них недетерминированно.

В целом программирование от приоритетов является мощным, но специфическим орудием для описания глобальных совместных, параллельных или распределенных процессов. Его элементы проникают и в традиционные системы программирования в виде обработки исключительных ситуаций в программах.

### Функциональное программирование

При функциональном программировании основным понятием является *функция*. Мы интересуемся определением и переопределением функции как целого. Соответственно, тогда сами функции становятся значениями.

Функциональное программирование базируется на исчислении, на теории рекурсивных схем и на денотационной семантике, дающей модели этих понятий.

Надо сказать, что при работе с функционалами высших типов возникают большие сложности, и для понимания их теории нужно хорошо знать логику, современную алгебру и топологию. Специалистов, владеющих и этой теорией, и практикой программирования, практически нет.

Наиболее богатый опыт и развитые традиции функционального программирования имеет язык LISP. Методы составления LISP-программ вполне сложились, и можно подвести некоторые итоги. Стиль программирования на LISP’е не удалось погубить даже внедрением совершенно чуждого функциональности присваивания. В то же время, этот стиль оказался гибок по отношению к освоению новых концепций, когда они не противоречат базовым средствам языка: и абстрактные типы данных, и объектная ориентированность вполне успешно прививаются на стройное дерево списочной структуры LISP’а.

Как показывает опыт LISP’а, взаимопроникновение стилей возможно, и особенно успешно в случае концептуально хорошо проработанной базы. Но далеко не всегда оно приводит к ожидаемому росту выразительности. Многочисленные примеры демонстрируют обратное. И всегда неудачи заимствований обусловлены нечеткостью проработки базовых концепций, отходом от того, что логически и теоретически предопределено для данного стиля. Вот почему крайне важно проанализировать существующие стили и выявить их сущность, а не поверхностные сходства и различия. Опыт показал, что владение функциональным стилем программирования является элементом фундаментального образования программиста и мощным средством проектирования программ на концептуальном уровне. Тот, кто осмыслил концепции функционального программирования, гораздо глубже овладевает современными высокоуровневыми методами объектно-ориентированного проектирования и дизайна и гораздо эффективнее их применяет.

Выбирая функциональное программирование, мы почти полностью отказываемся от UML в виду его объектной природы. Так, диаграммы классов, объектов и последовательности практически теряют смысл. В Haskell нет классов или объектов, нет инкапсулированной мутабельности, — а есть только процесс преобразования данных. Диаграмму последовательности, необходимую для описания взаимодействий этих самых объектов, можно было бы как-то использовать для цепочек функций, но штука в том, что сами цепочки будут более читабельны и понятны. Все прочие диаграммы, тем не менее, вполне применимы. В большой программе на ФП также имеются подсистемы или компоненты, а значит, в строю остаются диаграммы компонентов, пакетов и коммуникации. Диаграмма состояний — универсальна: процессы и конечные автоматы встречаются очень часто. Ее можно было бы применять даже в иных областях, не только в разработке ПО. Наконец, диаграмма вариантов использования вообще имеет мало отношения к дизайну ПО; она связывает бизнес-требования с системными требованиями на этапе анализа.

Но если внимательно присмотреться к «применимым» диаграммам, можно прийти к выводу, что они слабо помогут в высокоуровневом дизайне кода (который располагается ниже дизайна архитектуры), а то и вовсе могут навредить, подталкивая к императивному мышлению.

Во многих источниках делается вывод, что функциональные языки очень декларативные и можно сказать, что программы, на них написанные, документируют сами себя.

*Главным недостатком функциональных языков программирования* считается плохая совместимость с самыми популярными из императивных языков программирования, на которых сейчас написано подавляющее большинство широко используемого программного обеспечения. Также не стоит забывать про плохую переносимость программ на функциональных языках на различные платформы и низкую популярность этих языков (то есть, получается рекурсия - языки не популярны из-за того, что все опасаются серьёзно их использовать из-за их низкой популярности). Есть ещё ряд причин, но все они также, в большинстве своём, упираются в инерционность индустрии программного обеспечения, в том числе и в инерционность людей, которые работают в ней.

"Функциональный программист смотрится как средневековый монах, отвергающий удовольствия жизни в надежде, что это сделает его добродетельным. Для тех, кто заинтересован в материальных выгодах, эти "преимущества" не очень убедительны. (...) Ясно, что такая характеристика функционального программирования неадекватна". Джон Хьюз

### Объектно-ориентированный подход

В современном программировании *объектно-ориентированный подход (ООП)* является одним из популярнейших. Он превратился в стандарт во многих фирмах и во многих сообществах программистов. Этот стиль вместе с соответствующими организационными нововведениями позволил резко повысить эффективность коллективной работы программистов.

Сейчас можно сказать, что есть мода на ООП, так как по статистике это самый популярный стиль программирования. Но также как и в моде на одежду стоит руководствоваться не только какими-то тенденциями, но и задумываться о том, какой стиль в конкретной ситуации больше подходит. К примеру, если в моде открытые платья, то вряд ли лучшей идеей будет одеться по этой моде в холод -20 градусов. Также и ООП подходит отнюдь не всегда.

ООП базируется на интенсивном использовании сложных структур данных: объектов. Объекты соединяют внутри себя данные и методы. Чаще всего объекты организуются таким образом, что к данным прямого доступа извне нет, мы можем обращаться к ним лишь через методы. Это позволяет быстро заменять внутреннее представление данных, что исключительно важно для обеспечения перестраиваемости сложных программных систем.

Более того, для замены внутреннего представления данных в объектно- ориентированном программировании имеется механизм наследования. Можно описать новый тип объектов, базирующийся на уже имеющемся, и переопределяющий его методы. Как следствие, при присваивании переменной-объекту могут замениться не только значения данных, но и обрабатывающие их функции.

Объектный подход является прежде всего структурой достаточно высокоуровневых понятий, которые надстраиваются над базисом программных конструкций какого-то стиля первого уровня. В частности, объектная программа может с равным успехом базироваться и на структурном программировании, и на программировании от состояний, и на программировании от приоритетов.

Очень важной стадией развития ООП является объектно-ориентированное проектирование (дизайн) (ООД). Здесь система описывается с нескольких сторон, сначала со стороны пользователя (диаграммы использования), потом со стороны данных (диаграммы классов) и с других сторон реализации (диаграммы состояний и т. п.) Поддерживать целостность системы описаний и переходить от диаграмм к прототипам программ позволяет система *UML* (Unified Modelling Language). Это уже не язык программирования, а язык формализованных спецификаций программ.

### Три технологических стиля программирования

Настоящий параграф посвящен несколько иным стилям, чем представленные выше, зависящим не столько от языков и от задач, сколько от программного и технологического окружения программиста. Эти стили рассчитаны не на составление единственной программы, а на серийное их производство (в том или ином смысле).

В одном случае в качестве серии рассматривается набор программ, в которых повторяется применение одних и тех же фрагментов. Это стиль программирования от переиспользования.

В другом случае серия — это набор различных программ, которые в принципе могут быть построены автоматически (что в реальности означает либо полуавтоматически, либо вручную, как правило, по жестко фиксированному методу) из одной общей программы с учетом особенностей применения. Это стиль специализирующего программирования.

Наконец, третий случай, для которого серийность вырождается. Она подменяется парой, один компонент которой — макет, образец, демонстрационная разработка или же что-то отличное от программы, выполняющей решение задачи, но способное показать некоторые ее особенности либо в другом смысле приблизить разработку к цели. Это нечто называется образцом, макетом, шаблоном либо прототипом, по которому может строиться программа. Второй компонент пары — производственный вариант программы, изготавливаемый на основе информации о макетном образце, о его разработке или путем технологической процедуры работы с первым компонентом пары (например, это могут быть правила заполнения шаблона). Это стиль программирования от образца.Clean code

Робин Мартин (автор книги «Clean code» и не только) утверждает, что всех программистов, которые добиваются успеха в мире разработки ПО, отличает один общий признак: они больше всего заботятся о качестве создаваемого программного обеспечения. Это – основа для них. Потому что они являются профессионалами своего дела. То есть написание чистого кода является одним из важных аспектов программирования.

Некоторые принципы написания чистого кода, безусловно, есть у всех языков программирования, однако часто эти принципы выделяют для каждого языка в отдельности. Такой свод правил обычно называют “Best practices”, в интернете с легкостью можно найти “Best practices” для любого конкретного языка.

Однако для языков низкого уровня Clean Code не предусмотрен.

### Best practices JavaScript

* Избегайте глобальных переменных. Вместо этого используйте локальные переменные либо замыкания.
* Всегда объявляйте переменные, используйте ключевое слово “var”.
* Стараться делать все объявления как можно выше. Это сделает код чище, все переменные будут объявлены в одном месте, что поможет избежать возможности переобъявления переменных.

Это правило распространяется также и на циклы:

var i;

for (i = 0; i < 10; i++) {…}

* Инициализировать переменные при объявлении
* Не объявлять переменные как объектные типы, т.е. не использовать ключевые слова “String”, “Boolean”, “Number”.

var x = "John";               
var y = new String("John");  
(x === y) // is false because x is a string and y is an object.  
---------------------------------------------------------------  
var x = new String("John");               
var y = new String("John");  
(x == y) // is false because you cannot compare objects.

* Не использовать “new Object()”  
  Use {} instead of new Object()  
  Use "" instead of new String()  
  Use 0 instead of new Number()  
  Use false instead of new Boolean()  
  Use [] instead of new Array()  
  Use /()/ instead of new RegExp()  
  Use function (){} instead of new function()
* Использовать типы для того, для чего они предназначены. В числовых выражениях не следует использовать строки-числа и т.д.
* Использовать сравнение “===”
* Проверять аргументы функции на undefined. Это на случай, если при вызове функции аргумент не был передан.
* Заканчивать конструкцию “switch” с “default”.
* Не использовать функцию eval() – функция для запуска текста в качестве кода.
* Использовать “use strict”.

### Best Practices C++

* Названия своих типов с большой буквы: MyClass
* Названия функций с маленькой буквы: myMethod
* Константы большими буквами: const double PI = 3.14;
* Названия переменных писать либо сamelCase, либо snake\_case
* Private переменные называть, начиная с “m\_” (member)
* Параметры функции называть, начиная с “t\_”
* Не начинать названия с символа “\_”
* Для выходных и входных файлов создавать отдельные папки
* Использовать лишь однострочные комментарии “//”. Это поможет впоследствии избежать конфликтов с многострочными комментариями, добавленными для дебага.
* Не использовать “using namespace” в Header-файле
* Всегда использовать “{}”, даже если выполняется лишь одна команда в блоке
* Не писать код в ширину. Если, к примеру, имеет место длинное условие, следует раскидать его на несколько строк.
* Инициализировать переменные класса
* Всегда использовать namespace
* Не мешать табы с пробелами

### Знание Clean Code при устройстве на работу

Вакансии с такими требованиями имеют место. И можно предположить, раз Clean Code – отражение профессионализма программиста, то возможно на серьезные должности его знание является обязательным.

В Беларуси к фирмам с такими требованиями к разработчикам можно отнести:

* [Cedon BLR](http://jobs.tut.by/employer/757074)
* [Ciklum](http://jobs.tut.by/employer/8987)
* [Itransition](http://jobs.tut.by/employer/204511)
* [Specific-Group](http://jobs.tut.by/employer/679033)
* [DAROO](http://jobs.tut.by/employer/843780)
* [ИООО СИБ софтвэр](http://jobs.tut.by/employer/799984)
* [Intetics](http://jobs.tut.by/employer/550996)
* [Прогрессив Софт](http://jobs.tut.by/employer/1211290)
* [Иностр. п. ПИКСОЛИО](http://jobs.tut.by/employer/1244623)
* [Zagreby Ltd.](http://jobs.tut.by/employer/859156)
* [ООО МКСмедиа](http://jobs.tut.by/employer/1001985)

## Рефакторинг

*Рефа́кторинг* ([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) refactoring) или реорганизация кода  — процесс изменения внутренней структуры [программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и четкости.

### Цели рефакторинга

Цель рефакторинга — сделать код программы легче для понимания; без этого рефакторинг нельзя считать успешным.

Рефакторинг следует отличать от [оптимизации производительности](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BA%D0%BE%D0%B4%D0%B0). Как и рефакторинг, оптимизация обычно не изменяет поведение программы, а только ускоряет её работу. Но оптимизация часто затрудняет понимание кода, что противоположно рефакторингу.

С другой стороны, нужно отличать рефакторинг от [реинжиниринга](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%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), который осуществляется для расширения функциональности программного обеспечения. Как правило, крупные рефакторинги предваряют реинжиниринг.

### Трудности при отсутствии рефакторинга

Первая сложность состоит в том, что новички на проекте не смогут ничего понять в коде, а постоянный коллектив на одном проекте – это слишком идеальная ситуация, в любом случае имеет место текучка кадров.

Также трудности могут возникнуть на этапе введения нового функционала или расширения старого. Все равно придется провести рефакторинг хотя бы тех кусков, которые затронули новые изменения.

Из-за политики «сделать хоть как, не рефакторить впринципе» очень сильно может пострадать производительность программы.

Интересные мысли в статьях:

<http://www.codeart.ru/2010/12/05/refaktorit-nuzhno-ne-vsegda/>

<http://www.codeart.ru/2010/10/25/refaktorit-ili-ne-refaktorit/>

### Причины применения рефакторинга

Рефакторинг нужно применять постоянно при разработке кода. Основными стимулами его проведения являются следующие задачи:

1. необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение;
2. необходимо исправить ошибку, причины возникновения которой сразу не ясны;
3. преодоление трудностей в командной разработке, которые обусловлены сложной логикой программы.

### Примеры плохого кода

Во многом при рефакторинге лучше полагаться на интуицию, основанную на опыте. Тем не менее имеются некоторые видимые проблемы в коде ([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) code smells), требующие рефакторинга:

1. [дублирование кода](https://ru.wikipedia.org/wiki/%D0%94%D1%83%D0%B1%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0);
2. длинный метод;
3. большой класс;
4. длинный список параметров;
5. «жадные» функции — это метод, который чрезмерно обращается к данным другого объекта;
6. избыточные временные переменные;
7. классы данных;
8. несгруппированные данные.

### Методы рефакторинга

*Наиболее употребимые методы рефакторинга:*

* Изменение [сигнатуры метода](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%B3%D0%BD%D0%B0%D1%82%D1%83%D1%80%D0%B0_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8" \o "Сигнатура функции) (Change Method Signature)
* [Инкапсуляция](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_(%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)) [поля](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Поле данных (информатика)) (Encapsulate Field)
* Выделение [класса](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%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)" \o "Класс (программирование)) (Extract Class)
* Выделение [интерфейса](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%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%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)" \o "Интерфейс (объектно-ориентированное программирование)) (Extract Interface)
* Выделение локальной переменной (Extract Local Variable)
* Выделение [метода](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D1%8F%D0%B7%D1%8B%D0%BA%D0%B8_%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)) (Extract Method)

«Выделение метода» - один из наиболее часто проводимых типов рефакторинга. Находим метод, кажущийся слишком длинным, или код, требующий комментариев, объясняющих его назначение. Тогда преобразуем этот фрагмент кода в отдельный метод.

По ряду причин предпочтительны короткие методы с осмысленными именами. Во-первых, если выделен мелкий метод, повышается вероятность его использования другими методами. Во-вторых, методы более высокого уровня начинают выглядеть как ряд комментариев. Замена методов тоже упрощается, когда они мелко структурированы.

* Генерализация типа (Generalize Type)
* Встраивание (Inline)
* Введение [фабрики](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B1%D1%80%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)" \o "Фабричный метод (шаблон проектирования)) (Introduce Factory)
* Введение [параметра](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80_(%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)" \o "Параметр (программирование)) (Introduce Parameter)
* Подъём метода (Pull Up Method)
* Спуск метода (Push Down Method)
* Переименование метода (Rename Method)
* Перемещение метода (Move Method)
* Замена условного оператора [полиморфизмом](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%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)) (Replace Conditional with Polymorphism)

### Пример рефакторинга

Проблема

У вас есть условный оператор, который, в зависимости от типа или свойств объекта, выполняет различные действия.

class Bird {

//...

double getSpeed() {

switch (type) {

case EUROPEAN:

return getBaseSpeed();

case AFRICAN:

return getBaseSpeed() - getLoadFactor() \* numberOfCoconuts;

case NORWEGIAN\_BLUE:

return (isNailed) ? 0 : getBaseSpeed(voltage);

}

throw new RuntimeException("Should be unreachable");

}

}

Решение

Создайте подклассы, которым соответствуют ветки условного оператора. В них создайте общий метод и переместите в него код из соответствующей ветки условного оператора. Впоследствии замените условный оператор на вызов этого метода. Таким образом, нужная реализация будет выбираться через полиморфизм в зависимости от класса объекта.

abstract class Bird {

//...

abstract double getSpeed();

}

class European extends Bird {

double getSpeed() {

return getBaseSpeed();

}

}

class African extends Bird {

double getSpeed() {

return getBaseSpeed() - getLoadFactor() \* numberOfCoconuts;

}

}

class NorvegianBlue extends Bird {

double getSpeed() {

return (isNailed) ? 0 : getBaseSpeed(voltage);

}

}

// Somewhere in client code

speed = bird.getSpeed();

### Причины рефакторинга

Этот рефакторинг может помочь, если у вас в коде есть условные операторы, которые выполняют различную работу, в зависимости от:

* класса объекта или интерфейса, который он реализует;
* значения какого-то из полей объекта;
* результата вызова одного из методов объекта.

При этом если у вас появится новый тип или свойство объекта, нужно будет искать и добавлять код во все схожие условные операторы. Таким образом, польза от данного рефакторинга увеличивается, если условных операторов больше одного, и они разбросаны по всем методам объекта.

Достоинства

* Этот рефакторинг реализует принцип *говори, а не спрашивай*: вместо того, чтобы спрашивать объект о его состоянии, а потом выполнять на основании этого какие-то действия, гораздо проще просто сказать ему, что нужно делать, а как это делать он решит сам.
* Убивает дублирование кода. Вы избавляетесь от множества почти одинаковых условных операторов.

Если вам потребуется добавить новый вариант выполнения, все, что придётся сделать, это добавить новый подкласс, не трогая существующий код (*принцип открытости/закрытости*).

* Замена [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_(%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)" \o "Наследование (программирование)) [делегированием](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D0%B5%D0%B3%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%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)" \o "Делегирование (программирование)) (Replace Inheritance with Delegation)
* Замена кода типа подклассами (Replace Type Code with Subclasses)

### Авторефакторинг в различных IDE

Существует множество различных IDE, и не хватит целой книги, чтобы описать возможности авторефакторинга во всех, так что приведем пример *IntelliJ IDEA* и некоторых ее интересных возможностей.

* Ctrl + Alt + L - Сделать форматирование кода
* Ctrl + Alt + O - Удалить неиспользуемые импорты
* Alt + F7 - Найти использования кода
* Shift + F6 - Переименовать
* Ctrl + F6 - Изменить сигнатуру
* Ctrl + Alt + N - Встроить
* Ctrl + Alt + M - Поместить в метод
* Ctrl + Alt + V - Поместить в переменную
* Ctrl + Alt + F - Поместить в поле
* Ctrl + Alt + C - Поместить в константу
* Ctrl + Alt + P - Поместить в параметр
* И проч.

### Литература по рефакторингу

* Refactoring: Improving the Design of Existing Code («Рефакторинг: улучшение существующего кода»).Martin Fowler, Kent Beck, John Brant и William Opdyke (Мартин Фаулер, Кент Бек, Джон Брант и Вильям Опдайк)
* Чистый код. Создание, анализ и рефакторинг. Робин Мартин.
* Эффективная работа с унаследованным кодом. Майкл К. Физерс.