## ***Тема 3.2. Парадигмы технологии программирования***

[**3.2.1. Основные определения**](#_3.2.1._Основные_определения)[**3.2.2. Императивное программирование**](#_3.2.2._Императивное_программировани)[**3.2.3. Процедурное программирование**](#_3.2.3._Процедурное_программирование)[**3.2.4. Структурное программирование**](#_3.2.4._Структурное_программирование)[**3.2.5. Объектно-ориентированное программирование**](#_3.2.5.__Объектно-ориентированное)[**3.2.6. Функциональное программирование**](#_3.2.6.__Функциональное)[**3.2.7. Логическое программирование**](#_3.2.7.__Логическое)[**3.2.8. Декларативное программирование**](#_3.2.8._Декларативное_программирован)[**3.2.9. Cобытийно-управляемое программирование**](#_3.2.9._Событийно-управляемое_програ)

### **3.2.1. Основные определения**

Что же определяет ***структуру и стиль программных систем*** вообще и в частности проектов в среде **VS .Net**? Прежде всего, ***парадигма программирования***, которой необходимо следовать, работая в любой среде программирования.

***Парадигма программирования*** представляет и определяет то, как программист видит процесс создания и выполнения программного проекта – способ мышления, как представляются задачи, подлежащие решению на компьютере с помощью конкретной среды и языка программирования, как все должно быть структурировано и организовано.

В данном пособии предполагается, что используется среда программирования   
**VS .NET.** А поскольку в основе этой среды лежат различные технологии, в том числе технологии ***императивного программирования, структурного программирования, процедурного программирования, визуального программирования и объектно-ориентированного программирования и других технологий,*** то необходимо следовать, прежде всего, правилам этих технологий.

Таким образом, *«****парадигмы разные нужны, парадигмы разные важны****».* Все они – всего лишь различные инструменты, которые можно использовать при разработки ПО. Каждый из этих инструментов по-своему хорош. Но это не значит, что они «***эффективно универсальны***». То есть, на самом деле, различные методики «программирования» дают разный выигрыш для решения задач разных классов. Напомним, что этот выигрыш можно измерять по двум параметрам:

* эффективность программного обеспечения на современных ЭВМ;
* общие затраты на разработку программного обеспечения.

***Программирование*** – ***процесс создания компьютерных программ с помощью языков программирования. Программирование сочетает в себе элементы искусства, науки, математики и инженерии.***

В узком смысле***, программирование рассматривается как кодирование – реализация одного или нескольких взаимосвязанных алгоритмов на некотором языке программирования.***

В широком смысле***, программирование – процесс создания программ, то есть разработка программного обеспечения***. Причем большая часть работы программиста связана с написанием исходного кода проекта на одном из языков программирования.

Различные ***языки программирования*** поддерживают различные ***стили программирования (парадигмы программирования)***. Отчасти, искусство программирования состоит в том, чтобы выбрать один из языков программирования, наиболее полно подходящий для решения имеющейся задачи. Разные языки требуют от программиста различного уровня внимания к деталям при реализации алгоритма, результатом чего часто бывает компромисс между простотой и производительностью или между временем программиста и временем пользователя.

Напомним, что единственный язык, напрямую воспринимаемый процессором – это ***машинный язык.*** Программа, написанная на этом языке, называется ***машинным кодом***. Изначально, все программисты прорабатывали каждую мелочь в машинном коде, но сейчас эта трудная работа уже не делается. Вместо этого программисты пишут исходный код на языке высокого уровня, и компьютер (используя ***компилятор, интерпретатор*** или ***ассемблер***) ***транслирует*** его, в один или несколько этапов, уточняя все детали, в машинный код, готовый к исполнению на центральном процессоре. Если требуется полный низкоуровневый контроль над системой, программисты пишут программы на языке ассемблера, ***мнемонические инструкции*** которого преобразуются один к одному в соответствующие инструкции машинных команд центрального процессора.

Полезно напомнить также, что практически все современные компьютеры построены по принципам, заложенным еще **Фон Нейманом** в середине нашего века. То есть, есть процессор, есть память, есть внешние устройства, и все это работает под управлением последовательной выборки команд из памяти.

Кроме того, студенты, знакомые с архитектурой ЭВМ, должны знать, что компьютеры, на самом деле, недалеко ушли по своему внутреннему устройству от гипотетических последовательных вычислительных моделей, использовавшихся в начале века пионерами исследования алгоритмов для доказательств базовых утверждений современной ***теории вычислимости алгоритмов***.

В самом деле, ***процессор*** – это система логических элементов. То есть, – система булевых элементов, реализующая функцию переходов некоторого автомата (в данном случае сам компьютер и есть этот автомат) на множестве состояний (вся память процессора, какая есть, и будет этим самым состоянием). Поэтому современные компьютеры практически все ориентированы на последовательные вычисления. Следовательно, парадигмой, имеющей наиболее «естественную» реализацию семантики на современных компьютерах, является ***императивное программирование***. Оно заведомо выигрывает любой другой методологии в эффективности реализации. Хорошие трансляторы, например, с чистых объектно-ориентированных языков – вещь достаточно редкая.

Практически любой язык программирования в наши дни – это язык определений. Программы представляют собой множество определений программных объектов (типов данных, функций и т.д.), которые как-то взаимосвязаны. Причем методологии программирования, как правило, фокусируются на описании алгоритмической части определений, входящих в программу. Методология для работы с описательной частью всего одна: программа должна быть максимально структурирована. Это помогает многократно использовать единожды написанный код, да и просто облегчает понимание текста программы. Структурированность программы позволяет повысить уровень декларативности (то есть, еще сильнее оторваться от деталей архитектуры конкретного компьютера и программировать практически в терминах предметной области задачи). В алгоритмической же части многие современные языки на самом деле поддерживают в явном виде несколько парадигм программирования.

Итак, ***парадигма программирования***  – это совокупность идей и понятий, определяющая стиль написания программ. Парадигма, в первую очередь, определяется базовой программной единицей и самим принципом достижения модульности программы. В качестве этой единицы выступают:

* определение ([***декларативное***](http://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)***,*** [***функциональное программирование***](http://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));
* действие ([***императивное программирование***](http://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));
* правило (***продукционное программирование***);
* [диаграмма](http://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) переходов ([***автоматное программирование***](http://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%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))
* и др. сущности.

Кроме того в современном программировании очень часто парадигма программирования определяется набором инструментов программиста, а именно, [***языком программирования***](http://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) и используемыми [***библиотеками***](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B9_%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).

Парадигма программирования определяет и то, в каких терминах описывается логика программы. Например, в [***императивном программировании***](http://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) программа описывается как последовательность действий, в [***функциональном программировании***](http://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) представляется в виде выражения и множества определений функций, а в [***объектно-ориентированном программировании***](http://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%D0%B8%D0%B5) (**ООП**) программу принято рассматривать как набор взаимодействующих объектов. **ООП** есть, по сути, императивное программирование, дополненное принципом инкапсуляции данных и методов в объект (принцип модульности) и наследованием (принципом повторного использования разработанного функционала).

В тоже время, важно отметить, что парадигма программирования не определяется однозначно языком программирования  – многие современные языки программирования являются мультипарадигменными, то есть допускают использование различных парадигм.

Существует множество моделей, подходов и приёмов программирования, наиболее известными из которых являются следующие:

* [**Императивное программирование**](http://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)**;**
* [**Процедурное программирование**](http://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)**;**
* [**Структурное программирование**](http://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)**;**
* [**Объектно-ориентированное программирование;**](http://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%D0%B8%D0%B5)
* [**Функциональное программирование**](http://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)**;**
* [**Логическое программирование**](http://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)**;**
* [**Декларативное программирование**](http://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)**;**
* [**Событийно-управляемое программирование**](http://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%D0%BD%D0%B8%D0%B5)**.**

### **3.2.2. Императивное программирование**

***Императивное программирование*** – это [парадигма программирования](http://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](http://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](http://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 программы. Императивная программа очень похожа на приказы, выражаемые повелительным наклонением в [естественных языках](http://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](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80).

Одна из характерных черт императивного программирования – наличие переменных с операцией «**разрушающего присвоения**»". То есть, имеется переменная **А**, которая содержит значение **Х**. На очередном шаге алгоритм предписывает присвоить переменной А значение Y. То значение, которое было у переменной **А**, будет «навсегда забыто». На практике это означает «переход между состояниями под управлением функции переходов».

[Императивные языки программирования](http://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) противопоставляются также [функциональным](http://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%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) и [логическим](http://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%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) языкам программирования. Функциональные языки не представляют собой последовательность инструкций и не имеют глобального состояния. То есть, они обычно определяют, ***что*** надо вычислить, а не ***как*** это надо делать.

Первыми императивными языками были машинные коды – «родной язык» программирования для компьютера. В этих языках инструкции были крайне просты, что снижало нагрузку на компьютеры, однако затрудняло написание крупных программ. В [1954](http://ru.wikipedia.org/wiki/1954) появился первый язык программирования - [FORTRAN](http://ru.wikipedia.org/wiki/FORTRAN), а затем [ALGOL](http://ru.wikipedia.org/wiki/ALGOL), [Pascal](http://ru.wikipedia.org/wiki/Pascal) и [C](http://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).

Императивное программирование наиболее пригодно для реализации небольших подзадач, где очень важна скорость исполнения на современных компьютерах. Кроме этого, работа с внешними устройствами, как правило, описывается в терминах последовательного исполнения операций («открыть кран, набрать воды»), что делает такие задачи идеальными кандидатами на императивную реализацию.

### **3.2.3. Процедурное программирование**

***Процедурное программирование* –**  это [парадигма программирования](http://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 концепции ***описания*** и ***вызова процедур***. ***Процедуры*** также известны как [***подпрограммы***](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), ***методы*** или ***функции*** (это не математические функции). Процедуры просто содержат последовательность шагов для выполнения конкретной функциональной задачи или ее части. При этом в ходе выполнения программы любая процедура может быть вызвана из любой точки произвольное число раз. Процедурное программирование **–** это лучший выбор, чем просто последовательное или [***неструктурированное программирование***](http://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%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).

**Процедурное (императивное) программирование** является также отражением архитектуры традиционных [ЭВМ](http://ru.wikipedia.org/wiki/%D0%AD%D0%92%D0%9C). Теоретической моделью процедурного программирования служит алгоритмическая система под названием [Машина Тьюринга](http://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).

Программа на процедурном языке программирования состоит из последовательности операторов (инструкций), задающих процедуру решения задачи. Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, то есть значений исходных данных, в заключительное, то есть в результаты. Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней.

Процедурными языками программирования являются следующие: [**Basic**](http://ru.wikipedia.org/wiki/Basic) (начиная с [**Quick Basic**](http://ru.wikipedia.org/w/index.php?title=Quick_Basic&action=edit&redlink=1) до появления [**Visual Basic**](http://ru.wikipedia.org/wiki/Visual_Basic)), [**Си**](http://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)**,** [**КОБОЛ**](http://ru.wikipedia.org/wiki/%D0%9A%D0%9E%D0%91%D0%9E%D0%9B)**,** [**Фортран**](http://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D1%82%D1%80%D0%B0%D0%BD)**,** [**Pascal**](http://ru.wikipedia.org/wiki/Pascal) и т.д.

### **3.2.4. Структурное программирование** С*труктурное программирование – методология разработки* [*ПО*](http://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), в основе которой лежит представление программы в виде *иерархической структуры* [*блоков*](http://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA_%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). Она предложена в 70-х годах XX века [Э. Дейкстрой](http://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B0,_%D0%AD%D0%B4%D1%81%D0%B3%D0%B5%D1%80_%D0%92%D0%B0%D0%B9%D0%B1), разработана и дополнена [Н. Виртом](http://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82,_%D0%9D%D0%B8%D0%BA%D0%BB%D0%B0%D1%83%D1%81).

В соответствии с этой методологией

1. Любая программа представляет собой структуру, построенную из трёх типов базовых конструкций:

* ***последовательность*** – однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
* ***разветвление*** – однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;
* ***цикл*** – многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла); в программе базовые конструкции могут быть вложены друг в друга произвольным образом, но никаких других средств управления последовательностью выполнения операций не предусматривается.

1. Повторяющиеся фрагменты программы (либо не повторяющиеся, но представляющие собой логически целостные вычислительные блоки) могут оформляться в виде т. н. [подпрограмм](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) ([процедур](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0) или [функций](http://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)). В этом случае в тексте основной программы, вместо помещённого в подпрограмму фрагмента, вставляется инструкция вызова подпрограммы. При выполнении такой инструкции выполняется вызванная подпрограмма, после чего исполнение программы продолжается с инструкции, следующей за командой вызова подпрограммы.
2. Разработка программы ведётся пошагово, методом «сверху вниз».

Суть метода разработки программы «сверху вниз» заключается в следующем. Сначала пишется текст основной программы, в котором, вместо каждого связного логического фрагмента текста, вставляется вызов подпрограммы, которая будет выполнять этот фрагмент. Вместо настоящих, работающих подпрограмм, в программу вставляются «заглушки», которые ничего не делают. Полученная программа проверяется и отлаживается. После того, как [разработчик](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82) уверен, что подпрограммы вызываются в правильной последовательности (то есть общая структура программы верна), то подпрограммы-заглушки последовательно заменяются реально работающими. Причём разработка каждой подпрограммы ведётся тем же методом, что и основной программы. Разработка заканчивается тогда, когда не останется ни одной «затычки», которая не была бы удалена. Такая последовательность гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством фрагментов, и может быть уверен, что общая структура всех более высоких уровней программы верна. При сопровождении и внесении изменений в программу выясняется, в какие именно процедуры нужно внести изменения, и они вносятся, не затрагивая части программы, непосредственно не связанные с ними. Это позволяет гарантировать, что при внесении изменений и исправлении ошибок не выйдет из строя какая-то часть программы, находящаяся в данный момент вне зоны внимания программиста.

Технология нисходящего проектирования с пошаговой детализацией является неотъемлемой частью создания хорошо структурированных программ. При написании программы с использованием этой технологии вся задача рассматривается как единственное предложение (вершина), выражающее общее назначение программы. Так как вершина редко отображает достаточное количество деталей, на основании которых можно написать программу, то поэтому надо начинать процесс детализации – **функциональной декомпозиции**. Вершина разделяется на ряд более мелких задач (**функций**) в том порядке, в котором эти задачи должны выполнятся. В результате получим первую детализацию. Далее каждая из подзадач разбивается на подзадачи, принадлежащие второму уровню детализации. Программист завершает процесс нисходящей разработки с пошаговой детализацией, когда алгоритм настолько детализирован, чтобы его можно было бы преобразовать в программу.

Важно, что при декомпозиции используются только указанные выше три управляющих конструкции, что позволяет говорить о **структурной декомпозиции** или **структурном проектировании** программ.

Таким образом, пошаговая реализация это тактика разработки программы, а нисходящее проектирование это стратегия программирования.

Методология структурного программирования появилась как следствие возрастания сложности решаемых на компьютерах задач, и соответственного усложнения программного обеспечения. В 70-е годы XX века объёмы и сложность программ достигли такого уровня, что «интуитивная» (неструктурированная, или «рефлекторная») разработка программ, которая была нормой в более раннее время, перестала удовлетворять потребностям практики. Программы становились слишком сложными, чтобы их можно было нормально сопровождать, поэтому потребовалась какая-то систематизация процесса разработки и структуры программ.

Наиболее сильной критике со стороны разработчиков структурного подхода к программированию подвергся оператор [**GOTO**](http://ru.wikipedia.org/wiki/GOTO) (оператор безусловного перехода), имевшийся тогда почти во всех языках программирования. Неправильное и необдуманное использование произвольных переходов в тексте программы приводит к получению запутанных, плохо структурированных программ, по тексту которых практически невозможно понять порядок исполнения и взаимозависимость фрагментов.

Следование принципам структурного программирования сделало тексты программ, даже довольно крупных, нормально читаемыми. Серьёзно облегчилось понимание программ, появилась возможность разработки программ в нормальном промышленном режиме, когда программу может без особых затруднений понять не только её автор, но и другие программисты. Это позволило разрабатывать достаточно крупные для того времени программные комплексы силами коллективов разработчиков, и сопровождать эти комплексы в течение многих лет, даже в условиях неизбежных изменений в составе персонала.

Методология структурной разработки программного обеспечения была признана «самой сильной формализацией 70-х годов».

Перечислим некоторые достоинства структурного программирования:

1. Структурное программирование позволяет значительно сократить число вариантов построения программы по одной и той же спецификации, что значительно снижает сложность программы и, что ещё важнее, облегчает понимание её другими разработчиками.
2. В структурированных программах логически связанные операторы находятся визуально ближе, а слабо связанные – дальше, что позволяет обходиться без [схем-алгоритмов](http://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA-%D1%81%D1%85%D0%B5%D0%BC%D0%B0) и других графических форм изображения.
3. Сильно упрощается процесс [тестирования](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и [отладки](http://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BB%D0%B0%D0%B4%D0%BA%D0%B0) структурированных программ.

### **3.2.5. Объектно-ориентированное программирование**

***Объектно-ориентированное программирование (ООП)*** – [парадигма программирования](http://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 основными концепциями являются понятия [***объектов***](http://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) и [***классов***](http://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*** – это тип, описывающий устройство объектов. Понятие «класс» подразумевает некоторое поведение и способ представления. Понятие «объект» подразумевает нечто, что обладает определённым поведением и способом представления. Говорят, что объект — это экземпляр класса. Класс можно сравнить с чертежом, согласно которому создаются объекты. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области. **Класс** является описываемой на языке терминологии ([пространства имён](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE_%D0%B8%D0%BC%D1%91%D0%BD)) исходного кода моделью ещё не существующей сущности – объекта.

***Объект*** – сущность в [адресном пространстве](http://ru.wikipedia.org/wiki/%D0%90%D0%B4%D1%80%D0%B5%D1%81%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE) вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов [компиляции](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%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) (и [линковки](http://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%BA%D0%BE%D0%B2%D0%BA%D0%B0)) исходного кода на выполнение).

***Прототип*** – это объект-образец, по образу и подобию которого создаются другие объекты.

|  |
| --- |
|  |

**ООП** возникло в результате развития идеологии [процедурного программирования](http://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 процедуры (подпрограммы и функции) их обработки формально не связаны. Кроме того, в современном объектно-ориентированном программировании часто большое значение имеют понятия события (так называемое [событийно-ориентированное программирование](http://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%D0%BD%D0%B8%D0%B5)) и компонента ([компонентное программирование](http://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)).

Первым языком программирования, в котором были предложены принципы объектной ориентированности, была [**Симула**](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D1%83%D0%BB%D0%B0). В момент своего появления (в [1967](http://ru.wikipedia.org/wiki/1967) году), этот язык программирования предложил поистине революционные идеи: объекты, классы, виртуальные методы и др., однако это всё не было воспринято современниками как нечто грандиозное.

В настоящее время количество прикладных языков программирования, реализующих объектно-ориентированную парадигму, является наибольшим по отношению к другим парадигмам.

Структура данных «[класс](http://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)», представляющая собой объектный тип данных, внешне похожа на типы данных процедурно-ориентированных языков, таких, как *структура* в языке **Си** или *запись* в **Паскале** и **Quick Basic**. При этом элементы такой структуры (члены класса) могут сами быть не только данными, но и *методами* (то есть процедурами или функциями). Такое объединение называется [инкапсуляцией](http://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 «о**бъектности**» языка программирования, но ещё не означает его объектной ориентированности – для этого требуется наличие [наследования](http://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](http://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%B2_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0%D1%85_%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).

[***Абстракция данных***](http://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) заключается в том, что объекты представляют собою упрощенное, идеализированное описание реальных сущностей предметной области. Если соответствующие модели адекватны решаемой задаче, то работать с ними оказывается намного удобнее, чем с низкоуровневым описанием всех возможных свойств и реакций объекта.

***Инкапсуляция*** – это принцип, согласно которому любой [класс](http://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) должен рассматриваться как [чёрный ящик](http://ru.wikipedia.org/wiki/%D0%A7%D1%91%D1%80%D0%BD%D1%8B%D0%B9_%D1%8F%D1%89%D0%B8%D0%BA) – пользователь класса должен видеть и использовать только интерфейсную часть класса (т. е. список декларируемых свойств и методов [класса](http://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)) и не вникать в его внутреннюю реализацию. Поэтому данные принято инкапсулировать в классе таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а с помощью методов. Принцип инкапсуляции (теоретически) позволяет минимизировать число связей между [классами](http://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) и, соответственно, упростить независимую реализацию и модификацию классов.

***Сокрытие данных*** – неотделимая часть **ООП**, управляющая областями видимости. Является логическим продолжением инкапсуляции. Целью сокрытия является невозможность для пользователя узнать или испортить внутреннее состояние объекта.

***Наследованием*** называется возможность порождать один [класс](http://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) от другого с сохранением всех свойств и методов класса-предка (прародителя, иногда его называют суперклассом) и добавляя, при необходимости, новые свойства и методы. Набор классов, связанных отношением наследования, называют иерархией. Наследование призвано отобразить такое свойство реального мира, как иерархичность.

***Полиморфизмом*** называют явление, при котором функциям (методам) с одним и тем же именем соответствует разный программный код (полиморфный код) в зависимости от того, объект какого класса используется при вызове данного метода. Полиморфизм обеспечивается тем, что в классе-потомке изменяют реализацию метода класса-предка с обязательным сохранением [***сигнатуры***](http://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) метода. Это обеспечивает сохранение неизменным [интерфейса](http://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%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) класса-предка и позволяет осуществить связывание имени метода в коде с разными классами – из объекта какого класса осуществляется вызов, из того класса и берётся метод с данным именем. Такой механизм называется динамическим (или поздним) связыванием – в отличие от статического (раннего) связывания, осуществляемого на этапе компиляции .

По мнению одного из «отцов-основателей» **ООП**, объектно-ориентированный подход заключается в следующем наборе основных принципов:

1. Всё является объектом.
2. 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.
3. Каждый объект имеет независимую память, которая состоит из других объектов.
4. Каждый объект является представителем (экземпляром) класса, который выражает общие свойства объектов.
5. В классе задаётся поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.
6. 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.

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***](http://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** – 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**. Первоначально (например, в **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.

Однако общность механизма обмена сообщениями имеет и другую сторону — «полноценная» передача сообщений требует дополнительных накладных расходов, что не всегда приемлемо. Поэтому в большинстве ныне существующих объектно-ориентированных языков программирования используется концепция **«отправка сообщения как вызов метода»** – объекты имеют доступные извне методы, вызовами которых и обеспечивается взаимодействие объектов. 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.

**ООП** ориентировано на разработку крупных программных комплексов, разрабатываемых командой программистов (возможно, достаточно большой). Проектирование системы в целом, создание отдельных компонент 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 основывается на описании структуры и поведения проектируемой системы, то есть, фактически, в ответе на два основных вопроса:

* из каких частей состоит система;
* в чём состоит ответственность каждой из 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).

Кратко рассмотрим технологии родственные объектно-ориентированному программированию.

[***Компонентно-ориентированное программирование***](http://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%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) – 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***](http://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) – это технология, которая сохранив часть черт **ООП**, отказалось от базовых понятий – класса и наследования.

* вместо механизма описания классов и порождения экземпляров язык предоставляет механизм создания объекта и механизм клонирования объектов.
* каждый вновь созданный объект является «экземпляром без класса», то есть каждый объект может стать *прототипом*– быть использован для создания нового объекта с помощью операции *клонирования*, а 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.

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.

Современный объектно-ориентированный язык предлагает, как правило, следующий обязательный набор синтаксических средств:

* 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 **ООП**-языков поддерживают только единичное наследование);
* средства защиты внутренней структуры классов от несанкционированного использования извне – модификаторы доступа к полям и методам, типа **public**, **private** и **protected**;
* 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.

Большинство языков добавляют к указанному минимальному набору те или иные дополнительные средства. В их числе:

* конструкторы, деструкторы;
* свойства;
* индексаторы;
* интерфейсы – как альтернатива множественному наследованию;
* переопределение операторов для классов.

Часть языков, иногда называемых «чисто объектными», целиком построена вокруг объектных средств – в них любые данные являются объектами, любой код – методом какого-либо класса, и невозможно написать программу, в которой не использовались бы объекты. Примеры подобных языков – **C#, Java**. Другие языки включают **ООП**-подсистему в исходно процедурный язык. В них существует возможность программировать, не обращаясь к объектным средствам. Классические примеры – **C++, Pascal** и **VB**.

### **3.2.6. Функциональное программирование**

***Функциональное программирование*** – [парадигма программирования](http://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](http://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](http://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) в математическом понимании. Обычно противопоставляется парадигме [императивного программирования](http://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 последовательность изменения [состояний](http://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5). В отличие от [императивного](http://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) программирования, где одной из базовых концепций является [переменная](http://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), функциональное программирование не предполагает изменяемость данных.

На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции взаимодействуют и изменяют уже определённые данные. Таким образом, в императивном программировании, при вызове одной и той же функции с одинаковыми параметрами можно получить разные данные на выходе, из-за влияния на функцию внешних факторов. А в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат в обоих случаях, входные данные не могут измениться, выходные данные зависят только от них. Особенностью функциональных языков является то, что [цикл](http://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D0%BA%D0%BB_%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) обычно реализуется в виде рекурсии. Строго говоря, в функциональной парадигме программирования нет такого понятия как цикл. Рекурсивные функции вызывают сами себя, позволяя операции выполняться снова и снова.

Наиболее известными [языками функционального программирования](http://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 [Haskell](http://ru.wikipedia.org/wiki/Haskell) – чистый функциональный, [LISP](http://ru.wikipedia.org/wiki/LISP) и [F#](http://ru.wikipedia.org/wiki/F_Sharp) – функциональный язык для платформы [.NET](http://ru.wikipedia.org/wiki/.NET).

Такой широко распространенный декларативный язык как [SQL](http://ru.wikipedia.org/wiki/SQL) содержит некоторые элементы функционального программирования. В нем остерегаются использовать переменные. Языки работы с электронными таблицами также можно рассматривать как функциональные. Многие нефункциональные языки, такие как [C](http://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++](http://ru.wikipedia.org/wiki/C%2B%2B) и [C#](http://ru.wikipedia.org/wiki/C_Sharp) могут вести себя как функциональные при использовании указателей на функцию.

### **3.2.7. Логическое программирование**

***Логическое программирование*** – [парадигма программирования](http://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 на разделе [дискретной математики](http://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), который изучает принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате [математической логики](http://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**](http://ru.wikipedia.org/wiki/Prolog)**.**

Первым языком логического программирования был язык [Planner](http://ru.wikipedia.org/wiki/Planner), 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 и обеспечить возможность вывода фактов, без активного использования [стека](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA). Затем был разработан язык [**Prolog**](http://ru.wikipedia.org/wiki/Prolog), который не требовал плана перебора вариантов и был, в этом смысле, упрощением языка [**Planner**](http://ru.wikipedia.org/wiki/Planner).

### **3.2.8. Декларативное программирование**

***Декларативное программирование*** – термин с двумя различными значениями.

Согласно первому определению, программа «декларативна», если она описывает *каково́* нечто, а не *как его создать*. Например, веб-страницы на [**HTML**](http://ru.wikipedia.org/wiki/HTML) декларативны, так как они описывают, *что* должна содержать страница, а не *как отображать* страницу на экране. Этот подход отличается от языков [***императивного программировани*я**](http://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), требующих от программиста указывать алгоритм для исполнения.

Согласно второму определению, программа «декларативна», если она написана на исключительно [***функциональном***](http://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)***,*** [***логическом***](http://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) или [***константном языке программирования***](http://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%BE%D0%BD%D1%81%D1%82%D0%B0%D0%BD%D1%82%D0%BD%D0%BE%D0%BC_%D1%8F%D0%B7%D1%8B%D0%BA%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%D1%8F&action=edit&redlink=1)***.*** Выражение «декларативный язык» иногда употребляется для описания всех таких языков программирования как группы, чтобы подчеркнуть их отличие от [***императивных***](http://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) языков.

Программы на языках декларативного программирования легко поддаются методикам метапрограммирования – когда программа может генерироваться по её описанию. Например, **XSLT**-программа может быть сгенерирована из файла **XML**.

Таким образом, декларативное программирование подразумевает использование данных, а не написание кода для того, чтобы заставить приложение или компонент выполнить что-либо. Написание исходного кода иногда называют [*императивным программированием*](http://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). Очевидно, со временем декларативное программирование получит еще большее распространение. Примеры подобных технологий уже сейчас можно увидеть в технологиях [***Microsoft ASP.NET***](http://ru.wikipedia.org/wiki/ASP.NET) и [***Microsoft Windows Communication Foundation***](http://ru.wikipedia.org/wiki/Windows_Communication_Foundation). Даже в [***Microsoft Windows PresentationFoundation***](http://ru.wikipedia.org/wiki/Windows_Presentation_Foundation) программисты могут разрабатывать пользовательский интерфейс, объявляя (декларируя) его разметку и поведение, используя язык разметки**.**

### **3.2.9. Событийно-управляемое программирование**

***Событийно-(ориентированное)управляемое??? программирование*** – это способ построения компьютерной [программы](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), при котором в коде (как правило, в головной функции программы) явным образом выделяется **главный цикл приложения**, тело которого состоит из двух частей: **выборки события** и **обработки события**.

Как правило, в реальных задачах оказывается недопустимым длительное выполнение обработчика события, поскольку при этом программа не может реагировать на другие события. В связи с этим при написании событийно-ориентированных программ часто применяют [***автоматное программирование***](http://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%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)***.***