

Министерство науки и высшего образования Российской Федерации
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Марийский государственный университет»

МЕТОДИЧЕСКОЕ ПОСОБИЕ
ДЛЯ ПОДГОТОВКИ К ГОСУДАРСТВЕННЫМ
МЕЖДИСЦИПЛИНАРНЫМ ЭКЗАМЕНАМ

Направление подготовки 02.04.03 Математическое обеспечение и
администрирование информационных систем

Магистерская программа: Информационные системы и технологии

Настоящая программа предназначена для подготовки к вступительному
испытанию в магистратуру по направлению 02.04.03 Математическое
обеспечение и администрирование информационных систем (магистерская
программа Информационные системы и технологии).

Йошкар-Ола
2022 г.

Оглавление

1. Задачи поиска: исчерпывающий поиск, быстрый поиск, использование деревьев в задачах поиска.....	3
2. Уровни моделей и этапы проектирования баз данных. Инфологическое моделирование.....	7
3. Принципы построения и архитектура компьютерных сетей Протоколы, иерархия протоколов и режимы их работы	13
4. Назначение и основные функции операционных систем (ОС).....	20
5. Языки и системы программирования. Модели языков программирования	29
6. Модели и этапы разработки программного обеспечения	36
7. Реляционные СУБД. Объектно-ориентированные базы данных.....	41
8. Архитектуры вычислительных систем. Архитектура системы команд.....	51
9. Задачи сортировки Анализ сложности и эффективности алгоритмов сортировки	57
10. Современные технологии разработки программного обеспечения Управление версиями Документирование.....	62

1. Задачи поиска: исчерпывающий поиск, быстрый поиск, использование деревьев в задачах поиска

Задача данного действия заключается в нахождении одного или нескольких элементов в множестве, причем искомые элементы должны обладать определенным свойством.

Таким образом, в задаче поиска имеются следующие **шаги**:

- 1) вычисление свойства элемента;
- 2) сравнение свойства элемента с эталонным свойством (для абсолютных свойств) или сравнение свойств двух элементов (для относительных свойств);
- 3) перебор элементов множества.

Исчерпывающий поиск – это процесс нахождения в некотором множестве всех возможных вариантов, среди которых имеется решение конкретной задачи.

Перебор с возвратом (backtracking) – это один из методов организации исчерпывающего поиска, построение решения по одному компоненту и выяснение, может ли дальнейшее построение привести к решению.

Незначительные модификации метода перебора с возвратом, связанные с представлением данных или особенностями реализации, имеют и иные названия: метод ветвей и границ, поиск в глубину, метод проб и ошибок и т. д.

В основе **метода ветвей и границ** лежит следующая идея (для задачи минимизации): если нижняя граница для подобласти А дерева поиска больше, чем верхняя граница какой-либо ранее просмотренной подобласти В, то А может быть исключена из дальнейшего рассмотрения (правило отсева).

*Под **быстрым поиском** подразумевается: бинарный и последовательный поиски в массивах, хеширование.*

Под поиском в массиве будем понимать задачу нахождения индекса, по которому в массиве располагается некоторый заданный элемент. Если заранее известна некоторая информация о данных, среди которых ведется поиск, например, известно, что массив данных отсортирован, то удастся сократить время поиска, используя **бинарный поиск**.

Последовательный поиск – тривиальный алгоритм поиска, заключается в последовательном переборе элементов массива до тех пор, пока не будет обнаружен искомый или не будут просмотрены все элементы массива.

Поиск по хэшу – отказаться от поиска по данным и выполнить арифметические действия над K , позволяющие вычислить некоторую функцию $f(K)$. Последняя укажет адрес в таблице, где хранится K и связанная с ним информация. Недостаток – нужно заранее знать содержимое таблицы.

Использование деревьев в задачах поиска: бинарные и случайные бинарные, оптимальные и сбалансированные деревья поиска

При использовании в целях поиска элементов данных по значению уникального ключа применяются двоичные (бинарные) деревья поиска.

Дерево — одна из наиболее широко распространённых структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов. Является связанным графом, не содержащим циклы. Большинство источников также добавляют условие на то, что рёбра графа не должны быть ориентированными. В дополнение к этим трём ограничениям, в некоторых источниках указываются, что рёбра графа не должны быть взвешенными.

Дерево поиска – все вершины в правом поддереве $> x$ (вершина), все вершины в левом поддереве $< x$. Дерево поиска считается идеально **сбалансированным**, если число вершин в его левом и правом поддеревьях отличается не более, чем на 1.

Для поиска заданного ключа в дереве поиска достаточно пройти по одному пути от корня до (возможно, листовой) вершины.

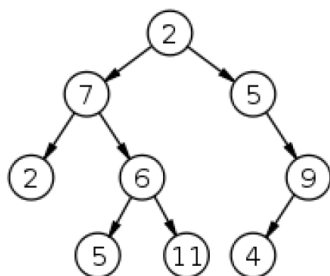
Представление деревьев

Существует множество различных способов представления деревьев. Наиболее общий способ представления изображает узлы как записи, расположенные в динамически выделяемой памяти с указателями на своих потомков, предков (или и тех и других), или как элементы массива, связанные между собой отношениями, определёнными их позициями в массиве (например, двоичная куча).

Методы обхода

Пошаговый перебор элементов дерева по связям между предками-узлами и потомками-узлами называется обходом дерева, а сам процесс называется обходом по дереву. Зачастую, операция может быть выполнена переходом указателя по отдельным узлам. Обход, при котором каждый узел-предок просматривается прежде его потомков называется предупорядоченным обходом или обходом в прямом порядке (pre-order walk), а когда просматриваются сначала потомки, а потом предки, то обход называется поступорядоченным обходом или обходом в обратном порядке (post-order walk). Существует также симметричный обход, при котором посещается сначала левое

поддереву, затем узел, затем — правое поддерево, и обход в ширину, при котором узлы посещаются уровень за уровнем (N-й уровень дерева — множество узлов с высотой N). Каждый уровень обходится слева направо.



Применение

- 1) управление иерархией данных;
- 2) упрощение поиска информации (см. обход дерева);
- 3) управление сортированными списками данных;
- 4) синтаксический разбор арифметических выражений (англ. parsing), оптимизация программ;
- 5) в качестве технологии компоновки цифровых картинок для получения различных визуальных эффектов;
- 6) форма принятия многоэтапного решения.

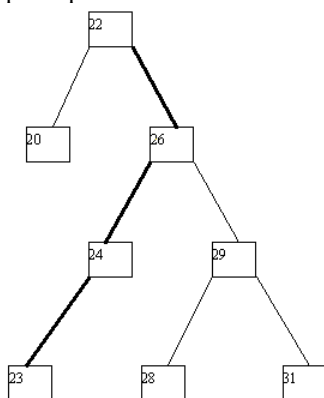


Рис.10.7. Путь поиска ключа по значению “23”

Случайные деревья поиска представляют собой упорядоченные бинарные деревья поиска, при создании которых элементы (их ключи) вставляются в случайном порядке.

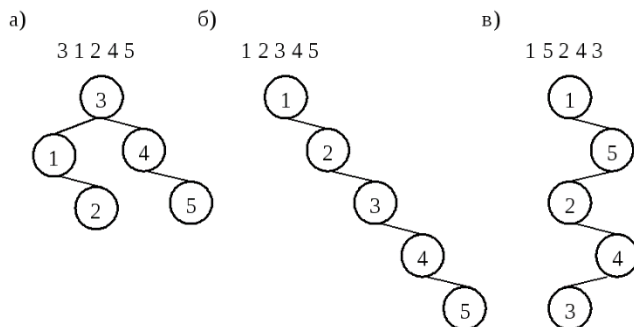


Рисунок 3.15 – Случайные и вырожденные деревья поиска

При поступлении элементов в случайном порядке получаем дерево с минимальной высотой h (см. рис. 3.15.а), а соответственно минимизируется время поиска элемента в таком дереве, которое пропорционально $O(\log n)$. При поступлении элементов в упорядоченном виде (см. рис. 3.12.б) или в несколько необычном порядке (см. рис. 3.12.в) происходит построение вырожденных деревьев поиска (оно вырождено в линейный список), что несколько не сокращает время поиска, которое составляет $O(n)$.

Оптимальное бинарное дерево поиска – это бинарное дерево поиска, построенное в расчете на обеспечение максимальной производительности при заданном распределении вероятностей поиска требуемых данных.

Процедура построения дерева оптимального поиска достаточно сложна и опирается на тот факт, что любое поддерево дерева оптимального поиска также обладает свойством оптимальности. Поэтому известный алгоритм строит дерево "снизу-вверх", т.е. от листьев к корню. Сложность этого алгоритма и расходы по памяти составляют $O(n^2)$. Имеется эвристический алгоритм, дающий дерево, близкое к оптимальному, со сложностью $O(n \cdot \log n)$ и расходами памяти - $O(n)$.

Таким образом, создание оптимальных деревьев поиска требует больших накладных затрат, что не всегда оправдывает выигрыш при быстром поиске.

По определению, двоичное дерево называется **сбалансированным (или АВЛ) деревом** в том и только в том случае, когда высоты двух поддеревьев каждой из вершин дерева отличаются не более, чем на единицу.

2. Уровни моделей и этапы проектирования баз данных. Инфологическое моделирование

В процессе научных исследований, посвященных тому, как именно должна быть устроена СУБД, предлагались различные способы реализации. Самым жизнеспособным из них оказалась предложенная американским комитетом по стандартизации ANSI (AmericanNationalStandardsInstitute) трехуровневая система организации БД, изображенная на рис. 2.1:

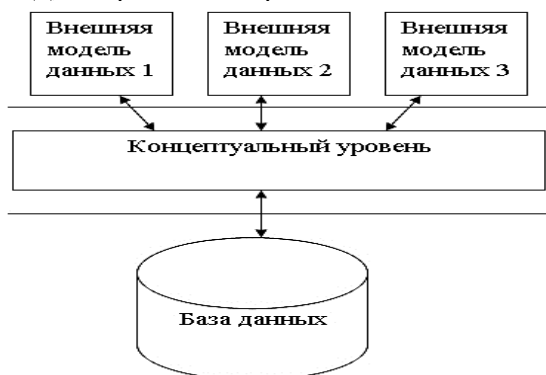


Рис. 2.1.Трехуровневая модель системы управления базой данных, предложенная ANSI

1. **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. Например, система распределения работ использует сведения о квалификации сотрудника, но ее не интересуют сведения об окладе, домашнем адресе и телефоне сотрудника, и наоборот, именно эти сведения используются в подсистеме отдела кадров.

2. **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.

3. **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.

Выделение концептуального уровня позволило разработать аппарат централизованного управления базой данных.



Модели данных по отношению к каждому уровню.

Физическая модель данных оперирует категориями, касающимися организации внешней памяти и структур хранения, используемых в данной операционной среде.

По отношению к моделям концептуального уровня внешние модели называются подсхемами и используют те же абстрактные категории, что и концептуальные модели данных.

Инфологическая или семантическая модель выражает информацию о предметной области в виде, независимом от используемой СУБД. Эти модели отражают в естественной и удобной

для разработчиков форме описание объектов предметной области, их свойства и их взаимосвязи.

Даталогическая модель данных создается на основе выбранной модели организации данных целевой СУБД. Созданная логическая модель данных является источником информации для этапа физического проектирования и обеспечивает разработчика физической базы данных средствами поиска компромиссов, необходимых для достижения поставленных целей.

Этапы проектирования баз данных

Реализацию сложных проектов по созданию информационных систем (ИС) принято разбивать на стадии анализа, проектирования, кодирования, тестирования и сопровождения.

Процесс разработки БД можно разбить на несколько этапов:

- Исследование предметной области;
- Инфологическое моделирование;
- Даталогическое проектирование;
- Даталогическое моделирование;
- Проектирование на физическом уровне.

Инфологическое моделирование заключается в создании концептуальной модели данных с использованием стандартных языковых средств, обычно графических, например ER-диаграмм (диаграмм «Сущность-связь»).

Семантическое моделирование данных. ER-диаграммы.

Моделирование структуры базы данных при помощи алгоритма нормализации имеет серьезные недостатки:

- Первоначальное размещение всех атрибутов в одном отношении является очень неестественной операцией. Интуитивно разработчик сразу проектирует несколько отношений в соответствии с обнаруженными сущностями. Даже если совершить насилие над собой и создать одно или несколько отношений, включив в них все предполагаемые атрибуты, то совершенно неясен смысл полученного отношения.

- Невозможно сразу определить полный список атрибутов. Пользователи имеют привычку называть разными именами одни и те же вещи или наоборот, называть одними именами разные вещи.

- Для проведения процедуры нормализации необходимо выделить зависимости атрибутов, что тоже очень нелегко, т.к. необходимо явно выписать все зависимости, даже те, которые являются очевидными.

В реальном проектировании структуры базы данных применяются другой метод - так называемое, семантическое моделирование. Семантическое моделирование представляет собой моделирование структуры данных, опираясь на смысл этих данных. В качестве инструмента семантического моделирования используются различные варианты диаграмм сущность-связь (ER - Entity-Relationship).

Определение 1. Сущность - это класс однотипных объектов, информация о которых должна быть учтена в модели.

Каждая сущность должна иметь наименование, выраженное существительным в единственном числе. Примерами сущностей могут быть такие классы объектов как "Поставщик", "Сотрудник", "Накладная".

Каждая сущность в модели изображается в виде прямоугольника с наименованием:

Сотрудник

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 именованная характеристика, являющаяся некоторым свойством 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:

Сотрудник
Табельный номер
Фамилия
Имя
Отчество
Должность
Зарплата

Определение 4. Ключ сущности - это избыточный набор атрибутов, значения которых в совокупности являются уникальными для каждого экземпляра сущности. Избыточность заключается в том, что удаление любого атрибута из ключа нарушается его уникальность.

Сущность может иметь несколько различных ключей.

Ключевые атрибуты изображаются на диаграмме подчеркиванием:

Сотрудник
<u>Табельный номер</u>
<u>Фамилия</u>
<u>Имя</u>
<u>Отчество</u>
<u>Должность</u>
<u>Зарплата</u>

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 ней.

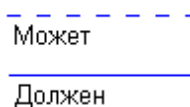
Например, связи между сущностями могут выражаться следующими фразами - "СОТРУДНИК может иметь несколько ДЕТЕЙ", "каждый СОТРУДНИК обязан числиться ровно в одном ОТДЕЛЕ".

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:



Модальность "может" означает, что экземпляр одной сущности может быть связан с одним или несколькими экземплярами другой сущности, а может быть и не связан ни с одним экземпляром.

Модальность "должен" означает, что экземпляр одной сущности обязан быть связан не менее чем с одним экземпляром другой сущности.

Связь может иметь разную модальность с разных концов (как на Рис. 4).

Описанный графический синтаксис позволяет однозначно читать диаграммы, пользуясь следующей схемой построения фраз:

<Каждый экземпляр СУЩНОСТИ 1> <МОДАЛЬНОСТЬ СВЯЗИ> <НАИМЕНОВАНИЕ СВЯЗИ> <ТИП СВЯЗИ> <экземпляр СУЩНОСТИ 2>.

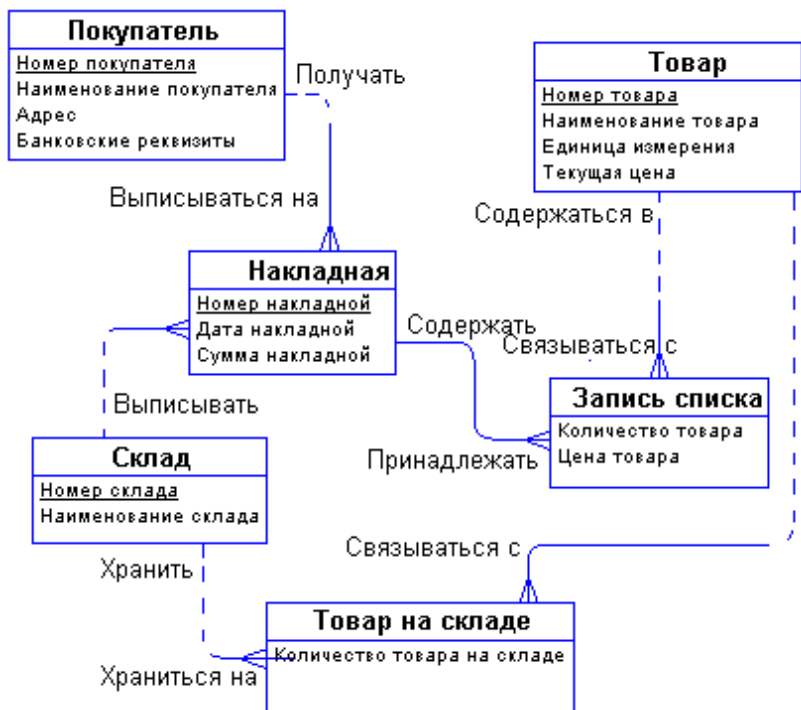
Каждая связь может быть прочитана как слева направо, так и справа налево. Связь на Рис. 4 читается так:

Слева направо: "каждый сотрудник может иметь несколько детей".

Справа налево: "Каждый ребенок обязан принадлежать ровно одному сотруднику".

Различают концептуальные и физические ER-диаграммы. Концептуальные диаграммы не учитывают особенностей конкретных СУБД. Физические диаграммы строятся по концептуальным и представляют собой прообраз конкретной базы данных. Сущности, определенные в концептуальной диаграмме становятся таблицами, атрибуты становятся колонками таблиц (при этом учитываются допустимые для данной СУБД типы данных и наименования столбцов), связи реализуются путем миграции ключевых атрибутов родительских сущностей и создания внешних ключей.

При правильном определении сущностей, полученные таблицы будут сразу находиться в ЗНФ. Основное достоинство метода состоит в том, модель строится методом последовательных уточнений первоначальных диаграмм. Пример концептуальной ER-диаграммы:



3. Принципы построения и архитектура компьютерных сетей Протоколы, иерархия протоколов и режимы их работы

Все многообразие компьютерных сетей можно классифицировать по следующим четырём признакам:

1. по типу среды передачи, то есть физической среды, которая используется для соединения компьютеров;
2. по скорости передачи информации;
3. по ведомственной принадлежности;
4. по территориальной распространенности.

1) Среда передачи называется еще "линией связи". Информация передается по линиям связи в виде различных сигналов, которые, испытывая сопротивление среды, затухают с расстоянием. Поэтому одной из важнейших характеристик линии связи является максимальная дальность, на которую может быть передана по ней информация без искажения.

В качестве линий связи могут использоваться:

- ИК-лучи (обеспечивают передачу информации между компьютерами, находящимися в пределах одной комнаты);
- электрические провода (кабель "витая пара" обеспечивает связь между компьютерами на расстояние до 100м, коаксиальные кабели – до 500м);
- оптоволоконные кабели (обеспечивают связь на расстояние нескольких десятков километров);
- телефонные линии, радиосвязь, спутниковая связь (позволяют соединять компьютеры, находящиеся в любой точке планеты).

2) По скорости передачи информации компьютерные сети делятся на низкоскоростные (скорость передачи информации до 10 Мбит/с), среднескоростные (скорость передачи информации до 100 Мбит/с), высокоскоростные (скорость передачи информации свыше 100 Мбит/с).

3) По принадлежности различают ведомственные и государственные сети. Ведомственные сети принадлежат одной организации и располагаются на ее территории. Государственные сети – это сети, используемые в государственных структурах.

4) По территориальной распространенности сети могут быть локальными, глобальными и региональными. Локальными называются сети, расположенные в одном или нескольких зданиях. Региональными называются сети, расположенные на территории города или области. Глобальными называются сети, расположенные на территории государства или группы государств, например, всемирная сеть Интернет.

В классификации сетей существует два основных термина: локальная сеть (LAN) и территориально-распределенная сеть (WAN).

Локальная сеть (LocalAreaNetwork) связывает компьютеры и принтеры, обычно находящиеся в одном здании (или комплексе зданий). Каждый компьютер, подключенный к локальной сети, называется рабочей станцией или сетевым узлом. Как правило, в локальных сетях практикуется использование высокоскоростных каналов.

Локальные вычислительные сети подразделяются на: одноранговые сети и иерархические (многоуровневые).

Одноранговая сеть представляет собой сеть равноправных компьютеров, каждый из которых имеет уникальное имя (имя компьютера).

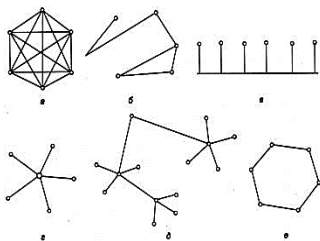
В иерархических локальных сетях имеется один или несколько специальных компьютеров – серверов, на которых хранится информация, совместно используемая различными пользователями.

Сервер в иерархических сетях – это постоянное хранилище разделяемых ресурсов. Сам сервер может быть клиентом только сервера более высокого уровня иерархии. Поэтому иерархические сети иногда называются сетями с выделенным сервером. Серверы обычно представляют собой высокопроизводительные компьютеры, возможно, с несколькими параллельно работающими процессорами. Компьютеры, с которых осуществляется доступ к информации на сервере, называются клиентами.

Территориально-распределенная сеть WAN (WideAreaNetwork) соединяет несколько локальных сетей, географически удаленных друг от друга. Территориально-распределенные сети обеспечивают те же преимущества, что и локальные, но при этом позволяют охватить большую территорию.

Архитектура сети – это набор параметров, правил, протоколов, алгоритмов, карт, которые позволяют изучать сеть, определяет принципы функционирования и интерфейс пользователя.

Организация обмена данными в сфере компьютерных и информационных технологий осуществляется согласно выбранной топологии. Конфигурация которой определяется соединением нескольких компьютеров и может отличаться от конфигурации логической связи.



Топология способ организации физических связей.

Под топологией ВС понимается конфигурация графа, вершинам (узлам) которого соответствуют компьютеры сети (иногда и другое оборудование, например концентраторы), а ребрам – физические связи между ними.

а) Полносвязная; б) Ячеистая; в) Общая шина; г) Звезда; д) Иерархическая звезда; е) Кольцевая конфигурация.

Для крупных сетей характерна смешанная топология.

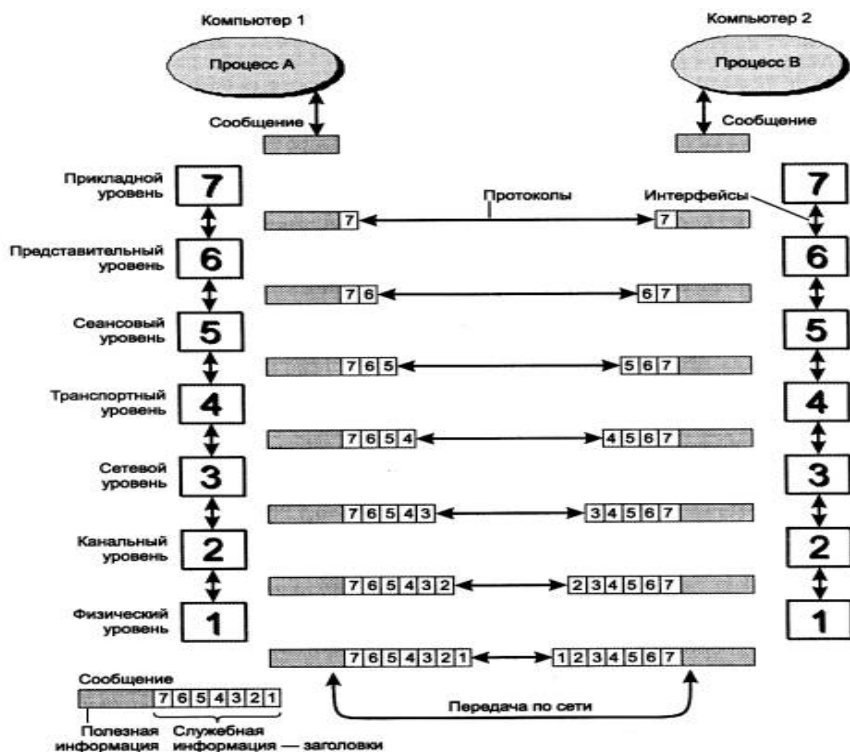
Процессы, происходящие в локальной вычислительной машине, не назовешь простыми. Представьте, что нашей задачей является необходимость обратиться к файлу, размещенному на магнитном диске и, в котором, находятся требуемые нам данные. Запрос вначале анализируется соответствующими программами состав символического имени файла и определяется его уникальный идентификатор. Затем по

уникальному имени определяются адрес его нахождения, права доступа к файлу и далее после расчета координат области файла, содержащей требуемые данные, выполняется физическая передача его внешнему устройству вывода - терминалу. Этот пример показывает, сколько много действий необходимо выполнить системе, чтобы выполнить сравнительно простую операцию. А если система (ЭВМ) работает в сети? Насколько усложняются процессы обработки информации, и возрастает их количество.

Известно, что для решения сложных задач используется универсальный прием – декомпозиция, то есть разбиение одной сложной задачи на несколько более простых задач-модулей. Каждому модулю определяется его основная функция и правила их взаимодействия для достижения общей конечной цели – получения результата решения.

При декомпозиции часто используется многоуровневый подход. Суть его заключается в том, что все множество модулей разбивается на уровни. Уровни образуют иерархию, то есть нижележащие уровни являются подчиненными по отношению к верхним. Модули одного уровня для выполнения своих задач обращаются с запросами только к модулям непосредственно примыкающего нижележащего уровня. Результаты же работы модулей некоторого уровня могут быть переданы только модулям соседнего вышележащего уровня. Правила взаимодействия уровней по вертикали называли интерфейсами. Интерфейс определяет набор функций, которые нижележащий уровень предоставляет вышележащему. Важным в такой структуре является возможность модификации отдельных модулей без изменения остальной части системы.

В начале 80-х годов рядом международных организаций по стандартизации при участии фирм разработчиков и изготовителей вычислительной техники была предложена модель, названная моделью взаимодействия открытых систем (Open System Interconnection, OSI) или моделью OSI. Указанная модель показана на рисунке 7.2.



Уровни двух компьютеров соединены стрелкой с надписью «Протокол». Это означает, что взаимодействие этих уровней выполняется в соответствии с некими формализованными правилами, называемыми протоколами, и определяющими последовательность и формат сообщений, которыми обмениваются компьютеры.

Под сетевым протоколом обычно понимают совокупность правил взаимодействия двух элементов сети при обмене информацией между ними.

Протокол определяет набор правил, которые описывают формат и назначение пакетов, передаваемых между одноранговыми сущностями внутри уровня. По сути, протокол определяет услуги уровня, на котором он работает. Протокол может претерпеть изменения, но предоставляемые услуги не должны меняться

Протокол управления физическим уровнем определяет форму представления, и порядок передачи данных через физический канал. Процесс управления сводится к формированию начала и конца кадра,

несущего в себе передаваемые данные, передаче и приему сигналов, определенной физической природы со скоростью, соответствующей пропускной способности канала.

Процессы, работающие на канальном уровне, принято делить на два подуровня. Один подуровень работает по управлению доступом к каналу, второй подуровень управляет непосредственно передачей данных – управление информационным каналом. Протокол управления доступом к каналам определяет процедуры передачи данных через канал, являющийся разделяемой средой между всеми пользователями локальной сети. Протокол управления информацией, устанавливает порядок обеспечения достоверности данных при передаче по каналу: формирует проверочные коды при передаче данных, проверяет эти коды в месте их приема и при обнаружении ошибки обеспечивает повторение передачи. Примерами протоколов канального уровня в современных сетях и ЭВМ можно назвать протоколы Ethernet, Token Ring, FDDI, 100VG-AnyLAN.

Протоколы сетевого уровня реализуются в виде программных модулей и выполняются на конечных узлах – компьютерах, называемых хостами, а также на промежуточных узлах – маршрутизаторах, называемых шлюзами.

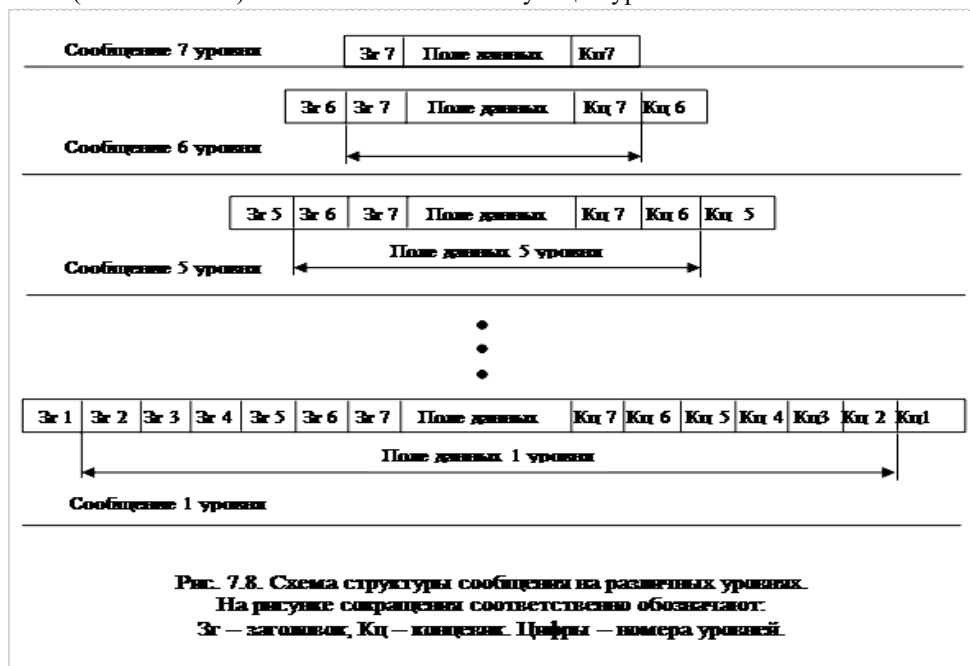
Протоколы физического, канального, сетевого и транспортного уровней обобщенно называют *сетевым транспортом* или *транспортной подсистемой*, так как они полностью обеспечивают решение задачи доставки сообщений в составных сетях с произвольными топологиями и различными технологиями.

Функции уровней (протоколы), начиная с транспортного, реализуются программными компонентами операционных систем компьютеров пользователей. Самыми известными транспортными протоколами в настоящее время являются протоколы TCP и UDP стека TCP/IP и протокол SPX стека Novell.

Многоуровневая организация управления процессами в компьютере и в сети порождает необходимость модифицировать на каждом уровне передаваемые сообщения, применительно к функциям, реализуемым на этом уровне. В этом случае сообщение преобразуется по схеме, представленной на рисунке 7.8. Как видно из многоуровневой модели компьютера, сообщение пользователя продвигается по уровням сверху вниз или снизу вверх. Необходимо твердо себе уяснить, что продвижение это виртуальное. Сообщение все время остается в буфере оперативной памяти и последовательно шаг за шагом обрабатывается соответствующими программами операционной системы до тех пор, пока «не доберется» до

физического или прикладного уровня. Тогда оно либо превращается в реальную последовательность битов и направляется в линию связи сети, либо в виде файла предстает перед глазами пользователя, который и выполняет запланированные заранее операции его обработки.

Виртуально перемещаясь с одного уровня на другой (вверх или вниз), сообщение уже реально в результате его обработки соответствующими программами операционной системы обрамляется (или лишается) заголовками соответствующих уровней.



Эту процедуру обрамления сравнивают с вложением в конверт обработанное сообщение на соответствующем уровне. На первом уровне в конверте оказываются еще шесть конвертов.

Пришедшее в компьютер сообщение в процессе его обработки перемещается с первого уровня на седьмой, последовательно освобождаясь «от конвертов». Таким образом, каждый уровень управления оперирует не с самими сообщениями, а только с «конвертами», в которых «упакованы» сообщения.

На нижнем, физическом уровне в заголовок включают специальные коды, например, 01111110, исполняющие роль

синхронизирующих сигналов и одновременно разделителей кадров. Такие же сочетания могут встретиться и в данных, которые могут быть восприняты обрабатывающими программами или аппаратурой как часть заголовка. Для исключения этой ошибки в данных, при встрече такой последовательности, после пятой единицы вставляется 0 (ноль), который называли – бит-стаффингом. При приеме данных выполняется обратное преобразование, бит-стаффинг заменяется 1, в результате чего данные принимают прежний вид. Этот прием называли обеспечением прозрачности физического канала по отношению к передаваемым данным.

4. Назначение и основные функции операционных систем (ОС)

Операционная система (ОС) - это комплекс программного обеспечения, предназначенный для снижения стоимости программирования, упрощения доступа к системе, повышения эффективности работы.

Цель создания операционной системы - получить экономический выигрыш при использовании системы, путем увеличения производительности труда программистов и эффективности работы оборудования.

Функции операционной системы:

- связь с пользователем в реальном времени для подготовки устройств к работе, переопределение конфигурации и изменения состояния системы.

- выполнение операций ввода-вывода; в частности, в состав операционной системы входят программы обработки прерываний от устройств ввода-вывода, обработки запросов к устройствам ввода-вывода и распределения этих запросов между устройствами.

- управление памятью, связанное с распределением оперативной памяти между прикладными программами.

- управление файлами; основными задачами при этом являются обеспечение защиты, управление выборкой и сохранение секретности хранимой информации.

- обработка исключительных условий во время выполнения задачи

- появление арифметической или машинной ошибки, прерываний, связанных с неправильной адресацией или выполнением привилегированных команд.

- вспомогательные, обеспечивающие организацию сетей, использование служебных программ и языков высокого уровня.

Дополнительные функции ОС:

Данные функции реализованы не непосредственно для удобства пользователя, а для обеспечения выполнения операций системы. Это следующие возможности.

Распределение ресурсов между пользователями, программами и процессами, работающими одновременно.

Ведение статистики использования ресурсов, с целью выставления пользователям счетов (например, за сетевой трафик) или для анализа эффективности работы системы.

Защита – обеспечение того, чтобы *доступ* к любым ресурсам был контролируемым.

Подходы к построению операционных систем: Монолитное ядро; Многоуровневые системы; Виртуальные машины; Микро ядерная архитектура; Смешанные системы; Системы реального времени.

Монолитное ядро

По сути дела, операционная система – это обычная программа, поэтому было бы логично и организовать ее так же, как устроено большинство программ, то есть составить из процедур и функций. В этом случае компоненты операционной системы являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура операционной системы называется монолитным ядром (monolithickernel). Монолитное ядро представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом, монолитное ядро – это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной операционной системы ядро совпадает со всей системой.

Во многих операционных системах с монолитным ядром сборка ядра, то есть его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается операционная система. При этом можно выбрать список оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция – это единственный способ добавить в него новые компоненты или исключить неиспользуемые.

Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонентов повышает надежность операционной системы в целом.

Монолитное ядро – старейший способ организации операционных систем. Примером систем с монолитным ядром является большинство Unix-систем.

Даже в монолитных системах можно выделить некоторую структуру. Как в бетонной глыбе можно различить вкрапления щебенки, так и в монолитном ядре выделяются вкрапления сервисных процедур, соответствующих системным вызовам. Сервисные процедуры выполняются в привилегированном режиме, тогда как пользовательские программы – в непривилегированном. Для перехода с одного уровня привилегий на другой иногда может использоваться главная сервисная программа, определяющая, какой именно системный вызов был сделан, корректность входных данных для этого вызова и передающая управление соответствующей сервисной процедуре с переходом в привилегированный режим работы. Иногда выделяют также набор программных утилит, которые помогают выполнять сервисные процедуры.

Многоуровневые системы (Layered systems)

Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты уровня N-1. Нижним уровнем в таких системах обычно является hardware, верхним уровнем – интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы THE (Technische Hogeschool Eindhoven) Дейкстры (Dijkstra) и его студентами в 1968 г. Эта система имела следующие уровни:

5	Интерфейс пользователя
4	Управление вводом выводом
3	Драйвер устройства связи оператора и консоли
2	Управление печатью
1	Планирование задач и процессов
0	Hardware

Рис. 1.2. Слоеная система THE

Слоеные системы хорошо реализуются. При использовании операций нижнего слоя не нужно знать, как они реализованы, нужно лишь понимать, что они делают. Слоеные системы хорошо тестируются. Отладка начинается с нижнего слоя и проводится послойно. При возникновении ошибки мы можем быть уверены, что

она находится в тестируемом слое. Слоеные системы хорошо модифицируются. При необходимости можно заменить лишь один слой, не трогая остальные. Но слоеные системы сложны для разработки: тяжело правильно определить порядок слоев и что к какому слою относится. Слоеные системы менее эффективны, чем монолитные. Так, например, для выполнения операций ввода-вывода программе пользователя придется последовательно проходить все слои от верхнего до нижнего.

Виртуальные машины

Пусть операционная система реализует виртуальную машину для каждого пользователя, но не упрощая ему жизнь, а, наоборот, усложняя. Каждая такая виртуальная машина предстает перед пользователем как голое железо – копия всего hardware в вычислительной системе, включая процессор, привилегированные и непривилегированные команды, устройства ввода-вывода, прерывания и т.д. И он остается с этим железом один на один. При попытке обратиться к такому виртуальному железу на уровне привилегированных команд в действительности происходит системный вызов реальной операционной системы, которая и производит все необходимые действия. Такой подход позволяет каждому пользователю загрузить свою операционную систему на виртуальную машину и делать с ней все, что душа пожелает.

Первой реальной системой такого рода была система CP/CMS, или VM/370, как ее называют сейчас, для семейства машин IBM/370.

Недостатком таких операционных систем является снижение эффективности виртуальных машин по сравнению с реальным компьютером, и, как правило, они очень громоздки. Преимущество же заключается в использовании на одной вычислительной системе программ, написанных для разных операционных систем.

Микроядерная архитектура

Современная тенденция в разработке операционных систем состоит в перенесении значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Речь идет о подходе к построению ядра, называемом микроядерной архитектурой (microkernelarchitecture) операционной системы, когда большинство ее составляющих являются самостоятельными программами. В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микро-ядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную

обработку прерываний, операции ввода-вывода и базовое управление памятью.

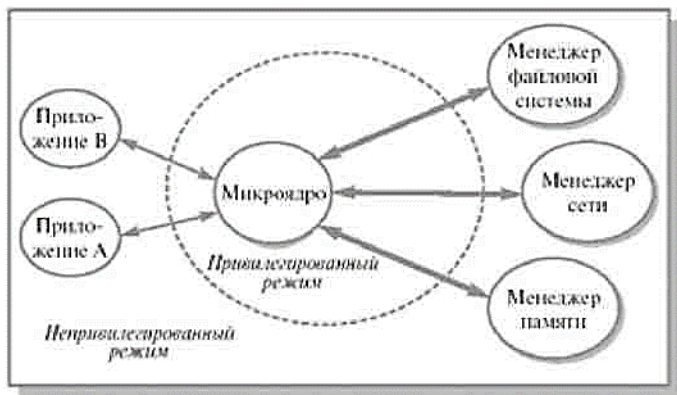


Рис. 1.4. Микроядерная архитектура операционной системы

Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

Основное достоинство микроядерной архитектуры – высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д. Существенно упрощается процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

В то же время микроядерная архитектура операционной системы вносит дополнительные накладные расходы, связанные с передачей сообщений, что существенно влияет на производительность. Для того чтобы микроядерная операционная система по скорости не уступала операционным системам на базе монолитного ядра, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных операционных систем – необходимость очень аккуратного проектирования.

Смешанные системы

Все рассмотренные подходы к построению операционных систем имеют свои достоинства и недостатки. В большинстве случаев современные операционные системы используют различные комбинации этих подходов. Так, например, ядро операционной системы Linux представляет собой монолитную систему с элементами микроядерной архитектуры. При компиляции ядра можно разрешить динамическую загрузку и выгрузку очень многих компонентов ядра – так называемых модулей. В момент загрузки модуля его код загружается на уровне системы и связывается с остальной частью ядра. Внутри модуля могут использоваться любые экспортируемые ядром функции.

Другим примером смешанного подхода может служить возможность запуска операционной системы с монолитным ядром под управлением микроядра. Так устроены 4.4BSD и MkLinux, основанные на микроядре Mach. Микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов. Все остальные функции, в том числе взаимодействие с прикладными программами, осуществляется монолитным ядром. Данный подход сформировался в результате попыток использовать преимущества микроядерной архитектуры, сохраняя по возможности хорошо отлаженный код монолитного ядра.

Наиболее тесно элементы микроядерной архитектуры и элементы монолитного ядра переплетены в ядре WindowsNT. Хотя WindowsNT часто называют микроядерной операционной системой, это не совсем так. Микроядро NT слишком велико (более 1 Мбайт), чтобы носить приставку "микро". Компоненты ядра WindowsNT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как и положено в микроядерных операционных системах. В то же время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром. По мнению специалистов Microsoft, причина проста: чисто микроядерный дизайн коммерчески невыгоден, поскольку неэффективен.

Таким образом, WindowsNT можно с полным правом назвать гибридной операционной системой. Многопроцессорные ОС разделяют на симметричные и асимметричные. В симметричных ОС на каждом процессоре функционирует одно и то же ядро, и задача может быть выполнена на любом процессоре, то есть обработка полностью децентрализована. При этом каждому из процессоров доступна вся память.

В асимметричных ОС процессоры неравноправны. Обычно существует главный процессор (master) и подчиненные (slave), загрузку и характер работы которых определяет главный процессор.

Системы реального времени

В разряд многозадачных ОС, наряду с пакетными системами и системами разделения времени, включаются также системы реального времени.

Они используются для управления различными техническими объектами или технологическими процессами. Такие системы характеризуются предельно допустимым временем реакции на внешнее событие, в течение которого должна быть выполнена программа, управляющая объектом. Система должна обрабатывать поступающие данные быстрее, чем они могут поступать, причем от нескольких источников одновременно.

Столь жесткие ограничения сказываются на архитектуре систем реального времени, например, в них может отсутствовать виртуальная память, поддержка которой дает непредсказуемые задержки в выполнении программ.

ОС отвечает за следующие действия, связанные с управлением памятью:

- Отслеживание того, какие части памяти в данный момент используются и какими процессами. Как правило, ОС организует для каждого процесса свою виртуальную память – расширение основной памяти путем хранения ее образа на диске и организации подкачки в *основную память* фрагментов (страниц или сегментов) виртуальной памяти процесса и ее откачки по мере необходимости.

- Стратегия загрузки процессов в основную память, по мере ее освобождения. При активизации процесса и его запуске или продолжении его выполнения процесс должен быть загружен в *основную память*, что и осуществляется операционной системой. При этом, возможно, какие-либо не активные в данный момент процессы приходится откачивать на диск.

- Выделение и освобождение памяти по мере необходимости. ОС обслуживает запросы вида "выделить область основной памяти длиной n байтов" и "освободить область памяти, начинающуюся с заданного адреса, длиной m байтов". Длина участков выделяемой и освобождаемой памяти может быть различной. ОС хранит список занятой и свободной памяти. При интенсивном использовании памяти может возникнуть ее фрагментация – дробление на мелкие свободные части, вследствие того, что при запросах на выделение памяти длина найденного сегмента оказывается немного больше, чем требуется, и

остаток сохраняется в списке свободной памяти как область небольшого размера (подчас всего 1 – 2 слова). При исчерпании основной памяти ОС выполняет сборку мусора – поиск не используемых фрагментов, на которые потеряны ссылки, и уплотнение (компактировку) памяти – сдвиг всех используемых фрагментов по меньшим адресам, с корректировкой всех адресов.

Поскольку размер основной памяти недостаточен для постоянного хранения всех программ и данных, в компьютерной системе должна быть предусмотрена вторичная (внешняя) *память* для отдачи части содержимого основной памяти.

В большинстве компьютерных систем в качестве главной вторичной памяти для хранения программ и данных используются диски.

ОС отвечает за выполнение следующих действий, связанных с управлением внешними дисками:

- Управление свободной дисковой памятью;
- Выделение дисковой памяти;
- Диспетчеризация дисков.

При управлении вторичной памятью возникают проблемы, аналогичные проблемам распределения основной памяти. Всякая *память*, даже самая большая *по* объему, рано или поздно может исчерпаться, либо фрагментироваться на множество мелких областей свободной памяти.

В распределенной системе ОС обеспечивает *доступ* пользователей к различным общим сетевым ресурсам – например, файловым системам или принтерам. Каждому общему ресурсу ОС присваивает определенное сетевое имя и управляет возможностью доступа к нему с различных компьютеров сети.

ОС обеспечивает также удаленный запуск программ на другом компьютере сети – возможность входа на другой *компьютер* и работы на нем, с использованием памяти, процессора и диска удаленной, как правило, более мощной машины, и использованием клиентского компьютера в качестве терминала. В *Windows* такая возможность называется удаленный рабочий стол.

Система защиты

Термин защита используется для обозначения механизма управления доступом программ, процессов и пользователей к системным и *пользовательским ресурсам*.

Механизм защиты в ОС должен обеспечивать следующие возможности:

Различать авторизованный, или санкционированный и несанкционированный *доступ*. Под авторизацией понимается предоставление операционной системой пользователю или программе какого-либо определенного набора полномочий, например, возможности чтения или изменения файлов в файловой *системе с общим доступом*.

Описывать предназначенные для защиты элементы *управления (конфигурации)*. Например, в *UNIX* используются специальные текстовые конфигурационные файлы для представления информации о файловых системах, к которым возможен сетевой *доступ*, с указанием списка машин (хостов), с которых возможен *доступ*, и набора действий, которые могут быть выполнены.

Обеспечивать средства выполнения необходимых для защиты действий (сигналы, исключения, *блокировка* и др.). Например, система защиты ОС должна фильтровать *сетевые пакеты*, получаемые извне локальной сети, выбирать и отсеивать "неблагонадежные" (получаемые с подозрительных IP-адресов), сообщать пользователю об обнаруженных и ликвидированных попытках *сетевых атак* с целью "взлома" Вашего компьютера (что и происходит на практике, например, при работе в *Windows*, когда Вы выходите в *Интернет* с Вашего компьютера). Если Вы нарушили условия защиты (например, Ваша *программа* попыталась обратиться к файлу, работать с которым у Вас нет полномочий), ОС должна выдать понятное сообщение и прекратить работу программы. В современных системах это делается с помощью генерации исключений.

Система поддержки командного интерпретатора

Сервисы (службы) ОС

Операционная система предоставляет для пользователей *целый* ряд сервисных возможностей, или, коротко, сервисов (служб):

Исполнение программ – *загрузка* программы в *память* и ее выполнение;

Поддержка ввода-вывода – обеспечение интерфейса для работы программ с *устройствами ввода-вывода*;

Работа с файловой системой – предоставление программам интерфейса для создания, именования, удаления *файлов*.

Коммуникация – *обмен информацией* между процессами, выполняемыми на одном компьютере или на других системах, связанных в *сеть*. В операционных системах реализуется с помощью общей памяти или передачи сообщений.

Обнаружение ошибок в работе *процессора, памяти, устройств ввода-вывода* и программах пользователей.

5. Языки и системы программирования. Модели языков программирования

Программа, написанная на языке программирования (ЯП), должна формально рассматриваться, во-первых, на соответствие синтаксическим правилам языка, а во-вторых, при синтаксической правильности проверяется на семантическую правильность, в-третьих (и это может сделать только человек) программа проверяется на смысловую правильность: является ли она осмысленной и если да, то какой смысл она несет. ЯП, следовательно, должен иметь ясные правила построения предложений (строгий синтаксис), и любое синтаксически верное предложение языка должно однозначно пониматься (однозначная семантика языка).

Язык программирования - Формализованный язык, предназначенный для описания программ и алгоритмов решения задач на ЭВМ. Языки программирования являются искусственными. В них синтаксис и семантика строго определены. Поэтому они не допускают свободного толкования выражения, что характерно для естественного языка. Язык программирования – ядро системы.



Языки высокого уровня не зависят от архитектуры компьютера. Чем более язык ориентирован на человека, тем выше его уровень.

Сравнительная характеристика функционального, логического и процедурного подхода к программированию.

Процедурное программирование

Процедурное (императивное) программирование является отражением архитектуры традиционных ЭВМ, которая была предложена фон Нейманом в 40-х годах. Теоретической моделью процедурного программирования служит алгоритмическая система под названием Машина Тьюринга (абстрактная вычислительная машина).

Программа на процедурном языке программирования состоит из последовательности операторов (инструкций), задающих процедуру решения задачи. Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, то есть значений исходных данных, в заключительное, то есть в результаты. Основным является оператор присваивания, служащий для изменения содержимого областей памяти.

Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней.

Процедурные языки характеризуются следующими особенностями:

- необходимостью явного управления памятью, в частности, описанием переменных;
- малой пригодностью для символьных вычислений;
- отсутствием строгой математической основы;
- высокой эффективностью реализации на традиционных ЭВМ.

Процедурный язык программирования предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на шаги и решаются шаг за шагом. Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов.

К процедурным языкам относятся: язык Ассемблера, C, Basic (версии начиная с QuickBasic до появления VisualBasic), Pascal.

Функциональное программирование

Функциональное программирование — раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний. При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов

других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

Первым функциональным языком программирования был LISP. Роль основной конструкции в функциональных (аппликативных) языках играет выражение. К выражениям относятся скалярные константы, структурированные объекты, функции, тела функций и вызовы функций.

Функциональный язык программирования включает следующие элементы:

- классы констант, которыми могут манипулировать функции;
- набор базовых функций, которые программист может использовать без предварительного объявления и описания;
- правила построения новых функций из базовых;
- правила формирования выражений на основе вызовов функций.

Программа представляет собой совокупность описаний функций и выражения, которые необходимо вычислить. Данное выражение вычисляется посредством редукции, то есть серии упрощений, до тех пор, пока это возможно по следующим правилам:

- вызовы базовых функций заменяются соответствующими значениями;
- вызовы не базовых функций заменяются их телами, в которых параметры замещены аргументами.

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

Преимущества:

- повышение надёжности;
- удобство организации модульного тестирования;
- возможности оптимизации при компиляции;
- возможности параллелизма.

Недостатки:

Недостатки функционального программирования вытекают из тех же самых его особенностей. Отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный сборщик мусора. Нестрогая модель вычислений приводит к непредсказуемому порядку вызова функций, что создает проблемы при вводе-выводе, где порядок выполнения операций важен. Кроме того, очевидно, функции ввода в своем естественном виде (например, `getchar` из стандартной библиотеки языка C) не являются чистыми, поскольку способны возвращать различные значения для одних и тех же аргументов, и для устранения этого требуются определенные ухищрения.

Для преодоления недостатков функциональных программ уже первые языки функционального программирования включали не только чисто функциональные средства, но и механизмы императивного программирования (присваивание, цикл, «неявный PROG» были уже в LISPe). Использование таких средств позволяет решить некоторые практические проблемы, но означает отход от идей (и преимуществ) функционального программирования и написание императивных программ на функциональных языках.

Логическое программирование

Логическое программирование — парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций.

Первым таким языком был язык Planner, в котором была заложена возможность автоматического вывода результата из данных и заданных правил перебора вариантов (совокупность которых называлась планом). Planner использовался для того, чтобы понизить требования к вычислительным ресурсам (с помощью метода *backtracking*) и обеспечить возможность вывода фактов, без активного использования стека. Затем был разработан язык Prolog, который не

требовал плана перебора вариантов и был, в этом смысле, упрощением языка Planner.

Языки логического программирования, в особенности Пролог, широко используются в системах искусственного интеллекта. Центральным понятием в логическом программировании является отношение. Программа представляет собой совокупность определений отношений между объектами (в терминах условий или ограничений) и цели (запроса). Процесс выполнения программы трактуется как процесс обще значимости логической формулы, построенной из программы по правилам, установленным семантикой используемого языка. Результат вычисления является побочным продуктом этого процесса. В реляционном программировании нужно только специфицировать факты, на которых алгоритм основывается, а не определять последовательность шагов, которые требуется выполнить.

Языки логического программирования характеризуются:

- высоким уровнем;
- строгой ориентацией на символьные вычисления;
- возможностью инверсных вычислений, то есть переменные в процедурах не делятся на входные и выходные;
- возможной логической неполнотой, поскольку зачастую невозможно выразить в программе определенные логические соотношения, а также невозможно получить из программы все выводы правильные.

Сравнительная характеристика языков программирования PASCAL, LISP, PROLOG:

Характеристика	PASCAL	LISP	PROLOG
тип языка	процедурный	функциональный	логический
типы данных	скаляры, структуры	атомы, списки	атомы, структуры
обработка данных	присвоение, передача по значению, передача по ссылке	значение функции, передача по значению	связь переменных через унификацию
управление программой	последовательное, ветвление, циклы, рекурсия	вычисление функций, условные вычисления, рекурсии, циклы	рекурсия, бэктрекинг
программы	блоки, процедуры	функции, LET-блоки	правила, факты
действия переменных	глобальные, локальные	локальные, свободные	область действия переменной - одно предложение
транслятор	компилятор	интерпретатор, компилятор	интерпретатор
длина программы	5	3	1
скорость	1	2	5
область	программы общего назначения	символьная обработка, ИИ	ЭС, ИИ, прототипы

Системы программирования - это комплекс инструментальных программных средств, предназначенный для работы с программами на одном из языков программирования. В их состав входят:

- трансляторы с языков высокого уровня;
- средства редактирования, компоновки и загрузки программ;
- макроассемблеры (машинно-ориентированные языки);
- отладчики машинных программ.

Системы программирования, включают в себя:

*Текстовый редактор,
Загрузчик программ;
Запускающий программ;
Компилятор;
Отладчик;
Диспетчер файлов и т.п.*

Транслятор – это программа, которая переводит входную программу на исходном (входном) языке в эквивалентную ей выходную программу на результирующем (выходном) языке.

Компилятор – это транслятор, который осуществляет перевод исходной программы в эквивалентную ей объектную программу на языке машинных команд или языке ассемблера (.exe).

Интерпретатор – это программа, которая воспринимает входную программу на исходном языке, переводит каждый оператор или строку в машинный язык и выполняет их. Интерпретатор не порождает результирующую программу и никакого результирующего кода.

Модели языков программирования

Рис. 21.1

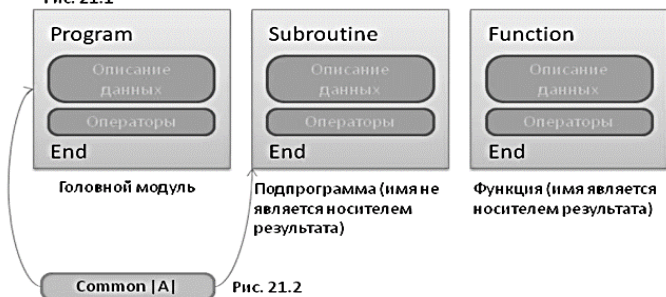


Рис. 21.2

Модульная модель языков программирования

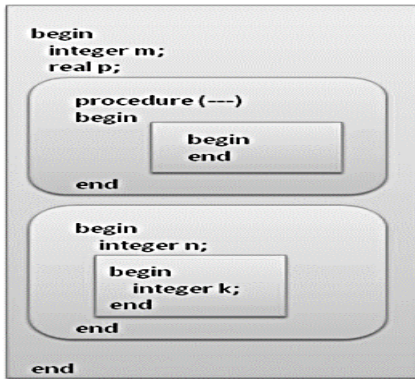
- Каждая подпрограмма (или функция) является самостоятельным модулем;
- Определенные внутри модуля переменные локализованы в нём

(не глобализованы);

- Память на объекты (переменные) выделяется статически;
- Наличие областей памяти, разделяемых различными подпрограммами.

Принцип модульности ляжет практически во все языки программирования. Сейчас понятие функции и модуля тождественно.

Блочная модель языков программирования



- Вся программа представляет собой единый моноблок;

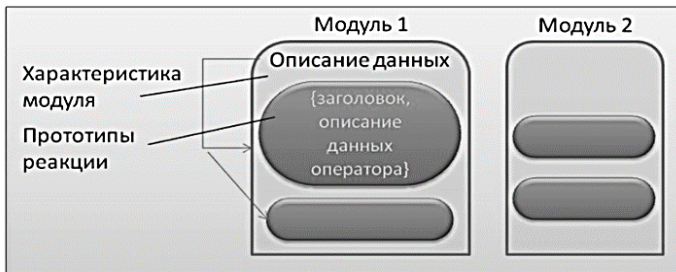
- Существует единственный программный «модуль»;

- Вся программа представляет собой систему вложенных блоков;

- Процедуры и функции вкладываются в блоки, их тела являются полноценными блоками;

- Данные локализованы в блоках;
- Внутренний блок наследует данные объемлющих его блоков;
- Память под переменные выделяется при входе в блок и удаляется при выходе из него (т.е. полная динамика – статическая модель отсутствует).

Блочно - модульная архитектура



Здесь есть блочная структура, но она сильно лимитированная. Блоки есть, но нельзя создавать операцию одну в другой;

- Все активные компоненты вкладываются внутрь модуля;
- Модулей уже несколько;
- Функцию в функции мы создать не можем, но блоки у нас уже есть.

6. Модели и этапы разработки программного обеспечения

Из этих моделей наиболее популярны пять основных: каскадная, V-образная, инкрементная, итерационная и спиральная.

Каскадная модель («Водопад»)

Разработка осуществляется поэтапно. Подходит для разработки проектов в *медицинской и космической отрасли, где уже сформирована обширная база документов*



Преимущества «водопада»

- *Разработку просто контролировать.* Заказчик всегда знает, чем сейчас заняты программисты, может управлять сроками и стоимостью.
- *Стоимость проекта определяется на начальном этапе.* Все шаги запланированы уже на этапе согласования договора, ПО пишется непрерывно «от и до».

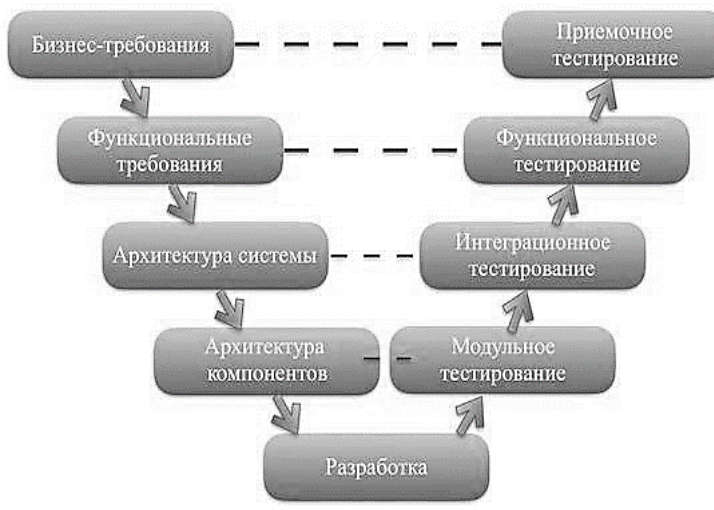
- *Не нужно нанимать тестировщиков с серьёзной технической подготовкой.* Тестировщики смогут опираться на подробную техническую документацию.

Недостатки каскадной модели

- *Тестирование начинается на последних этапах разработки.* Если в требованиях к продукту была допущена ошибка, то исправить её будет стоить дорого. Тестировщики обнаружат её, когда разработчик уже написал код, а технические писатели — документацию.
- *Заказчик видит готовый продукт в конце разработки и только тогда может дать обратную связь.* Велика вероятность, что результат его не устроит.
- *Разработчики пишут много технической документации, что задерживает работы.* Чем обширнее документация у проекта, тем больше изменений нужно вносить и дольше их согласовывать.

V-образная модель

Это усовершенствованная каскадная модель, в которой заказчик с командой программистов одновременно составляют требования к системе и описывают, как будут тестировать её на каждом этапе.



Преимущества V-образной модели

- Количество ошибок в архитектуре ПО сводится к минимуму.

Недостатки V-образной модели

- Если при разработке архитектуры была допущена ошибка, то вернуться и исправить её будет стоить дорого, как и в «водопаде».

V-модель подходит для проектов, в которых важна надёжность и цена ошибки очень высока. (разработка подушек безопасности или системы наблюдения за пациентами в клиниках).

Инкрементная модель разработки по частям



Преимущества инкрементной модели

- *Не нужно вкладывать много денег на начальном этапе.* Заказчик оплачивает создание основных функций, получает продукт, «выкатывает» его на рынок — и по итогам обратной связи решает, продолжать ли разработку.

- *Можно быстро получить фидбэк от пользователей и оперативно обновить техническое задание.* Так снижается риск создать продукт, который никому не нужен.

- *Ошибка обходится дешевле.* Если при разработке архитектуры была допущена ошибка, то исправить её будет стоить не так дорого, как в «водопаде» или V-образной модели.

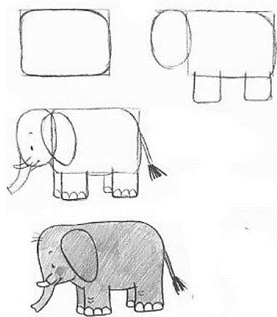
Недостатки инкрементной модели

- *Каждая команда программистов разрабатывает свою функциональность и может реализовать интерфейс продукта по-своему.* Чтобы этого не произошло, важно на этапе обсуждения техзадания объяснить, каким он будет, чтобы у всех участников проекта сложилось единое понимание.

- *Разработчики будут оттягивать доработку основной функциональности и «пилить мелочёвку».* Чтобы этого не случилось, менеджер проекта должен контролировать, чем занимается каждая команда.

Инкрементная модель подходит для проектов, в которых точное техзадание прописано уже на старте, а продукт должен быстро выйти на рынок.

Итерационная модель, при которой заказчик не обязан понимать, какой продукт хочет получить в итоге, и может не прописывать сразу подробное техзадание.



Преимущества итеративной модели

- *Быстрый выпуск минимального продукта* даёт возможность оперативно получать обратную связь от заказчика и пользователей. А значит, фокусироваться на наиболее важных функциях ПО и улучшать их в соответствии с требованиями рынка и пожеланиями клиента.

- *Постоянное тестирование пользователями* позволяет быстро обнаруживать и устранять ошибки.

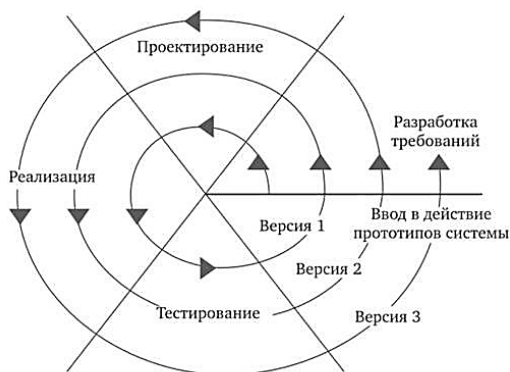
Недостатки итеративной модели

- *Использование на начальном этапе баз данных или серверов*— первые сложно масштабировать, а вторые не выдерживают нагрузку. Возможно, придётся переписывать большую часть приложения.

- *Отсутствие фиксированного бюджета и сроков.* Заказчик не знает, как выглядит конечная цель и когда закончится разработка.

Итеративная модель подходит для работы над *большими проектами с неопределёнными требованиями*, либо для задач с *инновационным подходом*, когда заказчик не уверен в результате.

Спиральная модель



Заказчик и команда разработчиков серьёзно анализируют риски проекта и выполняют его итерациями. Последующая стадия основывается на предыдущей, а в конце каждого витка – цикла итераций – принимается решение, продолжать ли проект.

Спиральная модель похожа на инкрементную, но здесь гораздо больше

времени уделяется оценке рисков. С каждым новым витком спирали процесс усложняется. Эта модель часто используется в *исследовательских проектах и там, где высоки риски*.

Преимущества спиральной модели

- *Большое внимание уделяется проработке рисков.*

Недостатки спиральной модели

- *Есть риск застрять на начальном этапе* — бесконечно совершенствовать первую версию продукта и не продвинуться к следующим.

Разработка длится долго и стоит дорого. (Система «Умный дом»).

Этапы разработки ПО

У любого программного обеспечения есть жизненный цикл – этапы, через которые оно проходит с начала создания до конца разработки и внедрения. Чаще всего это подготовка, проектирование, создание и поддержка.

Этапы разработки ПО

1. Понять природу и сферу применения продукта. Бизнес-моделирование - описывается деятельность компании и определяются требования к системе — те подпроцессы и операции, которые подлежат автоматизации в разрабатываемой системе. Используется SADT, RUP: диаграмма деятельности. Это кажется очевидным, однако для того, чтобы понять, чего хочет заказчик, требуется ощутимое время, особенно если заказчик сам не знает достаточно хорошо, чего он в действительности хочет. Нужно составить представление о масштабах проекта и с этой целью оценить, какими сроками, финансами и персоналом мы располагаем.

2. Анализ требований (Requirements Engineering) — это процесс сбора требований к системе, их систематизации, документирования, анализа, выявления противоречий, неполноты, разрешения конфликтов.

Требование – это потребность или ожидание, которое

- установлено в явном виде (например, заказчиком)
- наследуется, как обязательное из других систем требований (ГОСТа)
- подразумевается (например, выход по ESC) Описание системы

3. Проектирование ПО. Разработка архитектуры.

Архитектура – концепция, определяющая модель, структуру, выполняемые функции и взаимосвязь компонентов информационной системы.

Архитектура программной системы охватывает не только ее структурные и поведенческие аспекты, но и правила ее использования и интеграции с другими системами, функциональность, производительность, гибкость, надежность, возможность повторного применения, полноту, экономические и технологические ограничения, а также вопрос пользовательского интерфейса.

4. Кодирование – реализация одного или нескольких взаимосвязанных алгоритмов на некотором языке программирования.

5. Тестирование (частей ПО → интеграция и тестирование системы в целом) - процесс, позволяющий определить корректность, полноту и качество разработанного ПО; процесс формальной проверки или верификации может доказать, что дефекты отсутствуют, с точки зрения используемого метода.

В зависимости от используемого процесса разработки (модели разработки) шаги 2 - 5 могут быть выполнены несколько раз.

6. Документирование - это документы, сопровождающие некоторое ПО.

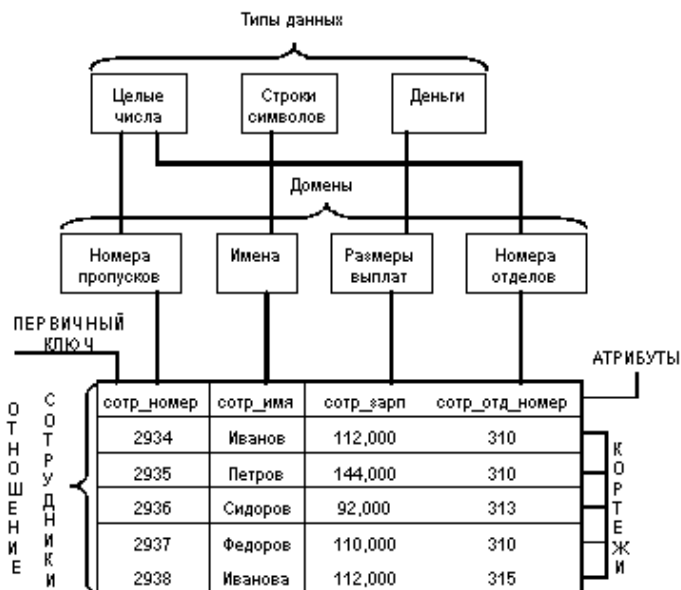
- Архитектурная/проектная - обзор программного обеспечения, включающий описание рабочей среды и принципов, которые должны быть использованы при создании ПО;
- Техническая - документация на код, алгоритмы, интерфейсы, API;
- Пользовательская – руководства для конечных пользователей, администраторов системы и другого персонала;
- Маркетинговая.

7. Внедрение - процесс настройки программного обеспечения под определенные условия использования, а также обучения пользователей работе с программным продуктом

8. Сопровождение - процесс улучшения, оптимизации и устранения дефектов (ПО) после передачи в эксплуатацию. Может потребовать до 80 % ресурсов, потребовавшихся на разработку.

7. Реляционные СУБД. Объектно-ориентированные базы данных

Основными понятиями реляционных баз данных являются тип данных, домен, атрибут, кортеж, первичный ключ и отношение. Для начала покажем смысл этих понятий на примере отношения СОТРУДНИКИ, содержащего информацию о сотрудниках некоторой организации:



Понятие тип данных в реляционной модели данных полностью адекватно понятию типа данных в языках программирования. Обычно в современных реляционных БД допускается хранение символьных, числовых данных, битовых строк, специализированных числовых данных (таких как "деньги"), а также специальных "темпоральных" данных (дата, время, временной интервал).

Понятие домена более специфично для баз данных, хотя и имеет некоторые аналогии с подтипами в некоторых языках программирования. В самом общем виде домен определяется заданием некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу типа данных. Если вычисление этого логического выражения дает результат "истина", то элемент данных является элементом домена.

Схема отношения - это именованное множество пар {имя атрибута, имя домена (или типа, если понятие домена не поддерживается)}. Степень или "арность" схемы отношения - мощность этого множества. Степень отношения СОТРУДНИКИ равна четырем, то есть оно является 4-арным. Если все атрибуты одного отношения определены на разных доменах, осмысленно использовать для именования атрибутов имена соответствующих доменов (не забывая, конечно, о том, что это является всего лишь удобным способом именования и не устраняет различия между понятиями домена и атрибута).

Схема БД (в структурном смысле) - это набор именованных схем отношений.

Кортеж, соответствующий данной схеме отношения, - это множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. "Значение" является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым, степень или "арность" кортежа, т.е. число элементов в нем, совпадает с "арностью" соответствующей схемы отношения. Попросту говоря, кортеж - это набор именованных значений заданного типа.

Отношение - это множество кортежей, соответствующих одной схеме отношения. Иногда, чтобы не путаться, говорят "отношение-схема" и "отношение-экземпляр", иногда схему отношения называют заголовком отношения, а отношение как набор кортежей - телом отношения. На самом деле, понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования.

Обычным житейским представлением отношения является таблица, заголовком которой является схема отношения, а строками - кортежи отношения-экземпляра; в этом случае имена атрибутов именуют столбцы этой таблицы. Поэтому иногда говорят "столбец таблицы", имея в виду "атрибут отношения".

Реляционная база данных - это набор отношений, имена которых совпадают с именами схем отношений в схеме БД.

Фундаментальные свойства отношений:

- Отсутствие кортежей-дубликатов. То свойство, что отношения не содержат кортежей-дубликатов, следует из определения отношения как множества кортежей. В классической теории множеств по определению каждое множество состоит из различных элементов. Из этого свойства вытекает наличие у каждого отношения так называемого первичного ключа - набора атрибутов, значения которых однозначно определяют кортеж отношения. Для каждого отношения по крайней мере полный набор его атрибутов обладает этим свойством. Однако при формальном определении первичного ключа требуется обеспечение его "минимальности", т.е. в набор атрибутов первичного ключа не должны входить такие атрибуты, которые можно отбросить без ущерба для основного свойства - однозначно определять кортеж.

- Отсутствие упорядоченности кортежей. Свойство отсутствия упорядоченности кортежей отношения также является следствием определения отношения-экземпляра как множества кортежей.

- Отсутствие упорядоченности атрибутов. Атрибуты отношений не упорядочены, поскольку по определению схема отношения есть множество пар {имя атрибута, имя домена}. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута.

- Атомарность значений атрибутов. Значения всех атрибутов являются атомарными. Это следует из определения домена как потенциального множества значений простого типа данных, т.е. среди значений домена не могут содержаться множества значений (отношения).

- Принято говорить, что в реляционных базах данных допускаются только нормализованные отношения или отношения, представленные в первой нормальной форме.

Наиболее распространенная трактовка реляционной модели данных, по-видимому, принадлежит Дейту, который воспроизводит ее (с различными уточнениями) практически во всех своих книгах. Согласно Дейту реляционная модель состоит из трех частей,

описывающих разные аспекты реляционного подхода: структурной части, манипуляционной части и целостной части.

В структурной части модели фиксируется, что единственной структурой данных, используемой в реляционных БД, является нормализованное n -арное отношение.

В манипуляционной части модели утверждаются два фундаментальных механизма манипулирования реляционными БД - реляционная алгебра и реляционное исчисление. Первый механизм базируется в основном на классической теории множеств (с некоторыми уточнениями), а второй - на классическом логическом аппарате исчисления предикатов первого порядка.

Наконец, в целостной части реляционной модели данных фиксируются два базовых требования целостности, которые должны поддерживаться в любой реляционной СУБД.

Первое требование называется требованием целостности сущностей. Объекту или сущности реального мира в реляционных БД соответствуют кортежи отношений. Конкретно требование состоит в том, что любой кортеж любого отношения отличим от любого другого кортежа этого отношения, т.е. другими словами, любое отношение должно обладать первичным ключом. Это требование автоматически удовлетворяется, если в системе не нарушаются базовые свойства отношений.

Второе требование называется требованием целостности по ссылкам и является несколько более сложным. Требование целостности по ссылкам, или требование внешнего ключа состоит в том, что для каждого значения внешнего ключа, появляющегося в ссылающемся отношении, в отношении, на которое ведет ссылка, должен найтись кортеж с таким же значением первичного ключа, либо значение внешнего ключа должно быть неопределенным (т.е. ни на что не указывать). Для примера это означает, что если для сотрудника указан номер отдела, то этот отдел должен существовать.

Существует много подходов к определению реляционной алгебры, которые различаются набором операций и способами их интерпретации, но в принципе, более или менее равносильны. Мы опишем немного расширенный начальный вариант алгебры, который был предложен Коддом. В этом варианте набор основных алгебраических операций состоит из восьми операций, которые делятся на два класса - теоретико-множественные операции и специальные реляционные операции. В состав теоретико-множественных операций входят операции:

- объединения отношений; При выполнении операции объединения двух отношений производится отношение, включающее все кортежи, входящие хотя бы в одно из отношений-операндов.

- пересечения отношений; Операция пересечения двух отношений производит отношение, включающее все кортежи, входящие в оба отношения-операнда.

- взятия разности отношений; Отношение, являющееся разностью двух отношений включает все кортежи, входящие в отношение - первый операнд, такие, что ни один из них не входит в отношение, являющееся вторым операндом.

- прямого произведения отношений. При выполнении прямого произведения двух отношений производится отношение, кортежи которого являются конкатенацией (сцеплением) кортежей первого и второго операндов.

Специальные реляционные операции включают:

- ограничение отношения; Результатом ограничения отношения по некоторому условию является отношение, включающее кортежи отношения-операнда, удовлетворяющие этому условию.

- проекцию отношения; При выполнении проекции отношения на заданный набор его атрибутов производится отношение, кортежи которого производятся путем взятия соответствующих значений из кортежей отношения-операнда.

- соединение отношений; При соединении двух отношений по некоторому условию образуется результирующее отношение, кортежи которого являются конкатенацией кортежей первого и второго отношений и удовлетворяют этому условию.

- деление отношений. У операции реляционного деления два операнда - бинарное и унарное отношения. Результирующее отношение состоит из одноатрибутных кортежей, включающих значения первого атрибута кортежей первого операнда таких, что множество значений второго атрибута (при фиксированном значении первого атрибута) совпадает со множеством значений второго операнда.

Кроме того, в состав алгебры включается операция присваивания, позволяющая сохранить в базе данных результаты вычисления алгебраических выражений, и операция переименования атрибутов, дающая возможность корректно сформировать заголовок (схему) результирующего отношения. Операция присваивания позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД. Операция переименования производит

отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены.

Объектно-ориентированные базы данных

Одной из новейших областей исследований баз данных является их создание на основе объектно-ориентированной парадигмы. В результате получается объектно-ориентированная база данных (object-oriented database), состоящая из объектов, связи между которыми отражают отношения между объектами. Например, объектно-ориентированная реализация базы данных сотрудников:

Отношение EMPLOYEE

Empl Id	Name	Address	SSN
25X15	Джо Бейкер	ул. Новая 33	111223333
34Y70	Шери Кларк	Верхнегородский пр. 563	999009999
23Y34	Джерри Смит	Круглый пер. 1555	111005555
⋮	⋮	⋮	⋮

Отношение JOB

Job Id	Job Title	Skill Code	Dept
S25X	Секретарь	T5	Отдел кадров
S26Z	Секретарь	T6	Бухгалтерия
F5	Нач. группы	FM3	Отдел сбыта
⋮	⋮	⋮	⋮

Отношение ASSIGNMENT

Empl Id	Job Id	Start Date	Term Date
23Y34	S25X	3-1-1999	4-30-2001
34Y70	F5	10-1-2001	*
23Y34	S25Z	5-1-2001	*
⋮	⋮	⋮	⋮

Рис. 9.5. База данных сотрудников, состоящая из трех отношений

могла бы включать три класса (то есть три типа объектов): EMPLOYEE, JOB и ASSGNMENT. Объект класса EMPLOYEE мог бы содержать такие элементы, как EmplId, Name, Address и SSNum; объект класса JOB — элементы JobId, JobTitle, Skill Code и Dept; объект класса ASSGNMENT — элементы StartDate и TermDate.

Концептуальное представление такой базы данных (рис. 11) образуется объектами и соединяющими их линиями, показывающими отношения между различными объектами. Рассмотрев объект типа EMPLOYEE, мы увидим, что он связан с набором объектов типа ASSIGNMENT, представляющих различные назначения сотрудника на должности. В свою очередь, каждый из объектов типа ASSIGNMENT связан с объектом типа JOB, обозначающим должность, которую занимал или занимает сотрудник. Таким образом, все назначения сотрудника на различные должности можно найти, проследив связи, идущие от объекта, представляющего сотрудника. Аналогично можно узнать, какие сотрудники занимали определенную должность, изучив ссылки от объекта, представляющего должность.

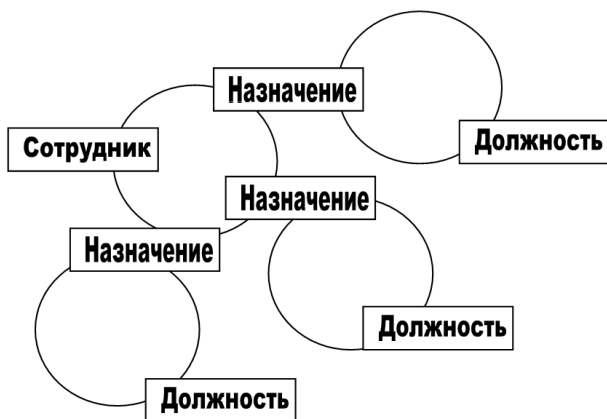


Рис. 11 – Связи между объектами в объектно-ориентированной базе данных

Связи между объектами в объектно-ориентированной базе данных обычно поддерживаются СУБД, поэтому подробности их реализации не касаются программиста, разрабатывающего приложения. Когда новый объект добавляется в базу данных, приложению необходимо только указать, с какими объектами его нужно связать в базе. СУБД сама создает необходимую для регистрации этих связей систему указателей. В частности, СУБД может связать объекты, обозначающие назначения на разные

должности определенного сотрудника, в стиле, схожем со связным списком.

Другая задача объектно-ориентированной СУБД — обеспечивать постоянное хранение переданных ей объектов. Такое требование может казаться очевидным, но в действительности хранение отличается от обычного способа работы с объектами. В обыкновенной объектно-ориентированной программе объекты, созданные в ходе выполнения, удаляются во время завершения программы. В этом смысле объекты считаются временными. Но объекты, созданные и помещенные в базу данных, должны быть постоянными — то есть их необходимо сохранить после того, как программа, создавшая их, завершится. Поэтому обеспечение постоянного хранения объектов — это существенное отклонение от нормы.

Сторонники объектно-ориентированных баз данных приводят множество аргументов в защиту того, что объектно-ориентированный подход к разработке базы данных лучше, чем реляционный. Один из аргументов заключается в том, что он обеспечивает единую парадигму разработки всей системы программного обеспечения (приложения, СУБД и самой базы данных). В противоположность этому, исторически приложения для обмена данными с реляционными базами данных создаются на императивном языке программирования. Но при этом всегда возникают конфликты между императивной и реляционной парадигмами. На нашем уровне изучения это различие едва ли заметно, но оно всегда являлось причиной множества ошибок в программном обеспечении. Однако мы можем оценить тот факт, что объектно-ориентированная база данных в сочетании с объектно-ориентированным приложением создают однородный образ объектов, общающихся между собой внутри единой системы, тогда как реляционная база данных и императивное приложение вызывают образ двух структур различной природы, пытающихся найти общий интерфейс.

Чтобы понять другое преимущество объектно-ориентированных баз данных над их реляционными конкурентами, рассмотрим проблему хранения имен сотрудников в реляционной базе данных. Если полное имя (фамилия, имя и отчество) целиком хранится в одном атрибуте отношения, то выполнять запросы, относящиеся только к фамилиям, становится неудобно. Если же имя хранится в трех разных атрибутах — фамилия, имя и отчество, — то вызывает неудобства обработки имен, не подходящих под шаблон «фамилия, имя, отчество». В объектно-ориентированной базе данных эти проблемы можно спрятать в объекте, где хранится имя сотрудника. Имя можно

записать в интеллектуальный объект, который сможет выдать имя сотрудника в различных форматах. Так, с использованием объектов обработка только фамилий становится такой же простой, как и работа с полными именами, девичьими фамилиями или прозвищами. Детали, связанные с каждой задачей, будут инкапсулированы в пределах объектов.

Возможность инкапсулировать технические подробности различных форматов данных полезна и в других случаях. В реляционной базе данных атрибуты отношения являются частью общего дизайна базы данных, поэтому типы, связанные с этими атрибутами, появляются во всей СУБД. (Необходимо объявлять временные переменные подходящих типов и разрабатывать процедуры для обработки данных различных типов.) Таким образом, расширение реляционной базы данных за счет добавления атрибутов новых типов (к примеру, звуковых и видео) может быть проблематичным. Действительно, это потребует расширения множества процедур во всей базе данных, чтобы они могли работать и с новыми типами. С другой стороны, в объектно-ориентированной модели процедуры, которые используются для получения объектов, обозначающих имена сотрудников, могут извлекать информацию, например из объекта, представляющего видеоклип, так как различие в типах может быть скрыто внутри соответствующих объектов. Следовательно, при помощи объектно-ориентированного подхода удобнее конструировать мультимедийные базы данных — эта возможность уже является существенным преимуществом.

Еще одно преимущество, которое объектно-ориентированная парадигма предлагает для разработки баз данных, — это возможность **хранения интеллектуальных объектов** вместо обычных данных. Это означает, что **объект может содержать методы, описывающие, как он будет отвечать на сообщения**, относящиеся к его содержимому и связям. Например, каждый объект класса EMPLOYEE (см. рис. 11) может содержать методы для вывода и обновления информации, хранящейся в объекте, а также для вывода предыдущих должностей сотрудника и, возможно, для регистрации перевода сотрудника на новую должность. Аналогично, каждый объект класса JOB может обладать методами для вывода характеристик должности и, возможно, для вывода списка сотрудников, которые занимали эту должность. Так, чтобы получить историю работы сотрудника в компании, нам не нужно будет создавать внешние процедуры, описывающие, как получить информацию. Вместо этого мы просто попросим соответствующий объект сотрудника рассказать его историю работы.

Таким образом, способность создавать базы данных, объекты которых могут интеллектуально отвечать на запросы, предлагает исключительный набор возможностей, далеко превосходящий возможности традиционных реляционных баз данных.

В объектно-ориентированных базах данных, в отличие от реляционных, хранятся не записи, а объекты. ОО-подход представляет более совершенные средства для отображения реального мира, чем реляционная модель, естественное представление данных.

К сожалению, в объектно-ориентированном программировании отсутствуют общие средства манипулирования данными, такие как реляционная алгебра или реляционное счисление. Работа с данными ведется с помощью одного из объектно-ориентированных языков программирования общего назначения, обычно это SmallTalk, C++ или Java.

Подведем теперь некоторые итоги:

В реляционной модели все отношения принадлежат одному уровню, именно это осложняет преобразование иерархических связей модели «сущность-связь» в реляционную модель. ОО-модель можно рассматривать послойно, на разных уровнях абстракции.

Имеется возможность определения новых типов данных и операций с ними.

В то же время, ОО-модели присущ и ряд недостатков:

1. · Отсутствуют мощные непроедурные средства извлечения объектов из базы. Все запросы приходится писать на процедурных языках, проблема их оптимизации возлагается на программиста;

2. · Вместо чисто декларативных ограничений целостности (типа явного объявления первичных и внешних ключей реляционных таблиц с помощью ключевых слов PRIMARY KEY и REFERENCES) или полудекларативных триггеров для обеспечения внутренней целостности приходится писать процедурный код.

Очевидно, что оба эти недостатка связаны с отсутствием развитых средств манипулирования данными. Эта задача решается двумя способами — расширение ОО-языков в сторону управления данными (стандарт ODMG), либо добавление объектных свойств в реляционные СУБД (SQL-3, а также так называемые объектно-реляционных СУБД).

Основными принципами объектно-ориентированной технологии являются: • абстракция; • инкапсуляция; • модульность; • иерархия; • типизация; • полиморфизм; • наследование.

Объектно-ориентированной БД (ООБД) является БД, которая основывается на принципах объектно-ориентированной технологии.

8. Архитектуры вычислительных систем. Архитектура системы команд

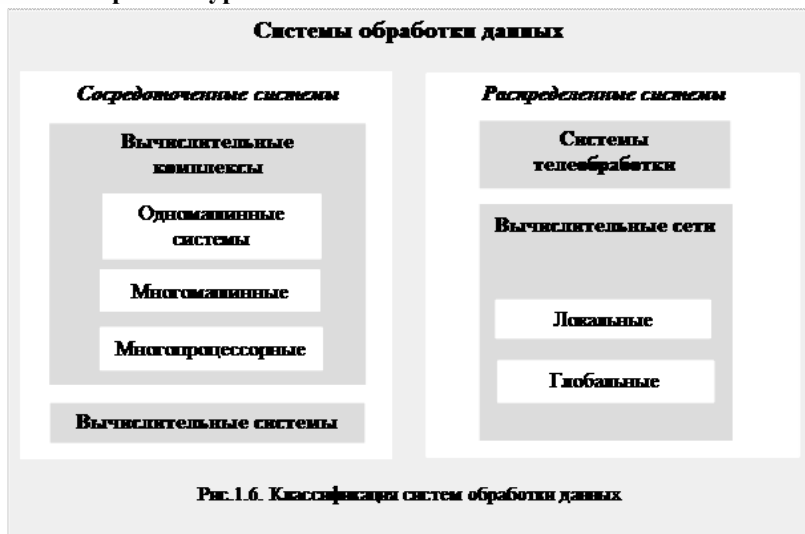
Архитектура ВС – совокупность характеристик и параметров, определяющих функционально-логическую и структурную организацию системы. Важна возможность систем, а не детали их технического исполнения.

Архитектура вычислительной машины (англ. computerarchitecture) – концептуальная структура вычислительной машины, определяющая проведение обработки информации и включающая методы преобразования информации в данные и принципы взаимодействия технических средств и программного обеспечения.

Архитектура системы – совокупность свойств системы, существенных для пользования.

Архитектурой компьютера называется его описание на некотором общем уровне, включающее описание пользовательских возможностей программирования, системы команд, системы адресации, организации памяти и т.д. Архитектура определяет принципы действия, информационные связи и взаимное соединение основных логических узлов компьютера: процессора, оперативного ЗУ, внешних ЗУ и периферийных устройств. Общность архитектуры разных компьютеров обеспечивает их совместимость с точки зрения пользователя.

Общая классификация вычислительных систем обработки данных – архитектур



Классифицировать системы, в том числе и вычислительные, можно по одному или нескольким характерным признакам. В зависимости от этого классификационная структура может выглядеть совершенно по-разному. Если классифицировать системы обработки данных по способу построения, то есть в том порядке, как они здесь представлены, классификация будет выглядеть как на рисунке 1.6.

Все системы обработки данных можно разделить на два класса. Отдельные ЭВМ, вычислительные комплексы или специализированные системы образуют класс *сосредоточенных (централизованных) систем*, в которых реализуется вся обработка данных.

Тогда системы телеобработки и вычислительные сети следует отнести к классу *распределенных систем*, где обработка данных рассредоточена по многим составляющим системы. При этом системы телеобработки относят к распределенным до некоторой степени условно, поскольку обработка реализуется централизованно, на одной ЭВМ или вычислительном комплексе.

Архитектура системы команд

Системой команд вычислительной машины называют полный перечень команд, которые способна выполнять данная ЭВМ. В свою очередь, под архитектурой системы команд (АСК) принято определять те средства вычислительной машины, которые видны и доступны программисту.

При классифицировании систем параллельной обработки данных оказалось полезным ввести понятие *множественности команд* и *множественности данных*. Под *множеством* надо понимать наличие нескольких последовательностей команд (программ или частей программ), находящихся одновременно в стадии выполнения, или последовательностей данных, подвергающихся обработке. Исходя из выше приведенных условий, все параллельные системы разбили на четыре больших класса:

- системы с одиночным потоком команд и одиночным потоком данных – ОКОД;
- системы с множественным потоком команд и одиночным потоком данных – МКОД;
- системы с одиночным потоком команд и множественным потоком данных – ОКМД;
- системы с множественным потоком команд и множественным потоком данных – МКМД;

Рассмотрим кратко особенности каждого класса.
Системы ОКОД.



Схематически такую систему можно представить так, как показано на рисунке 2.8. Системы этого класса – обычные однопроцессорные ЭВМ. В таких системах организация параллельной обработки заключается в совмещении во времени различных этапов разных задач, при одновременной обработке их разными устройствами. Например, одновременно работают печатающие устройства по нескольким каналам, вводятся данные с накопителей на магнитных дисках, работает процессор и т.д., и все это делается по различным программам.

Для улучшения производительности применяют:

- Конструирование оперативных запоминающих устройств (ОЗУ) в виде многомодульной структуры обеспечивает каждому модулю независимое функционирование и дает определенный, ощутимый эффект увеличения производительности. При этом уменьшается количество конфликтов при обращении к ОЗУ нескольких устройств ввода-вывода сразу.
- Конфликты разрешаются так же путем введения приоритетов устройств, которые определяют очередность их обращений к ОЗУ. При возникновении очередей возможны простои оборудования.
- Конвейер команд также можно рассматривать как совмещение во времени работы нескольких устройств – блоков, выполняющих отдельные части операции.

Если к этому еще добавить использование многомодульных ОЗУ можно добиться существенного увеличения производительности

системы. При этом программы и данные можно разместить в разных модулях и совмещать во времени выборку команд и операндов.

- Кроме того, можно сократить время доступа к памяти, если применить способ чередования адресов (расслоение обращений) памяти – один модуль будет иметь четные адреса, другой нечетные и начало цикла памяти смещено на половину цикла.

Системы класса МКОД.



Рис. 2.9. Схема системы МКОД

По определению такая система должна была бы выглядеть так, как представлено на рисунке 2.9, однако, не существует таких задач, в которых одна и та же последовательность данных обрабатывалась бы различными программами. По этой причине на практике реализована система, представленная рисунком 2.10.

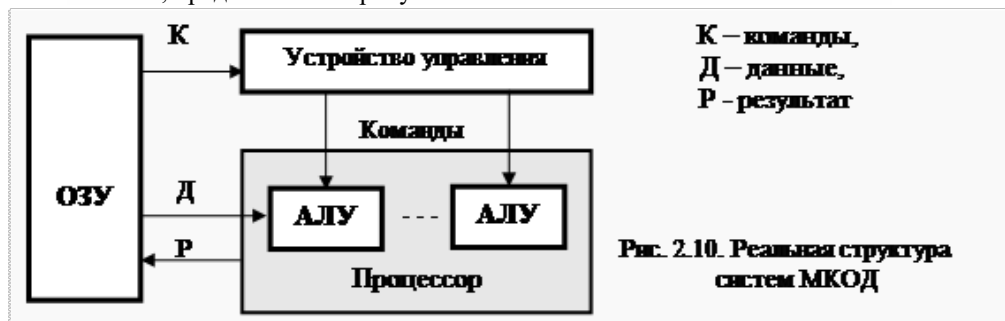


Рис. 2.10. Реальная структура систем МКОД

Фактический, одиночный поток команд устройством управления разделяется на несколько потоков микрокоманд (микроопераций). Каждая из микроопераций реализуется специализированным, настроенным на выполнение именно данной микрооперации устройством.

Поток данных последовательно проходит через все или часть этих специализированных АЛУ. Именно такие системы называют

конвейерными или системами с магистральной обработкой информации. Определяющим признаком таких систем является наличие конвейера арифметических или логических операций. Напомним, что такие системы максимально производительны для определенных типов задач, имеющих длинные последовательности (цепочки) однотипных операций при длинной последовательности данных (например, координаты, выдаваемые радиолокационной станцией), то есть когда имеет место параллелизм объектов и данных.

Системы класса ОКМД.

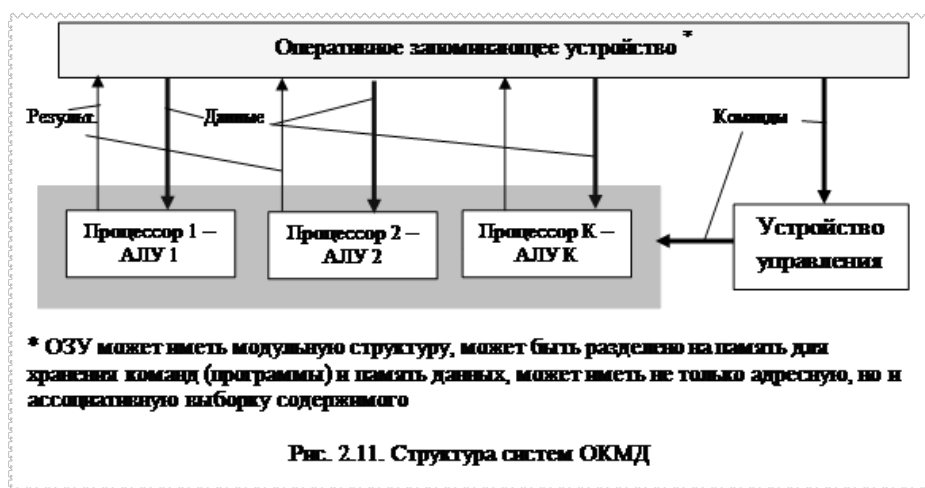
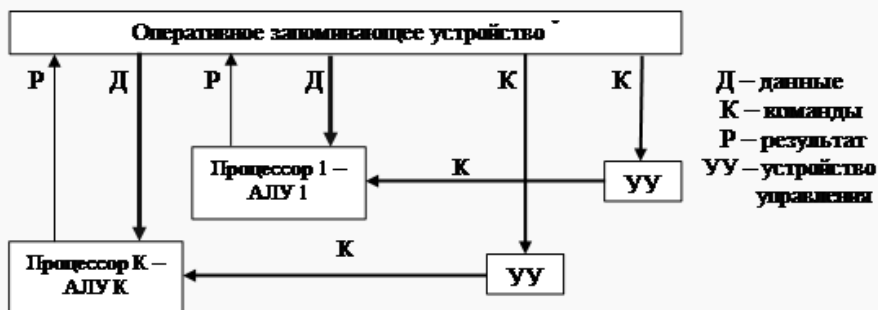


Рис. 2.11. Структура систем ОКМД

Системы, относящиеся к этому классу также как и предыдущие, ориентированы на параллелизм объектов и данных. В таких системах по одной программе обрабатываются сразу несколько потоков данных, каждый из которых обрабатывается своим процессором, но работающим под общим управлением - по одной команде разные процессоры выполняют одну и ту же операцию над различными данными. Схематически такие системы можно иллюстрировать рисунком 2.11. В архитектуре систем, представленных на рисунке управление может быть реализовано и с помощью отдельной ЭВМ, имеющей свою память команд. Используются и другие виды параллельной обработки, например, с помощью матричных или ассоциативных систем.



*** Оперативное запоминающее устройство может иметь модульную структуру, может быть разделено на память для хранения программ (команд) и память данных, может иметь не только адресную выборку, но и ассоциативную (по содержанию памяти)**

Рис. 2.12. Структура систем МКМД

В вычислительных системах этого класса, обработка данных может быть организована либо на многомашинных комплексах, либо на многопроцессорных комплексах.

Объединенные в систему с помощью тех или иных видов связи вычислительные машины, наиболее приспособлены для решения потока независимых задач, но при этом существенно увеличивают производительность вычислительных работ за счет большего количества решаемых задач. Многопроцессорные системы (комплексы) повышают производительность при использовании всех видов параллелизма, но, наибольший эффект достигается на задачах, запрограммированных по типу параллелизма независимых ветвей.

Архитектура систем, показанная на рисунке 2.12, является наиболее универсальной в отношении решаемых задач. Программное обеспечение таких систем не ориентируется на решение определенного класса задач, и поэтому они строятся как универсальные вычислительные комплексы, имеющие наиболее широкое распространение.

В 70 – 80-х годах прошлого столетия, вычислительные центры организовывали обработку данных по приведенной архитектурной схеме, стараясь обеспечить вычислительными мощностями как можно большее число пользователей, работающих в то время в режиме телеобработки данных.

9. Задачи сортировки Анализ сложности и эффективности алгоритмов сортировки

Задачи сортировки

Задачу сортировки следует понимать, как процесс перегруппировки однотипных элементов структуры данных в некотором определенном порядке. Цель сортировки - облегчить последующий поиск, обновление, исключение, включение элементов в структуру данных.

Разработано множество алгоритмов сортировки, однако нет алгоритма, который был бы наилучшим в любом случае.

Эффективность алгоритма сортировки может зависеть от ряда факторов, таких, как: число сортируемых элементов; диапазон и распределение значений сортируемых элементов; степень отсортированности элементов; характеристики алгоритма (сложность, требования к памяти и тп.); место размещения элементов (в оперативной памяти или на ВЗУ).

Сортируемые элементы часто представляют собой записи, данных определенной структуры. Каждая запись имеет поле ключа, по значению которого осуществляется сортировка, и поля данных. При рассмотрении алгоритмов сортировки нас интересует только поле ключа, поэтому другие поля опускаются из рассмотрения.

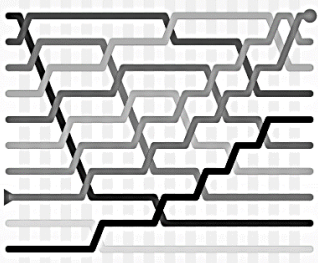
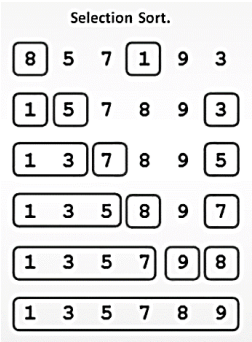
Метод сортировки называется устойчивым, если в процессе сортировки относительное расположение элементов с одинаковыми (равными) ключами не изменяется. Устойчивость сортировки желательна, если речь идет об элементах, уже отсортированных по некоторым другим ключам (свойствам), не влияющим на ключ, по которому сейчас осуществляется сортировка.

Внутренняя и внешняя сортировки

В зависимости от фактора размещения элементов все методы сортировки разбивают на два класса: внутренняя сортировка (сортировка массивов) и внешняя сортировка (сортировка файлов, или сортировка последовательностей).

Внешняя сортировка оперирует с запоминающими устройствами большого объема, но с доступом не произвольным, а последовательным, т. е. в данный момент мы «видим» только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью.

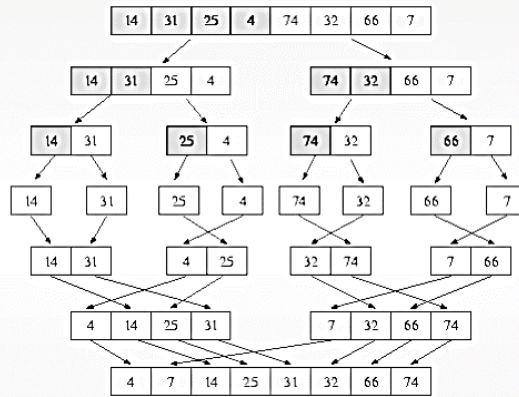
Алгоритмы сортировки

<p>Сортировка пузырьком</p>	<p>Один из простейших методов сортировки. Заключается в постепенном смещении элементов с большим значением в конец массива. Элементы последовательно сравниваются попарно, и если порядок в паре нарушен – меняются местами.</p> <p>Алгоритм проходит по заданному массиву множество раз, каждый раз сравнивая пару соседних друг с другом чисел. Если числа стоят не в том порядке, в котором должны, он меняет их местами.</p> <p>Элементы сортируемого массива «всплывают», как пузырьки воздуха в воде.</p> 
<p>Сортировка выбором</p>	<p>Алгоритм ищет наименьший элемент в текущем списке и производит обмен его значения со значением первой неотсортированной позиции. То же самое происходит со вторым элементом с наименьшим значением. Цикл повторяется до тех пор, пока все элементы не займут нужную последовательность.</p> <p>Стандартный поиск минимума для каждого из n элементов – это квадрат. Сложность: $O(n^2)$</p> <div style="text-align: center;"> <p>Selection Sort.</p>  </div>

<p>Быстрая сортировка</p>	<p>Считается одним из самых быстрых алгоритмов сортировки. Как и сортировка слиянием, работает по принципу «разделяй и властвуй». Временная сложность алгоритма может достигать $O(n \log n)$.</p> <p>Ещё один рекурсивный алгоритм. Сначала проводится грубая оценка массива на основе некоторого элемента, называемого опорным. Все элементы, если сортируем по возрастанию, большие опорного, перекидываются направо от него, а все меньшие – налево. Массив делится на две части, каждая из которых сортируется отдельно.</p> <p> Ч.Э.Хоар</p> <p><i>Идея: выгоднее переставлять элементы, который находится дальше друг от друга.</i></p> 
<p>Сортировка кучей (Пирамидальная сортировка)</p>	<p>Имея построенную пирамиду, несложно реализовать сортировку. Так как корневой узел пирамиды имеет самое большое значение, мы можем отделить его и поместить в конец массива. Вместо корневого узла можно поставить последний узел дерева и, просеив его вниз, снова получить пирамиду. В новом дереве корень имеет самое большое значение среди оставшихся элементов. Его снова можно отделить, и так далее. Алгоритм получается следующий:</p> <ol style="list-style-type: none"> 1. поменять местами значения первого и последнего узлов пирамиды; 2. отделить последний узел от дерева, уменьшив размер дерева на единицу (элемент остаётся в массиве); 3. восстановить пирамиду, просеив вниз её новый корневой элемент; 4. перейти к пункту 1; <p>По мере работы алгоритма, часть массива, занятая деревом, уменьшается, а в конце массива накапливается отсортированный результат.</p> <p>Процесс просеивания:</p>

	<div style="display: flex; justify-content: space-around; align-items: flex-start;"> <div style="text-align: center;"> <p>1)</p> </div> <div style="text-align: center;"> <p>2)</p> </div> <div style="text-align: center;"> <p>3)</p> </div> </div>
Сортировка вставками	<p>Применяется для вставки элементов массива на «свое место». Сортировка вставками представляет собой простой метод сортировки и используется для раскладки колоды во время игры в бридж.</p> <p>Каждый элемент массива вставляется на правильное (относительно остальных) место в новом, отсортированном массиве.</p> <p>Так как кроме обхода всех элементов массива, для каждого элемента нужно найти место в новом массиве и сдвинуть некоторые элементы (после вставки, но до старого расположения элемента), сложность задачи – $O(n^2)$</p>
Сортировка слиянием	<p>Следует принципу «разделяй и властвуй», согласно которому массив данных разделяется на равные части, которые сортируются по-отдельности. После они сливаются, в результате получается отсортированный массив.</p> <p>Сортируемый массив разбивается на две части, каждая из которых сортируется отдельно (возможно, этим же самым алгоритмом), а потом результаты сортировок объединяются в один. Это – рекурсивный алгоритм.</p>

Главный принцип этого алгоритма: массив из одного элемента уже отсортирован.



Анализ сложности и эффективности алгоритмов поиска и сортировки

Критериями оценки эффективности алгоритма сортировки является пространственная и временная сложность.

Пространственная сложность

Означает количество памяти, затраченной на выполнение алгоритма. Пространственная сложность включает вспомогательную память и память для хранения входных данных.

Временная сложность

Означает время, за которое алгоритм выполняет поставленную задачу с учетом входных данных.

В таблице представлена оценка сложности алгоритмов

Алгоритм сортировки	Время работы в худшем случае	Время работы в среднем случае	Время работы в лучшем случае	Пространственная сложность
Сортировка пузырьком	n^2	n^2	n	1
Сортировка выбором	n^2	n^2	n^2	1
Быстрая сортировка	n^2	$n \cdot \log n$	$n \cdot \log n$	$n \cdot \log n$
Сортировка кучей	$n \cdot \log n$	$n \cdot \log n$	$n \cdot \log n$	1
Сортировка вставками	n^2	n^2	n	1
Сортировка слиянием	$n \cdot \log n$	$n \cdot \log n$	$n \cdot \log n$	n

У каждого алгоритма сортировки своя временная и пространственная сложность. Использовать можно любой из

представленных алгоритмов в зависимости от поставленных задач. Лучшим алгоритмом является быстрая сортировка. Она позволяет выбрать опорный элемент и разделяет массив на 3 части: меньше, равно и больше опорного элемента.

10. Современные технологии разработки программного обеспечения Управление версиями Документирование

Экстремальное программирование.

Экстремальное программирование (далее XP) – методология создания ПО, позволяет сделать этот процесс более прогнозируемым, гибким и быстрым, в соответствии с требованиями современного бизнеса. XP основывается на идее адаптации изменений в программном проекте вместе с отказом от детального планирования. XP отходит от традиционного процесса создания программной системы и вместо единовременного планирования, анализа и проектирования с расчетом на долгосрочную перспективу при XP все эти операции реализуются постепенно в ходе разработки, добиваясь тем самым значительного сокращения времени разработки и стоимости изменений в программе. Возникло в первой половине 90-х годов. Автор термина XP Кент Бек – пришел к выводу, что разработку любого программного проекта можно сделать более эффективной, если приложить усилия в четырех основных направлениях: усовершенствовать взаимосвязь разработчиков, упростить проектные решения, усилить обратную связь с заказчиком и проявлять больше активности. Эти четыре направления и стали приоритетными в XP. Основные концепции Экстремального Программирования:

Планирование:

- Пишутся User Stories. Через которые заказчик рассказывает какую программу он хочет получить.
- Собственно план создается в результате Планирования Релиза - определяет даты релизов и формулировки заказчика, которые будут воплощены в каждом из них.
- Выпускать частые небольшие Релизы – Заказчик всегда имеет работающую версию программы с последними реализованными фичами.
- Измеряется Скорость проекта – позволяет понять все ли мы делаем правильно и укладываемся ли в срок.
- Проект делится на Итерации – это позволяет получать отдачу от заказчика и корректировать программу.
- Каждая итерация начинается с собрания по планированию.

- Люди постоянно меняются задачами – знания по проекту распространяются в команде.

- Каждый день начинается с утреннего Собрания стоя.

- XP правила корректируются, если что-то не так – это добавляет гибкость методологии.

- Дизайн:

- Простота. Все фичи реализуются максимально простым способом.

- Писать Пробные решения для уменьшения риска.

- Не добавлять никаких функций раньше времени. Добавлять только ту функциональность которая действительно требуется в данный момент.

- Рефакторить безжалостно - Упрощать написанный код.

Кодирование:

- Заказчик всегда рядом. Заказчик является членом команды и отвечает на вопросы разработчиков.

- Весь код должен соответствовать принятому стандарту.

- Любая строчка программы написана 2 программистами.

- Частая интеграция кода – избавляет от трудностей интеграции модулей.

- Оставлять оптимизацию на потом.

- Тестирование:

- Любой код должен иметь Unit Test. Тесты пишутся до написания кода.

- Все Unit тесты должны проходить перед отдачей.

- Если найден баг то тесты корректируются или создаются.

- Функциональные тесты периодически выполняются и их результаты публикуются.

Плюсы:

- XP строится на том, что создавать простую и понятную программу выгоднее, чем сложную и запутанную.

- В XP тестированию уделяется особое внимание. Тесты разрабатываются до того, как начнется написание программы, во время работы и после того, как кодирование завершено.

- Готовность программистов к постоянным изменениям в проекте. За счет постоянной обратной связи с заказчиком ЭП позволяет вносить изменения именно на той стадии, когда это действительно эффективно.

Минусы:

- Невозможно использовать XP на гигантских проектах — оно

подходит для небольших групп программистов (от 2 до 10 человек).

- Не подходит для распределенных команд, связанных между собой с помощью Интернета.

Ключевые концепции ХР

Ценности это то, что отличает набор индивидуалов от команды. Кент Бек в своей книге "Extreme Programming Explained: Embrace Change" выделил основные ценности ХР:

Общение. Довольно часто в проектах возникают проблемы, если кто-то не сказал кому-то что-то с некоторой точки зрения важное. ХР делает практически невозможным не общаться.

Простота. ХР предлагает в процессе написания кода всегда делать самую простую вещь, которая смогла бы работать. Бек описывает это так: "ХР делает ставку на то, что лучше сегодня сделать простую вещь ... чем более сложную, но которая все равно никогда не будет использоваться".

Обратная связь. Постоянная обратная связь с заказчиком, группой или реальными конечными пользователями дает Вам больше возможностей регулировать вашу работу. Обратная связь позволяет Вам придерживаться правильного пути.

Смелость. Смелость подразумевается в контексте трех остальных положений, которые поддерживают друг друга.

-Требуется смелость предположить, что постоянная обратная связь лучше, чем попытка знать все с самого начала.

-Требуется смелость общаться с другими членами группы, что может продемонстрировать часть вашего собственного незнания.

-Требуется смелость сохранить простоту системы, откладывая завтрашние решения на завтра.

Без простой системы постоянного обмена знаниями и обратной связи для исправления ошибок, трудно быть смелым.

12 практик ХР

Практики ХР действуют в совокупности поэтому изучение одной из них влечет за собой понимание и изучение других. После знакомства с этими принципами, станут понятны приемы, используемые в методике экстремального программирования

- 1) Игра планирования ХР признает, в самом начале Вы не можете знать абсолютно все. Главная идея этого принципа состоит в том, чтобы быстро сделать приблизительную схему и дорабатывать ее, как только все становится более понятным.

- 2) Программирование в парах.

- В ХР весь программный код пишут пары разработчиков, что предоставляет много экономических и прочих выгод:

- Все проектные решения принимаются, по крайней мере, двумя мозгами.

- Как минимум, два человека знакомы с каждой частью системы.

- Имеется меньшее количество шансов, что оба человека станут пренебрегать тестированием или другими задачами.

- Замена в парах распространяет знания внутри группы.

- Код всегда проверяется, по крайней мере, одним человеком.

- Исследования также показывают, что программирование в парах является действительно более эффективным, чем одиночное программирование.

3) Тестирование. Есть два вида тестов в XP: функциональные тесты и тесты модулей. Функциональные тесты (приемочные тесты) пишутся на основе директивы заказчика. Они рассматривают систему как черный ящик. Заказчик ответственен за проверку корректности функциональных тестов.

Никакой код не может быть выпущен без Unit теста. Перед отдачей кода разработчик должен удостовериться что все тесты проходят без ошибок. Никто не может отдать код, если все не прошли 100%.

4) Рефакторинг (разложение программы на элементарные операции) - это методика улучшения кода без изменения его функциональных возможностей. XP-группа постоянно занимается рефакторингом.

5) Простой дизайн. XP выбирает самое простое решение.

Самый простой дизайн, который сможет работать это такой дизайн, который (по Кенту Беку):

- Выполняет все тесты;

- Не содержит дублирующийся код;

- Ясно отражает цели программистов для всего кода;

- Содержит наименьшее количество возможных классов и методов.

6) Коллективное владение кодом стимулирует разработчиков подавать идеи для всех частей проекта, а не только для своих модулей. Любой разработчик может изменять любой код для расширения функциональности, исправления ошибок или рефакторинга.

7) Непрерывное интегрирование кода помогает избежать кошмаров интегрирования. XP-группы интегрируют свой код несколько раз в день, после того, как они выполнили все тестирования модулей.

8) Доступный для связи заказчик. Чтобы оптимально функционировать, ХР-группа должна иметь заказчика, расположенного недалеко, чтобы разъяснять директивы и принимать важные деловые решения.

9) Частые Релизы. Разработчики должны выпускать версии системы пользователям (или бета-тестерам) как можно чаще.

10) 40-часовая рабочая неделя. Постоянное напряжение и интенсификация труда быстро истощает силы разработчиков, что заметно снижает эффективность труда.

11) Стандарт кодирования предохраняет группу от отвлечения на несущественные параметры тех вещей, которые не имеют такого значения, как продвижение с максимальной скоростью.

12) Метафора системы в ХР аналогична тому, что большинство методологий называет архитектурой. Метафора дает группе непротиворечивое изображение, которое они могут использовать, чтобы описать, как работает существующая система, где нужны новые части и какую форму они должны принять.

RUP

Статическую структуру RUP составляют описания работ и задач (части работы), описания создаваемых артефактов, а также рекомендации по их выполнению, которые группируются в дисциплины: шесть основных — бизнес-моделирование (Business Modeling), управление требованиями (Requirements), анализ и проектирование (Analysis and Design), реализация (Implementation), тестирование (Test), внедрение (Deployment), и три вспомогательных — управление конфигурациями и изменениями (Configuration and Change Management), управление проектом (Project Management), поддержка среды разработки (Environment).

Динамическую структуру процесса составляют фазы и итерации. Проект, как правило, делится на четыре фазы: начало (Inception), проработка (Elaboration), построение (Construction) и передача (Transition). Фазы, в свою очередь, делятся на итерации. В ходе каждой итерации выполняются работы и задачи из различных дисциплин; соотношение этих работ меняется в зависимости от фазы.

Работы и задачи в RUP привязаны к стандартному набору ролей участников процесса. Роли объединяют более узкие группы работ и задач, которые могут выполняться одним человеком с узкой специализацией. Как правило, реальный исполнитель выполняет одну или несколько ролей в соответствии со своей квалификацией. Скажем, менеджер проекта может выполнять также обязанности архитектора. Одну роль в рамках проекта могут выполнять и несколько человек.

Например, в проекте, как правило, участвует несколько разработчиков.

Создатели RUP определяют его как итеративный, архитектурно-ориентированный и управляемый прецедентами использования процесс разработки программного обеспечения. Согласно последней доступной автору версии RUP (Version 2003.06.00.65) к этому надо добавить, что RUP использует лучшие практические методы (итеративная разработка, управление требованиями, использование компонентной архитектуры, визуальное моделирование, непрерывный контроль качества, управление изменениями) и десять элементов, представляющих квинтэссенцию RUP (разработка концепции; управление по плану; снижение рисков и отслеживание их последствий; тщательная проверка экономического обоснования; использование компонентной архитектуры; прототипирование, инкрементное создание и тестирование продукта; регулярные оценки результатов; управление изменениями; создание продукта, пригодного к употреблению; адаптация RUP под нужды своего проекта).

Пользоваться таким объемным определением, конечно, неудобно. Поэтому для характеристики RUP Пером Кроллом введено понятие «Дух RUP». Хотя оно не входит в «канонический» текст RUP, но предложено человеком, который, являясь директором соответствующего направления в компании IBM, связан с RUP самым непосредственным образом.

Дух RUP заключен в восьми принципах:

- атаковать риски как можно раньше, пока они сами не перешли в атаку;
- разрабатывать именно то, что нужно заказчику;
- главное внимание — исполняемой программе;
- приспосабливаться к изменениям с самого начала проекта;
- создавать архитектурный каркас как можно раньше;
- разрабатывать систему из компонентов;
- работать как одна команда;
- сделать качество стилем жизни.

Эти принципы весьма полно характеризуют RUP и в наибольшей степени соответствуют современному стилю разработки программного обеспечения.

Особенностью RUP является то, что в результате работы над проектом создаются и совершенствуются модели. Вместо создания громадного количества бумажных документов, RUP опирается на разработку и развитие семантически обогащенных моделей, всесторонне представляющих разрабатываемую систему. RUP — это

руководство по тому, как эффективно использовать UML. Стандартный язык моделирования, используемый всеми членами группы, делает понятными для всех описания требований, проектирование и архитектуру системы.

RUP поддерживается инструментальными средствами, которые автоматизируют большие разделы процесса. Они используются для создания и совершенствования различных промежуточных продуктов на различных этапах процесса создания ПО, например, при визуальном моделировании, программировании, тестировании и т. д.

RUP — это конфигурируемый процесс, поскольку, вполне понятно, что невозможно создать единого руководства на все случаи разработки ПО. RUP пригоден как для маленьких групп разработчиков, так и для больших организаций, занимающихся созданием ПО. В основе RUP лежит простая и понятная архитектура процесса, которая обеспечивает общность для целого семейства процессов. Более того, RUP может конфигурироваться для учёта различных ситуаций. В его состав входит Development Kit, который обеспечивает поддержку процесса конфигурирования под нужды конкретных организаций.

RUP описывает, как эффективно применять коммерчески обоснованные и практически опробованные подходы к разработке ПО для коллективов разработчиков, где каждый из членов получает преимущества от использования передового опыта в:

- итерационной разработке ПО,
- управлении требованиями,
- использовании компонентной архитектуры,
- визуальном моделировании,
- тестировании качества ПО,
- контроле за изменениями в ПО.

RUP организует работу над проектом в терминах последовательности действий (workflows), продуктов деятельности, исполнителей и других статических аспектов процесса с одной стороны, и в терминах циклов, фаз, итераций и временных отметок завершения определенных этапов в создании ПО (milestones), т. е. в терминах динамических аспектов процесса, с другой.

При итерационном подходе, каждая из фаз процесса разработки состоит из нескольких итераций, целью которых является последовательное осмысление стоящих проблем, наращивание эффективных решений и снижение риска потенциальных ошибок в проекте. В то же время, каждая из последовательностей действий по созданию ПО выполняется в течение нескольких фаз, проходя пики и

спады активности.

Каждый цикл итерации проекта начинается с планирования того, что должно быть выполнено. Результатом выполнения должен быть значимый продукт. Заканчивается же цикл оценкой того, что было сделано и были ли цели достигнуты.

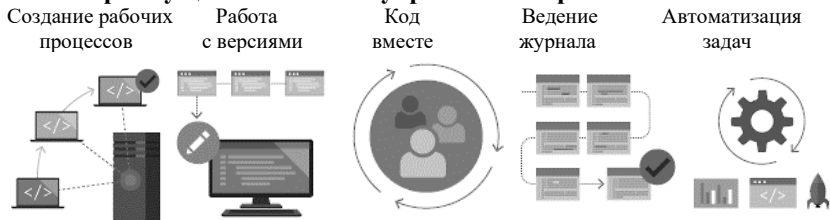
RUP достаточно обширен. Это набор рекомендаций и примеров по всем стадиям и фазам разработки программ. Хотя в основу этих рекомендаций положен многолетний опыт разработки программных систем, не для каждого проекта RUP подходит на сто процентов. Каждый программный проект по-своему уникален. Нельзя бездумно копировать чужой проект, создавая артефакты, имеющие незначительную ценность. Во многих небольших организациях по разработке программного обеспечения, особенно в тех, которые не имеют собственной мощной системы разработки, RUP можно использовать «как есть» или в готовом виде. Также для максимального его приближения к нуждам, требованиям, характеристикам и ограничениям организации-разработчика процесс может быть уточнен, расширен и специфически настроен

Управление версиями, организация коллектива разработчиков, документирование

Системы управления версиями — это программное обеспечение, помогающее отслеживанию изменений, внесенных в код с течением времени. Когда разработчик редактирует код, система управления версиями создает моментальный снимок файлов. Затем этот моментальный снимок сохраняется, чтобы при необходимости им было можно воспользоваться позже.

Без контроля версий разработчики могут удерживать на своем компьютере несколько копий кода. Это опасно, так как можно легко изменить или удалить файл в неправильной копии кода, что может привести к потере работы. Системы управления версиями решают эту проблему, управляя всеми версиями кода, но выполнив рабочую группу с одной версией за раз.

Преимущества системы управления версиями



Рабочие процессы управления версиями предотвращают Chaos всех пользователей, использующих собственный процесс разработки с различными и несовместимыми средствами. Системы управления версиями обеспечивают принудительную обработку и разрешения, чтобы все пользователи оставались на одной странице.

Каждая версия имеет описание того, что делают изменения в версии, например исправление ошибки или добавление функции. Эти описания помогают команде отслеживать изменения в коде по версии, а не по отдельным изменениям файла. Код, хранящийся в версиях, можно в любой момент при необходимости просмотреть и восстановить из системы управления версиями. Это позволяет легко создать новую работу с любой версией кода.

Управление версиями синхронизирует версии и гарантирует, что изменения не конфликтуют с другими изменениями других. Группа использует систему управления версиями для устранения конфликтов, даже если пользователи делают изменения одновременно.

Управление версиями сохраняет историю изменений по мере того, как команда сохраняет новые версии кода. Этот журнал позволяет узнать, кто, зачем и когда внес изменения. Журнал дает группам уверенность в экспериментах, так как в любое время можно легко выполнить откат до предыдущей хорошей версии. Журнал позволяет любому пользователю выполнять базовые операции из любой версии кода, например для исправления ошибки в предыдущем выпуске.

Функции автоматизации системы управления версиями сохраняют время и создают противоречивые результаты. Автоматизируйте тестирование, анализ кода и развертывание, сохраняя новые версии в системе управления версиями.

Документирование проекта

Документирование — это написание и поддержание документов, сопровождающих программное обеспечение. Существует следующие основные типы документации на ПО:

- требования — документация свойств, возможностей, характеристик и качеств системы;
- архитектурная/проектная — обзор программного обеспечения, включающий описание рабочей среды и принципов, которые должны быть использованы при создании ПО;
- техническая - документация на код, алгоритмы, интерфейсы, API;
- пользовательская — руководства для конечных пользователей, администраторов системы и другого персонала;

- маркетинговая — документация для маркетинга продукта и анализа рыночного спроса, рекламные материалы.

Стандарты документации

Стандарты обеспечивают совместимость между проектами. Это означает, что идеи или артефакты, разработанные для одного случая, могут быть перенесены и на другой. Стандарты улучшают понимание среди инженеров. Наиболее крупные компании создали стандарты разработки программного обеспечения. Некоторые заказчики, такие как Министерство обороны США, настаивают, чтобы подрядчики следовали их стандартам.

Практика многих компаний показывает, что просто издание и распространение стандартов не приводит к их принятию. Для того чтобы быть эффективными, стандарты должны восприниматься инженерами как нечто полезное для них, а не как набор препятствий. Кроме того, четкие и измеримые цели, требующие дисциплинированного и документированного подхода, обычно являются хорошим мотивом для разработчиков.

Хэмфри предлагает командам коллективно решать, какой из стандартов ведения документации им применять. Преимуществом является то, что компания при этом перебирает различные подходы, а в результате этого процесса, могут быть приняты наиболее выгодные варианты на долгое время работы.

Недостатком самостоятельного выбора стандарта командами является то, что группы, работающие в одной компании, зачастую выбирают разные стандарты. Это уменьшает возможности сравнения проектов и требует от инженеров, переключающихся на другой проект, изучения нового стандарта документации.

Организации могут допускать некоторую гибкость и автономность в вопросе создания документации, но при этом рассчитывать на получение определенной стандартной информации для улучшения всего процесса производства. Улучшение процесса включает в себя эволюционный мета-процесс (процесс, имеющий дело с другими процессами) внутри организации. Одним из примеров является модель зрелости возможностей (CMM), которая классифицирует организации, занимающиеся разработкой программного обеспечения, по пяти категориям возрастающих возможностей.

Перечисленные ниже организации публикуют важные стандарты. Не все стандарты могут быть абсолютно актуальны, так как совещания, требуемые для их создания, проходят намного медленнее, чем появление новых технологий на рынке.

- Институт инженеров по электротехнике и радиоэлектронике (IEEE) в течение многих лет остается очень активным в создании стандартов документации программного обеспечения. Большинство стандартов разработаны различными комитетами, состоящими из опытных и ответственных инженеров-профессионалов. Некоторые из стандартов IEEE стали также стандартами ANSI.

- Международная организация по стандартизации (ISO) имеет огромное влияние во всем мире, особенно среди организаций-производителей, имеющих дело с Евросоюзом (ЕС). ЕС предписывает следование стандартам ISO любой компании, имеющей дело со странами-членами Евросоюза, что является мощным стимулом для поддержания этих стандартов странами всего мира.

- Институт технологий разработки программного обеспечения (SEI) был учрежден Министерством обороны США в университете Карнеги-Меллон для поднятия уровня технологии программного обеспечения у подрядчиков Министерства обороны. Работа SEI также была принята многими коммерческими компаниями, которые считают улучшение процесса разработки программного обеспечения своей стратегической корпоративной задачей.

- Консорциум по технологии манипулирования объектами (OMG) является некоммерческой организацией, в которую в качестве членов входят около 700 компаний. OMG устанавливает стандарты для распределенных объектно-ориентированных вычислений. В частности, OMG использует унифицированный язык моделирования UML в качестве своего стандарта для описания проектов.

Документы, сопровождающие проект, сильно различаются среди организаций, но примерно соответствуют водопадным фазам. Стандарт ISO 12207 является одним из примеров такого набора документов.

Когда стандарты ведения документации не применяются, инженерам приходится затрачивать огромное количество времени, самостоятельно приводя документы в порядок. Типичный набор документации с использованием терминологии IEEE показан на рис.



Использование набора документации для водопадного процесса не означает, что обязательно должна использоваться водопадная модель. Однако если она не используется, то необходимо провести обновление всех документов и пополнять их каждый раз, когда применяются водопадные фазы. Это означает, что документы должны быть очень хорошо организованы.

Ниже приводится описание каждого документа из набора IEEE. Другие стандарты внутренне организованы по тому же принципу.

- SVVP (Software Verification and Validation Plan): план экспертизы программного обеспечения. Этот план определяет, каким образом и в какой последовательности должны проверяться стадии проекта, а также сам продукт на соответствие поставленным требованиям. Верификация — это процесс проверки правильности сборки приложения; валидация проверяет тот факт, что собран требуемый продукт. Зачастую валидацию и верификацию осуществляют сторонние организации, в этом случае экспертиза называется независимой (IV&V — Independent V&V).

- SQAP (Software Quality Assurance Plan): план контроля качества программного обеспечения. Этот план определяет, каким

образом проект должен достигнуть соответствия установленному уровню качества.

- SCMP (Software Configuration Management Plan): план управления конфигурациями программного обеспечения. SCMP определяет, как и где должны храниться документы, программный код и их версии, а также устанавливает их взаимное соответствие. Было бы крайне неразумным начинать работу без такого плана, так как самый первый созданный документ обречен на изменения, но необходимо знать, как управлять этими изменениями до того, как будет начато составление документа. Средние и большие компании, как правило, стараются выработать единое управление конфигурациями для всех своих проектов. Таким образом, инженерам требуется только научиться следовать предписанным процедурам в соответствии с SCMP.

- SPMP (Software Project Management Plan): план управления программным проектом. Этот план определяет, каким образом управлять проектом. Обычно он соответствует известному процессу разработки, например стандартному процессу компании.

- SRS (Software Requirements Specification): спецификация требований к программному обеспечению. Этот документ определяет требования к приложению и является подобием контракта и руководства для заказчика и разработчиков.

- SDD (Software Design Document): проектная документация программного обеспечения. SDD представляет архитектуру и детали проектирования приложения, обычно с использованием диаграмм объектных моделей и потоков данных.

- STD (Software Test Documentation): документация по тестированию программного обеспечения. Этот документ описывает, каким образом должно проводиться тестирование приложения и его компонентов.

Иногда в проектах привлекается дополнительная документация. Документация для итеративной разработки может быть организована двумя способами. Некоторые документы, в частности SDD, могут содержать свою версию для каждой итерации. Другой способ — дописывать дополнения, которые появляются по мере развития приложения.

Унифицированный язык моделирования (UML) был разработан для стандартизации описания программных проектов, в особенности объектно-ориентированных. UML был принят в качестве стандарта консорциумом OMG.