**Содержание**

[1. Алгоритм: базовые конструкции, способы задания, оценка временной сложности на примере алгоритмов поиска](#h.w1g6utvik2fc)

[2. Тип данных. Структуры данных. Классификация структур данных](#h.315hmzjxgn9x)

[3. Базовые и улучшенные сортировки на основе выбора, включения и обмена. Сравнительный анализ алгоритмов сортировки](#h.ladilxi2x1n3)

[4. Структуры данных типа стек. Реализация стека как отображения на массив и односвязный список. Примеры применения](#h.w78q7dgjcbeq)

[5. Структуры данных типа очередь. Реализация очереди как отображение на массив и односвязный список. Примеры применения.](#h.15y4gwh6nupw)

[6. Структуры данных типа таблица. Прямого доступа, хеш-таблица. Разрешение коллизий с помощью цепочек и открытой адресации и анализ их алгоритмов](#h.4vj7oo81u386)

[7. Структуры данных бинарное дерево. Операции включения, исключения. Алгоритмы поиска и прохождения](#h.g1yr5lr3bnvj)

[8. Программные продукты как сложные системы. Признаки сложных систем](#h.5one8lxve30g)

[9. Декомпозиция. Преимущества и недостатки объектно-ориентированного подхода в программировании](#h.opxhu5m8vbhy)

[9. Основные понятия объектно-ориентированного программирования: объект, класс, виды отношений между классами](#h.bd4co06wmsuj)

[10. Основные принципы объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм](#h.hffu154yrxs4)

[11. Общая характеристика объектов в объектно-ориентированном программировании. Объектная декомпозиция. Виды отношений между объектами](#h.kt39txk0c7wp)

[12. Массивы, коллекции, обобщения: основные особенности, реализуемые интерфейсы.](#h.9dto3771kt7w)

[13. Понятие «базы данных». Основные компоненты базы данных](#h.ipki02lfg55r)

[14. Системы баз данных. Архитектура систем баз данных. Современные СУБД.](#h.h2m7z36cm66j)

[15. Нормальные формы БД. Нормализация данных.](#h.k2twcyhxorhz)

[16. Инфологическая модель данных. Диаграмма «сущность — связь».](#h.54lm0e2cnhcx)

[17. Язык SQL для работы с реляционными базами данных.](#h.ciy5dif8sigw)

[18. Хранимые процедуры, триггеры, транзакции.](#h.rxxmt4js4j0x)

[19. Поиск информации в базе данных. Полнотекстовый поиск.](#h.or477r1j9bbz)

[20. Понятие операционной системы. Классификация. Функциональные компоненты.](#h.pf9twjkcdmdj)

[21. Архитектура операционной системы на примере одной из современных систем.](#h.ahv1l4qxrkyx)

[22. Процессы и потоки в многозадачной операционной системе. Алгоритмы планирования и управления.](#h.hh436tojwty3)

[23. Управление памятью в операционной системе. Физическая память. Виртуальная память. Адресное пространство. Алгоритмы замещения страниц памяти.](#h.8j88xobk5bi9)

[24. Межпроцессное взаимодействие. Механизмы взаимодействия процессов.](#h.9qwv529vp9z1)

[25. Файловые системы. Основные особенности и физическая организация на примере одной из современных файловых систем.](#h.m4r4sy1rjulz)

[26. Каналы передачи данных. Физический канал. Логический канал. Понятие блока данных. Пример формата блока данных любого протокола](#h.lykol7tf66jq)

[27. Структуризация сетей. Понятие и характеристики основных сетевых топологий. Структурообразующие аппаратные средства и программное обеспечение](#h.ei1lkifdctu0)

[28. Понятие адресации в сетях. Типы адресов. Адресация в IP-сетях](#h.6et20adnhjjr)

[29. Характеристика протоколов IP, TCP, ARP, ICMP, POP3, SMTP](#h.qxvpplqhddmm)

[30. Эталонная модель взаимодействия открытых систем. Понятие уровня и уровневых сетевых сервисов. Характеристика уровней](#h.qjfit9spujzr)

[31. Этапы сбора и анализа требований к программной системе, требований заказчика и детальных требований](#h.jybajph9kid)

[32. Требования к программному обеспечению. Типы требований. Уровни описания требований](#h.fo5r2sxm3acc)

[33. Критерии качества программного обеспечения](#h.ge7gbib6fb5g)

[34. Модели жизненного цикла программного обеспечения](#h.xs7kzdi6k4ja)

[35. Технологическая документация. Пользовательская документация программных средств. Документация по сопровождению программных средств](#h.cy939gd277vu)

[36. Принципы S.O.L.I.D. Примеры использования](#h.vy853q9w09cg)

[37. Паттерны проектирования. Группы паттернов. Примеры использования](#h.w87xz4e6phoj)

[38. Эволюция программного обеспечения. Работа с унаследованными системами, возвратное проектирование (reverse engineering)](#h.eiqvxbtwtt5x)

[39. Принципы объектно-ориентированного конструирования программных систем.](#h.nt9dm42d8yv4)

[40. Рефакторинг программных систем](#h.xu1sy1hrhvl9)

[41. Миграция программных систем](#h.9q55wzay2hjj)

[42. Концепция тестирования программного обеспечения. Основные проблемы тестирования](#h.488npsu6lubm)

[43. Оценка покрытия тестами программы и проекта. Методика интегральной оценки тестированности.](#h.oizl356cdr4d)

[44. Модульное и интеграционное тестирование.](#h.govaflaexcew)

[45. Статическое и динамическое тестирование](#h.20d5j3nyk2sb)

[46. Методы поиска и устранения ошибок](#h.fwmqtxusz48c)

# 

# 1. Алгоритм: базовые конструкции, способы задания, оценка временной сложности на примере алгоритмов поиска

Базовые конструкции: последовательное выполнение, цикл, условие. Цикл с постусловием, предусловием.

Способы задания:

* словесный — произвольное изложение на естественном языке.
* Графический — последовательность связанных между собой функциональных блоков, каждый из которых соответствует выполнению одного или нескольких действий.
* Псевдокод занимает промежуточное положение между естественным языком и формальным. С одной стороны, он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст. С другой стороны, в псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи. В псевдокоде не приняты строгие синтаксические правила для записи команд, присущие формальным языкам, что облегчает запись алгоритма на стадии его проектирования и дает возможность использовать более широкий набор команд, рассчитанный на абстрактного исполнителя.

Оценка временной сложности — функция времени выполнения алгоритма от размера входных данных. Это функция размера входных и выходных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера. Сложность бинарного поиска: Θ( log( n ) ). Поиск продолжаться, пока не достигнем единицы. Если мы захотим узнать, на какой итерации это произошло, нам нужно будет просто решить следующее уравнение:

1 = n / 2^i

2^i = n

i = log( n )

# 2. Тип данных. Структуры данных. Классификация структур данных

Тип данных - множество значений и операций на этих значениях.

Структуры данных - программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных в вычислительной технике.

Классификация структур данных

Простые — состоят из простых типов. Составные — состоят из других структур.

Физическая структура данных - это способ физического представления данных в памяти компьютера.

Логическая или абстрактная структура - это рассмотрение структуры данных без учета его представления в машинной памяти.

Несвязные структуры характеризуются отсутствием связей между элементами структуры.

Связные структуры характеризуются наличием связи. Примерами несвязных структур есть векторы, массивы, строки, стеки, очереди; примеры связных структур - связные списки.

Статические - к этой группе относят массивы, множества, записи, таблицы.

Полустатические - это стеки, очереди, деки, дерева.

Динамические - линейные и разветвленные связные списки, графы, деревья.

Структуры данных для оперативной памяти - это данные, размещенные в статической и динамической памяти компьютера. Все вышеприведенные структуры данных - это структуры для оперативной памяти.

Структуры данных для внешней памяти называют файловыми структурами или файлами. Примерами файловых структур есть последовательные файлы, файлы, организованные разделами, В- деревья.

Линейные структуры с **последовательным** распределением элементов в памяти (векторы, строки, массивы, стеки, очереди) и структуры с **произвольным** связным распределением элементов в памяти (односвязные и двухсвязные линейные списки).

Нелинейные структуры - многосвязные списки, деревья, графы.

# 3. Базовые и улучшенные сортировки на основе выбора, включения и обмена. Сравнительный анализ алгоритмов сортировки

**Сортировка пузырьком**. Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами. После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию...

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.

Среднее число сравнений и обменов имеют квадратичный порядок роста: Theta(n^2), отсюда можно заключить, что алгоритм пузырька очень медленнен и малоэффективен.

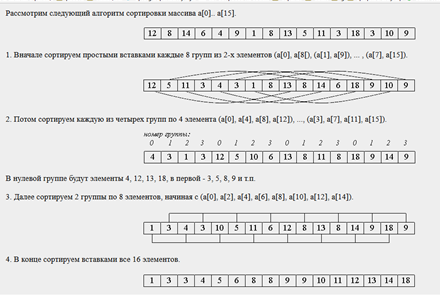
**Сортировка выбором**. Идея метода состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней одного элемента за другим в правильном порядке. На i-м шаге выбираем наименьший из элементов a[i] ... a[n] и меняем его местами с a[i]. С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций: Theta(n^2).

**Сортировка простыми вставками**. На следующем, (i+1)-м каждом шаге алгоритма берем a[i+1] и вставляем на нужное место в готовую часть массива.

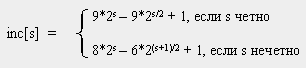
Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним. В зависимости от результата сравнения элемент либо остается на текущем месте (вставка завершена), либо они меняются местами и процесс повторяется. Аналогично сортировке выбором, среднее, а также худшее число сравнений и пересылок оцениваются как Theta(n^2).



**Сортировка Шелла** является довольно интересной модификацией алгоритма сортировки простыми вставками.



Использованный в примере набор ..., 8, 4, 2, 1 - неплохой выбор, особенно, когда количество элементов - степень двойки. Однако гораздо лучший вариант предложил Р. Седжвик. Его последовательность имеет вид:



При использовании таких приращений среднее количество операций: O(n^(7/6)), в худшем случае - порядка O(n^(4/3)).

При использовании формулы Седжвика следует остановиться на значении inc[s-1], если 3\*inc[s] > size.

**Пирамидальная сортировка**. O(n log n). Строим из массива пирамиду.

* в a[0] хранится корень дерева
* левый и правый сыновья элемента a[i] хранятся, соответственнно, в a[2i+1] и a[2i+2]

Плюсы такого хранения пирамиды очевидны:

* никаких дополнительных переменных, нужно лишь понимать схему.
* узлы хранятся от вершины и далее вниз, уровень за уровнем.
* узлы одного уровня хранятся в массиве слева направо.

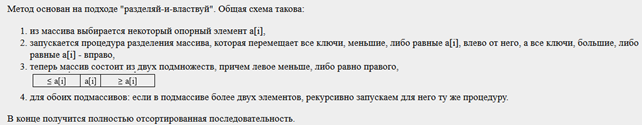
Итак, задача построения пирамиды из массива успешно решена. Как видно из свойств пирамиды, в корне всегда находится максимальный элемент. Отсюда вытекает алгоритм фазы 2:

Берем верхний элемент пирамиды a[0]...a[n] (первый в массиве) и меняем с последним местами. Теперь "забываем" об этом элементе и далее рассматриваем массив a[0]...a[n-1]. Для превращения его в пирамиду достаточно просеять лишь новый первый элемент.

Повторяем шаг 1, пока обрабатываемая часть массива не уменьшится до одного элемента.

Очевидно, в конец массива каждый раз попадает максимальный элемент из текущей пирамиды, поэтому в правой части постепенно возникает упорядоченная последовательность.

**Быстрая сортировка**



На входе массив a[0]...a[N] и опорный элемент p, по которому будет производиться разделение.

1. Введем два указателя: i и j. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель i с I шагом в 1 элемент по направлению к концу массива, пока не будет найден элемент a[i] >= p. Затем аналогичным образом начнем двигать указатель j от конца массива к началу, пока не будет найден a[j] <= p.
3. Далее, если i <= j, меняем a[i] и a[j] местами и продолжаем двигать i,j по тем же правилам...
4. Повторяем шаг 3, пока i <= j.

быстродействие: O(n log n), что и имеет место на практике.

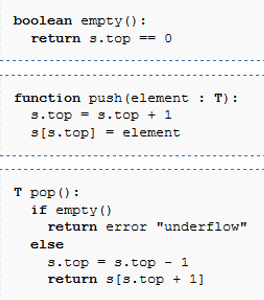
# 4. Структуры данных типа стек. Реализация стека как отображения на массив и односвязный список. Примеры применения

Стек - тип данных, представляющий собой список элементов, организованных по принципу последним пришёл — первым вышел. Программный вид стека используется для обхода структур данных, например, дерево или граф. При использовании рекурсивных функций также будет применяться стек, но аппаратный его вид. Кроме этих назначений, стек используется для организации стековой машины, реализующей вычисления в обратной инверсной записи.

Перед реализацией стека выделим ключевые поля:

* {s[1] … n]} — массив, с помощью которого реализуется стек, способный вместить не более n элементов,
* s.top — индекс последнего помещенного в стек элемента.

Стек состоит из элементов {s[1] … s.top]}, где s[1] — элемент на дне стека, [s.top] — элемент на его вершине. Если s.top = 0, то стек не содержит ни одного элемента и является пустым.



Стек можно реализовать и на списке. Для этого необходимо создать список и операции работы стека на созданном списке. Ниже представлен пример реализации стека на односвязном списке. Стек будем "держать" за голову. Добавляться новые элементы посредством операции push будут перед головой, сами при этом становясь новой головой, а элементом для изъятия из стека с помощью pop будет текущая голова. После вызова функции push текущая голова уже станет старой и будет являться следующим элементом за добавленным, то есть ссылка на следующий элемент нового элемента будет указывать на старую голову. После вызова функции pop будет получена и возвращена информация, хранящаяся в текущей голове. Сама голова будет изъята из стека, а новой головой станет элемент, который следовал за изъятой головой.

Ключевые поля:

* head.data — значение в верхушке стека,
* head.next — значение следующее за верхушкой стека.



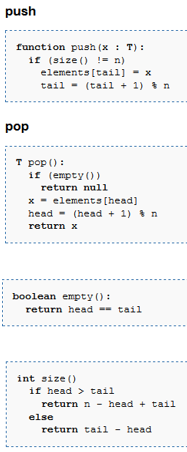
# 5. Структуры данных типа очередь. Реализация очереди как отображение на массив и односвязный список. Примеры применения.

О́чередь — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел». Очередь в программировании используется, как и в реальной жизни, когда нужно совершить какие-то действия в порядке их поступления, выполнив их последовательно. Примером может служить организация событий в Windows. Когда пользователь оказывает какое-то действие на приложение, то в приложении не вызывается соответствующая процедура (ведь в этот момент приложение может совершать другие действия), а ему присылается сообщение, содержащее информацию о совершенном действии, это сообщение ставится в очередь, и только когда будут обработаны сообщения, пришедшие ранее, приложение выполнит необходимое действие.

Очередь, способную вместить не более n элементов, можно реализовать с помощью массива [0 …n-1]. Она будет обладать следующими полями:

* head — голова очереди,
* tail — хвост очереди.

ТУТ КОД ДЛЯ КОЛЬЦЕВОЙ ОЧЕРЕДИ



**На списке**.

* ListItem(data : T, next : ListItem) — конструктор,
* x.value — поле, в котором хранится значение элемента,
* x.next — указатель на следующий элемент очереди.

# 6. Структуры данных типа таблица. Прямого доступа, хеш-таблица. Разрешение коллизий с помощью цепочек и открытой адресации и анализ их алгоритмов

Таблица — набор элементов одинаковой организации, каждый из которых можно представить в виде двойки <K, V>, где K — ключ, а V — тело (информационная часть) элемента.

Ключ уникален для каждого элемента, то есть в таблице нет элементов с одинаковыми ключами. Ключ используется для доступа к элементам при выполнении операций.

В таблице прямого доступа ключ – адрес элемента.

Хеш-таблица — это таблица, в которой положение адреса элемента определяется с помощью некоторой функции H (хеш-функции), аргументом которой является значение ключа элемента.

**Метод цепочек.** Каждая ячейка массива H является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.

Операции поиска или удаления элемента требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления элемента нужно добавить элемент в конец или начало соответствующего списка, и, в случае, если коэффициент заполнения станет слишком велик, увеличить размер массива H и перестроить таблицу. Коэффициент заполнения высчитывается как отношение количества уже внесенных элементов к количеству возможных значений хэш-функции.

При предположении, что каждый элемент может попасть в любую позицию таблицы H с равной вероятностью и независимо от того, куда попал любой другой элемент, среднее время работы операции поиска элемента составляет Θ(1 + α), где α — коэффициент заполнения таблицы.

**Открытая адресация.** В массиве H хранятся сами пары ключ-значение. Алгоритм вставки элемента: сначала вычисляется хэш-функция, если возникает коллизия, то алгоритм проверяет ячейки массива H в некотором порядке до тех пор, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент. Этот порядок вычисляется на лету, что позволяет сэкономить на памяти для указателей, требующихся в хеш-таблицах с цепочками.

Вставка - О( 1 / (1-α) ); поиск - О( (1/α) \* ln( 1/(1+α) ) ); удаление - О(1). 0 < α < 1. α опт. = 0,5.

# 7. Структуры данных бинарное дерево. Операции включения, исключения. Алгоритмы поиска и прохождения

Бинарное дерево - иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками.

Существует два вида обхода графа: 1) в глубину; 2) в ширину.

Поиск в ширину (BFS) идет из начальной вершины, посещает сначала все вершины, находящиеся на расстоянии одного ребра от начальной, потом посещает все вершины на расстоянии два ребра от начальной и так далее.

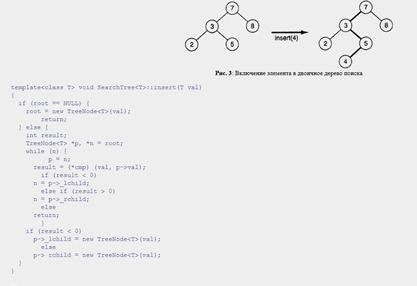
Поиск в глубину (DFS) идет из начальной вершины, посещая еще не посещенные вершины без оглядки на удаленность от начальной вершины.

Cложность обхода и в ширину и в глубину O(V + E), где V – количество вершин, E – количество ребер. То есть является линейной по количеству вершин и ребер.

Прямой обход идет в следующем порядке: корень, левый потомок, правый потомок. Симметричный — левый потомок, корень, правый потомок. Обратный – левый потомок, правый потомок, корень.

**Включение** **и удаление элемента в дерево поиска.**

**Включение.** Для включения нового элемента в двоичное дерево поиска вначале нужно определить его точное положение — а именно внешний узел, который должен быть заменен путем отслеживания пути поиска элемента, начиная с корня. Кроме сохранения указателя n на текущий узел мы будем хранить указатель р на предка узла n. Таким образом, когда n достигнет некоторого внешнего узла, р будет указывать на узел, который должен стать предком нового узла. Для осуществления включения узла мы создадим новый узел, содержащий новый элемент, и затем свяжем предок р с этим новым узлом (рис. 3).



**Удаление**. Чтобы удалить элемент из дерева поиска, вначале мы отслеживаем путь поиска элемента, начиная с корня и вниз до узла n, содержащего элемент. В этот момент могут возникнуть три ситуации:

* Узел n имеет пустой левый потомок. В этом случае ссылка на n (записанная в предке n, если он есть) заменяется на ссылку на правого потомка n.
* У узла n есть не пустой левый потомок, но правый потомок пустой. В этом случае ссылка вниз на n заменяется ссылкой на левый потомок узла n.
* Узел n имеет два не пустых потомка. Для этого нужно обменять его ключ с самым левым потомком правого поддерева или с самым правым левого поддерева.

# 8. Программные продукты как сложные системы. Признаки сложных систем

Накопившийся к настоящему времени опыт разработки программного обеспечения, показывает, что большинство программных продуктов представляют собой сложные системы. Быть сложным, значит включать множество частей, аспектов, деталей, понятий, требующих для понимания серьёзного исследования или рассмотрения.

Сложные системы являются иерархическими, т. е. состоят из взаимосвязанных подсистем, которые в свою очередь могут быть разделены на подсистемы и т. д. вплоть до самого низкого уровня.

* Выбор, какие компоненты в данной системе являются элементарными относительно произволен и в большей степени оставляется на усмотрение исследователя.
* Внутрикомпонентные связи обычно сильнее, чем связи между компонентами системы. Это различие внутрикомпонентных и межкомпонентных взаимодействий обуславливает разделение функций между частями системы и даёт возможность относительно независимо изучать каждую часть.
* Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных. Иными словами, разные сложные системы содержат одинаковые структурные части. Эти части могут использовать общие более мелкие компоненты, такие как клетки, или более крупные структуры, типа сосудистых систем, имеющиеся и у растений, и у животных.
* Любая работающая сложная система является результатом развития работающей более простой системы. В процессе развития системы объекты первоначально рассматривающиеся как сложные становятся элементарными из них строят более сложные системы.

# 9. Декомпозиция. Преимущества и недостатки объектно-ориентированного подхода в программировании

Декомпозиция — разделение целого на части. Также декомпозиция — это научный метод, использующий структуру задачи и позволяющий заменить решение одной большой задачи решением серии меньших задач, пусть и взаимосвязанных, но более простых.

Декомпозиция, как процесс расчленения, позволяет рассматривать любую исследуемую систему как сложную, состоящую из отдельных взаимосвязанных подсистем, которые, в свою очередь, также могут быть расчленены на части. В качестве систем могут выступать не только материальные объекты, но и процессы, явления и понятия.

Преимущества объектно-ориентированного подхода:

* Классы позволяют проводить конструирование из полезных компонент, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
* Данные и операции вместе образуют определенную сущность и они не «размазываются» по всей программе, как это нередко бывает в случае процедурного программирования.
* Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
* Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.
* Мы сокращаем время на разработку, которое с выгодой может быть отдано другим проектам.
* Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
* Когда некая компонента используется сразу несколькими клиентами, то улучшения, вносимые в ее код, одновременно оказывают свое положительное влияние и на множество работающих с ней программ.
* Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.
* Объектная модель вполне естественна, поскольку в первую очередь ориентирована на человеческое восприятие мира, а не на компьютерную реализацию.

Недостатки:

* Необходимо понимать базовые концепции, такие как классы, наследование и динамическое связывание. Для программистов, уже знакомых с понятием модуля и с абстрактными типами данных, это потребует минимальных усилий. Для тех же, кто никогда не использовал инкапсуляцию данных, это может означать изменения мировоззрения и может отнять на изучение значительное количество времени.
* Многоразовое использование требует от программиста познакомиться с большими библиотеками классов. А это может оказаться сложнее, чем даже изучение нового языка программирования. Библиотека классов фактически представляет собой виртуальный язык, который может включать в себя сотни типов и тысячи операций. В языке Smalltalk, к примеру, до того, как перейти к практическому программированию, нужно изучить значительную часть его библиотек и классов. А это тоже требует времени.
* Проектирование классов — задача куда более сложная, чем их использование. Проектирование класса, как и проектирование языка, требует большого опыта. Это итеративный процесс, где приходится учиться на своих же ошибках.

# 9. Основные понятия объектно-ориентированного программирования: объект, класс, виды отношений между классами

**Класс** – тип данных, описывающий структуру и поведение объектов.

**Объект** – экземпляр класса.

**Ассоциация** означает, что объекты двух классов могут ссылаться один на другой, иметь некоторую связь между друг другом.

**Агрегация** – отношение, когда один объект является частью другого. Объект, являющий частью может существовать отдельно.

**Композиция** - более сильная связь между классами, чем агрегация. Между агрегацией и композицией довольно тонкая грань. Особенностью композиции является то, что объекты, из которых создаётся композиция, могут принадлежать только классу, с которым они образуют композицию. При этом время жизни объекта и класса, в который встраивается объект, совпадает.

**Обобщение/расширение (наследование).** Диаграмма классов, показывающая наследование двух подклассов от одного суперкласса. Обобщение показывает, что один из двух связанных классов (подтип) является частной формой другого (надтипа), который называется обобщением первого. На практике это означает, что любой экземпляр подтипа является также экземпляром надтипа.

# 10. Основные принципы объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм

**Инкапсуляция** - это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. Инкапсуляция используется для управления скрытием данных от неправильного использования.

**Полиморфизм** - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Полиморфный метод будет иметь одинаковое имя, но разные типы параметров и будет осуществлять логически одинаковые действия над ними и достигать одной цели.

**Наследование** — механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса или интерфейса. Потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами. Позволяет строить иерархии.

# 11. Общая характеристика объектов в объектно-ориентированном программировании. Объектная декомпозиция. Виды отношений между объектами

**Общая характеристика объектов в объектно-ориентированном программировании.** Объект в ООП — это сущность, способная сохранять свое состояние (информацию) и обеспечивающая набор операций (поведение) для проверки и изменения этого состояния. Объект в объектно-ориентированном программировании — это модель или абстракция реальной сущности в программной системе. Предмет моделирования при построении объекта в ООП может быть любым.

Каждый объект в ООП характеризуется своим состоянием. Состояние объекта характеризуется текущим значением его атрибутов. Стоит заметить, что атрибутами объекта в ООП могут быть не только простейшие значения (число, логическое значение и т.д.), но и сложные величины или другие объекты.

Важнейшей характеристикой объекта в ООП является описание того, как он может взаимодействовать с окружающим миром. Это описание называется интерфейсом объекта.

Объекты в ООП взаимодействуют между собой с помощью сообщений. Принимая сообщение, объект выполняет соответствующее действие. Эти действия обычно называются методами.

**Объектная декомпозиция.** Объектной декомпозицией называют процесс представления предметной области задачи в виде совокупности функциональных элементов (объектов), обменивающихся в процессе выполнения программы входными воздействиями (сообщениями). Каждый выделяемый объект предметной области отвечает за выполнение некоторых действий, зависящих от полученных сообщений и параметров самого объекта. Совокупность значений параметров объекта называют его состоянием, а совокупность реакций на получаемые сообщения — поведением. Параметры состояния и элементы поведения объектов определяются условием задачи. В процессе решения задачи объект, получив некоторое сообщение, выполняет заранее определенные действия, например, может изменить собственное состояние, выполнить некоторые вычисления, нарисовать окно или график и, в свою очередь, сформировать сообщения другим объектам. Таким образом, процессом решения задачи управляет последовательность сообщений. Передавая эти сообщения от объекта к объекту, программа выполняет необходимые действия.

**Ассоциация.** Ассоциации обеспечивают взаимодействия объектов, принадлежащих разным классам. Ассоциация обозначает только семантическую связь. Она не указывает направление и точную реализацию отношения. Ассоциация пригодна для анализа проблемы, когда нам требуется лишь идентифицировать связи. 1 – 1. 1 – n. n – n.

**Агрегация** – связь часть – целое. Уничтожение владельца не приводит к уничтожению части.

**Композиция** – связь часть – целое. Уничтожение владельца приводит к уничтожению части.

**Зависимость** — это слабая форма отношения использования, при котором изменение в спецификации одного влечёт за собой изменение другого, причём обратное не обязательно. Возникает, когда объект выступает, например, в форме параметра или локальной переменной.

# 12. Массивы, коллекции, обобщения: основные особенности, реализуемые интерфейсы.

Массив — набор компонентов (элементов), расположенных в памяти непосредственно друг за другом, доступ к которым осуществляется по индексу (индексам). Размерность массива — количество индексов, необходимое для однозначного доступа к элементу массива.

Массив — упорядоченный набор данных, для хранения данных одного типа, идентифицируемых с помощью одного или нескольких индексов. В простейшем случае массив имеет постоянную длину и хранит единицы данных одного и того же типа.

Количество используемых индексов массива может быть различным. Массивы с одним индексом называют одномерными, с двумя — двумерными и т. д. Одномерный массив (колонка, столбец) нестрого соответствует вектору в математике, двумерный — матрице. Чаще всего применяются массивы с одним или двумя индексами, реже — с тремя, ещё большее количество индексов встречается крайне редко.

Достоинства

1) лёгкость вычисления адреса элемента по его индексу (поскольку элементы массива располагаются один за другим)

2) одинаковое время доступа ко всем элементам

3) малый размер элементов: они состоят только из информационного поля.

Недостатки

1) для статического массива — отсутствие динамики, невозможность удаления или добавления элемента без сдвига других

2) для динамического и/или гетерогенного массива — более низкое (по сравнению с обычным статическим) быстродействие и дополнительные накладные расходы на поддержку динамических свойств и/или гетерогенности.

3) при работе с массивом в стиле C (с указателями) и при отсутствии дополнительных средств контроля — угроза выхода за границы массива и повреждения данных

Все массивы являются потомками класса System.Array, который в свою очередь наследует ряд интерфейсов: ICloneable, IList, ICollection, IEnumerable, а, следовательно, реализует все их методы и свойства. Кроме того, класс Array имеет большое число собственных методов и свойств.

Значения интерфейсов:



Коллекция — программный объект, содержащий в себе набор значений одного или различных типов, и позволяющий обращаться к этим значениям.

Коллекции состоят из интерфейсов, реализации и алгоритмов. Система хранилищ данных с унифицированной архитектурой

Коллекция позволяет записывать в себя значения и извлекать их. Назначение коллекции — служить хранилищем объектов и обеспечивать доступ к ним. Обычно коллекции используются для хранения групп однотипных объектов, подлежащих стереотипной обработке. Для обращения к конкретному элементу коллекции могут использоваться различные методы, в зависимости от её логической организации. Реализация может допускать выполнение отдельных операций над коллекциями в целом. Наличие операций над коллекциями во многих случаях может существенно упростить программирование.

По общим характеристикам

Коллекция может иметь постоянный либо динамически изменяющийся размер. В первом случае в коллекцию может быть записано только строго определённое количество объектов, во втором — коллекция допускает размещение стольких объектов, сколько необходимо (разумеется, в пределах, задаваемых техническими ограничениями). В большинстве случаев, говоря о коллекции, имеют в виду динамическую коллекцию, то есть коллекцию второго вида, хотя, например, обычный статический массив — это тоже коллекция.

Коллекция может хранить объекты только одного или различных типов. Во втором случае говорят о гетерогенной коллекции.

По логике организации

В зависимости от того, как логически организован доступ к данным коллекции, выделяются следующие основные типы:

Вектор — элементы коллекции упорядочены, каждый имеет собственный номер, называемый индексом, по которому к нему можно в любой момент обратиться. Как правило, в качестве индексов выступают последовательные целые числа либо значения, приводимые к ним. Для обращения к элементу вектора используется имя вектора и значение индекса. При добавлении нового элемента он добавляется либо в конец вектора, либо в позицию с заданным индексом. Удаление элемента из вектора приводит к образованию пустого элемента.

Матрица — элементы имеют два упорядоченных индекса, каждый из которых является целым числом или значением, приводимым к целому. Для доступа к элементу нужно указать имя матрицы и оба индекса. Новый элемент может быть добавлен только в позицию с заданной парой индексов. Удаление приводит к оставлению пустого элемента.

Многомерный массив — логическое развитие идеи вектора и матрицы до большего (в общем случае — произвольного) числа индексов.

Список — элементы коллекции упорядочены, идентификаторов у элементов нет. Список — коллекция с последовательным доступом. В любой момент доступен первый элемент коллекции (обычно также доступен и последний). От любого элемента коллекции можно получить доступ к следующему по порядку, таким образом, можно последовательно дойти от первого элемента списка до любого желаемого. Возможна реализация, допускающая обратный проход (к предыдущему элементу от известного). Новый элемент может добавляться в начало или в конец списка. При удалении элемента из начала списка первым элементом становится следующий за ним, при удалении из конца — предыдущий, из середины — предыдущий и последующий элементы становятся, соответственно, предыдущим и последующим один для другого.

Стек — коллекция, реализующая принцип хранения «LIFO» («последним пришёл — первым вышел»). В стеке постоянно доступен только один элемент — тот, который был добавлен последним. Новый элемент может быть добавлен в стек, он станет текущим. Текущий элемент всегда можно удалить («взять») из стека, после этого становится доступен элемент, который был добавлен непосредственно перед ним.

Очередь — коллекция, реализующая принцип хранения «FIFO» («первым пришёл — первым вышел»). В очереди постоянно доступен только один элемент — тот, который был добавлен самым первым из имеющихся. При добавлении нового элемента он попадает в очередь. Текущий элемент можно удалить — тогда текущим станет элемент, добавленный первым из оставшихся.

Ассоциативный массив (словарь) — неупорядоченная коллекция, хранящая пары «ключ — значение». Доступ к элементам производится по ключу. В качестве ключа могут использоваться значения различных типов, единственное ограничение — тип ключа должен допускать сравнение на равенство. Любая пара может быть в любой момент удалена. Добавляться может только пара (с определённым ключом). Может вводиться запрет на дублирование ключей в коллекции. Если такого ограничения нет, то при обращении по дублирующемуся ключу может выдаваться либо n-е найденное значение (где n либо постоянно, либо определяется формой запроса), либо все значения с данным ключом.

Множество — неупорядоченная коллекция, хранящая набор уникальных значений и поддерживающая для них операции добавления, удаления и определения вхождения. Как правило, для множеств поддерживаются операции, аналогичные операциям с математическими множествами: объединение, пересечение, симметричная разность множеств и несимметричная разность множеств.

Мультимножество — неупорядоченная коллекция, аналогичная множеству, но допускающая наличие в коллекции одновременно двух и более одинаковых значений.

По реализации

На уровне реализации коллекция может представлять собой одну из следующих структур данных:

* Массив
* Односвязный список
* Двусвязный список
* Стек
* Хеш-таблица
* Битовый массив

Обобщения - это параметризованные типы. С их помощью можно объявлять классы, интерфейсы и методы, где тип данных указан в виде параметра. Обобщения добавили в язык безопасность типов.

class Gen<T> {}

В угловых скобках используется T - имя параметра типа. Это имя используется в качестве заполнителя, куда будет подставлено имя реального типа, переданного классу Gen при создании реальных типов. То есть параметр типа T применяется в классе всякий раз, когда требуется параметр типа. Угловые скобки указывают, что параметр может быть обобщён. Сам класс при этом называется обобщённым классом или параметризованным типом.

Далее тип T используется для объявления объекта по имени ob:

T ob; // объявляет объект типа T

Вместо T подставится реальный тип, который будет указан при создании объекта класса Gen. Объект ob будет объектом типа, переданного в параметре типа T. Если в параметре T передать тип String, то экземпляр ob будет иметь тип String.

Рассмотрим конструктор Gen().

Get(T o) {

ob = o;}

Параметр o имеет тип T. Это значит, что реальный тип параметра o определяется типом, переданным параметром типа T при создании объекта класса Gen.

Параметр типа T также может быть использован для указания типа возвращаемого значения метода:

T getob() {

return ob; }

Как использовать обобщённый класс. Можно создать версию класса Gen для целых чисел:

Gen<Integer> iOb;

В угловых скобках указан тип Integer, т.е. это аргумент типа, который передаётся в параметре типа T класса Gen. Фактически мы создаём версию класса Gen, в которой все ссылки на тип Tстановятся ссылками на тип Integer.

Когда мы присваиваем ссылку на экземпляр, то угловые скобки также требуется указывать.

iOb = new Gen<Integer>(77);

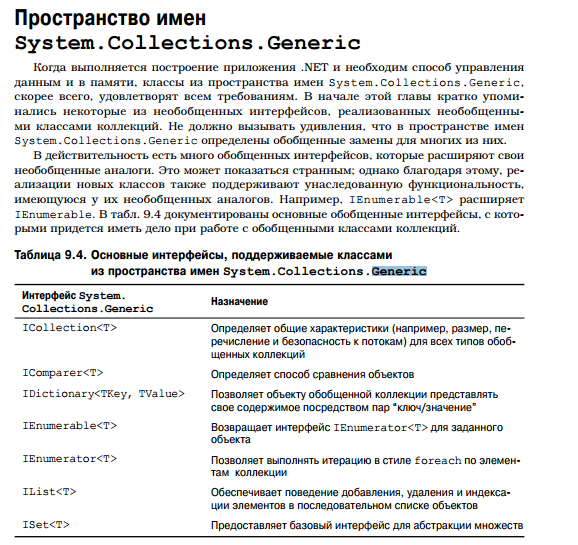
Обобщения работают только с объектами. Поэтому нельзя использовать в качестве параметра элементарные типы вроде int или char:

Gen<int> intOb = new Gen<int>(44); // нельзя!

Хотя объекты iOb и strOb имеют тип Gen<T>, они являются ссылками на разные типы и их сравнивать нельзя.

iOB = strOb; // нельзя!

Использование обобщений автоматически гарантирует безопасность типов во всех операциях, где они задействованы.



# 13. Понятие «базы данных». Основные компоненты базы данных

БД – совокупность данных, организованных по определённым правилам, предусматривающим общие принципы описания, хранения и манипулирования данными, независимая от прикладных программ. Эти данные относятся к определённой предметной области и организованы таким образом, что могут быть использованы для решения многих задач многими пользователями.

Широкое распространение получили БД, имеющие табличную структуру. Базы данных, имеющие связанные таблицы, называют *реляционными* базами данных. Координация осуществляется путем установления связей между таблицами. В объектно-ориентированных БД данные с различными отношениями рассматриваются как единый объект. Поэтому разработчик может не беспокоиться о связывании и разделении данных.

**Основные компоненты:**

Реляционная база данных состоит из множества таблиц. Структуру каждой таблицы разрабатывают отдельно. В этих таблицах столбцы называются *полями* или *атрибутами*, а строки - *записями* или *кортежами*. Поля образуют структуру базы данных, а записи составляют информацию, которая в ней содержится.

Для того чтобы связи между таблицами работали надежно, и по записи из одной таблицы можно было однозначно найти записи в другой таблице, надо предусмотреть в таблице *уникальные поля (идентификаторы)*.

Уникальное поле - это поле, значения в котором не могут повторяться. Если ни одно поле таблицы не приемлемо в качестве уникального, его можно создать искусственно. Кроме этого, существует понятие *ключевого поля*. При создании структуры таблиц одно поле (или одну комбинацию полей) можно назначить ключевым. С ключевыми полями СУБД работает особо. Она проверяет их уникальность и быстрее выполняет сортировку по таким полям. Ключевое поле - очевидный кандидат для создания связей. Иногда ключевое поле называют *первичным ключом*.

В реляционных базах данных важную роль играет еще один ключ - *внешний ключ*. Внешний ключ - это поле одной таблицы, которое ссылается на первичный ключ другой таблицы. Существует три разновидности связей между таблицами базы данных:

* «один-ко-многим»,
* «один-к-одному»,
* «многие-ко-многим».

В качестве первичного ключа в таблицах часто используют поле, имеющее тип "Счетчик". Ввести два одинаковых значения в такое поле нельзя по определению, поскольку приращение значения поля производится автоматически. Структура связей между таблицами называется *схемой данных*.

*Индекс* - это средство автоматической сортировки записей в таблице по значению индексируемого поля. Существует два вида индексов: допускающие и не допускающие повторение значений поля. Почти так же, как и по одному полю, индекс можно составить по двум и более полям. Применение индексов значительно ускоряет просмотр и выборку данных. Недостатки индексов - каждый из них занимает дополнительное место на жестком диске, а также в оперативной памяти. Кроме этого, увеличивается время добавления или удаления записи (особенно если в таблице содержится большое количество записей), поскольку индексы приходится создавать заново. Хотя эффект снижения быстродействия здесь не так важен, как эффект ускорения операций поиска и сортировки, все же следует следить за тем, чтобы не индексировать в таблице слишком много полей.

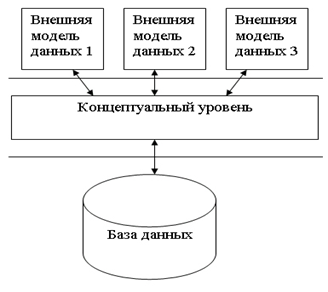
# 14. Системы баз данных. Архитектура систем баз данных. Современные СУБД.

Понятие **система баз данных** используется как в широком, так и в узком смысле.

В *широком смысле* система баз данных понимается фактически как синоним понятия информационная система и включает в себя данные, аппаратное обеспечение,программное обеспечение и пользователей.

В *узком смысле* система баз данных понимается как СУБД с управляемой ею базой данных, возможно, уже наполненной. Систе́ма управле́ния ба́зами да́нных (СУБД) — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных.

Трёхуровневая архитектура.



*Трехуровневая модель системы управления базой данных, предложенная ANSI*

* **Уровень внешних моделей** - самый верхний уровень, где каждая модель имеет свое «видение» данных. Этот уровень определяет точку зрения на БД отдельных приложений. Каждое приложение видит и обрабатывает только те данные, которые необходимы именно этому приложению.
* **Концептуальный уровень** - центральное управляющее звено, здесь база данных представлена в наиболее общем виде, который объединяет данные, используемые всеми приложениями, работающими с данной базой данных. Фактически концептуальный уровень отражает обобщенную модель предметной области (объектов реального мира), для которой создавалась база данных. Как любая модель, концептуальная модель отражает только существенные, с точки зрения обработки, особенности объектов реального мира.
* **Физический уровень** - собственно данные, расположенные в файлах или в страничных структурах, расположенных на внешних носителях информации.

Эта архитектура позволяет обеспечить логическую (между уровнями 1 и 2) и физическую (между уровнями 2 и 3) независимость при работе с данными. Логическая независимость предполагает возможность изменения одного приложения без корректировки других приложений, работающих с этой же базой данных. Физическая независимость предполагает возможность переноса хранимой информации с одних носителей на другие при сохранении работоспособности всех приложений, работающих с данной базой данных.

**Современные СУБД.**

Обычно современная СУБД содержит следующие компоненты:

* **ядро**, которое отвечает за управление данными во внешней и оперативной памяти и журнализацию,
* **процессор языка базы данных**, обеспечивающий оптимизацию запросов на извлечение и изменение данных и создание, как правило, машинно-независимого исполняемого внутреннего кода,
* **подсистему поддержки времени исполнения**, которая интерпретирует программы манипуляции данными, создающие пользовательский интерфейс с СУБД
* а также **сервисные программы** (внешние утилиты), обеспечивающие ряд дополнительных возможностей по обслуживанию информационной системы.

К современным СУБД предъявляются следующие требования:

* масштабируемость — отсутствие существенного снижения скорости выполнения пользовательских запросов при пропорциональном росте количества запросов и аппаратных ресурсов используемых данной СУБД (таких как объем оперативной памяти, количество процессоров и серверов);
* доступность — возможность всегда выполнить запрос;
* надежность — минимальная вероятность сбоев, наличие средств восстановления данных после сбоев, инструментов резервного копирования и дублирования данных (в последнее время производители СУБД нередко предлагают инструменты, позволяющие осуществлять подобные операции, не прерывая работу пользователей);
* управляемость — простота администрирования, наличие средств автоматического конфигурирования (типичный современный набор средств администрирования включает средства создания баз данных и их объектов, инструменты описания правил репликации данных между различными серверами, утилиты управления пользователями, группами и их правами, средства мониторинга событий, средства просмотра планов выполнения запросов, утилиты миграции из других СУБД);
* наличие средств защиты данных от потери и несанкционированного доступа;
* поддержка доступа к данным с помощью Web-служб;
* поддержка стандартных механизмов доступа к данным (таких как ODBC, JDBC, OLE DB, ADO .NET), позволяющая создавать приложения для СУБД с помощью различных средств разработки.

Несоответствие СУБД какому-либо из этих требований приводит к тому, что даже у неплохой по другим потребительским свойствам СУБД область применения оказывается весьма ограниченной.

# 15. Нормальные формы БД. Нормализация данных.

В рамках реляционной модели данных Э.Ф. Коддом был разработан аппарат нормализации отношений и предложен механизм, позволяющий любое отношение преобразовать к третьей нормальной форме. Нормализация схемы отношения выполняется путём декомпозиции схемы.



**Первая нормальная форма (1НФ).**

Отношение приведено к 1НФ, если все его атрибуты простые.

Отношение КНИГИ содержит сложные атрибуты Author ("Авторы") и Editor ("Редакторы"). Для приведения к 1НФ требуется сделать ключ отношения составным – атрибуты ID, Author и Editor (табл. 3.2).



Понятие функциональной зависимости. Пусть X и Y – атрибуты (группы атрибутов) некоторого отношения. Говорят, что Y функционально зависит от X, если в любой момент времени каждому значению X=х соответствует единственное значение Y=y (X®Y). (При этом любому значению Y=y может соответствовать несколько значений Х=(х1, х2,…)).

В нормализованном отношении все неключевые атрибуты функционально зависят от ключа отношения. Говорят, что неключевой атрибут функционально полно зависит от составного ключа, если он функционально зависит от ключа, но не находится в функциональной зависимости ни от какой части составного ключа.

**Вторая нормальная форма (2НФ).**

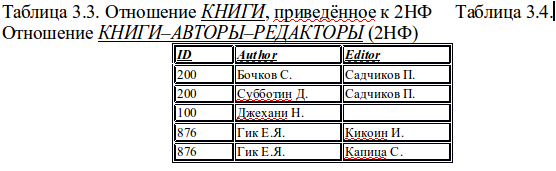
Отношение находится во 2НФ, если оно приведено к 1НФ и каждый неключевой атрибут функционально полно зависит от составного ключа.

Для того чтобы привести отношение ко 2НФ, нужно:

построить его проекцию, исключив атрибуты, которые не находятся в функционально полной зависимости от составного ключа;

построить дополнительные проекции на часть составного ключа и атрибуты, функционально зависящие от этой части ключа.

Ключом отношения КНИГИ (табл. 3.2) является комбинация полей (ID, Author, Editor). Все поля, не входящие в состав ключа, зависят только от идентификатора книги. Поэтому отношение должно быть разбито на два: КНИГИ (табл. 3.3) и КНИГИ–АВТОРЫ–РЕДАКТОРЫ (табл. 3.4). Эти отношения связаны по внешнему ключу, которым является поле ID.



Рассмотрим понятие транзитивной зависимости. Пусть X, Y, Z – атрибуты некоторого отношения. При этом X® Y и Y® Z, но обратное соответствие отсутствует, т.е. Z не зависит от Y или Y не зависит от X. Тогда говорят, что Z транзитивно зависит от X (X®® Z).

**Третья нормальная форма (3НФ).**

Отношение находится в 3НФ, если оно находится во 2НФ и в нем отсутствуют транзитивные зависимости.

Для отношения КНИГИ (табл. 3.3) атрибут Theme зависит от атрибута Code, а не от ключа (хотя название рубрики, естественно, соответствует её шифру). Поэтому для приведения отношения к 3НФ (табл. 3.5) нужно выделить из него ещё одно отношение РУБРИКАТОР (табл. 3.6).



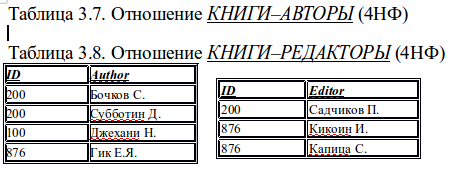
Введём понятие многозначной зависимости. Многозначная зависимость существует, если заданным значениям атрибута X соответствует множество, состоящее из нуля (или более) значений атрибута Y (X–»Y). Если в отношении присутствуют многозначные зависимости, то схема отношения должна находиться в 4НФ.

Различают тривиальные и нетривиальные многозначные зависимости. Тривиальной называется такая многозначная зависимость X–»Y, для которой Y Ì X или X U Y = R, где R – рассматриваемое отношение. Тривиальная многозначная зависимость не нарушает 4НФ. Если хотя бы одно из двух этих условий не выполняется (т.е. Y не является подмножеством X или X U Y состоит не из всех атрибутов R), то такая многозначная зависимость называется нетривиальной.

**Четвертая нормальная форма (4НФ).**

Отношение находится в 4НФ, если оно находится в 3НФ и в нем отсутствуют нетривиальные многозначные зависимости.

Для отношения КНИГИ–АВТОРЫ–РЕДАКТОРЫ (табл. 3.4) атрибуты Author и Editor зависит образуют две многозначные зависимости от первичного ключа, и при этом значения этих атрибутов не зависят друг от друга. Поэтому для приведения отношения к 4НФ нужно разбить его на два отношения КНИГИ–АВТОРЫ и КНИГИ–РЕДАКТОРЫ (табл. 3.7, 3.8).



Нормализация сокращает дублирование данных, но появление новых отношений усложняет поддержку логической целостности данных.

# 16. Инфологическая модель данных. Диаграмма «сущность — связь».

Инфологическая модель данных – описание предметной области, выполняемое с использованием естественного языка, математических формул, таблиц, графиков и других средств, понятных всем людям, работающих над проектированием БД (обобщенное неформальное описание данных). Выполняется на первом этапе проектирования.

Инфологическая модель данных является человеко-ориентированной моделью и полностью независима от физических параметров среды хранения данных. Такой средой хранения данных может быть память человека, а не компьютер. Поэтому инфологическая модель не изменяется до тех пор, пока какие-то изменения в реальном мире не потребуют внесения в нее соответствующих изменений так, чтобы эта модель продолжала отражать предметную область.

Цель инфологического этапа проектирования состоит в получении семантических (концептуальных) моделей, отражающих предметную область и информационные потребности пользователей. Поэтому этот этап называют еще как семантическое моделирование. Семантическое моделирование представляет собой моделирование структуры данных, опираясь на смысл этих данных.

Понятие “Предметная область” - базисное в теории БД и не имеет строгого определения. Оно вытекает из понятий “объект” и “предмет”. Предметная область (ПО) – часть реального мира, подлежащая изучению с целью организации управления и, в конечном итоге, автоматизации. ПО представляется множеством фрагментов, которые характеризуются множеством объектов, множеством процессов, использующих объекты, а также множеством пользователей, характеризуемых единым взглядом на предметную область.

Объектом называется явление внешнего мира. Это либо нечто реально существующее - человек, товар, изделие, либо процесс - учет рождаемости, получение товаров, выпуск изделий. Каждый объект обладает огромным количеством свойств.

**Примеры.**

Объект "Человек" обладает свойствами: рост, имя, дата рождения … ,

объект - "Изделие" обладает свойствами: качество, дата изготовления, внешний вид….

Между объектами существуют многочисленные связи. Например:

Человек покупает, продает, производит Изделие

Изделие создается, покупается, продается Человеком.

Предмет – модель реального объекта, в котором зафиксированы лишь выделенные для ИС свойства и связи. Совокупность отобранных предметов образует объектное ядро предметной области, а совокупность их взаимосвязей - структуру фрагмента действительности. Т.о. понятие “Предметная область” соответствует точке зрения потребителя на объектное ядро: в ней выделены только те объекты, свойства объектов и связи между объектами, которые представляют ценность для ИС и должны быть сохранены в БД.

Все действия по выявлению ядра предметной области производятся на этапе анализа ИС.

Объектное ядро системы в течение ЖЦ ИС не остается постоянным: пропадают и возникают объекты, меняются их свойства и взаимосвязи. Зафиксированные во времени цепочки этих изменений называются траекториями предметной области, а совокупность общих свойств траекторией – семантикой предметной области

Имеется целый ряд методик моделирования предметной области. Одна из наиболее популярных в настоящее время методик базируется на использовании графических диаграмм, включающих небольшое число разнородных компонентов ERD (Entity-Relationship Diagrams). В русскоязычной литературе эти диаграммы называют "объект – отношение" либо "сущность - связь".

Модель ERD была предложена в 1976 г. Питером Пин-Шэн Ченом. В дальнейшем многими авторами были разработаны свои варианты подобных моделей: нотация (notation – система обозначения, записи) Мартина, нотация IDEF1X, нотация Баркера), но все они базируются на графических диаграммах, предложенных Ченом.

На использовании разновидностей ER-модели основано большинство современных подходов к проектированию реляционных баз данных.

По сути, все варианты диаграмм сущность-связь исходят из одной идеи - рисунок всегда нагляднее текстового описания. Все такие диаграммы используют графическое изображение сущностей предметной области, их свойств (атрибутов), и взаимосвязей между сущностями.

Мы познакомимся с ER-диаграммами в нотации Баркера, как довольно легкой в понимании основных идей.

**Основные понятия ER-диаграмм (Entity-Relationship Diagrams)**

Основными понятиями ER-модели являются сущность, связь и атрибут.

Для большей выразительности и лучшего понимания имя сущности может сопровождаться примерами конкретных объектов этого типа.

**Определение 1.** Сущность - это реальный или представляемый объект, информация о котором должна сохраняться и быть доступна. Сущностями могут быть люди, места, самолеты, рейсы, вкус, цвет и т.д.

Каждая сущность должна иметь наименование, выраженное существительным в единственном числе. При этом имя сущности - это имя типа, а не некоторого конкретного экземпляра этого типа. Понятие тип сущности относится к набору однородных личностей, предметов, событий или идей, выступающих как целое.

Примерами сущностей могут быть такие классы объектов как "Поставщик", "Сотрудник", "Накладная".

Каждая сущность в модели изображается в виде прямоугольника, содержащего имя сущности:



**Определение 2.** Экземпляр сущности - это конкретный представитель данной сущности.

Например, представителем сущности "Сотрудник" может быть "Сотрудник Иванов".

Экземпляры сущностей должны быть различимы, т.е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности.

**Определение 3.** Атрибут сущности - это поименованная характеристика сущности. Его наименование должно быть уникальным для конкретного типа сущности, но может быть одинаковым для различного типа сущностей (например, ЦВЕТ может быть определен для многих сущностей: СОБАКА, АВТОМОБИЛЬ, КРАСКА и т.д.). Атрибуты используются для определения того, какая информация должна быть собрана о сущности. Примерами атрибутов для сущности АВТОМОБИЛЬ являются ТИП, МАРКА, НОМЕРНОЙ ЗНАК, ЦВЕТ и т.д.

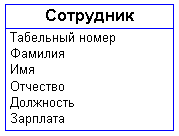
Здесь также существует различие между типом атрибута и экземпляром. Тип атрибута ЦВЕТ имеет много экземпляров или значений: Красный, Синий, Банановый, Белая ночь и т.д., однако каждому экземпляру сущности присваивается только одно значение атрибута.

Абсолютное различие между типами сущностей и атрибутами отсутствует. Атрибут является таковым только в связи с типом сущности. В другом контексте атрибут может выступать как самостоятельная сущность. Например, для автомобильного завода цвет – это только атрибут продукта производства, а для лакокрасочной фабрики цвет – тип сущности.

Каждый атрибут обеспечивается именем, уникальным в пределах сущности. Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с характеризующими прилагательными).

Примерами атрибутов сущности "Сотрудник" могут быть такие атрибуты как "Табельный номер", "Фамилия", "Имя", "Отчество", "Должность", "Зарплата" и т.п.

Атрибуты изображаются в пределах прямоугольника, определяющего сущность:



Атрибуты могут классифицироваться по принадлежности к одному из трех различных типов: описательные, указывающие, вспомогательные.

Описательные атрибуты представляют факты, внутренне присущие каждому экземпляру сущности.

Указывающие атрибуты используются для присвоения имени или обозначения экземплярам сущности.

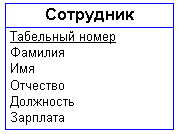
Вспомогательные атрибуты используются для связи экземпляра одной сущности с экземпляром другого. Атрибуты подчиняются строго определенным правилам.

**Определение 4.** Ключ сущности - минимальный набор атрибутов, по значениям которых можно однозначно найти требуемый экземпляр сущности. Минимальность означает, что исключение из набора любого атрибута не позволяет идентифицировать сущность по оставшимся.

Например, для сущности Расписание ключом является атрибут Номер\_рейса или набор: Пункт\_отправления, Время\_вылета и Пункт\_назначения (при условии, что из пункта в пункт вылетает в каждый момент времени один самолет).

Сущность может иметь несколько различных ключей.

Ключевые атрибуты изображаются на диаграмме подчеркиванием:

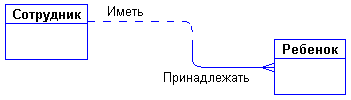


**Определение 5.** Связь - это некоторая ассоциация между двумя сущностями. Одна сущность может быть связана с другой сущностью или сама с собою. Связи позволяют по одной сущности находить другие сущности, связанные с нею.

Если бы назначением базы данных было только хранение отдельных, не связанных между собой данных, то ее структура могла бы быть очень простой. Однако одно из основных требований к организации базы данных – это обеспечение возможности отыскания одних сущностей по значениям других, для чего необходимо установить между ними определенные связи. А так как в реальных базах данных нередко содержатся сотни или даже тысячи сущностей, то теоретически между ними может быть установлено более миллиона связей. Наличие такого множества связей и определяет сложность инфологических моделей.

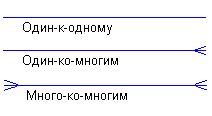
Например, связи между сущностями могут выражаться следующими фразами - "СОТРУДНИК может иметь несколько ДЕТЕЙ", "каждый СОТРУДНИК обязан числиться ровно в одном ОТДЕЛЕ".

Графически связь изображается линией, соединяющей две сущности:



Каждая связь имеет два конца и одно или два наименования. Наименование обычно выражается в неопределенной глагольной форме: "иметь", "принадлежать" и т.п. Каждое из наименований относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности.

Каждая связь может иметь один из следующих типов связи:



Связь типа **один-к-одному** означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой). Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, неправильно разделенную на две.

Связь типа **один-ко-многим** означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой). Это наиболее часто используемый тип связи. Левая сущность (со стороны "один") называется родительской, правая (со стороны "много") - дочерней. (см. рис. графического изображения связи)

Связь типа **много-ко-многим** означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Тип связи много-ко-многим является временным типом связи, допустимым на ранних этапах разработки модели. В дальнейшем этот тип связи должен быть заменен двумя связями типа один-ко-многим путем создания промежуточной сущности.

# 17. Язык SQL для работы с реляционными базами данных.

Реляционная база данных — база данных, основанная на реляционной модели данных.

Реляционная модель данных: данные передаются пользователю в виде таблиц и пользователю предоставляются операторы, которые позволяют создавать новые таблицы на основе существующих.

SQL (structured query language) - Структурированный Язык Запросов. Это - язык который дает вам возможность создавать и работать в реляционных базах данных, которые являются наборами связанной информации сохраняемой в таблицах.

SQL это язык ориентированный специально на реляционные базы данных. Он устраняет много работы которую вы должны были бы сделать если бы вы использовали универсальный язык программирования, напрмер C.

Интерактивный SQL используется для функционирования непосредственно в базе данных чтобы производить вывод для использования его заказчиком. В этой форме SQL, когда вы введете команду, она сейчас же выполнится и вы сможете увидеть вывод - немедленно.

Вложенный SQL состоит из команд SQL помещенных внутри программ, которые обычно написаны на некотором другом языке.

Запрос - команда которую вы даете вашей программе базы данных, и которая сообщает ей чтобы она вывела определенную информацию из таблиц в память. Эта информация обычно посылается непосредственно на экран компьютера или терминала которым вы пользуетесь, хотя, в большинстве случаев, ее можно также послать принтеру, сохранить в файле ( как объект в памяти компьютера ), или представить как вводную информацию для другой команды или процесса.

Операторы SQL делятся на:

- операторы определения данных (Data Definition Language, DDL)

CREATE создает объект БД (саму базу, таблицу, представление, пользователя и т. д.)

ALTER изменяет объект

DROP удаляет объект

- операторы манипуляции данными (Data Manipulation Language, DML)

SELECT считывает данные, удовлетворяющие заданным условиям

INSERT добавляет новые данные

UPDATE изменяет существующие данные

DELETE удаляет данные

- операторы определения доступа к данным (Data Control Language, DCL)

GRANT предоставляет пользователю (группе) разрешения на определенные операции с объектом

REVOKE отзывает ранее выданные разрешения

- операторы управления транзакциями (Transaction Control Language, TCL)

COMMIT применяет транзакцию.

ROLLBACK откатывает все изменения, сделанные в контексте текущей транзакции.

Общий вид оператора выборки SELECT выглядит следующим образом. С помощью этого оператора можно выбирать данные.

SELECT выражение [FROM table\_references] [WHERE условие]

[GROUP BY {имя столбца | номер столбца } [ASC | DESC], ...]

[HAVING условие]

[ORDER BY { имя столбца | номер столбца } [ASC | DESC], ...]

Операторы модификации данных.

Добавление данных. Оператор INSERT.

INSERT INTO < таблица > [(список столбцов)] VALUES (вставляемые значения);

Список столбцов таблицы необязателен и обязан присутствовать лишь тогда, когда вставляемых значений меньше чем столбцов таблицы.

Изменение данных. Оператор UPDATE.

Для изменения данных следует применять оператор UPDATE.

UPDATE < имя таблицы > SET < атрибут > = значение,... [WHERE <условное выражение>];

Удаление данных. Оператор DELETE.

Для удаления кортежей из таблиц следует применять оператор DELETE.

DELETE FROM < имя таблицы > [WHERE <условное выражение>]

Пример:

USE [KinoFilms]

CREATE TABLE Director (

ID\_director int NOT NULL PRIMARY KEY IDENTITY(1, 1),

Name varchar(50) NOT NULL CHECK (Name <> ''),

Surname varchar(50) NOT NULL CHECK (Surname <> ''),

Birthyear date NOT NULL,

)

CREATE TABLE Country (

ID\_country int NOT NULL PRIMARY KEY IDENTITY(1, 1),

Country\_name varchar(50) NOT NULL CHECK (Country\_name <> ''),

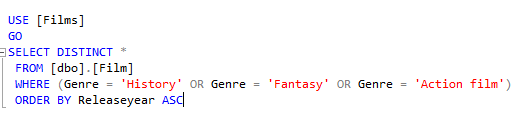
)

SELECT

Name\_film, Genre, Releaseyear, Name\_Adward, Year\_handing, Nomination

FROM Film, Adward

WHERE Film.ID\_film = Adward.ID\_film



# 18. Хранимые процедуры, триггеры, транзакции.

**Храни́мая процеду́ра** — объект базы данных, представляющий собой набор SQL-инструкций, который компилируется один раз и хранится на сервере. Хранимые процедуры очень похожи на обыкновенные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные, в них могут производиться числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам. В хранимых процедурах могут выполняться стандартные операции с базами данных (как DDL, так и DML). Кроме того, в хранимых процедурах возможны циклы и ветвления, то есть в них могут использоваться инструкции управления процессом исполнения.

Кроме собственно выполнения запроса, хранимые процедуры позволяют также производить вычисления и манипуляцию данными — изменение, удаление, выполнять DDL-операторы (не во всех СУБД!) и вызывать другие хранимые процедуры, выполнять сложную транзакционную логику. Один-единственный оператор позволяет вызвать сложный сценарий, который содержится в хранимой процедуре, что позволяет избежать пересылки через сеть сотен команд и, в особенности, необходимости передачи больших объёмов данных с клиента на сервер.

В большинстве СУБД при первом запуске хранимой процедуры она компилируется (выполняется синтаксический анализ и генерируется план доступа к данным). В дальнейшем её обработка осуществляется быстрее.

Использование хранимых процедур позволяет ограничить или вообще исключить непосредственный доступ пользователей к таблицам базы данных, оставив пользователям только разрешения на выполнение хранимых процедур, обеспечивающих косвенный и строго регламентированный доступ к данным. Кроме того, некоторые СУБД поддерживают шифрование текста (wrapping) хранимой процедуры. Эти функции безопасности позволяют изолировать от пользователя структуру базы данных, что обеспечивает целостность и надежность базы. Снижается вероятность таких действий как «внедрение SQL-кода», поскольку хорошо написанные хранимые процедуры дополнительно проверяют входные параметры перед тем, как передать запрос СУБД.

Пример:

Так, в ходе выполнения были разработаны две хранимые процедуры, которые позволяют узнать, если ли у определенного фильма награды и сколько компаний образовалось до 1939 года. Запросы в SQL сервере приведены ниже:

CREATE PROCEDURE Adwards\_films (@Name\_film varchar(50))

AS

BEGIN

DECLARE @count int;

SET @count = 0;

SET @count = (SELECT COUNT(\*) FROM Adward WHERE ID\_film =

(SELECT ID\_film FROM Film WHERE Name\_film = @Name\_film));

IF (@count > 0)

RETURN 1;

ELSE

RETURN 0;

END

Вторая хранимая процедура:

CREATE PROCEDURE CountOldCompany

AS

BEGIN

DECLARE @count int;

SET @count = (SELECT COUNT(\*) FROM Company WHERE Foundyear < 1939);

RETURN @count;

END

**Три́ггер** — это хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено наступлением определенного события (действием) — по сути добавлением INSERT или удалением DELETE строки в заданной таблице, или модификации UPDATE данных в определенном столбце заданной таблицы реляционной базы данных. Триггеры применяются для обеспечения целостности данных и реализации сложной бизнес-логики. Триггер запускается сервером автоматически при попытке изменения данных в таблице, с которой он связан. Все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера. Соответственно, в случае обнаружения ошибки или нарушения целостности данных может произойти откат этой транзакции.

Момент запуска триггера определяется с помощью ключевых слов BEFORE (триггер запускается до выполнения связанного с ним события; например, до добавления записи) или AFTER(после события). В случае, если триггер вызывается до события, он может внести изменения в модифицируемую событием запись (конечно, при условии, что событие — не удаление записи). Некоторые СУБД накладывают ограничения на операторы, которые могут быть использованы в триггере (например, может быть запрещено вносить изменения в таблицу, на которой «висит» триггер, и т. п.)

Кроме того, триггеры могут быть привязаны не к таблице, а к представлению (VIEW). В этом случае с их помощью реализуется механизм «обновляемого представления». В этом случае ключевые слова BEFORE и AFTER влияют лишь на последовательность вызова триггеров, так как собственно событие (удаление, вставка или обновление) не происходит.

В некоторых серверах триггеры могут вызываться не для каждой модифицируемой записи, а один раз на изменение таблицы. Такие триггеры называются табличными.

Пример:

1) При добавлении нового фильма триггер увеличивает число снятых режиссером фильмов на 1

DELIMITER //

CREATE TRIGGER ins\_film AFTER INSERT ON film

FOR EACH ROW

BEGIN

UPDATE director SET countfilms = countfilms + 1 WHERE director.ID\_director = New.ID\_director;

END//

DELIMITER ;

2) При удалении фильма триггер уменьшает число снятых режиссером фильмов на 1

DELIMITER //

CREATE TRIGGER del\_film AFTER DELETE ON film

FOR EACH ROW

BEGIN

UPDATE director SET countfilms = countfilms - 1 WHERE director.ID\_director = OLD.ID\_director;

END//

DELIMITER ;

**Транза́кция** — группа последовательных операций, которая представляет собой логическую единицу работы с данными. Транзакция может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще и тогда она не должна произвести никакого эффекта. Транзакции обрабатываются транзакционными системами, в процессе работы которых создаётся история транзакций. Различают последовательные (обычные), параллельные и распределённые транзакции. Распределённые транзакции подразумевают использование больше чем одной транзакционной системы и требуют намного более сложной логики (например, two-phase commit — двухфазный протокол фиксации транзакции). Также, в некоторых системах реализованы автономные транзакции, или под-транзакции, которые являются автономной частью родительской транзакции. Одним из наиболее распространённых наборов требований к транзакциям и транзакционным системам является набор ACID (Atomicity, Consistency, Isolation, Durability). Вместе с тем, существуют специализированные системы с ослабленными транзакционными свойствам. В идеале транзакции разных пользователей должны выполняться так, чтобы создавалась иллюзия, что пользователь текущей транзакции — единственный. Однако в реальности, по соображениям производительности и для выполнения некоторых специальных задач, СУБД предоставляют различные уровни изоляции транзакций.

Пример:

Транзакция используется при удалении режиссера, всех его фильмов и актеров, задействованных в этих фильмах.

if ((dataGridView6.Rows.Count > 1) && (dataGridView6.CurrentRow.Index < dataGridView6.Rows.Count - 1))

{

TDirector Dir = new TDirector();

Dir.ConnectToDBase(Connect);

MySqlTransaction trans = Dir.Connection.BeginTransaction(System.Data.IsolationLevel.Serializable);

Dir.Command.Transaction = trans;

try

{

Dir.DeleteDirector(dataGridView6.CurrentRow.Cells[0].Value.ToString(), dataGridView6.CurrentRow.Cells[1].Value.ToString());

trans.Commit();

}

catch

{

MessageBox.Show("Не удалось удалить данные");

trans.Rollback();

}

Dir.DisconnectFromDBase();

}

ShowDirector();

Уровни описаны в порядке увеличения изоляции транзакций и надёжности работы с данными

0 — Неподтверждённое чтение (Read Uncommitted, Dirty Read, грязное чтение) — чтение незафиксированных изменений своей транзакции и параллельных транзакций, возможны нечистые, неповторяемые чтения и фантомы

1 — Подтверждённое чтение (Read Committed) — чтение всех изменений своей транзакции и зафиксированных изменений параллельных транзакций, нечистые чтения невозможны, возможны неповторяемые чтения и фантомы;

2 — Повторяемое чтение (Repeatable Read, Snapshot) — чтение всех изменений своей транзакции, любые изменения, внесённые параллельными транзакциями после начала своей, недоступны, нечистые и неповторяемые чтения невозможны, возможны фантомы;

3 — Упорядоченный — (Serializable, сериализуемый) — упорядоченные (сериализуемые) транзакции. Идентичен ситуации, при которой транзакции выполняются строго последовательно, одна после другой, то есть результат действия которых не зависит от порядка выполнения шагов транзакции (запрещено чтение всех данных, изменённых с начала транзакции, в том числе и своей транзакцией). Фантомы невозможны.

Чем выше уровень изоляции, тем больше требуется ресурсов, чтобы их поддерживать.

# 19. Поиск информации в базе данных. Полнотекстовый поиск.

Поиск информации в базах данных – это процесс отбора из них множества описаний объектов, удовлетворяющих сформулированным в запросе условиям. При этом в качестве результатов поиска могут выдаваться не все признаки объектов, а только часть их – в соответствии с условиями запроса. Объект может выбираться из массива по значению одного идентифицирующего его (ключевого) признака или по сочетанию значений нескольких ключевых признаков. Он может также выбираться по сочетанию любых других (неключевых) признаков, если это сочетание однозначно выделяет его из множества всех объектов массива.

Различаются первичные и производные (в частности, обобщенные) признаки объектов. Первичные признаки назначаются при первоначальном описании объектов, а производные являются функциями первичных. Поиск может вестись как по первичным, так и по производным признакам. Чаще всего в процессе поиска информации выбирается не один объект, а множество объектов. Оно может быть задано различными способами:

1) перечнем значений ключевых признаков или сочетаний ключевых признаков;

2) значением или интервалом (перечнем) значений одного неключевого признака;

3) булевой функцией значений или интервалов (перечней) значений любых признаков объекта (как ключевых, так и неключевых);

4) отношением между признаками, выраженным с помощью арифметических и логических операции (операций типа “И”, “ИЛИ”, “НЕ”), а также отношений =, >, < и их отрицаний. Условия выборки признаков у найденных объектов задаются в виде перечней наименований этих признаков.

Важной проблемой, возникающей при реализации процедур поиска информации, является проблема отождествления признаков объектов и установления парадигматических отношений между ними (отношений типа род-вид, целое-часть и др.). Общее решение этой проблемы связано с возможностью распознавания смыслового тождества и парадигматических отношений различных форм наименований понятий на основе их морфологического, синтаксического и семантического анализа.

Поиск информации может выполняться за один или несколько шагов. В первом случае он ведется по одному запросу, во втором – по серии запросов. При многошаговом поиске возможны три основных способа организации процесса выполнения запросов:

**композиция запросов** – запросы выполняются в строго определенной последовательности, а результаты поиска по предыдущему запросу используются в качестве исходных данных для формирования следующего за ним запроса. При этом первый запрос в серии запросов определяется полностью, а остальные – не полностью и доопределяются в процессе поиска

**объединение запросов** – когда результаты поиска по нескольким запросам объединяются в одну общую выдачу.

**разветвление запросов** – когда после выполнения очередного запроса есть возможность перехода к одному из нескольких запросов в зависимости от выполнения тех или иных условий.

Перечисленные способы организации процесса выполнения запросов могут применяться в различных сочетаниях, что позволяет строить различные процедуры многошагового поиска.

SQL использует специальные операторы IN, BETWEEN, LIKE, и ISNULL.

Оператор IN определяет набор значений, в котором данное значение может или не может быть включено

SELECT \*

FROM Salespeople

WHERE city IN ('Barcelona','London');

BETWEEN определяет диапазон, значений. Вы должны ввести ключ слово BETWEEN с начальным значением, ключ AND и конечное значение.

SELECT \* FROM Salespeople

WHERE comm BETWEEN .10 AND .12;

ОПЕРАТОР LIKE применим только к полям типа CHAR или VARCHAR. Он ищет поле символов, чтобы видеть, совпадает ли с условием часть его строки. Имеются два типа групп символов использования с LIKE:

символ подчерк ( \_ ) замещает любой одиночный символ.

знак процента (%) замещает последов люб числа символов.

Пример:

SELECT

FROM Customers

WHERE cname LIKE 'G%';

**Полнотекстовый поиск.**

Полнотекстовые индексы обозначаются как индексы типа FULLTEXT. Эти индексы могут быть созданы в столбцах VARCHAR и TEXT во время создания таблицы командой CREATE TABLE или добавлении позже с помощью команды ALTER TABLE или CREATE INDEX. Загрузка больших массивов данных в таблицу будет происходить намного быстрее, если таблица не содержит индекс FULLTEXT, который затем создается командой ALTER TABLE (или CREATE INDEX). Загрузка данных в таблицу, уже имеющей индекс FULLTEXT, будет более медленной.

Полнотекстовый поиск выполняется с помощью ф-ции MATCH().

CREATE TABLE articles (

id INT UNSIGNED AUTO\_INCREMENT NOT NULL PRIMARY KEY,

title VARCHAR(200),

body TEXT,

FULLTEXT (title,body)

);

SELECT \* FROM articles

WHERE MATCH (title,body) AGAINST ('database');

Ф-ция MATCH() выполняет поиск в естественном языке, сравнивая строку с содержанием текста (совокупность одного или более столбцов, включенных в индекс FULLTEXT). Строка поиска задается как аргумент в выражении AGAINST(). Поиск выполняется без учета регистра символов.

В логическом режиме полнотекстового поиска поддерживаются след операторы:

+ это слово должно присут.

- что это слово не должно присут

( ) группирует слова в подвыражения.

~ воздейстует как оператор отрицания, обуславливает негативный вклад данного слова в релевантность строки. Им отмечают нежелательные слова.

\* оператор усечения. Он должен добавляться в конце слова.

" соответствует только строкам, содержащим эту фразу, написанную буквально.

# 20. Понятие операционной системы. Классификация. Функциональные компоненты.

**Операционная система (ОС)** – Программное обеспечение, которое с одной стороны выступает как интерфейс между аппаратурой компьютера и пользователя с его задачей, а с другой стороны предназначено для организации эффективного использования ресурсов этого компьютера и организации надежных вычислений.

**Классификация ОС**

*По количеству пользователей:*

1) Однопользовательские (MS DOS)

2) Многопользовательские (Windows 2003)

*По доступу (по типу) организации*

1) Системы распределения времени

2) Системы реального времени (QNX)

3) Пакетные (MS DOS)

*По количеству решаемых задач*

1) Однозадачные (MS DOS)

2) Многозадачные (Windows 2003)

*По виду интерфейса*

1) Символьные

2) Графические

*По назначению*

1) Общего назначения (повсеместно)

2) Сетевые ОС

3) Специального назначения (для управления БД, для решения задач реального времени)

**Функциональные компоненты ОС.**

*Основные подсистемы управления ресурсами – это подсистемы:*

1) управления процессами;

2) управления памятью;

3) управления файлами и внешними устройствами.

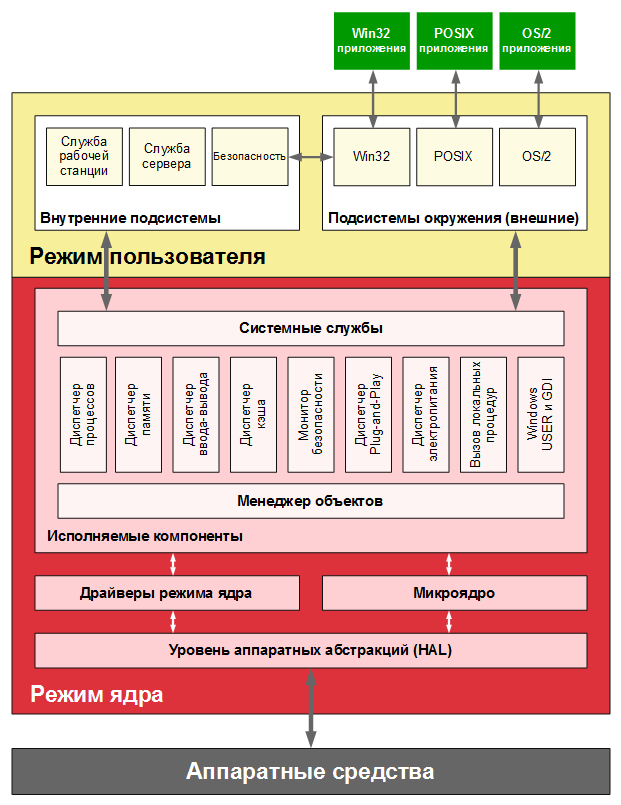
*Общие для всех ресурсов – это подсистемы:*

1) прикладного программного и пользовательского интерфейсов;

2) защиты данных и администрирования.

# 21. Архитектура операционной системы на примере одной из современных систем.

Архитектура Windows NT (New Technology)



Архитектура Windows NT имеет модульную структуру и состоит из двух основных уровней — компоненты, работающие в режиме пользователя, и компоненты режима ядра. Программы и подсистемы, работающие в режиме пользователя, имеют ограничения на доступ к системным ресурсам. Режим ядра имеет неограниченный доступ к системной памяти и внешним устройствам. Ядро системы NT называют гибридным ядром или макроядром. Архитектура включает в себя само ядро, уровень аппаратных абстракций (HAL), драйверы и ряд служб (Executives), которые работают в режиме ядра (Kernel-mode drivers) или в пользовательском режиме (User-mode drivers)[1][2].

Пользовательский режим Windows NT состоит из подсистем, передающих запросы ввода-вывода соответствующему драйверу режима ядра посредством менеджера ввода-вывода. Есть две подсистемы на уровне пользователя: подсистема окружения (запускает приложения, написанные для разных операционных систем) и интегрированная подсистема (управляет особыми системными функциями от имени подсистемы окружения). Режим ядра имеет полный доступ к аппаратной части и системным ресурсам компьютера. И также предотвращает доступ к критическим зонам системы со стороны пользовательских служб и приложений.

Подсистема окружения Win32 запускает 32-разрядные Windows приложения.

Подсистема окружения OS/2 поддерживает неграфические 16-разрядные приложения операционной системы OS/2 и эмулирует систему OS/2 2.1.x.

Подсистема окружения POSIX поддерживает приложения, написанные в соответствии со стандартом POSIX.1.

Hardware Abstraction Layer (HAL, Слой аппаратных абстракций) — слой абстрагирования, реализованный в программном обеспечении, находящийся между физическим уровнем аппаратного обеспечения и программным обеспечением, запускаемом на этом компьютере. HAL предназначен для скрытия различий в аппаратном обеспечении от основной части ядра операционной системы, таким образом, чтобы большая часть кода, работающая в режиме ядра, не нуждалась в изменении при её запуске на системах с различным аппаратным обеспечением. На персональных компьютерах HAL, по существу, может рассматриваться как драйвер материнской платы, позволяющий взаимодействовать инструкциям высокоуровневых языков программирования с низкоуровневыми компонентами, такими, как аппаратное обеспечение.

Менеджер объектов - Это исполнительная подсистема, к которой обращаются все остальные модули исполнительной подсистемы, в частности, системные вызовы, когда им необходимо получить доступ к ресурсам Windows NT. Менеджер объектов служит для уменьшения дублирования объектов, что может привести к ошибкам в работе системы. Для менеджера объектов каждый ресурс системы является объектом — будь то физический ресурс типа периферийного устройства, файловой системы, или логический ресурс — файл и др. Каждый объект имеет свою структуру, или тип объекта.

Менеджер безопасности обеспечивает идентификацию при регистрации пользователя

Менеджер кэша – хранит блоки диска использованных недавно для ускорения доступа

Менеджер локальных процедур – обеспечивает производительную работу процессора

Менеджер Plug-and-Play – уведомляет о новых появившихся устройствах

Windows USER и GDI – интерфейс графических устройств – позволяет пользователю данных выводить данные на монитор и принтер

Системные службы – предоставление доступа интерфейса к исполнительным системам

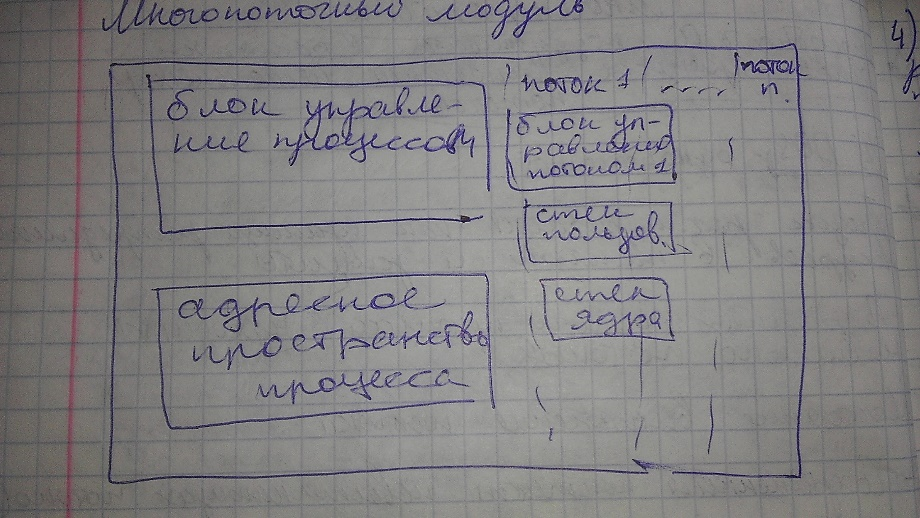
# 22. Процессы и потоки в многозадачной операционной системе. Алгоритмы планирования и управления.

Процесс – выполняемая программа, включающая значение счетчика команд, регистров и данных. В процессе работы компьютера процессор должен переключаться с одного процесса на другой (мультизадачность).

Многопоточность – способность ОС поддерживать в рамках одного процесса выполнение нескольких потоков. MS DOS – однопоточная.

Процесс рассматривается как единица распределения ресурсов памяти и защиты.

Многопоточный модуль



Стек ядра и пользователя предназначено для вызова процедур.

Переход потока из одного состояния в другое осуществляется в результате планирования диспетчеризации.

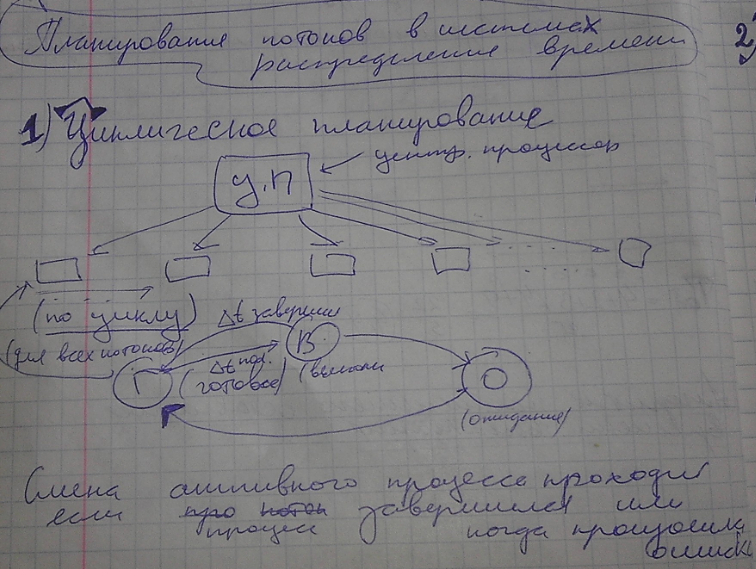
Под планированием понимается работа по определению того в какой момент времени необходимо прервать выполнение активного потока и предоставить возможность выполнения какому-то следующему потоку.

2 вида планировщика:

Статический – решение принимается по расписанию (светофор)

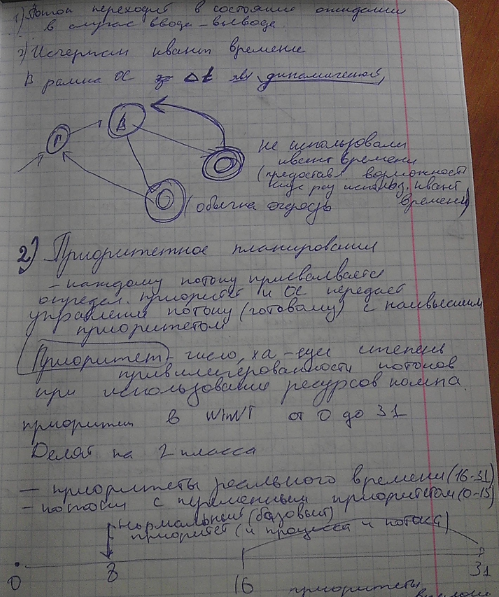
Динамический – решение в режиме онлайн на основе анализа ситуации (регулировщик)

В современных ОС используется вытесняющие алгоритмы, в которых решение переключения процессора с выполнения одного потока на другой выполняет ОС.



Смена активного процесса проходит если:

* поток завершился или когда произошла ошибка
* поток переходит в состояние ожидания
* исчерпан квант времени



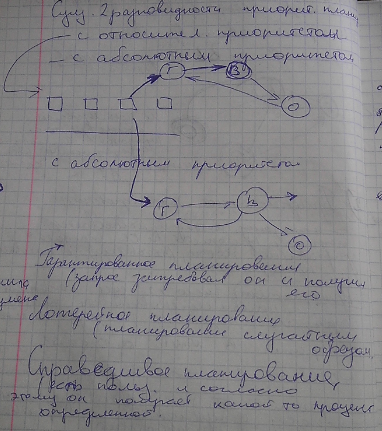
2) Приоритетное планирование

Каждому потоку присваивается определённый приоритет и ОС передает управление потоку (готовому) с наивысшим приоритетом.

Приоритет - число, характеризующее степень привилегированности потока при использовании ресурсов компьютера.

Приоритеты в WinNT - от 0 до 31. 2 класса:

* приоритеты реального времени (16-31)
* потоки с переменным приоритетом (0-15)



Существуют 2 разновидности приоритетов планирования:

* с относительным приоритетом (2 рисунок)
* с абсолютным приоритетом (1 рисунок)

**Гарантированное планирование**

Подход к планированию заключается в предоставлении пользователям реальных обещаний относительно производительности, а затем в выполнении этих обещаний. Одно из обещаний, которое можно дать и просто выполнить, заключается в следующем: если в процессе работы в системе зарегистрировано п пользователей, то вы получите 1/я от мощности центрального процессора. Аналогично этому, в однопользовательской системе, имеющей я работающих процессов, при прочих равных условиях каждый из них получит 1/я от общего числа процессорных циклов. Это представляется вполне справедливым решением.

Чтобы выполнить это обещание, система должна отслеживать, сколько процессорного времени затрачено на каждый процесс с момента его создания. Затем она вычисляет количество процессорного времени, на которое каждый из них имел право, а именно время с момента его создания, деленное на п. Поскольку также известно и количество времени центрального процессора, уже полученное каждым процессом, нетрудно подсчитать соотношение израсходованного и отпущенного времени центрального процессора. Соотношение 0,5 означает, что процесс получил только половину от того, что должен был получить, а соотношение 2,0 означает, что процесс получил вдвое больше времени, чем то, на которое он имел право. Согласно алгоритму, после этого будет запущен процесс с самым низким соотношением, который будет работать до тех пор, пока его соотношение не превысит соотношение его ближайшего конкурента.

**Лотерейное планирование**

Планирование случайным образом. Основная идея состоит в раздаче процессам лотерейных билетов на доступ к различным системным ресурсам, в том числе и к процессорному времени. Когда планировщику нужно принимать решение, в случайном порядке выбирается лотерейный билет, и ресурс отдается процессу, обладающему этим билетом. Применительно к планированию процессорного времени система может проводить лотерейный розыгрыш 50 раз в секунду, и каждый победитель будет получать в качестве приза 20 мс процессорного времени.

**Справедливое планирование**

В этой модели каждому пользователю распределяется некоторая доля процессорного времени и планировщик выбирает процессы, соблюдая это распределение. Таким образом, если каждому из двух пользователей было обещано по 50% процессорного времени, то они его получат, независимо от количества имеющихся у них процессов.



**Планирование потоков в Windows NT**

Вытесняющий алгоритм планирования на основе уровней приоритета (будет выполнен поток из очереди готовых с большим приоритетом и он будет работать определённый квант времени; если появится поток с большим приоритетом, то активный поток вытесняется. Если заканчивается квант времени, то из очереди берётся следующий с этим же приоритетом, если в очереди готовых таких потоков нет, тот этот же поток будет выполняться снова)

Состояния потоков:

* Инициализация - создание структуры потоков
* Готовность - состояние, в котором есть все, кроме кванта времени
* Простаивание - поток проходит через это состояние из состояния готовности перед выполнением
* Выполнение - поток выполняется на процессоре, он заканчивает выполнение, когда квант времени закончился или завершился код программы (или произошел ввод / вывод)
* Ожидание - операция ввода / вывода (поток ждет)
* Транзит - планировщик переводит ожидающий поток в состояние готовности сразу же, как только соответствующие дескрипторы оказываются в сигнальном состоянии, хотя при этом поток фактически проходит через промежуточное переходное состояние
* Завершение - поток выполнен

**32 уровневая система приоритетов**

16-31 - приоритеты реального уровня

1-15 - динамический уровень

0 - системный уровень (idle)

*Приоритет потока устанавливается при инициализации (создании) потока*

24 - приоритет реального времени

13 - высокий приоритет

10 - выше обычного (среднего)

8 - обычный (нормальный / средний)

6 - ниже обычного

4 - простаивающий приоритет

# 23. Управление памятью в операционной системе. Физическая память. Виртуальная память. Адресное пространство. Алгоритмы замещения страниц памяти.

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Распределению подлежит вся оперативная память, не занятая операционной системой. Обычно ОС располагается в самых младших адресах, однако может занимать и самые старшие адреса.

**Функциями ОС по управлению памятью являются:**

1) отслеживание свободной и занятой памяти,

2) выделение памяти процессам и освобождение памяти при завершении процессов,

3) вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место,

4) настройка адресов программы на конкретную область физической памяти.

Типы памяти в ОС. Иерархическая структура памяти. Кэш-память. ОЗУ, ПЗУ, внешняя память на дисках. Физическая и виртуальная память.

Память:

1 - основная (физическая – оперативная);

2 - дополнительная(внешняя, диски).

Основная память – это упорядоченный массив 1-байтовых ячеек, каждая из которых имеет свой уникальный адрес. Внешняя память – 1-мерное линейное адресное пространство, состоящее из последовательности байтов. Иерархия памяти:

1) Регистровая память процессора(самая быстрая)

2) КЭШ процессора

3) Основанная память

4) Внешняя(эл. диски)

5) Внешняя (магн диски)

6) Магнитная лента

**Физическая память**

Физическая память (physical memory) - это реальные микросхемы RAM (Random Access Memory), установленные в компьютере. Каждый байт физической памяти имеет физический адрес (physical address), который представляет собой число от нуля до числа на единицу меньшего, чем количество байтов физической памяти. Например, ПК с установленными 64 Мб RAM, имеет физические адреса &Н00000000-&Н04000000 в шестнадцатеричной системе счисления, что в десятичной системе будет 0-67 108 863.

Физическая память (в отличие от файла подкачки и виртуальной памяти) является исполняемой(executable), то есть памятью, из которой можно читать и в которую центральный процессор может посредством системы команд записывать данные.

**Виртуальная память**

Виртуальная память (virtual memory) - это просто набор чисел, о которых говорят как о виртуальных адресах. Программист может использовать виртуальные адреса, но Windows не способна по этим адресам непосредственно обращаться к данным, поскольку такой адрес не является адресом реального физического запоминающего устройства, как в случае физических адресов и адресов файла подкачки. Для того чтобы код с виртуальными адресами можно было выполнить, такие адреса должны быть отображены на физические адреса, по которым действительно могут храниться коды и данные. Эту операцию выполняет диспетчер виртуальной памяти (Virtual Memory Manager - VMM). Операционная система Windows обозначает некоторые области виртуальной памяти как области, к которым можно обратиться из программ пользовательского режима. Все остальные области указываются как зарезервированные. Какие области памяти доступны, а какие зарезервированы, зависит от версии операционной системы (Windows 9x или Windows NT).

Каждый процесс Win32 получает виртуальное адресное пространство (совокупность виртуальных адресов процесса) (virtual address space), называемое также адресным пространством, или пространством процесса (process space), объем которого равен 4 Гб (для 32 разрядных систем). Таким образом, код процесса может ссылаться на адреса с &Н00000000 по &HFFFFFFFF (или с 0 по 2 в степени 32 - 1 = 4 294 967 295 в десятичной системе счисления). Конечно, так как виртуальные адреса - это просто числа, заявление о том, что каждый процесс получает свое собственное виртуальное адресное пространство, выглядит довольно бессмысленным. (Это все равно, что сказать, что каждый человек получает свой собственный диапазон возраста от 0 до 150).

Тем не менее, это утверждение должно означать, что Windows не видит ни какой взаимосвязи в том, что и процесс А, и процесс В используют один и тот же виртуальный адрес, например &Н40000000. В частности, Windows может сопоставить (или не сопоставить) виртуальным адресам каждого процесса разные физические адреса

**Алгоритмы замещения страниц**

Когда происходит страничное прерывание, операционная система должна выбрать страницу для удаления из памяти, чтобы освободить место для страницы, кото­рую нужно перенести в память. Если удаляемая страница была изменена за время своего присутствия в памяти, ее необходимо переписать на диск, чтобы обновить копию, хранящуюся там. Однако если страница не была модифицирована (на­пример, она содержит текст программы), копия на диске уже является самой но­вой и ее не надо переписывать. Тогда страница, которую нужно прочитать, просто считывается поверх выгружаемой страницы.

**Алгоритмы замещения страниц. «Оптимальный».**

Предположим, что в ОП находится некоторый набор страниц. К каждой из этих страниц происходит обращение команд процессора. Поэтому каждая страница может быть помечена числом равным кол-ву команд, которые будут выполняться перед первым обращением к этой странице. Оптимальный алгоритм предполагает выгружать из ОП виртуальную страницу с наибольшим этим числом. Однако, с этим алгоритмом связана проблема: его нельзя реализовать с 1-го раза, т.к. в момент страничного прерывания ОС не знает, когда произойдёт следующее обращение к виртуальной странице.

Метод полезен для оценки других алгоритмов замещения страниц, а сам не используется в практике ОС.

**Алгоритмы замещения страниц. «NRU (Not Recently Used)».**

Алгоритм не использующихся в последнее время страниц.

Исходная информация-таблица страниц.

В таблице страниц есть бит обращения Reference R. Бит R принимает значение=1, когда к этой странице происходит обращение

M-модификация (когда происходит в страничке изменения)

Биты R и M находятся в дескрипторе таблиц страниц.

Биты R и M во всех страницах=0 периодически(например при каждом прерывании по таймеру) бит R очищается, чтобы отличить страницы, к которым давно не происходило обращение от тех, на которые были ссылки. Когда возникает страничное прерывание, необходимо отобразить страницу из ОП, то ОС проверяет все страницы и делит их на 4 категории на основании текущих значений битов R и M.

0-M=0 R=0(не было ни обращений, ни изменений)

1-M=1 R=0

2-R=1 M=0

3-R=1 M=1

Алгоритм NRU удаляет страницу из ОП с помощью случайного поиска в непустом классе с наименьшим номером.

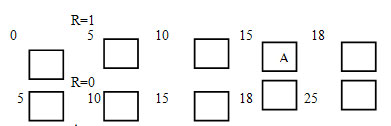
Алгоритмы замещения страниц. «FIFO».

Для всех стр. находящихся в оперативной памяти ОС поддерживает список в котором первая стр. является старейшей, а стр. в хвосте списка попала в него недавно. Когда происходит страничное прерывание удаляется стр. находящаяся в голове списка, а новая стр. добавляется в хвост.

Алгоритм FIFO может выгрузить из памяти часть используемой страниц.

**Алгоритмы замещения страниц. «Вторая попытка».**

Алгоритм FIFO может выгрузить из памяти часть используемой страниц, поэтому в алгоритме «вторая попытка» предлагается использовать значение бита R. Если он =0, значит стр. недолго находиться в памяти и не используется, поэтому она замещается новым, если же он =1, то ему присваивается значение 0 и стр. переносится в конец списка, а время ее загрузки обнуляется.



Алгоритм «вторая попытка» ищет в списке самую старую страницу, к которой не было обращений.

**Алгоритмы замещения страниц. «Часы».**

Алгоритм «вторая попытка» является неэффективным, потому что он постоянно передвигает стр. по списку лучше хранить все стр. в кольцевом списке в форме часов, стрелка указывает на старейшую страницу. Если происходит страничное прерывание, то та стр. на которую направлена стрелка. Если ее бит R =0, то стр. выгружается, а на ее место в часовой круг встает новая стр., а стрелка сдвигается вперед на 1 позицию. Если R=1, то он сбрасывается и стрелка перемещается к следующей стр. Этот процесс повторяется до тех пор пока не найдется стр. с битом R=0.

**Алгоритмы замещения страниц. «LRU (Lveast Recently Used) – не использующаяся дольше всего».**

В основе лежит предположение, что страницы к которым происходило многократное обращение в нескольких последних командах вероятно так же будут использоваться и в дальнейшем. Те стр. к которым ранее не было обращений будут еще долго не использоваться. В этом алгоритме строится матрица n x n бит. Когда происходит обращение к k-й странице, все биты строки k=1, а все биты столбца k=0.

Удаляется стр. у которой в строке наименьшее значение. Этот алгоритм сложно реализавать.

**Алгоритмы замещения страниц. «Старение».**

Это модификация NFU (редко используемая страница) (к счетчику при каждом прерывание прибавляется значение модификатора R(обращение). Каждый счетчик сдвигается вправо на 1 разряд перед прибавлением значения бита R, бит R добавляется в крайний слева, а не в крайний справа бит счетчика.

Пример:

бит R 101011 110010 110101

Тик0 1 2

0 1000 1100 1110

1 0000 1000 1100

2 1000 0100 0010

3 1000 0000 1000

4 1000 1100 0110

5 1000 0100 1010

Когда происходит страничное прерывание, удаляется та страница, чей счетчик имеет наименьшую величину.

**Алгоритмы замещения страниц. «Рабочий набор».**

«рабочий набор» - множество страниц которое процесс использует в данный момент.

Текущее виртуальное время – время работы процессора, которое фактически использует процесс с момента запуска. В этом алгоритме выводится новое поле (для времени последнего использования).

При каждом страничном прерывании исследуется таблица страниц и ищется страница, подходящая для удаления из памяти.(проверяется R). Если он=1, то текущее виртуальное время записывается в поле «время последнего использования». Если R=0, то к этой странице не было обращений в течение последнего тика часов. Она может быть кандидатом на удаление. Из списка кандидатов удаляется та страница, у которой максимальный возраст, который определяется как разность между текущим виртуальным временем и значением в поле Время последнего использования. Если у всех страниц R=1, то удаляется случайная страница, у которой m=0. Если R=1 и m=1, то случайным образом выбирается и удаляется страница.

**Алгоритмы замещения страниц. «WSClock».**

Модификация «рабочего набора». Основывается на часовом алгоритме. Для алгоритма необходима структура страниц в виде кольцевого списка. Как и в случае алгоритма «часы» при страничном прерывании первой проверяется та страница, на которую указывает стрелка. Если бит R=1, то это означает, что эта страница не является кандидатом на удаление, если R=0, стрелка передвигается на следующую страницу. Страница проверяется m. Если m=0, то страница чистая, если m=1, страница «грязная» и стрелка делает движение вперед на 1 позицию. Если стрелка делает целый круг и возвращается в начальное положение, то выбирается страница с максимальным возрастом.

# 24. Межпроцессное взаимодействие. Механизмы взаимодействия процессов.

**Межпроцессное взаимодействие** (***i****nter-****p****rocess* ***c****ommunication*, **IPC**) — обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

Из механизмов, предоставляемых ОС и используемых для IPC, можно выделить:

* механизмы обмена сообщениями;
* механизмы синхронизации;
* механизмы разделения памяти;
* механизмы удалённых вызовов (RPC).

Для оценки производительности различных механизмов IPC используют следующие параметры:

* пропускная способность (количество сообщений в единицу времени, которое ядро ОС или процесс способна обработать);
* задержки (время между отправкой сообщения одним потоком и его получением другим потоком).

IPC может называться терминами ***межпотоковое взаимодействие*** (***i****nter-****t****hread* ***c****ommunication*) и ***межпрограммное взаимодействие*** (***i****nter-****a****pplication****c****ommunication*).

**Механизмы взаимодействия процессов:**

**1. Каналы.** Типичным механизмом обмена данными для процессов является использование потоков: стандартного потока ввода, вывода, ошибок, файловых потоков и т.п. Стандартные потоки ввода, вывода и ошибок, по сути, являются *каналами*, и такие каналы можно или создавать между родственными процессами (*безымянные каналы*), или привязывать к узлам файловой системы, позволяя ими пользоваться любым процессам, которые имеют к ним доступ (*именованные каналы*).

*Полудуплексным каналом* называется системный буфер, очередь, представляемая парой дескрипторов. Один из этих дескрипторов доступен на запись и позволяет помещать данные в канал, другой — доступен на чтение и позволяет получать данные из канала. Так как открытые дескрипторы сохраняются при использовании функции [fork](http://pubs.opengroup.org/onlinepubs/009695399/functions/fork.html), каналы можно использовать для обмена данными между родственными процессами.

**2. Сигналы.** Сигналы являются программными прерываниями, которые посылаются процессу, когда случается некоторое событие. Сигналы могут возникать синхронно с ошибкой в приложении. Сигналы могут посылаться процессу, если система обнаруживает программное событие, например, когда пользователь дает команду прервать или остановить выполнение, или получен сигнал на завершение от другого процесса. Сигналы могут прийти непосредственно от ядра ОС, когда возникает сбой аппаратных средств ЭВМ. Система определяет набор сигналов, которые могут быть отправлены процессу. При этом каждый сигнал имеет целочисленное значение и приводит к строго определенным действиям.

Известно три варианта реакции на сигналы:

1. вызов собственной функции обработки;
2. игнорирование сигнала;
3. использование предварительно установленной функции обработки по умолчанию.

Чтобы реагировать на разные сигналы, необходимо знать концепции их обработки. Процесс должен организовать так называемый обработчик сигнала в случае его прихода.

**3. Очереди сообщений.** Очереди сообщений как средство межпроцессной связи позволяют процессам взаимодействовать, обмениваясь данными. Данные передаются между процессами дискретными порциями, называемыми сообщениями. Процессы, использующие этот тип межпроцессной связи, могут выполнять две операции: *послать или принять сообщение*.

Процесс, прежде чем послать или принять какое-либо сообщение, должен запросить систему породить программные механизмы, необходимые для обработки данных операций.

Процессы, имеющие права на операции и пытающиеся послать или принять сообщение, *могут приостанавливаться*, если выполнение операции не было успешным. В частности это означает, что процесс, пытающийся послать сообщение, может ожидать, пока процесс-получатель не будет готов; и наоборот, получатель может ждать отправителя. Если указано, что процесс в таких ситуациях должен приостанавливаться, говорят о выполнении над сообщением «*операции с блокировкой*». Если приостанавливать процесс нельзя, говорят, что над сообщением выполняется ''*операция без блокировки*''.

**4. Семафоры.** Семафоры являются одним из классических примитивов синхронизации. Семафор (semaphore) - это целая переменная, значение которой можно опрашивать и менять только при помощи неделимых (атомарных) операций.

Двоичный семафор может принимать только значения 0 или 1. Считающий семафор может принимать целые неотрицательные значения.

В приложениях как правило требуется использование более одного семафора, ОС должна представлять возможность создавать множества семафоров.

Над каждым семафором, принадлежащим некоторому множеству, можно выполнить любую из трех операций:

* увеличить значение;
* уменьшить значение;
* дождаться обнуления.

Для выполнения первых двух операций у процесса должно быть право на изменение, для выполнения третьей достаточно права на чтение. Для третьей операции нужно передать 0; если текущее значение семафора отлично от нуля, операция не может быть успешно выполнена.

**5. Разделяемая память.** Разделяемая память может быть наилучшим образом описана как отображение участка (сегмента) памяти, которая будет разделена между более чем одним процессом. Это гораздо более быстрая форма IPC, потому что здесь нет никакого посредничества (т.е. каналов, очередей сообщений и т.п.). Вместо этого, информация отображается непосредственно из сегмента памяти в адресное пространство вызывающего процесса. Сегмент может быть создан одним процессом и впоследствии использован для чтения/записи любым количеством процессов.

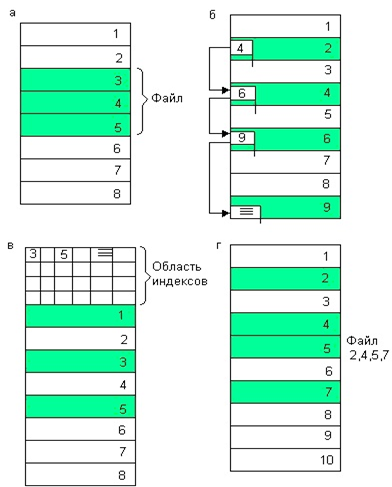
**6. Удалённый вызов процедур**, реже Вызов удалённых процедур (от *Remote Procedure Call, RPC*) — класс технологий, позволяющих компьютерным программам вызывать функции или процедуры в другом адресном пространстве (как правило, на удалённых компьютерах). Обычно, реализация RPC технологии включает в себя два компонента: сетевой протокол для обмена в режиме клиент-сервер и язык сериализации объектов (или структур, для необъектных RPC). Различные реализации RPC имеют очень отличающуюся друг от друга архитектуру и разнятся в своих возможностях. На транспортном уровне RPC используют в основном протоколы TCP и UDP, однако, некоторые построены на основе HTTP (что нарушает архитектуру ISO/OSI, так как HTTP изначально не транспортный протокол).

# 25. Файловые системы. Основные особенности и физическая организация на примере одной из современных файловых систем.

**Фа́йловая систе́ма** (*file system*) — порядок, определяющий способ организации, хранения и именования данных на носителях информации в компьютерах, а также в другом электронном оборудовании: цифровых фотоаппаратах, мобильных телефонах и т. п. Файловая система определяет формат содержимого и способ физического хранения информации, которую принято группировать в виде файлов. Конкретная файловая система определяет размер имен файлов и (каталогов), максимальный возможный размер файла и раздела, набор атрибутов файла. Некоторые файловые системы предоставляют сервисные возможности, например, разграничение доступа или шифрование файлов.

Файловая система связывает носитель информации с одной стороны и API для доступа к файлам — с другой. Именно файловая система устанавливает, где и как будет записан файл на физическом носителе (например, жёстком диске).

В общем случае жесткий диск состоит из набора пластин, покрытых магнитным слоем.



Непрерывное размещение — простейший вариант физической организации *(рисунок (а))*, при котором файлу предоставляется последовательность кластеров диска, образующих непрерывный участок дисковой памяти.

Следующий способ физической организации — размещение файла в виде связанного списка кластеров дисковой памяти *(рисунок (б))*. При таком способе в начале каждого кластера содержится указатель на следующий кластер. В этом случае адресная информация минимальна: расположение файла может быть задано одним числом — номером первого кластера.

Популярным способом, применяемым, например, в файловой системе FAT, является использование связанного списка индексов *(рисунок (в)).* Этот способ является некоторой модификацией предыдущего. Файлу также выделяется память в виде связанного списка кластеров. Номер первого кластера запоминается в записи каталога, где хранятся характеристики этого файла. Остальная адресная информация отделена от кластеров файла. С каждым кластером диска связывается некоторый элемент — индекс. Индексы располагаются в отдельной области диска.

Еще один способ задания физического расположения файла заключается в простом перечислении номеров кластеров, занимаемых этим файлом *(рисунок (г)).* Этот перечень и служит адресом файла. Недостаток данного способа очевиден: длина адреса зависит от размера файла и для большого файла может составить значительную величину. Достоинством же является высокая скорость доступа к произвольному кластеру файла, так как здесь применяется прямая адресация, которая исключает просмотр цепочки указателей при поиске адреса произвольного кластера файла. Фрагментация на уровне кластеров в этом способе также отсутствует.

**Пример:**

NTFS превосходно справляется с обработкой больших массивов данных и достаточно хорошо проявляет себя при работе с томами объемом от 300-400 Мбайт и выше. Максимально возможные размеры тома и файла составляют 16ЭБ (220ТБ). Количество файлов в каталогах не ограничено. В основу структуры каталогов NTFS заложено B-Tree.

Как и многие другие системы, NTFS делит все полезное дисковое пространство тома на кластеры - блоки данных, адресуемые как единицы данных. NTFS поддерживает размеры кластеров от 512Б до 64КБ; стандартом же считается кластер размером 2 или 4 КБ.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| M  F  T | Зона MFT | Зона для размещения файлов и каталогов | Копия | Зона для размещения файлов и каталогов |

**Структура тома NTFS**

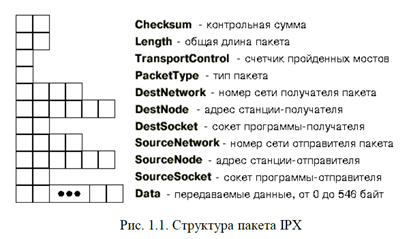
Все дисковое пространство в NTFS делится на две неравные части. Первые 12% диска отводятся под так называемую MFT-зону - пространство, которое может занимать, увеличиваясь в размере, главный служебный *метафайл* MFT. MFT (master file table) - это специальный файл, главная системная структура данных, которая и позволяет определять местонахождение всех остальных файлов. MFT поделен на записи фиксированного размера 1КБ (минимум 1КБ и максимум 4КБ), и каждая запись соответствует какому-либо файлу (в общем смысле этого слова). Первые 16 файлов носят служебный характер и недоступны ОС - они называются *метафайлами*, причем самый первый метафайл - сам MFT. Эти 16 элементов MFT - единственная часть диска, имеющая строго фиксированное положение. Копия первых 16 записей хранится в середине тома для надежности. Каждый из упомянутых первых 16 файлов NTFS отвечает за какой-либо аспект работы системы и находится в корневом каталоге NTFS-тома. Все они начинаются с символа "$".

# 26. Каналы передачи данных. Физический канал. Логический канал. Понятие блока данных. Пример формата блока данных любого протокола

**Канал передачи данных** - это средства двухстороннего обмена данными, которые включают в себя линии связи и аппаратуру передачи (приема) данных. Каналы передачи данных связывают между собой источники информации и приемники информации.

Каналы связи (data link) создаются по линиям связи при помощи сетевого оборудования и физических средств связи. **Физические** средства связи построены на основе витых пар, коаксиальных кабелей, оптических каналов или эфира. Между взаимодействующими информационными системами через физические каналы коммуникационной сети и узлы коммутации устанавливаются логические каналы. **Логический канал** – это путь для передачи данных от одной системы к другой. Логический канал прокладывается по маршруту в одном или нескольких физических каналах. Логический канал можно охарактеризовать как маршрут, проложенный через физические каналы и узлы коммутации.

Информация в сети передается блоками (пакетами) данных по процедурам обмена между объектами. Эти процедуры называют протоколами передачи данных.

Пакет - основная единица информации в компьютерных сетях. При разбиении данных на пакеты скорость их передачи возрастает настолько, что каждый компьютер в сети получает возможность принимать и передавать данные практически одновременно с остальными компьютерами. На целевом компьютере (компьютере-получателе) пакеты накапливаются и выстраиваются в должном порядке для восстановления исходного вида данных.

Пример 2. Пакет HTTP:

Каждое HTTP-сообщение состоит из трех частей, которые передаются в следующем порядке:

1. Стартовая строка – определяет тип сообщения; Стартовые строки HTTP-сообщения различаются для запроса и ответа.
2. Заголовки – характеризуют тело сообщения, параметры передачи и прочие сведения;
3. Тело сообщения – непосредственно данные сообщения.

# 27. Структуризация сетей. Понятие и характеристики основных сетевых топологий. Структурообразующие аппаратные средства и программное обеспечение

**Сетевая тополо́гия** — это конфигурация графа, вершинам которого соответствуют конечные узлы сети (компьютеры) и коммуникационное оборудование (маршрутизаторы), а рёбрам — физические или информационные связи между вершинами.

**Основные сетевые топологии**

**Полносвязная**

Сеть, в которой каждый компьютер непосредственно связан со всеми остальными. Однако этот вариант громоздкий и неэффективный, потому что каждый компьютер в сети должен иметь большое количество коммуникационных портов, достаточное для связи с каждым из остальных компьютеров.

**Неполносвязная**

Неполносвязных топологий существует несколько. В них, в отличие от полносвязных может применяться передача данных не напрямую между компьютерами, а через дополнительные узлы.

**Шина (Bus)**

Топология данного типа, представляет собой общий кабель (называемый шина или магистраль), к которому подсоединены все рабочие станции. На концах кабеля находятся терминаторы, для предотвращения отражения сигнала.

Преимущества сетей шинной топологии:

* расход кабеля существенно уменьшен
* отказ одного из узлов не влияет на работу сети в целом;
* сеть легко настраивать и конфигурировать;
* сеть устойчива к неисправностям отдельных узлов.

Недостатки сетей шинной топологии:

* разрыв кабеля может повлиять на работу всей сети;
* ограниченная длина кабеля и количество рабочих станций;
* недостаточная надежность сети из-за проблем с разъемами кабеля.
* низкая производительность, обусловлена разделением канала между всеми абонентами.

**Звезда**

В сети построенной по топологии типа «звезда» каждая рабочая станция подсоединяется кабелем (витой парой) к концентратору или хабу ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *hub*). Концентратор обеспечивает параллельное соединение ПК и, таким образом, все компьютеры, подключенные к сети, могут общаться друг с другом.

Преимущества сетей топологии звезда:

* легко подключить новый ПК;
* имеется возможность централизованного управления;
* сеть устойчива к неисправностям отдельных ПК и к разрывам соединения отдельных ПК.

Недостатки сетей топологии звезда:

* отказ хаба влияет на работу всей сети;
* большой расход кабеля.

Звезда – централизованная топология, так же встречаются и децентрализованные топологии. Децентрализация: в сетевой топологии существуют по крайней мере два узла с двумя или больше путями между ними, чтобы обеспечить дополнительные пути, которые будут использоваться в случае, если один из путей выйдет из строя. Эта децентрализация часто используется, чтобы компенсировать недостаток выхода из строя одного пункта, используя единственное устройство в качестве центрального узла (например, в звезде и сетях дерева). Специальный вид сети, ограничивающий число путей между двумя узлами, называется гиперкубом.

**Кольцо**

В сети с топологией кольцо все узлы соединены каналами связи в неразрывное кольцо (необязательно окружность), по которому передаются данные. Выход одного ПК соединяется со входом другого ПК. Начав движение из одной точки, данные, в конечном счете, попадают на его начало. Данные в кольце всегда движутся в одном и том же направлении.

Принимающая рабочая станция распознает и получает только адресованное ей сообщение. В сети с топологией типа физическое кольцо используется маркерный доступ, который предоставляет станции право на использование кольца в определенном порядке. Логическая топология данной сети — логическое кольцо. Данную сеть очень легко создавать и настраивать.

К основному недостатку сетей топологии кольцо является то, что повреждение линии связи в одном месте или отказ ПК приводит к неработоспособности всей сети.

Как правило, в чистом виде топология «кольцо» не применяется из-за своей ненадёжности, поэтому на практике применяются различные модификации кольцевой топологии.

**Ячеистая топология**

Получается из полносвязной топологии путём удаления некоторых связей. Допускает соединения большого количества компьютеров и характерна для крупных сетей.

Дополнительные способы являются комбинациями базовых. В общем случае такие топологии называются смешанными или гибридными, но некоторые из них имеют собственные названия, например «Дерево».

**Смешанная топология** — сетевая топология, преобладающая в крупных сетях с произвольными связями между компьютерами. В таких сетях можно выделить отдельные произвольно связанные фрагменты (*подсети*), имеющие типовую топологию, поэтому их называют сетями со смешанной топологией.

**ТЕХНИЧЕСКИЕ СРЕДСТВА СЕТЕЙ**

Технические средства локальных сетей включают в себя следующие функциональные группы оборудования:

1. Средства линий передачи данных (кабель, витая пара, оптоволокно) – реализуют собственно перенос сигналов;
2. Средства увеличения дистанции передачи данных (репитер, усилитель, модемы) – осуществляют усиление сигнала или преобразование в форму, удобную для дальнейшей передачи;
3. Средства повышения ёмкости линий передачи (мульплексоры) – позволяют реализовывать несколько логических каналов в рамках одного физического соединения путём разделения частот передачи, чередования пакетов во времени и т.д.
4. Средства управления информационными потоками в сети (коммутация каналов, коммутация пакетов, разветвление линий передачи) – осуществляют адресацию сообщений;
5. Средства соединения линий передачи с сетевым оборудованием узлов (сетевые платы, адаптеры) Предназначены для: взаимодействия с другими устройствами в локальной сети; повышения производительности, назначают приоритеты для ответственного трафика, поддерживают удалённую активизацию связи с центральной рабочей станцией, поддерживают удалённое изменение конфигурации, что значительно экономит время и силы администраторов постоянно растущих сетей.

Операционная система сети включает в себя набор управляющих и обслуживающих программ, обеспечивающих:

* межпрограммный метод доступа (возможность организации связи между отдельными прикладными программами комплекса, реализуемыми в различных узлах сети);
* доступ отдельных прикладных программ к ресурсам сети (и в первую очередь к устройствам ввода-вывода);
* синхронизацию работы прикладных программных средств в условиях их обращения к одному и тому же вычислительному ресурсу;
* обмен информацией между программами с использованием сетевых "почтовых ящиков";
* выполнение команд оператора с терминала, подключенного к одному из узлов сети, на каком-либо устройстве, подключенном к другому удаленному узлу вычислительной сети;
* удаленный ввод заданий, вводимых с любого терминала, и их выполнение на любой ЭВМ в пакетном или оперативном режиме;
* обмен наборами данных (файлами) между ЭВМ сети;
* доступ к файлам, хранимым в удаленных ЭВМ, и обработку этих файлов;
* защиту данных и вычислительных ресурсов сети от несанкционированного доступа;
* выдачу различного рода справок об использовании информационных, программных и технических ресурсов сети;
* передачу текстовых сообщений с одного терминала пользователя на другие (электронная почта).

Отдельным видом сетевого программного обеспечения является **драйвер**:

Драйвер (англ. driver) (множественное число драйверы) - это компьютерная программа, с помощью которой другая программа (обычно операционная система) получает доступ к аппаратному обеспечению некоторого устройства. В общем случае, для использования любого сетевого устройства (как внешнего, так и внутреннего) необходим драйвер.

# 28. Понятие адресации в сетях. Типы адресов. Адресация в IP-сетях

Для того, чтобы компьютеры могли идентифицировать друг друга в информационно-вычислительной сети, им присваиваются явные адреса. Основными типами адресов являются следующие:

* MAC-адрес (физический адрес);
* IP-адрес (сетевой адрес);
* доменный адрес (символьные имена, DNS-имена);

Для обмена данными в Интернете (между различными локальными сетями) узлу необходим IP-адрес. Это логический сетевой адрес конкретного узла. Для обмена данными с другими устройствами, подключенными к Интернету, необходим правильно настроенный, уникальный IP-адрес. IP-адрес присваивается сетевому интерфейсу узла.

IP-адрес – представляет собой основной тип адресов, на основании которых сетевой уровень передаёт пакеты между сетями. Эти адреса состоят из 4 байт. Назначаются администратором во время конфигурирования компьютеров и маршрутизатора. Он состоит из двух частей:

1. Номер сети – выбирается администратором произвольно или назначается службой InterNic;
2. Номер узла в сети – назначается независимо от локального адреса узла.

Маршрутизатор имеет столько адресов, сколько сетевых связей.

Классы IP-адресов:

Если адрес начинается с 0, то сеть относят к классу А и номер сети занимает 1 байт, номер узла 3 байта. Сети класса А имеют номера в диапазоне от 1 до 126. Таких сетей немного, зато количество узлов в них может достигать 2^24 .

Если первые два бита равны 10, то сеть относится к классу В. Является сетью средних размеров, максимальное количество узлов в которой равняется 2^16 .

Если адрес начинается последовательностью 110, то он относится к классу С, количество узлов в котором равняется 2^8 .

Если адрес начинается последовательностью 1110, то это сеть класса D. Она означает групповой адрес—Multicast. Если в пакете в качестве адреса назначения указан адрес класса D, то такой пакет получают все узлы, которым присвоен данный адрес.

Если адрес начинается с 11110, то эта сеть относится к классу Е. Адреса этого класса зарезервированы для будущего применения.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Класс** | **Первые биты** | **Наименьший номер сети** | **Наибольший номер сети** | **Максимальное число узлов в сети** |
| A | 0 | 1.0.0.0 | 126.0.0.0 | 224 |
| B | 10 | 128.0.0.0 | 191.255.0.0 | 216 |
| C | 110 | 192.0.1.0 | 223.255.255.0 | 28 |
| D | 1110 | 224.0.0.0 | 239.255.255.255 | Multicast |
| E | 11110 | 240.0.0.0 | 247.255.255.255 | Зарезервирован |

**ИСПОЛЬЗОВАНИЕ МАСОК В IP-АДРЕСАЦИИ**

Маска – число, которое используется в паре с IP адресом, двоичная запись маски содержит единицы в тех разделах, которые должны в IP адресе интерпретироваться как номер сети. Поскольку номер сети является цельной частью адреса, 1 в маске представляют непрерывную последовательность.

Для стандартных классов сетей маски имеют след. значения:

1. Класс А – 11111111.00000000.00000000.00000000 (255.0.0.0)
2. Класс В – 11111111.11111111.00000000.00000000 (255.255.0.0)

**ОПРЕДЕЛЕНИЕ IP-АДРЕСА**

Для определения локального адреса по IР адресу используется протокол разрешения (ARP). Существует также протокол, решающий обратную задачу – нахождение IP адреса по известному локальному адресу. Он называется реверсивный ARP (RARP).

Работа протокола ARP начинается с просмотра ARP-таблицы. Каждая строка таблицы устанавливает соответствие между IP адресом и локальным адресом. Для каждой сети, подключённой к сетевому адаптеру или к порту маршрутизатора, строится отдельная таблица.

Статические записи создаются вручную с помощью утилиты ARP и не имеет срока устаревания (существует до тех пор, пока компьютер или маршрутизатор не будут выключены). Динамические записи создаются модулем протокола ARP и периодически обновляются. Если в течение нескольких минут запись не обновляется, то она исключается из таблицы (ARP-кэш).

# 29. Характеристика протоколов IP, TCP, ARP, ICMP, POP3, SMTP

**Протокол IP (**протокол межсетевого взаимодействия**)**

IP компонента TCP/IP определяет, куда пакеты данных будут направлены, основываясь на адресе источника назначения, у IP есть определенные характеристики, связанные с тем, как работает эта функция.

Именно IP стал тем протоколом, который объединил отдельные компьютерные сети во всемирную сеть Интернет. Неотъемлемой частью протокола является *адресация* сети.

IP использует пакеты для передачи информации через сеть. Пакет - независимый контейнер, содержащий достаточное количество информации, для передачи его от источника до точки назначения без необходимости предварительной обработки.

IP характеризуется следующим:

* IP работает на третьем уровне модели OSI и стека TCP/IP.
* IP - протокол, работающий без установки соединения, в котором данные отправляются к точке назначения без дополнительного уведомления точки назначения. Принимающие устройство получает данные и не возвращает информацию о статусе полученного пакета, отправляющему устройству.
* IP использует иерархическую адресацию, в которой сетевую часть адреса можно представить как название улицы, а хостовую часть адреса, как номер дома на этой улице.
* IP предоставляет доставку пакетов без подтверждения и не гарантирует доставку пакета. Поэтому пакет может быть неправильно направлен, получен дважды или потерян на пути к месту назначения.
* IP не предоставляет специальных возможностей, которые позволили бы восстановить поврежденный пакет. Это обеспечивается оконечными система в сети.

В реальном масштабе времени голосовая или видео передача, без проблем переносит потерю нескольких пакетов; в данном случае наиболее важна скорость, чем гарантированное получение пакетов, потому что восстановление пакетов приведет к задержке по времени.

**Протокол TCP**

Transmission Control Protocol (TCP) (протокол управления передачей) - является обязательным протоколом стандарт TCP/IP, определенный в стандарте RFC 793, "Transmission Control Protocol (TCP)".

TCP — это протокол транспортного уровня, предоставляющий транспортировку (передачу) потока данных, с необходимостью предварительного установления соединения, благодаря чему гарантирует уверенность в целостности получаемых данных, также выполняет повторный запрос данных в случае потери данных или искажения. Помимо этого протокол TCP отслеживает дублирование пакетов и в случае обнаружения - уничтожает дублирующиеся пакеты.

Протокол гарантирует целостность передаваемых данных и подтверждения отправителя о результатах передачи. Используется при передаче файлов, где потеря одного пакета может привести к искажению всего файла.

TCP обеспечивает свою надежность благодаря следующему:

* Данные от приложения разбиваются на блоки определенного размера, которые будут отправлены.
* Когда TCP посылает сегмент, он устанавливает таймер, ожидая, что с удаленного конца придет подтверждение на этот сегмент. Если подтверждение не получено по истечении времени, сегмент передается повторно.
* Когда TCP принимает данные от удаленной стороны соединения, он отправляет подтверждение. Это подтверждение не отправляется немедленно, а обычно задерживается на доли секунды
* TCP осуществляет расчет контрольной суммы для своего заголовка и данных. Это контрольная сумма, рассчитываемая на концах соединения, целью которой является выявить любое изменение данных в процессе передачи. Если сегмент прибывает с неверной контрольной суммой, TCP отбрасывает его и подтверждение не генерируется. (Ожидается, что отправитель отработает тайм-аут и осуществит повторную передачу.)
* Так как TCP сегменты передаются в виде IP датаграмм, а IP датаграммы могут прибывать беспорядочно, также беспорядочно могут прибывать и TCP сегменты. После получения данных TCP может по необходимости изменить их последовательность, в результате приложение получает данные в правильном порядке.
* Так как IP датаграмма может быть продублирована, принимающий TCP должен отбрасывать продублированные данные.
* TCP осуществляет контроль потока данных. Каждая сторона TCP соединения имеет определенное пространство буфера. TCP на принимающей стороне позволяет удаленной стороне посылать данные только в том случае, если получатель может поместить их в буфер. Это предотвращает от переполнения буферов медленных хостов быстрыми хостами.
* Перед началом передачи каких-либо данных, согласно протоколу TCP, стороны должны установить соединение.

**Протокол ARP**

Для отображения IP-адресов в Ethernet адреса используется протокол ARP (Address Resolution Protocol - адресный протокол). Отображение выполняется только для отправляемых IP-пакетов, так как только в момент отправки создаются заголовки IP и Ethernet.

Преобразование адресов выполняется путем поиска в таблице. Эта таблица, называемая ARP-таблицей, хранится в памяти и содержит строки для каждого узла сети. В двух столбцах содержатся IP- и Ethernet-адреса. Если требуется преобразовать IP-адрес в Ethernet-адрес, то ищется запись с соответствующим IP-адресом.

ARP-таблица необходима потому, что IP-адреса и Ethernet-адреса выбираются независимо, и нет какого-либо алгоритма для преобразования одного в другой.

**Протокол ICMP**

Протокол ICMP - это протокол *сообщения об ошибках*, а не протокол *коррекции ошибок*. Конечный узел может предпринять некоторые действия для того, чтобы ошибка больше не возникала, но эти действия протоколом ICMP не регламентируются.

Каждое сообщение протокола ICMP передается по сети внутри пакета IP. Пакеты IP с сообщениями ICMP маршрутизируются точно так же, как и любые другие пакеты, без приоритетов, поэтому они также могут теряться. Кроме того, в загруженной сети они могут вызывать дополнительную загрузку маршрутизаторов. Для того, чтобы не вызывать лавины сообщения об ошибках, потери пакетов IP, переносящие сообщения ICMP об ошибках, не могут порождать новые сообщения ICMP. Сообщение ICMP всегда содержит заголовок и первые 64 бита данных пакета IP, который вызвал ошибку.

Сообщение, прибывшее в узел-источник, может быть обработано там либо ядром операционной системы, либо протоколами транспортного и прикладного уровней, либо приложениями, либо просто проигнорированы. Важно, что обработка ICMP-сообщений не входит в обязанности протоколов IP и ICMP.

Заметим, что некоторые из пакетов могут исчезнуть в сети, не вызвав при этом никаких оповещений. В частности, протокол ICMP не предусматривает передачу сообщений о проблемах, возникающих при обработке IP-пакетов, несущих ICMP-сообщения об ошибках. Такое решение было принято разработчиками протокола, чтобы не порождать «штормы» в сетях, когда количество сообщений об ошибках лавинообразно возрастает.

**Протокол POP3**

POP3 (англ. Post Office Protocol Version 3 — протокол почтового отделения, версия 3) — стандартный интернет-протокол прикладного уровня, используемый клиентами электронной почты для получения почты с удаленного сервера по TCP/IP-соединению.

POP и IMAP (Internet Message Access Protocol) — наиболее распространённые интернет-протоколы для извлечения почты. Практически все современные клиенты и серверы электронной почты поддерживают оба стандарта. Протокол POP был разработан в нескольких версиях, нынешним стандартом является третья версия (POP3). Большинство поставщиков услуг электронной почты (такие как Hotmail, Gmail и Yahoo! Mail) также поддерживают IMAP и POP3. Предыдущие версии протокола (POP, POP2) устарели.

POP3 поддерживает простые требования «загрузи-и-удали» для доступа к удаленным почтовым ящикам. Хотя большая часть POP-клиентов предоставляют возможность оставить почту на сервере после загрузки, использующие POP клиенты обычно соединяются, извлекают все письма, сохраняют их на пользовательском компьютере как новые сообщения, удаляют их с сервера, после чего разъединяются.

POP3-сервер прослушивает порт 110. Шифрование связи для POP3 запрашивается после запуска протокола. Доступные сообщения клиента фиксируются при открытии почтового ящика POP-сессией и определяются количеством сообщений для сессии, или, по желанию, с помощью уникального идентификатора, присваиваемого сообщению POP-сервером. Этот уникальный идентификатор является постоянным и уникальным для почтового ящика и позволяет клиенту получить доступ к одному и тому же сообщению в разных POP-сессиях. Почта извлекается и помечается для удаления с помощью номера сообщения. При выходе клиента из сессии помеченные сообщения удаляются из почтового ящика.

Команды РОРЗ состоят из ключевых слов, состоящих из ASCII-символов, и одним или несколькими параметрами, отделяемыми друг от друга символом "пробела" - <SP>. Все команды заканчиваются символами "возврата каретки" и "перевода строки" - <CRLF>. Длина ключевых слов не превышает четырех символов, а каждого из аргументов может быть до 40 символов.

Ответы РОРЗ-сервера на команды состоят из строки статус-индикатора, ключевого слова, строки дополнительной информации и символов завершения строки. Длина строки ответа может достигать 512 символов. Строка статус-индикатора принимает два значения: положительное ("+ОК") и отрицательное ("-ERR"). Любой сервер РОРЗ обязан отправлять строки статус-индикатора в верхнем регистре, тогда как другие команды и данные могут приниматься или отправляться как в нижнем, так и в верхнем регистрах.

Ответы РОРЗ-сервера на отдельные команды могут составлять несколько строк. В этом случае строки разделены символами <CRLF>. Последнюю строку информационной группы завершает строка, состоящая из символа "." (код — 046) и <CRLF>, т. е. последовательность "CRLF.CRLF"

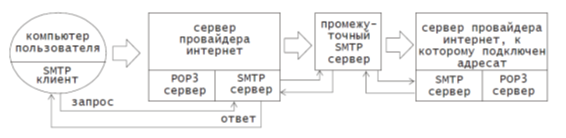
Команды протокола POP3:

* USER - идентифицирует пользователя с указанным именем;
* PASS - указывает пароль для пары клиент-сервер;
* QUIT - закрывает TCP-соединение;
* STAT - сервер возвращает количество сообщений в почтовом ящике плюс размер почтового ящика;
* LIST - сервер возвращает идентификаторы сообщений вместе с размерами сообщений;
* RETR - извлекает сообщение из почтового ящика;
* DELE - отмечает сообщение для удаления;
* NOOP - Сервер возвращает положительный ответ, но не совершает никаких действий;
* LAST - Сервер возвращает наибольший номер сообщения из тех, к которым ранее уже обращались;
* RSET - Отменяет удаление сообщения, отмеченного ранее командой DELE.

**Протокол SMTP**

**SMTP (Simple Mail Transfer Protocol)** – широко используемый сетевой протокол, предназначенный для передачи электронной почты в сетях TCP/IP.

Упрощенно схема взаимодействия представлена на рисунке ниже (объемными стрелками показано направление движения почтовых сообщений).



Со стороны пользователя обычно одна и та же программа выступает в роли и POP3 клиента, и SMTP клиента отправителя.

На схеме у пользователя есть клиентское ПО, а у провайдера – серверная часть приложения. На самом деле это немного не так. Протокол SMTP делает возможным смену сторон даже в ходе одного сеанса. Условно принято считать клиентом ту сторону, которая начинает взаимодействие и хочет отослать почту, а сервером ту, что принимает запросы. После того, как клиент посылает серверу несколько служебных команд и получает положительные ответы на них, он отправляет SMTP серверу собственно тело сообщения. SMTP сервер получает сообщение, вносит в него дополнительные заголовки, указывающие на то, что он обработал данное послание, устанавливает связь со следующим SMTP сервером по пути следования письма.

Общение между любыми SMTP серверами происходит по той же схеме. Инициирует переговоры клиент, сервер на них отвечает, а затем получает корреспонденцию и "ставит штампик" в теле письма (в его заголовочной части).

Как только почта достигнет конечного пункта (SMTP сервера адресата сообщения), она будет сложена в почтовый ящик абонента, который всегда сможет в удобное для него время изъять ее по протоколам POP3 или IMAP, в зависимости от того, какой из них поддерживается провайдером. В арсенал SMTP клиента, равно как и сервера, входит около 10 команд, но, воспользовавшись только пятью из них, уже можно легально послать почтовое сообщение. Это HELO, MAIL, RCPT, DATA, QUIT. Их использование подразумевается именно в такой последовательности. HELO предназначена для идентификации отправителя, MAIL указывает адрес отправителя, RCPT – адрес назначения. После команды DATA и ответа на нее, клиент посылает серверу тело сообщения, которое должно заканчиваться строкой, содержащей лишь одну точку.

# 30. Эталонная модель взаимодействия открытых систем. Понятие уровня и уровневых сетевых сервисов. Характеристика уровней

Эталонная модель взаимодействия открытых систем (БЭМВОС) – это концептуальная основа, определяющая характеристики и средства открытых систем. Она обеспечивает работу в одной сети систем, выпускаемых различными производителями. Разработана ISO (международной организацией стандартов) и широко используется во всём мире как основа концепций информационных сетей и их ассоциаций. На базе этой модели описываются правила и процедуры передачи данных между открытыми системами. Она также описывает структуру открытой системы и комплекс стандартов, которым она должна удовлетворять.

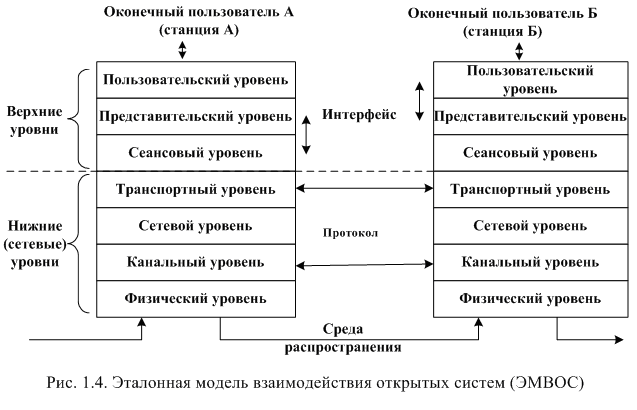
Основными элементами модели являются: уровни, объекты, соединения, физические средства соединений.

В соответствии с этой моделью процесс передачи сообщений в системах связи последовательно разбивается на принципиально различающиеся операции. Каждую ив этих операций относят к своему уровню.

Каждый уровень определяется сервисом, который он предоставляет вышестоящему уровню, и протоколом - набором правил и форматов данных для взаимодействия между собой объектов одного уровня, работающих на разных компьютерах.

Уровни строятся по принципу строгой иерархии: на высшем уровне находятся источник и получатель информации - пользователи системы связи, на нижнем - среда распространения электромагнитных волн. Высший уровень управляет работой низшего. Каждому уровню соответствует свое техническое устройство или организационная единица системы связи пользователь или должностное лицо, обеспечивающее функционирование системы связи. В некоторых системах связи часть этих устройств может отсутствовать либо выполнять не все функции некоторого уровня.

В ЭМВОС выделяют 7 уровней: пользовательский, представительский, сеансовый, транспортный, сетевой, канальный, физический (рис. 1.4). Полную совокупность средств у одного пользователя, выполняющих операции различных уровней, называют станцией.



Характеристика уровней:

* **Физический уровень** (Physical layer) определяет способ физического соединения компьютеров в сети. Функциями средств, относящихся к данному уровню, являются побитовое преобразование цифровых данных в сигналы, передаваемые по физической среде (например, по кабелю), а также собственно передача сигналов.
* **Канальный уровень**(Data Link layer) отвечает за организацию передачи данных между абонентами через физический уровень, поэтому на данном уровне предусмотрены средства адресации, позволяющие однозначно идентифицировать отправителя и получателя во всем множестве абонентов, подключенных к обще линии связи. В функции данного уровня также входит упорядочивание передачи с целью параллельного использования одной линии связи несколькими парами абонентов. Кроме того, средства канального уровня обеспечивают проверку ошибок, которые могут возникать при передаче данных физическим уровнем.
* **Сетевой уровень**(Network layer) обеспечивает доставку данных между компьютерами сети, представляющей собой объединение различных физических сетей. Данный уровень предполагает наличие средств логической адресации, позволяющих однозначно идентифицировать компьютер в объединенной сети. Одной из главных функций, выполняемых средствами данного уровня, является целенаправленная передача данных конкретному получателю.
* **Транспортный уровень**(Transport layer) реализует передачу данных между двумя программами, функционирующими на разных компьютерах, обеспечивая при этом отсутствие потерь и дублирования информации, которые могут возникать в результате ошибок передачи нижних уровней. В случае, если данные, передаваемые через транспортный уровень, подвергаются фрагментации, то средства данного уровня гарантируют сборку фрагментов в правильном порядке.
* **Сессионный (или сеансовый) уровень** (Session layer) позволяет двум программам поддерживать продолжительное взаимодействие по сети, называемое **сессией** (session) или **сеансом**. Этот уровень управляет установлением сеанса, обменом информацией и завершением сеанса. Он также отвечает за идентификацию, позволяя тем самым только определенным абонентам принимать участие в сеансе, и обеспечивает работу служб безопасности с целью упорядочивания доступа к информации сессии.
* **Уровень представления (или представительский)** (Presentation layer) осуществляет промежуточное преобразование данных исходящего сообщения в общий формат, который предусмотрен средствами нижних уровней, а также обратное преобразование входящих данных из общего формата в формат, понятный получающей программе.
* **Прикладной (или пользовательский) уровень** (Application layer) предоставляет высокоуровневые функции сетевого взаимодействия, такие, как передача файлов, отправка сообщений по электронной почте и т.п.

# 31. Этапы сбора и анализа требований к программной системе, требований заказчика и детальных требований

**Сбор требований**

На этапе сбора требований основная работа ведется с заказчиком системы и еѐ будущими пользователями. Цель этапа — точно определить функции продукта и способы его интеграции в существующие процессы.

Для того чтобы продукт был удобен пользователям и делал «то, что надо», сначала надо определить, кто же им будет пользоваться. Для этого определяются профили пользователей.

Когда определены профили пользователей продукта, следует найти людей, соответствующих этим профилям и желающих помочь вам в разработке продукта. Для проектов под заказ это достаточно простая задача — нужно выбрать одного или двух наиболее грамотных и инициативных людей для каждого профиля.

Далее, необходимо начать общение с конечными пользователями и узнать для достижения каких целей будет использоваться будущий продукт. Лучшим методом для достижения этой цели является сбор пользовательских историй. Пользовательская история — это вариант использования будущего продукта в конкретной ситуации с целью достижения измеримого результата. Каждая пользовательская история должна приносить пользу — не должно быть историй, которые выполняют действия ради действий. Пользовательские истории могут содержать как сложные инструкции с ответвлениями, так и конкретные примеры.

Каждая пользовательская история имеет одно или больше родительских бизнес требований и ее главная цель описать наиболее удачные способы их удовлетворения. Именно на основе родительского требования устанавливается приоритет истории.

Нужно очень четко контролировать поступающую от заказчика информацию. Требования должны быть продиктованы бизнесом и быть нацелены на достижение реальных и измеряемых результатов. Одним словом реализация требования должна приносить пользу, а не являться самоцелью. Если пользователь говорит, что неплохо было бы сделать «мастер того-то и того-то» нужно мгновенно среагировать и задать вопрос — «зачем этот мастер ему нужен». Как правило, он с радостью расскажет о реальных мотивах, которые и интересуют.

Качественное выполнение работ на этом этапе гарантирует то, что будущий продукт будет соответствовать ожиданиям заказчика. Четкая расстановка приоритетов обеспечивает реализацию наиболее востребованной функциональности и исключение второстепенной/невостребованной функциональности, что сэкономит бюджет и сроки.

В процессе этапа сбора требований формируются первичные требования (или требования заказчика).

**Анализ требований**

На этапе анализа требований проходит структуризация уже собранных ранее требований. Цель этапа — предоставить четкий список недублируемых требований к системе, которые должны быть выделены из избыточных и частично дублирующихся сценариев и пользовательских историй, которые были полученных на предыдущем этапе. Правильно сгруппированные требования помогут обойтись минимальным количеством функционала для удовлетворения максимально большего количества целей, а это, в свою очередь, поможет сэкономить бюджет и не даст расползтись рамкам проекта.

По окончании этапа анализа требований, многостраничный документ, содержащий сотни пользовательских историй, будет разбит на части. Каждая часть будет освещать только необходимую функциональность, а в еѐ основе будет стоять диаграмма вариантов использования, на основе которой можно будет легко увидеть все требуемые функции системы. Описание вариантов использования не будут дублироваться, а лишь дополнять друг друга.

Работу над этапом анализа можно считать законченной, когда вы выделили все пользовательские истории в отдельные пакеты. Каждый пакет имеет понятное название, а описание содержит критерии, на основе которых в него попали или не попали пользовательские истории.

**Существует два основных метода проектирования** — проектирование на основе вариантов использования и проектирование на основе требований.

Проектирование на базе вариантов использования считается более эффективным, так как этот метод позволяет не терять связь с пользовательскими историями и прекрасно иллюстрирует требуемое поведение системы в целом, а, следовательно, гарантирует востребованность всего функционала, который будет создан (очень расточительно и болезненно для разработчиков писать код, которым никогда не удастся воспользоваться). Но все же есть условия, при которых аналитик может пренебречь этим методом в пользу проектирования на базе требований:

1. Проектирование на основе требований следует предпочесть, если необходимо сократить затраты на стадию анализа до минимума, а количество пользовательских историй в пакете не велико (их содержимое можно просмотреть не дольше чем за 10 минут)
2. Команда разработки не умеет или не хочет работать с вариантами использования и требует предоставление требований к системе в классическом виде.

**Проектирование на базе вариантов использования:**

Как правило, у всех пользовательских историй в пакете есть одна или несколько главных целей, и есть история, которая описывает наиболее простой способ их достижения. Такая история называется — базовой, а описание еѐ действия — базовый путь. Главным критерием определения базового варианта использования является наличие общих со всеми остальными вариантами использования действий. Остальные истории описывают альтернативный способ достижения результата или содержат дополнительные действия для достижения специфического результата и по большому счету являются дополнениями к базовой истории.

Пользовательские истории это разновидность стандартных вариантов использования, с той лишь особенностью, что они описывают взаимодействие не с реальной, а с гипотетической системой.

Таким образом, в этом методе выделяются базовые пользовательские истории, а так же альтернативные способы достижение результатов.

**Проектирование на основе требований:** Для нормальной разработки нужен список требований, который однозначно идентифицирует потребности пользователей и не имеет множественных повторений, которые присутствуют с избытком в пользовательских историях (если не произвести структурирование историй, описанное в предыдущем подходе). Кроме этого, для более гибкого проектирования необходим как можно более детализированный (раздробленный) список.

Благодаря дроблению списка, приобретается возможность запланировать реализацию наиболее значимых требований на начало разработки, а низкоприоритетных - на конец. Как результат, в случае отставания от графика выполнения в продукте уже будут реализованы наиболее приоритетные требования, и вы будете иметь возможность пожертвовать низкоприоритетными требованиями с целью сокращения отставания.

В процессе анализа требований формируются детальные требования.

# 32. Требования к программному обеспечению. Типы требований. Уровни описания требований

**Требования к ПО**

Требования к ПО – это возможности или условия, которым должна соответствовать система или проект, совокупность утверждений относительно атрибутов, свойств или качеств программной системы, подлежащей реализации.

**Классификация требований ПО**

**1) Функциональные.** Это требования, которые описывают поведение системы и ее функции. Если функциональные требования оформлены как пользовательские, они, как правило описывают систему в обобщенном виде. В противоположность этому функциональные требования, оформленные как системные описывают систему максимально подробно включая ее функции, исключения и т.д.

Спецификация функциональных требований должна быть комплексной и непротиворечивой, но на практике этого добиться сложно. Причем зачастую выявить несогласованность на начальных этапах очень трудно.

Это перечень сервисов, которые должна выполнять система, причем должно быть указано, как система реагирует на те или иные входные данные, как она ведет себя в определенных ситуациях и т.д. В некоторых случаях указывается, что система не должна делать.

**2) Нефункциональные.** Описывают характеристики системы и ее окружения, а не поведение системы. В них обычно приводится перечень ограничений накладываемых на действия, функции выполняемых в системе (загруженность процессора и т.д.)

Многие нефункциональные требования относятся к системе в целом, а не к отдельным ее функциям. Это означает, что они более значимы и критичны, чем отдельные функциональные требования. Ошибка допущенная в функциональных требованиях может снизить качество системы, ошибка допущенная в нефункциональных может привести к неработоспособности системы.

Нефункциональные требования можно разбить на следующие группы:

1. Требования к продукту
2. Организационные требования
3. Внешние требования

Здесь также может быть приведен перечень ограничений, накладываемых на действия и функции, выполняемые системой. Они включают временные ограничения, ограничения на процесс разработки системы, стандарты и тд.

**3)** **Требования предметной области**

Характеризуют ту предметную область, где будет эксплуатироваться система, эти требования могут быть функциональными и нефункциональными.

**Уровни описания требований**

1. **Пользовательские требования.** Должны описывать функциональные и нефункциональные требования так, чтобы они были понятны даже пользователю не имеющему специальных технических знаний. Пользовательские требования - описание на естественном языке (плюс поясняющие диаграммы) функций, выполняемых системой, и ограничений, накладываемых на неё.
2. **Системные требования** - детализированное описание системных функций и ограничений, которое иногда называют функциональной спецификацией. Она служит основой для заключения контракта между покупателем системы и разработчиками ПО.
3. **Проектная системная спецификация** - обобщённое описание структуры программной системы, которое будет основой для детализованного проектирования системы и её последующей реализации. Эта спецификация дополняет и детализирует спецификацию системных требований.

Пользовательские требования пишутся для заказчика ПО и для лица, заключающего контракт на разработку программной системы, причём они могут не иметь детальных технических знаний по разрабатываемой системе. Спецификация системных требований предназначена для руководящего технического состава компании-разработчика и для менеджеров проекта. Она также необходима заказчику ПО и субподрядчикам по разработке. Эти оба документа также предназначены для конечных пользователей программной системы. Наконец, проектная системная спецификация является документом, который ориентирован на разработчиков ПО.

# 33. Критерии качества программного обеспечения

**Ка́чество програ́ммного обеспечения** — способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям.

Основные критерии:

* функциональная пригодность;
* уровень производительности;
* совместимость;
* удобство пользования;
* надёжность;
* защищённость;
* сопровождаемость;
* переносимость (мобильность).

Критерии при использовании:

* результативность;
* производительность;
* удовлетворенность;
* свобода от риска;

покрытие контекста.

# 34. Модели жизненного цикла программного обеспечения

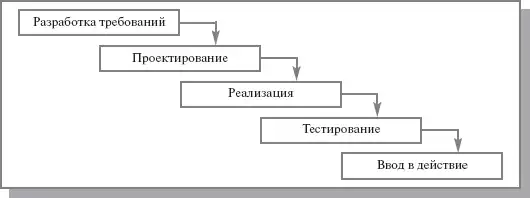
**Жизненный цикл ПО** — период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации.

**Модель** жизненного цикла ПО — структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении жизненного цикла. Модель жизненного цикла зависит от специфики, масштаба и сложности проекта и специфики условий, в которых система создается и функционирует.

Основные модели:

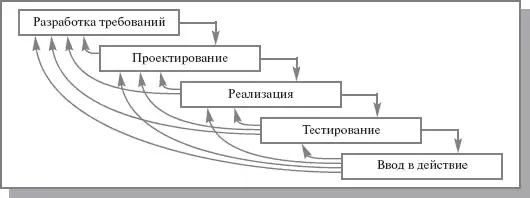
* каскадная,
* поэтапная с промеж. контролем,
* спиральная.

**Каскадная** предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Для **ПРОСТЫХ** систем.

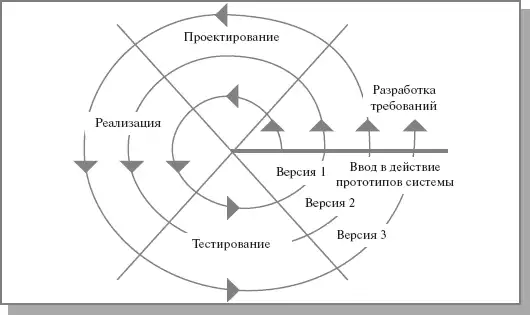


**Поэтапная модель с промежуточным контролем**

Разработка ведется итерациями с циклами обратной связи между этапами. **Межэтапные** корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах; время жизни каждого из этапов растягивается на весь период разработки.



**Спиральная модель.** На каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка.Особое внимание уделяется начальным этапам разработки - анализу и проектированию, где реализуемость тех или иных технических решений проверяется и обосновывается посредством создания прототипов (макетирования).



# 

# 35. Технологическая документация. Пользовательская документация программных средств. Документация по сопровождению программных средств

**Документа́ция** на программное обеспечение — документы, описывающие, как пользоваться программным продуктом.

**Техническая** документация — текст, описывающий различные аспекты, что именно делает код. Такая документация часто включается непосредственно в исходный код или предоставляется вместе с ним.

Подобная документация имеет сильно выраженный технический характер и в основном используется для определения и описания API, структур данных и алгоритмов.  
  
Часто при составлении технической документации используются автоматизированные средства — генераторы документации. Они получают информацию из специальным образом оформленных комментариев в исходном коде, и создают справочные руководства в каком-либо формате, например, в виде текста или HTML.

В отличие от технической документации, **пользовательская** документация описывает лишь то, *как* использовать программу. Пользовательская документация представляет собой руководство пользователя, которое описывает каждую функцию программы, а также *шаги*, которые нужно выполнить для использования этой функции. Что делать в случае возникновения проблем.

Документация по **сопровождению** ПО описывает ПО с точки зрения ее **разработки**. Сопровождение — это продолжающаяся разработка. Эта документация необходима, если ПО предполагает изучение того, как оно устроена (сконструирована), и модернизацию его компонентов.

Документация по сопровождению ПС можно разбить на две группы:

* документация, определяющая строение программ и структур данных ПС и технологию их разработки;
* документацию, помогающую вносить изменения в ПС.

# 

# 36. Принципы S.O.L.I.D. Примеры использования

Принципы ООП предназначены для повышения помощи в разработке системы, которую легко поддерживать и удобно расширять в течение долгого времени. Принципы SOLID - это руководства, которые могут применяться во время работы над программным обеспечением для удаления «кода с запашком».

* **S Принцип единственной обязанности (Single responsibility principle)**

Класс должен иметь только одну ответственность (то есть повлиять на спецификацию класса должно быть способно только одно потенциальное изменение в спецификации ПО).

Пример: есть класс, который работает с изображениями. Причём он умеет и печатать их, и менять размеры. Если захотим в другой части приложение использовать только часть функционала, то придется использовать весь компонент. При соблюдении принципы данный компонент был бы разбит на несколько (работа с файл. сист, преобразования и тд).

* **O Принцип открытости/закрытости (Open/closed principle)**

Программные сущности должны быть открыты для расширения, но закрыты для модификации.

Пример: компонент отправляет оповещения пользователю на e-mail. Чтобы изменить или добавить ещё один способ оповещения, придется изменять компонент. При соблюдении принципа этот компонент должен уметь работать с любым абстрактным способом оповещения.

* **L Принцип подстановки Барбары Лисков (Liskov substitution principle)**

Объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.

Пример с высчитыванием площади у квадрата и прямоугольника

* **I Принцип разделения интерфейса (Interface segregation principle)**

Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.

Пример: интерфейс класса фигуры содержит слишком много методов и свойств, которые нужны реализовать. (метод получения площади, метод рисования), но при реализации не всегда нужно, например, рисовать фигуру, поэтому лучше сделать несколько интерфейсов.

* **D Принцип инверсии зависимостей (Dependency inversion principle)**

Зависимость на Абстракциях. Нет зависимости на что-то конкретное. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракции.  
Пример: сервис зависит от класса конкретной СУБД. Вместо этого он должен зависеть от интерфейса, от которой наследуется класс конкретной СУБД.

# 

# 37. Паттерны проектирования. Группы паттернов. Примеры использования

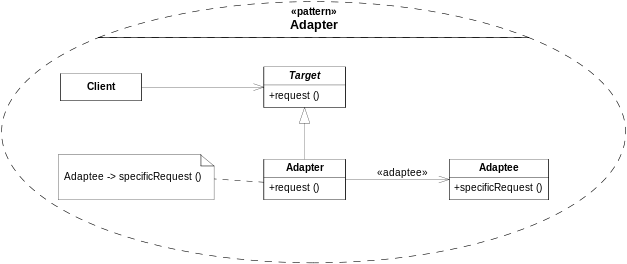
Шаблон проектирования или паттерн (англ. design pattern) в разработке программного обеспечения — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.  
  
Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

\*Привожу примеры реализации ниже, названия можно заменить чем-то конкретным из вашей предметной области\*.

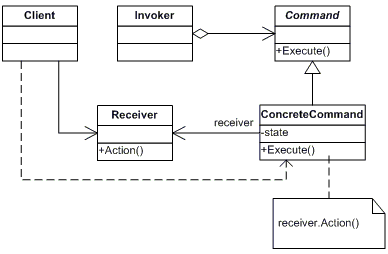
|  |  |  |
| --- | --- | --- |
| Название | Оригинальное название | Описание |
| **Основные шаблоны** (Fundamental) | | |
| [Шаблон делегирования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%B4%D0%B5%D0%BB%D0%B5%D0%B3%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) | Delegation pattern | Объект внешне выражает некоторое поведение, но в реальности передаёт ответственность за выполнение этого поведения связанному объекту. |
| [Шаблон функционального дизайна](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%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%B4%D0%B8%D0%B7%D0%B0%D0%B9%D0%BD%D0%B0) | Functional design | Гарантирует, что каждый модуль компьютерной программы имеет только одну обязанность и исполняет её с минимумом побочных эффектов на другие части программы. |
| [**Порождающие шаблоны**](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D0%BE%D0%B6%D0%B4%D0%B0%D1%8E%D1%89%D0%B8%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) **(**[**Creational**](https://en.wikipedia.org/wiki/Creational_pattern)**)** — шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту. | | |
| [Абстрактная фабрика](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%B0%D1%8F_%D1%84%D0%B0%D0%B1%D1%80%D0%B8%D0%BA%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Abstract factory | Класс, который представляет собой интерфейс для создания компонентов системы. |
| [Строитель](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Builder | Класс, который представляет собой интерфейс для создания сложного объекта. |
| [Фабричный метод](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B1%D1%80%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Factory method | Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. |
| [Одиночка](https://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%B8%D0%BD%D0%BE%D1%87%D0%BA%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Singleton | Класс, который может иметь только один экземпляр. |
| [**Структурные шаблоны**](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) **(Structural)** определяют различные сложные структуры, которые изменяют [интерфейс](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу. | | |
| [Адаптер](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0%D0%BF%D1%82%D0%B5%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Adapter / Wrapper | Объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс. |
| [Мост](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D1%81%D1%82_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Bridge | Структура, позволяющая изменять интерфейс обращения и интерфейс реализации класса независимо. |
| [Компоновщик](https://ru.wikipedia.org/wiki/Composite) | Composite | Объект, который объединяет в себе объекты, подобные ему самому. |
| [Декоратор](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) или Wrapper/Обёртка | Decorator | Класс, расширяющий функциональность другого класса без использования наследования. |
| [Фасад](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D1%81%D0%B0%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Facade | Объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое. |
| [Заместитель](https://ru.wikipedia.org/wiki/Proxy_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Proxy | Объект, который является посредником между двумя другими объектами, и который реализует/ограничивает доступ к объекту, к которому обращаются через него. |
| [**Поведенческие шаблоны**](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) **(Behavioral)** определяют взаимодействие между объектами, увеличивая таким образом его гибкость. | | |
| [Цепочка обязанностей](https://ru.wikipedia.org/wiki/Chain_of_Responsibility) | Chain of responsibility | Предназначен для организации в системе уровней ответственности. |
| [Команда](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), Action, Transaction | Command | Представляет действие. Объект команды заключает в себе само действие и его параметры. |
| [Итератор](https://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), [Cursor](https://ru.wikipedia.org/wiki/Cursor) | Iterator | Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящих в состав агрегации. |

Пример одиночки:

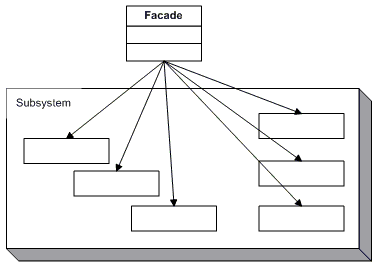
public class Singleton  
{  
 private static readonly Singleton instance = new Singleton();  
  
 public static Singleton Instance  
 {  
 get { return instance; }  
 }  
 /// защищённый конструктор нужен, чтобы предотвратить создание экземпляра класса Singleton  
 protected Singleton() { }

Схема адаптера:  


Команда:



Фасад:

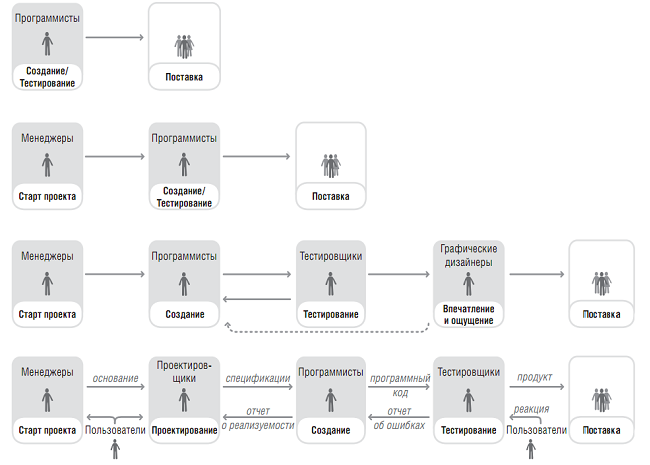


# 38. Эволюция программного обеспечения. Работа с унаследованными системами, возвратное проектирование (reverse engineering)

В ранние дни развития индустрии программного обеспечения процесс разработки сводился к тому, что программисты вынашивали идею продукта, а затем создавали и самостоятельно тестировали его.

В более поздние времена к процессу стали подключаться профессиональные управленцы, их задачи сводились к оценке потребностей рынка и формулированию основных требований к разрабатываемому программному обеспечению.

С развитием индустрии в самостоятельную дисциплину выделилось тестирование, а также широкое распространение получили графические интерфейсы пользователя, появилась необходимость разработки различных визуальных элементов, в связи с чем к процессу разработки ПО подключились графические дизайнеры.



Обра́тная разрабо́тка (обратный инжиниринг, реверс-инжиниринг; англ. reverse engineering) — исследование некоторого готового устройства или программы, а также документации на него с целью понять принцип его работы; например, чтобы обнаружить недокументированные возможности (в том числе программные закладки), сделать изменение или воспроизвести устройство, программу или иной объект с аналогичными функциями, но без прямого копирования.

Применяется обычно в том случае, если создатель оригинального объекта не предоставил информации о структуре и способе создания (производства) объекта. Правообладатели таких объектов могут заявить, что проведение обратной разработки или использование её результатов нарушает их исключительное право по закону об авторском праве и патентному законодательству.

# 39. Принципы объектно-ориентированного конструирования программных систем.

При разработке ОО систем используется принципы:

* абстрагирование;
* ограничение доступа;
* модульность;
* иерархичность.

**Абстрагирование** — процесс выделения абстракций в предметной области задачи. Абстракция — совокупность существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

**Ограничение доступа** — сокрытие отдельных элементов реализации абстракции, не затрагивающих существенных характеристик ее как целого. Необходимость ограничения доступа предполагает разграничение двух частей в описании абстракции:

* интерфейс — совокупность доступных извне элементов реализации абстракции (основные характеристики состояния и поведения);
* реализация — совокупность недоступных извне элементов реализации абстракции (внутренняя организация абстракции и механизмы реализации ее поведения).

Ограничение доступа в ООП позволяет разработчику:

* выполнять конструирование системы поэтапно, не отвлекаясь на особенности реализации используемых абстракций;
* легко модифицировать реализацию отдельных объектов, что в правильно организованной системе не потребует изменения других объектов.

**Модульность** — принцип разработки программной системы, предполагающий реализацию ее в виде отдельных частей (модулей). При выполнении декомпозиции системы на модули желательно объединять логически связанные части, по возможности обеспечивая сокращение количества внешних связей между модулями. Принцип унаследован от модульного программирования, следование ему упрощает проектирование и отладку программы.

**Иерархия** — ранжированная или упорядоченная система абстракций. Принцип иерархичности предполагает использование иерархий при разработке программных систем.

# 

# 40. Рефакторинг программных систем

**Рефакторинг** (реорганизация) кода — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований.

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

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

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

# 

# 41. Миграция программных систем

Миграция ПС — это перенос ПС, который может выполняться аппаратно (смена серверов), так и программно (смена технологий, операционных систем) с сохранением всей работоспособности системы и данных.

Самый распространённый вид миграции — миграция данных. Миграция данных выполняется в рабочей системе, когда требуется произвести изменение представления хранения данных.

Например, в процессе разработки в новой версии системы у сущности «Сотрудник» появился новый атрибут — место жительства. В этом случае вместе с изменением версии приложения должно быть произведеная миграция данных на на новую схему хранения. При использования реляционной СУБД требуется добавить в таблице “Сотрудник” новый столбец “место жительства” (ALTER TABLE). Набор таких скриптов, привязанных к конкретной версии приложения, и есть миграция. Также можно реализовать скрипт для миграции данных в случае понижения версии приложения (удаление столбца).

# 42. Концепция тестирования программного обеспечения. Основные проблемы тестирования

Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий две различные цели:

1. продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям;

2. выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.

Основная проблема тестирования — определение достаточности множества тестов для истинности вывода о правильности реализации программы, а также нахождения множества тестов, обладающего этим свойством.

* Тестирование программы на всех входных значениях невозможно.
* Невозможно тестирование и на всех путях алгоритма.
* Следовательно, надо отбирать конечный набор тестов, позволяющий проверить программу на основе наших интуитивных представлений
* Требования к тестам — программа на любом из них должна останавливаться, т.е. не зацикливаться.
* Тесты могут содержать ошибки

# 

# 43. Оценка покрытия тестами программы и проекта. Методика интегральной оценки тестированности.

**Оценка покрытия тестами программы и проекта.**

Тестирование программы *P* по некоторому критерию *C* означает покрытие множества компонентов программы *P*, представляющих из себя множество модулей по элементам или связям.

При этом — кортеж неизбыточных тестов. Тест не избыточен, если существует покрытый им компонент из пар программы и критерия: M(P, C) непокрытый ни одним из предыдущих тестов t\_1, ..., t\_(j-1) каждому t\_j соответсвует неизбыточный путь p\_i, при этом p\_i - последовательность вершин от выхода или входа программ. Здесь j = 1..n

V(P, C) — сложность тестирования P по критерию C. Измеряется максимальным числом неизбыточных тестов, покрывающих все элементы множества M(P, C)

DV(P, C) - остаточная сложность тестирования P по критерию C. Измеряется максимальным числом неизбыточных тестов, покрывающих элементы множества M(P, C), оставшиеся непокрытыми после прогона набора тестов T. Величина DV строго и монотонно убывает от V до 0.

TV(P, C, T) = (V - DV) / V - оценка степени тестрованния P по критерию C.

Критерий окончания тестирования: TV(P, C, T) ≥ L; (0 ≤ L ≤ 1) задан в требованиях к программному продукту.

**Методика интегральной оценки тестированности**

Включает в себя 8 шагов:

1. Выбор критерия C и приемочные оценки тестрованности программного проекта L
2. Построение дерева классов проекта и построение графа управления для каждого модуля
3. Модульное тестирование и оценка степени тестрованности TV на модульном уровне
4. Построение графо-управления, интегрирующего модули в единую иерарахическую модель проекта.
5. Выбор тестовых путей для проведения инетграционного или системного тестирования.
6. Генерация тестов, покрывающих тестовые пути на 5 шаге.
7. Интегральная оценка тестированности проекта с учетом оценки тестированности модулей компонент.
8. Повторение шага 5-7 до достижения заданного уровня тестированности L.

# 44. Модульное и интеграционное тестирование.

**Модульное тестирование**

**Модульное тестирование** — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

**Цель модульного тестирования** — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Модульное тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно (регрессионное тестирование). Это поощряет программистов к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений.

Модульное тестирование помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх»: сначала тестируя отдельные части программы, а затем программу в целом.

Модульные тесты можно рассматривать как «живой документ» для тестируемого класса. Клиенты, которые не знают, как использовать данный класс, могут использовать юнит-тест в качестве примера.

Поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним. Например, класс пользуется базой данных; в ходе написания теста программист обнаруживает, что тесту приходится взаимодействовать с базой. Это ошибка, поскольку тест не должен выходить за границу класса. В результате разработчик абстрагируется от соединения с базой данных и реализует этот интерфейс, используя свой собственный mock-объект. Это приводит к менее связанному коду, минимизируя зависимости в системе.

**Интеграционное тестирование**

**Интеграционное тестирование** — одна из фаз тестирования программного обеспечения, при которой отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию.

Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования.

**Целью интеграционного тестирования** является проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надежности. Тестирование этих проектируемых единиц — объединения, множества или группы модулей — выполняется через их интерфейс, с использованием тестирования «чёрного ящика».

Для автоматизации интеграционного тестирования применяются ***системы непрерывной интеграции***. Принцип действия таких систем состоит в следующем:

1. CIS производит мониторинг системы контроля версий;
2. При изменении исходных кодов в репозитории производится обновление локального хранилища;
3. Выполняются необходимые проверки и модульные тесты;
4. Исходные коды компилируются в готовые выполняемые модули;
5. Выполняются тесты интеграционного уровня;
6. Генерируется отчет о тестировании.

Таким образом, автоматические интеграционные тесты выполняются сразу же после внесения изменений, что позволяет обнаруживать и устранять ошибки в короткие сроки.

# 45. Статическое и динамическое тестирование

Статическое тестирование производится без запуска программного кода продукта. Тестирование осуществляется путем анализа программного кода (code review) или скомпилированного кода. Анализ может производиться как вручную, так и с помощью специальных инструментальных средств. Целью анализа является раннее выявление ошибок и потенциальных проблем в продукте.

С помощью code review на раннем этапе могут быть выявлены ошибки в коде продукта. Как правило code review производится самими разработчиками.

Примерами ошибок, которые потенциально можно выявить с помощью автоматического статического тестирования, могут быть:

* утечки ресурсов (утечки памяти, неосвобождаемые файловые дескрипторы и т.д.)
* возможность переполнения буфера (buffer overflows)
* ситуации частичной (неполной) обработки ошибок

Как правило, результатом автоматического анализа кода является список рекомендаций для ручного review некоторых участков кода, потенциально содержащих ошибки.

В отличии от статического, динамическое тестирование производится путем запуска продукта и проверки его функционала. Проверка осуществляется с помощью ручного или автоматического выполнения заранее подготовленного набора тестов.

Примеры динамического тестирования:

* модульное тестирование (unit testing)
* интеграционное тестирование (integrated testing)
* приемочное тестирование (acceptance testing)

# 46. Методы поиска и устранения ошибок

Для локализации и установки точной природы ошибки в программе используют статические и динамические методы отладки программ.

К статическим методам относятся методы отладки, при которых не требуется выполнение отлаживаемой программы на ЭВМ. Они обычно требуют больших усилий от программиста и незначительных затрат машинного времени. Они универсальны и пригодны для отладки программ, написанных на любом языке программирования и используемых на любой ЭВМ.

Статические методы включают:

* ручную прокрутку программы;
* прокрутку программы программными анализаторами ( нап­ример, компилятором); автоматизированный анализ программы в этом случае проводится без выполнения ее на ЭВМ и поэтому попадает в категорию «статических»;
* коллективную проверку программ;
* проверку программы программистом-технологом с целью выявления и исправления в ней технологических ошибок.

Экспериментально установлено, что в программах ручными методами удается обнаруживать от 30 до 70 % программных и алгоритмических ошибок из общего числа ошибок, выявленных при отладке. При этом одновременно осуществляется доработка программ с целью улучшения их структуры, логики обработки данных и для снижения сложности последующего автомати­зированного тестирования на ЭВМ.

Динамические методы связаны со значительным расходом машинного времени и, возможно, не меньшими затратами труда программиста. В этом случае отладка программ происходит сов­местно с их выполнением на ЭВМ. Динамические методы отладки программ, как правило, привязаны к конкретной ЭВМ и к конкретному транслятору (компилятору).

К динамическим методам относятся:

* тестирование;
* поиск ошибок с использованием системных средств;
* отладка программы в интерактивном режиме.

Важнейшее правило отладки: не делать следующего выхода на ЭВМ, пока не будет разобрана каждая найденная ошибка. Из этого правила существует единственное исключение: если най­дены 5—6 ошибок, которые не дают эффекта, то можно сделать новый выход на машину (устранив эти ошибки), чтобы получить эффект в чистом виде (если он есть),поскольку наложение нес­кольких ошибок иногда может дать самый неожиданный ре­зультат.

Если программист исчерпал все возможности поиска ошибки, но не нашел ее, то как крайнее средство, можно сделать выход на машину, ничего не изменив в программе, но добавив печати, выдающие значения идентификаторов, участвующих в фор­мировании неверного значения. И снова произвести анализ полученных результатов. Квалификация программиста в области отладки определяется тем, сколько информации об ошибках он сможет получить из одной выдачи ЭВМ.