Понятие алгоритма. Программирование

Любое управление процессом требует определенных правил и четких действий. Компьютер – это устройство, предназначенное для автоматизации создания, хранения, обработки и передачи данных, а значит здесь должны выполняться четкие предписания для выполнения той или иной задачи.

Для создания программ, предназначенной для решения на ЭВМ какой-либо задачи, требуется составление алгоритма ее решения.

> **Алгоритм** является описанием информационного процесса, связанного с изменением состояния объекта (от начального состояния к конечному), в виде последовательности элементарных команд.

Алгоритмами, например, являются правила сложения, умножения, решения алгебраических уравнений, умножения матриц и т.п. Слово алгоритм происходит от algoritmi, являющегося латинской транслитерацией арабского имени хорезмийского математика IX века аль-Хорезми. Благодаря латинскому переводу трактата аль-Хорезми европейцы в XII веке познакомились с позиционной системой счисления, и в средневековой Европе алгоритмом называлась десятичная позиционная система счисления и правила счета в ней.

Иными словами, алгоритм – это точная инструкция, а инструкции встречаются практически во всех областях человеческой деятельности. Возможны алгоритмы проведения физического эксперимента, сборки шкафа или телевизора, обработки детали. Однако не всякая инструкция есть алгоритм.

Инструкция становится алгоритмом только тогда, когда она удовлетворяет определенным требованиям. Одно из требований алгоритма однозначность, т.е. если при применении к одним и тем же данным он даст один и тот же результат.

Применительно к ЭВМ алгоритм позволяет формализовать вычислительный процесс, начинающийся с обработки некоторой совокупности возможных исходных данных и направленный на получение определенных этими исходными данными результатов. Термин вычислительный процесс распространяется и на обработку других видов информации, например, символьной, графической или звуковой.

Если вычислительный процесс заканчивается получением результатов, то говорят, что данный алгоритм применим к рассматриваемой совокупности исходных данных. В противном случае говорят, что алгоритм неприменим к совокупности исходных данных. Любой применимый алгоритм обладает следующими основными свойствами:

- дискретностью;
- определенностью;
- результативностью;
- массовостью.

Дискретность – последовательное выполнение простых или ранее определённых (подпрограммы) шагов. Преобразование исходных данных в результат осуществляется дискретно во времени.

Определенность состоит в совпадении получаемых результатов независимо от пользователя и применяемых технических средств (однозначность толкования инструкций).

Результативность означает возможность получения результата после выполнения конечного количества операций.

Массовость заключается в возможности применения алгоритма к целому классу однотипных задач, различающихся конкретными значениями исходных данных (разработка в общем виде).

Для задания алгоритма необходимо описать следующие его элементы:

- набор объектов, составляющих совокупность возможных исходных данных, промежуточных и конечных результатов;
- правило начала;
- правило непосредственной переработки информации (описание последовательности действий);
- правило окончания;
- правило извлечения результатов.

Алгоритм всегда рассчитан на конкретного исполнителя. В нашем случае таким исполнителем является ЭВМ. Для обеспечения возможности реализации на ЭВМ алгоритм должен быть описан на языке, понятном компьютеру, то есть на языке программирования.

Понятия алгоритма и программы разграничены не очень чётко. Обычно программой называют окончательный вариант алгоритма решения задачи, ориентированный на конкретного пользователя.

Таким образом, можно дать следующее определение программы для ЭВМ:

> **Программа –** это описание алгоритма и данных на некотором языке программирования, предназначенное для последующего автоматического выполнения.

К основным способам описания алгоритмов можно отнести следующие:

- словесно-формульный (на естественном языке);
- структурный или блок-схемный;
- с использованием специальных алгоритмических языков;
- с помощью граф-схем (граф – совокупность точек и линий, в которой каждая линия соединяет две точки. Точки называются вершинами, линии – рёбрами).

Перед составлением программ чаще всего составляют алгоритм решения поставленной задачи одним из вышеописанных способов.

При **словесно-формульном способе** алгоритм записывается в виде текста с формулами по пунктам, составляющих последовательность действий.

Пусть, например, необходимо найти значение следующего выражения:

$y = 4a – (x + 3)$.

Словесно-формульным способом алгоритм решения этой задачи может быть записан в следующем виде:

1. Ввести значения a и x.
2. Сложить x и 3.
3. Умножить a на 4.

4. Вычесть из 4a сумму (x+3).

5. Вывести y как результат вычисления выражения.

При **блок-схемном описании** алгоритм изображается геометрическими фигурами (блоками), связанными по управлению линиями (направлениями потока) со стрелками. В блоках записывается последовательность действий.

Такой вид записи алгоритма обладает наибольшими достоинствами. Он наиболее нагляден: каждая операция вычислительного процесса изображается отдельной геометрической фигурой. Кроме того, графическое изображение алгоритма наглядно показывает разветвления путей решения задачи в зависимости от различных условий, повторение отдельных этапов вычислительного процесса и другие детали.

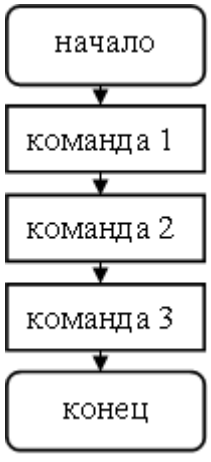
Оформление программ должно соответствовать определенным требованиям (рис. 2.). В настоящее время действует единая система программной документации (ЕСПД), которая устанавливает правила разработки, оформления программ и программной документации. В ЕСПД определены и правила оформления блок-схем алгоритмов (ГОСТ 10.002-80 ЕСПД, ГОСТ 10.003-80 ЕСПД).

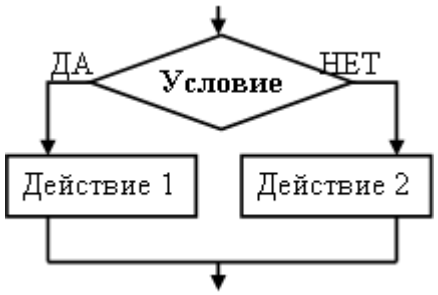
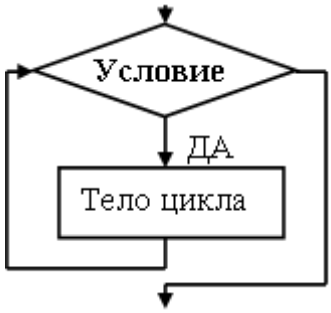
Одним из свойств алгоритма является дискретность, т.е. представление процесса вычислений на отдельные шаги и выделения отдельных участков программы на определенные структуры.

Любой вычислительный процесс может быть представлен как комбинация элементарных алгоритмических структур:

- Следование. Предполагает последовательное выполнение команд сверху вниз. Если алгоритм состоит только из структур следования, то он является линейным.
- Ветвление. Выполнение программы идет по одной из двух, нескольких или множества ветвей. Выбор ветви зависит от условия на входе ветвления и поступивших сюда данных.
- Цикл. Предполагает возможность многократного повторения определенных действий. Количество повторений зависит от условия цикла.
- Функция (подпрограмма). Команды, отделенные от основной программы, выполняются лишь в случае их вызова из основной программы (из любого ее места). Одна и та же функция может вызываться из основной программы сколь угодно раз.

При этом выделят три основных вида алгоритмов:

1) линейные – алгоритмы, в которых команды выполняются последовательно одна за другой;	

2) ветвящиеся – алгоритмы, в которых те или иная серия команд реализуется в зависимости от выполнения условия;	
3) циклические – алгоритмы, в которых серия команд выполняется многократно.	

Система программирования – это система для разработки новых программ на конкретном языке программирования.

Современные системы программирования обычно предоставляют пользователям мощные и удобные средства разработки программ. В них входят:

- компилятор или интерпретатор;
- интегрированная среда разработки;
- средства создания и редактирования текстов программ;
- обширные библиотеки стандартных программ и функций;
- отладочные программы, т.е. программы, помогающие находить и устранять ошибки в программе;
- "дружественная" к пользователю диалоговая среда;
- многооконный режим работы;
- мощные графические библиотеки; утилиты для работы с библиотеками;
- встроенный ассемблер;
- встроенная справочная служба;
- другие специфические особенности.

Методология и технология программирования.

Приведем основные определения.

Программа — завершенный продукт, пригодный для запуска своим автором на системе, на которой он был разработан.

Программный продукт — программа, которую любой человек может запускать, тестировать, исправлять и развивать. Такая программа должна быть написана в обобщенном стиле, тщательно оттестирована и сопровождена подробной документа-

цией. (С учетом модной в настоящее время концепции авторских прав, здесь необходимо уточнить – любой человек, имеющий разрешение работать с исходными текстами программ)

Программный комплекс — набор взаимодействующих программ, согласованных по функциям и форматам, точно определенным интерфейсам, и в целом составляющих полное средство для решения больших задач.

Жизненный цикл программного обеспечения – это весь период его разработки и эксплуатации, начиная с момента возникновения замысла и заканчивая прекращением ее использования.

Методология программирования – совокупность методов, применимых в жизненном цикле программного обеспечения и объединенных общим философским подходом.

Существует четыре широко известных в настоящее время методологии программирования – императивного, объектно-ориентированного, логического, функционального.

Технология программирования изучает технологические процессы и порядок их прохождения – стадии (с использованием знаний, методов и средств).

Процесс — совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Процессы состоят из набора действий, а каждое действие из набора задач. Вертикальное измерение отражает статические аспекты процессов и оперирует такими понятиями, как рабочие процессы, действия, задачи, результаты деятельности и исполнители.

Стадия — часть действий по созданию программного обеспечения, ограниченная некоторыми временными рамками и заканчивающаяся выпуском конкретного продукта, определяемого заданными для данной стадии требованиями. Стадии состоят из этапов, которые обычно имеют итерационный характер. Иногда стадии объединяют в более крупные временные рамки, называемые фазами. Итак, горизонтальное измерение представляет время, отражает динамические аспекты процессов и оперирует такими понятиями, как фазы, стадии, этапы, итерации и контрольные точки.

Технологический подход определяется спецификой комбинации стадий и процессов, ориентированной на разные классы программного обеспечения и на особенности коллектива разработчиков.

2. Императивное программирование.

Императивное программирование — это исторически первая методология программирования, которой пользовался каждый программист, программирующий на любом из «массовых» языков программирования – Basic, Pascal, C.

Она ориентирована на классическую фон Неймановскую модель, остававшуюся долгое время единственной аппаратной архитектурой. Методология императивного программирования характеризуется принципом последовательного изменения состояния вычислителя пошаговым образом. При этом управление изменениями полностью определено и полностью контролируемо.

Методы и концепции.

- Метод *изменения состояний* — заключается в последовательном изменении состояний. Метод поддерживается концепцией алгоритма.
- Метод *управления потоком исполнения* — заключается в пошаговом контроле управления. Метод поддерживается концепцией потока исполнения.

Вычислительная модель. Если под вычислителем понимать современный компьютер, то его состоянием будут значения всех ячеек памяти, состояние процессора (в том числе — указатель текущей команды) и всех сопряженных устройств. Единственная структура данных — последовательность ячеек (пар «адрес» - «значение») с линейно упорядоченными адресами.

В качестве математической модели императивное программирование использует машину Тьюринга-Поста — абстрактное вычислительное устройство, предложенное на заре компьютерной эры для описания алгоритмов.

Синтаксис и семантика. Языки, поддерживающие данную вычислительную модель, являются как бы средством описания функции переходов между состояниями вычислителя. Основным их синтаксическим понятием является оператор. Первая группа — простые операторы, у которых никакая их часть не является самостоятельным оператором (например, оператор присваивания, оператор безусловного перехода, вызова процедуры и т. п.). Вторая группа — структурные операторы, объединяющие другие операторы в новый, более крупный оператор (например, составной оператор, операторы выбора, цикла и т. п.).

Традиционное *средство структурирования* — подпрограмма (процедура или функция). Подпрограммы имеют параметры и локальные определения и могут быть вызваны рекурсивно. Функции возвращают значения как результат своей работы.

Если в данной методологии требуется решить некоторую задачу для того, чтобы использовать ее результаты при решении следующей задачи, то типичный подход будет таким. Сначала исполняется алгоритм, решающий первую задачу. Результаты его работы сохраняются в специальном месте памяти, которое известно следующему алгоритму, и используются им.

Императивные языки программирования. Императивные языки программирования манипулируют данными в пошаговом режиме, используя последовательные инструкции и применяя их к разнообразным данным. Считается, что первым алгоритмическим языком программирования был язык Plankalkuel (от plan calculus), разработанный в 1945—1946 годах Конрадом Цузе (Konrad Zuse).

Большинствои из наиболее известных и распространенных императивных языков программирования было создано в конце 50-х — середине 70-х годов XX века. Это период 80-х и 90-х годов соответствует увлечениям новыми парадигмами, и императивных языков в это время практически не появлялось.

Класс задач. Императивное программирование наиболее пригодно для решения задач, в которых последовательное исполнение каких-либо команд является естественным. Примером здесь может служить управление современными аппаратными средствами. Поскольку практически все современные компьютеры императивны, эта методология позволяет порождать достаточно эффективный исполняемый код. С ростом сложности задачи императивные программы становятся все менее и менее читаемыми.

Программирование и отладка действительно больших программ (например, компиляторов), написанных исключительно на основе методологии императивного программирования, может затянуться на долгие годы.

2.1. Модульное программирование.

Модульное программирование — это такой способ программирования, при котором вся программа разбивается на группу компонентов, называемых модулями, причем каждый из них имеет свой контролируемый размер, четкое назначение и детально проработанный интерфейс с внешней средой. Единственная альтернатива модульности — монолитная программа, что, конечно, неудобно. Таким образом, наиболее интересный вопрос при изучении модульности — определение критерия разбиения на модули.

Концепции модульного программирования. В основе модульного программирования лежат три основных концепции:

Принцип утаивания информации Парнаса. Всякий компонент утаивает единственное проектное решение, т. е. модуль служит для утаивания информации. Подход к разработке программ заключается в том, что сначала формируется список проектных решений, которые особенно трудно принять или которые, скорее всего, будут меняться. Затем определяются отдельные модули, каждый из которых реализует одно из указанных решений.

Аксиома модульности Коуэна. Модуль — независимая программная единица, служащая для выполнения некоторой определенной функции программы и для связи с остальной частью программы. Программная единица должна удовлетворять следующим условиям:

- блочность организации, т. е. возможность вызвать программную единицу из блоков любой степени вложенности;
- синтаксическая обособленность, т. е. выделение модуля в тексте синтаксическими элементами;
- семантическая независимость, т. е. независимость от места, где программная единица вызвана;

- общность данных, т. е. наличие собственных данных, сохраняющихся при каждом обращении;
- полнота определения, т. е. самостоятельность программной единицы.

Сборочное программирование Цейтина. Модули — это программные кирпичи, из которых строится программа. Существуют три основные предпосылки к модульному программированию:

- стремление к выделению независимой единицы программного знания. В идеальном случае всякая идея (алгоритм) должна быть оформлена в виде модуля;
- потребность организационного расчленения крупных разработок;
- возможность параллельного исполнения модулей (в контексте параллельного программирования).

Определения модуля и его примеры. Приведем несколько дополнительных определений модуля.

- Модуль — это совокупность команд, к которым можно обратиться по имени.
- Модуль — это совокупность операторов программы, имеющая граничные элементы и идентификатор (возможно агрегатный).

Функциональная спецификация модуля должна включать:

- синтаксическую спецификацию его входов, которая должна позволять построить на используемом языке программирования синтаксически правильное обращение к нему;
- описание семантики функций, выполняемых модулем по каждому из его входов.

Разновидности модулей. Существуют три основные разновидности модулей:

1) *"Маленькие" (функциональные)* модули, реализующие, как правило, одну какую-либо определенную функцию. Основным и простейшим модулем практически во всех языках программирования является процедура или функция.

2) *"Средние" (информационные)* модули, реализующие, как правило, несколько операций или функций над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Примеры "средних" модулей в языках программирования:

a)задачи в языке программирования Ada;

b)кластер в языке программирования CLU;

c)классы в языках программирования C++ и Java.

3) *"Большие" (логические) модули*, объединяющие набор "средних" или "маленьких" модулей. Примеры "больших" модулей в языках программирования:

a)модуль в языке программирования Modula-2;

b)пакеты в языках программирования Ada и Java.

Набор **характеристик модуля** предложен Майерсом [Майерс 1980]. Он состоит из следующих конструктивных характеристик:

1) размера модуля;

В модуле должно быть 7 (+/-2) конструкций (например, операторов для функций или функций для пакета). Это число берется на основе представлений психологов о среднем оперативном буфере памяти человека. Символьные образы в человеческом мозгу объединяются в "чанки" — наборы фактов и связей между ними, запоминаемые и извлекаемые как единое целое. В каждый момент времени человек может обрабатывать не более 7 чанков.

Модуль (функция) не должен превышать 60 строк. В результате его можно поместить на одну страницу распечатки или легко просмотреть на экране монитора.

2) прочности (связности) модуля;

Существует гипотеза о глобальных данных, утверждающая, что глобальные данные вредны и опасны. Идея глобальных данных дискредитирует себя так же, как и идея оператора безусловного перехода goto. Локальность данных дает возможность легко читать и понимать модули, а также легко удалять их из программы.

Связность (прочность) модуля (cohesion) — мера независимости его частей. Чем выше связность модуля — тем лучше, тем больше связей по отношению к оставшейся части программы он упрятывает в себе. Можно выделить типы связности, приведенные ниже.

Функциональная связность. Модуль с функциональной связностью реализует одну какую-либо определенную функцию и не может быть разбит на 2 модуля с теми же типами связностей.

Последовательная связность. Модуль с такой связностью может быть разбит на последовательные части, выполняющие независимые функции, но совместно реализующие единственную функцию. Например, один и тот же модуль может быть использован сначала для оценки, а затем для обработки данных.

Информационная (коммуникативная) связность. Модуль с информационной связностью — это модуль, который выполняет несколько операций или функций над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Эта информационная связность применяется для реализации абстрактных типов данных.

Обратим внимание на то, что средства для задания информационно прочных модулей отсутствовали в ранних языках программирования (например, FORTRAN и даже в оригинальной версии языка Pascal). И только позже, в языке программирования Ada, появился пакет — средство задания информационно прочного модуля.

3) сцепления модуля с другими модулями;

Сцепление (coupling) — мера относительной независимости модуля от других модулей. Независимые модули могут быть модифицированы без переделки других модулей. Чем слабее сцепление модуля, тем лучше. Рассмотрим различные типы сцепления.

Независимые модули — это идеальный случай. Модули ничего не знают друг о друге. Организовать взаимодействие таких модулей можно, зная их интерфейс и соответствующим образом перенаправив выходные данные одного модуля на вход другого. Достичь такого сцепления сложно, да и не нужно, поскольку сцепление по данным (параметрическое сцепление) является достаточно хорошим.

Сцепление *по данным* (параметрическое) — это сцепление, когда данные передаются модулю, как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Этот вид сцепления реализуется в языках программирования при обращении к функциям (процедурам). Две разновидности этого сцепления определяются характером данным.

- Сцепление по простым элементам данных.
- Сцепление по структуре данных. В этом случае оба модуля должны знать о внутренней структуре данных.

4) рутинности (идемпотентность, независимость от предыдущих обращений) модуля.

Рутинность — это независимость модуля от предыдущих обращений к нему (от предыстории). Будем называть модуль рутинным, если результат его работы зависит только от количества переданных параметров (а не от количества обращений).

Модуль должен быть рутинным в большинстве случаев, но есть и случаи, когда модуль должен сохранять историю. В выборе степени рутинности модуля пользуются тремя рекомендациями.

- В большинстве случаев делаем модуль рутинным, т. е. независимым от предыдущих обращений.
- Зависящие от предыстории модули следует использовать только в тех случаях, когда это необходимо для сцепления по данным.
- В спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость, чтобы пользователи имели возможность прогнозировать поведение такого модуля.

2.2. Структурное программирование.

Структурное программирование (СП) возникло как вариант решения проблемы уменьшения СЛОЖНОСТИ разработки программного обеспечения.

В начале эры программирования работа программиста ничем не регламентировалась. Решаемые задачи не отличались размахом и масштабностью, использовались в

основном машинно-ориентированные языки и близкие к ним язык типа Ассемблера, разрабатываемые программы редко достигали значительных размеров, не ставились жесткие ограничения на время их разработки.

По мере развития программирования появились задачи, для решения которых определялись ограниченные сроки все более сложных задач с привлечением групп программистов. И как следствие, разработчики столкнулись с тем, что методы, пригодные для разработки небольших задач, не могут быть использованы при разработке больших проектов в силу сложности последних.

Таким образом, цель структурного программирования - повышение надежности программ, обеспечение *сопровождения и модификации*, облегчение и ускорение разработки.

Методология структурного императивного программирования — подход, заключающийся в задании хорошей топологии императивных программ, в том числе отказе от использования глобальных данных и оператора безусловного перехода, разработке модулей с сильной связностью и обеспечении их независимости от других модулей.

Подход базируется на двух основных принципах:

- Последовательная декомпозиция алгоритма решения задачи сверху вниз.
- Использование структурного кодирования.

Напомним, что данная методология является важнейшим развитием императивной методологии.

Происхождение, история и эволюция. Создателем структурного подхода считается Эдсгер Дейкстра. Ему также принадлежит попытка (к сожалению, совершенно неприменимая для массового программирования) соединить структурное программирование с методами доказательства правильности создаваемых программ. В его разработке участвовали такие известные ученые как Х. Милс, Д.Э. Кнут, С. Хоор.

Методы и концепции, лежащие в основе структурного программирования. Их три

Метод *алгоритмической декомпозиции сверху вниз* — заключается в пошаговой детализации постановки задачи, начиная с наиболее общей задачи. Данный метод обеспечивает хорошую структурированность. Метод поддерживается концепцией алгоритма.

Метод *модульной организации частей программы* — заключается в разбиении программы на специальные компоненты, называемые модулями. Метод поддерживается концепцией модуля.

Метод *структурного кодирования* — заключается в использовании при кодировании трех основных управляющих конструкций. Метки и оператор безусловного

перехода являются трудно отслеживаемыми связями, без которых мы хотим обойтись. Метод поддерживается концепцией управления

Структурные языки программирования. Основное отличие от классической методологии императивного программирования заключается в отказе (точнее, той или иной степени отказа) **от оператора безусловного перехода**.

[Пратт Т., 1979] "Важным для программиста свойством синтаксиса является возможность *отразить в структуре программы структуру лежащего в ее основе алгоритма.* При использовании для построения программы метода, известного под названием **структурное программирование**, программа конструируется иерархически - сверху вниз (от главной программы к подпрограммам самого нижнего уровня), с употреблением на каждом уровне только ограниченного набора управляющих структур: простых последовательностей инструкций, циклов и некоторых видов условных разветвлений. При последовательном проведении этого метода структуру результирующих алгоритмов легко понимать, отлаживать и модифицировать. В идеале у нас должна появиться возможность перевести построенную таким образом схему программы прямо в соответствующие программные инструкции, отражающие структуру алгоритма."

Теорема о структурировании (Бёма-Джакопини (Boet-Jacopini)): Всякую правильную программу (т.е. программу с одним входом и одним выходом без зацикливаний и недостижимых веток) можно записать с использованием следующих логических структур - последовательность, выбора и повторение цикла

Следствие 1: *Всякую программу можно привести к форме без оператора goto.*

Следствие 2: *Любой алгоритм можно реализовать в языке, основанном на трех управляющих конструкциях -последовательность, цикл, повторение.*

Следствие 3: *Сложность структурированных программ ограничена, даже в случае их неограниченного размера.*

Структурное программирование- это не самоцель. Его основное назначение- это получение хорошей ("правильной") программы, однако даже в самой хорошей программе операторы перехода goto иногда нужны: например - выход из множества вложенных циклов.

Практически на всех языках, поддерживающих императивную методологию, можно разрабатывать программы и по данной методологии. В ряде языков введены специальные заменители оператора goto, позволяющие облегчить управление циклами (например, Break и Continue в языке C).

Класс задач. Класс задач для данной методологии соответствует классу задач для императивной методологии. Заметим, что при этом удается разрабатывать более сложные программы, поскольку их легко воспринимать и анализировать.

3. Метод объектно-ориентированного программирования.

Метод структурного программирования оказался эффективен при написании программ «ограниченной сложности». Однако с возрастанием сложности реализуемых программных проектов и, соответственно, объема кода создаваемых программ, возможности метода структурного программирования оказались недостаточными.

Основной причиной возникших проблем можно считать то, что в программе не отражалась непосредственно структура явлений и понятий реального мира и связей межу ними. При попытке анализа и модификации текста программы программист вынужден был оперировать искусственными категориями.

Чтобы писать все более сложные программы, необходим был новый подход к программированию. В итоге были разработаны принципы Объектно-Ориентированного Программирования. ООР аккумулирует лучшие идеи, воплощённые в структурном программировании, и сочетает их с мощными новыми концепциями, которые позволяют по-новому организовывать ваши программы.

Надо сказать, что теоретические основы ООП были заложены еще в 70-х годах прошлого века, но практическое их воплощение стало возможно лишь в середине 80-х, с появлением соответствующих технических средств.

Методология ООП использует **метод объектной декомпозиции**, согласно которому структура системы (статическая составляющая) описывается в терминах *объектов и связей* между ними, а поведение системы (динамическая составляющая) - в терминах обмена сообщениями между объектами. Сообщения могут быть как реакцией на события, вызываемые как внешними факторами, так и порождаемые самими объектами.

Объектно-ориентированные программы называют «программами, управляемыми от событий», в отличие от традиционных программ, называемых «программам, управляемыми от данных».

Основные **методы и концепции ООП**

- Метод объектно-ориентированной декомпозиции – заключается в выделении объектов и связей между ними. Метод поддерживается концепциями инкапсуляции, наследования и полиморфизма.
- Метод абстрактных типов данных – метод, лежащий в основе инкапсуляции. Поддерживается концепцией абстрактных типов данных.
- Метод пересылки сообщений – заключается в описании поведения системы в терминах обмена сообщениями между объектами. Поддерживается концепцией сообщения.

Вычислительная модель чистого ООП поддерживает только одну операцию – *посылку сообщения объекту*. Сообщения могут иметь параметры, являющиеся объектами. Само сообщение тоже является объектом.

Объект имеет набор обработчиков сообщений (набор методов). У объекта есть поля – персональные переменные данного объекта, значениями которых являются ссылки на другие объекты. В одном из полей объекта хранится ссылка на объект-предок, которому переадресуются все сообщения, не обработанные данным объектом. Структуры, описывающие обработку и переадресацию сообщений, обычно выделяются в отдельный объект, называемый классом данного объекта. Сам объект называется экземпляром указанного класса.

Синтаксис и семантика

В синтаксисе чистых объектно-ориентированных языков все может быть записано в форме посылки сообщений объектам. Класс в объектно-ориентированных языках описывает структуру и функционирование множества объектов с подобными характеристиками, атрибутами и поведением. Объект принадлежит к некоторому классу и обладает своим собственным внутренним состоянием. Методы — функциональные свойства, которые можно активизировать.

В объектно-ориентированном программировании определяют три основных свойства:

Инкапсуляция. Это сокрытие информации и комбинирование данных и функций (методов) внутри объекта.

Наследование. Построение иерархии порожденных объектов с возможностью для каждого такого объекта-наследника доступа к коду и данным всех порождающих объектов-предков. Построение иерархий является достаточно сложным делом, так как при этом приходится выполнять классифицирование.

Большинство окружающих нас объектов относится к категориям, рассмотренным в книге [Шлеер, Меллор 1993]:

- Реальные объекты – абстракции предметов, существующих в физическом мире;
- Роли – абстракции цели или назначения человека, части оборудования или организации;
- Инциденты – абстракции чего-то произошедшего или случившегося;
- Взаимодействия – объекты, получающиеся из отношения между другими объектами.

Полиморфизм (полиморфизм включения) — присваивание действию одного имени, которое затем разделяется вверх и вниз по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, подходящим именно ему.

У каждого объекта есть ссылка на класс, к которому он относится. При приеме сообщения объект обращается к классу для обработки данного сообщения. Сообщение может быть передано вверх по иерархии наследования, если сам класс не располагает методом для его обработки. Если обработчик событий для сообщения выбирается динамически, то методы, реализующие обработчиков событий, принято называть виртуальными.

Естественным средством структурирования в данной методологии являются классы. Классы определяют, какие поля и методы экземпляра доступны извне, как обрабатывать отдельные сообщения и т. п. В чистых объектно-ориентированных языках извне доступны только методы, а доступ к данным объекта возможен только через его методы.

Взаимодействие задач в данной методологии осуществляется при помощи обмена сообщениями между объектами, реализующими данные задачи.

Для поддержки концепции ООР были разработаны специальные **объектно-ориентированные языки** программирования. Все языки ООР можно разделить на три группы.

Чистые языки, в наиболее классическом виде поддерживающие объектно-ориентированную методологию. Такие языки содержат небольшую языковую часть и существенную библиотеку, а также набор средств поддержки времени выполнения.

Гибридные языки, которые появились в результате внедрения объектно-ориентированных конструкций в популярные императивные языки программирования.

Урезанные языки, которые появились в результате удаления из гибридных языков наиболее опасных и ненужных с позиций ООП конструкций.

4. Логическое программирование.

В 70-х годах возникла ветвь языков *декларативного программирования*, связанная с проектами в области искусственного интеллекта, а именно языки логического программирования.

Согласно логическому подходу к программированию, программа представляет собой совокупность правил или логических высказываний. Кроме того, в программе допустимы *логические причинно-следственные связи*, в частности, на основе операции импликации.

Таким образом, языки логического программирования базируются на классической логике и применимы для систем логического вывода, в частности, для так называемых экспертных систем. На языках логического программирования естественно формализуется логика поведения, и они применимы для описаний правил принятия решений, например, в системах, ориентированных на поддержку бизнеса.

Важным преимуществом такого подхода является достаточно высокий уровень машинной независимости, а также возможность откатов – возвращения к предыдущей подцели при отрицательном результате анализа одного из вариантов в процессе поиска решения (скажем, очередного хода при игре в шахматы), что избавляет от необходимости поиска решения путем полного перебора вариантов и увеличивает эффективность реализации.

Одним из недостатков логического подхода в концептуальном плане является специфичность класса решаемых задач.

Другой недостаток практического характера состоит в сложности эффективной реализации для принятия решений в реальном времени, скажем, для систем жизнеобеспечения.

Нелинейность структуры программы является особенностью декларативного подхода и, строго говоря, представляет собой оригинальную особенность, а не объективный недостаток.

в качестве примеров языков логического программирования можно привести Prolog (название возникло от слов PROgramming in LOGic) и Mercury.

5. Функциональное программирование.

Функциональный подход к программированию появился в результате проведения фундаментальных математических исследований.

Время появления теоретических работ, обосновывающих *функциональный подход*, относится к 20-м – 30-м годам XX столетия. Как мы убедимся впоследствии, теория часто значительно опережает практику программирования, и важнейшие работы, которые сформировали математическую основу подхода, были написаны задолго до появления компьютеров и языков программирования, которые потенциально могли бы реализовать эту теорию.

Что касается первой реализации, то она появилась в 50-х годах XX столетия в форме языка LISP, о котором речь пойдет далее.

Важнейшей характеристикой функционального подхода является то обстоятельство, что всякая программа, разработанная на языке функционального программирования, может рассматриваться как функция, аргументы которой, возможно, также являются функциями.

Функциональный подход породил целое семейство языков, родоначальником которых, как уже отмечалось, стал язык программирования LISP. Позднее, в 70-х годах, был разработан первоначальный вариант языка ML, который впоследствии развился, в частности, в SML, а также ряд других языков. Из них, пожалуй, самым «молодым» является созданный уже совсем недавно, в 90-х годах, язык Haskell.

Важным преимуществом реализации языков функционального программирования является автоматизированное динамическое распределение памяти компьютера для хранения данных. При этом программист избавляется от необходимости контролировать данные, а если потребуется, может запустить функцию **«сборки мусора»** – очистки памяти от тех данных, которые больше не понадобятся программе.

Сложные программы при функциональном подходе строятся посредством агрегирования функций. При этом текст программы представляет собой функцию, некоторые аргументы которой можно также рассматривать как функции. Таким образом, повторное использование кода сводится к вызову ранее описанной функции, структура которой, в отличие от процедуры императивного языка, математически прозрачна.

Поскольку функция является естественным формализмом для языков функционального программирования, реализация различных аспектов программирования, связанных с функциями, существенно упрощается. Интуитивно прозрачным становится написание **рекурсивных функций**, т.е. функций, вызывающих самих себя в качестве аргумента. Естественной становится и реализация обработки рекурсивных структур данных.

Благодаря реализации механизма сопоставления с образцом, такие языки функционального программирования как ML и Haskell хорошо использовать для символьной обработки.

Естественно, языки функционального программирования не лишены и некоторых недостатков.

Часто к ним относят нелинейную структуру программы и относительно невысокую эффективность реализации. Однако первый недостаток достаточно субъективен, а второй успешно преодолен современными реализациями, в частности, рядом последних трансляторов языка SML, включая и компилятор для среды Microsoft .NET.

Для профессиональной разработки программного обеспечения на языках функционального программирования необходимо глубоко понимать природу функции.

Заметим, что под термином «функция» в математической формализации и программной реализации имеются в виду различные понятия.

Функцией в языке программирования называется конструкция этого языка, описывающая правила преобразования аргумента (так называемого фактического параметра) в результат.

Для формализации понятия «функция» была построена математическая теория, известная под названием ламбда-исчисления. Более точно это исчисление следует именовать **исчислением ламбда-конверсий.**

Под **конверсией** понимается преобразование объектов исчисления (а в программировании – функций и данных) из одной формы в другую. Исходной задачей в математике было стремление к упрощению формы выражений. В программировании именно эта задача не является столь существенной, хотя, как мы увидим в дальнейшем, использование ламбда-исчисления как исходной формализации может способствовать упрощению вида программы, т.е. вести к оптимизации программного кода.

Кроме того, конверсии обеспечивают переход к вновь введенным обозначениям и, таким образом, позволяют представлять предметную область в более компактном либо более детальном виде, или, говоря математическим языком, изменять уровень абстракции по отношению к предметной области. Эту возможность широко используют также языки объектно-ориентированного и структурно-модульного программирования в иерархии объектов, фрагментов программ и структур данных. На этом же принципе основано взаимодействие компонентов приложения в .NET. Именно в этом смысле переход к новым обозначениям является одним из важнейших элементов программирования в целом, и именно ламбда-исчисление (в отличие от многих дру-

гих разделов математики) представляет собой адекватный способ формализации переобозначений.

Систематизируем эволюцию теорий, лежащих в основе современного подхода к ламбда-исчислению.

Рассмотрим эволюцию языков программирования, развивающихся в рамках *функционального подхода*.

Ранние языки функционального программирования, которые берут свое начало от классического языка LISP (LISt Processing), были предназначены, для обработки списков, т.е. символьной информации. При этом основными типами были атомарный элемент и список из атомарных элементов, а основной акцент делался на анализе содержимого списка.

Развитием ранних языков программирования стали языки функционального программирования с сильной типизацией, характерным примером здесь является классический ML, и его прямой потомок SML. В языках с сильной типизацией каждая конструкция (или выражение) должна иметь тип.

При этом в более поздних языках функционального программирования нет необходимости в явном приписывании типа, и типы изначально неопределенных выражений, как в SML, могут выводиться (до запуска программы), исходя из типов связанных с ними выражений.

Следующим шагом в развитии языков функционального программирования стала поддержка *полиморфных функций*, т.е. функций с параметрическими аргументами (аналогами математической функции с параметрами). В частности, полиморфизм поддерживается в языках SML, Miranda и Haskell.

На современном этапе развития возникли языки функционального программирования «нового поколения» со следующими расширенными возможностями: сопоставление с образцом (Scheme, SML, Miranda, Haskell), параметрический полиморфизм (SML) и так называемые «ленивые» (по мере необходимости) вычисления (Haskell, Miranda, SML).

Семейство языков функционального программирования довольно многочисленно. Об этом свидетельствует не столько значительный список языков, сколько тот факт, что многие языки дали начало целым направлениям в программировании. Напомним, что LISP дал начало целому семейству языков: Scheme, InterLisp, COMMON Lisp и др.

Не стал исключением и язык программирования SML, который был создан в форме языка ML Р. Милнером (Robin Milner) в MIT (Massachusetts Institute of Technology) и первоначально предназначен для логических выводов, в частности, доказательства теорем. Язык отличается строгой типизацией, в нем отсутствует параметрический полиморфизм.

Развитием «классического» ML стали сразу три современных языка с практически одинаковыми возможностями (параметрический полиморфизм, сопоставление с образцом, «ленивые» вычисления). Это язык SML, разработанный в Великобритании и США, CaML, созданный группой французских ученых института INRIA, SML/NJ – диалект SML из New Jersey, а также российская разработка – mosml («московский» диалект ML).

Близость к математической формализации и изначальная функциональная ориентированность послужили причиной следующих преимуществ *функционального подхода:*

1. *простота тестирования и верификации программного кода* на основе возможности построения строгого математического доказательства корректности программ;
2. *унификация представления программы и данных* (данные могут быть инкапсулированы в программу как аргументы функций, означивание или вычисление значения функции может производиться по мере необходимости);
3. *безопасная типизация:* недопустимые операции с данными исключены;
4. *динамическая типизация:* возможно обнаружение ошибок типизации во время выполнения (отсутствие этого свойства в ранних языках функционального программирования может приводить к переполнению оперативной памяти компьютера);
5. *независимость программной реализации от машинного представления данных и системной архитектуры программы* (программист сосредоточен на деталях реализации, а не на особенностях машинного представления данных).