Лекция 4. Объектно-ориентированное проектирование, Этапы проектирования ИС.

Аннотация: Объектно-ориентированное проектирование. Этапы проектирования ИС.

1. Введение

От структур и подпрограмм к объектам.

Целями **структурного программирования** являлись структуризация данных и декомпозиция кода. Объединение данных в структуры позволяло, с одной стороны, более естественно описывать сущности реального мира, имеющие самые разнотипные наборы атрибутов. А, с другой стороны, делало процесс разработки программ более простым, поскольку однородные данные были сгруппированы в одном месте и под одним общим именем.

Декомпозиция кода заключалась в разделении исходного кода программы на отдельные подпрограммы. Подпрограммы снижали количество возможных связей между отдельными операторами (локализация кода) и, кроме того, позволяли устанавливать необходимую область видимости для переменных, используемых в подпрограммах (локализация данных). Локализация кода не позволяла произвольно переходить от одного оператора в одной подпрограмме к любому оператору в другой подпрограмме. Вызов подпрограммы приводил к передаче управления только в определенную точку входа, хотя некоторые языки допускали более одной точки входа в подпрограмму.

Такое решение не только повышало надежность программ, но и позволяло многократно использовать одни и те же подпрограммы в различных программах. Как следствие, стали появляться библиотеки подпрограмм, как самостоятельные программные продукты. Часть библиотек поставлялась вместе с компиляторами с языков программирования и операционными системами, другая часть распространялась свободно или на коммерческой основе. Применение библиотек существенно ускоряло процесс разработки программного обеспечения, так как подпрограммы, применяемые многократно, требовалось отлаживать только единожды.

Локализация данных имела ничуть не меньшее значение, чем локализация кода. Три вида данных в подпрограмме:

- Локальные данные подпрограммы, которые автоматически создавались при ее вызове и уничтожались при возврате из подпрограммы;
- Локальные данные, сохраняемые между несколькими вызовами одной подпрограммы;
- Глобальные данные, доступные нескольким или всем подпрограммам, и доступ к которым осуществлялся прямо из подпрограммы;
- Фактические параметры, передаваемые подпрограммам, замещающие объявленные формальные параметры.

Глобальные данные применялись в основном для организации связей между подпрограммами, но они существенно ограничивали гибкость подпрограмм. Действительно, если подпрограмма работала с какими-то глобальными данными, то эти данные должны были совпадать по имени, типу во всех программах, которые использовали данную подпрограмму.

В отличие от этого, локальные данные, наоборот, позволяли подпрограмме быть независимой от программ, использующих ее. Но они не обеспечивали передачу данных между различными подпрограммами. Для передачи данных или ссылок на них использовался механизм формальных/фактических параметров. При описании подпрограммы объявляются формальные параметры, которые она принимает при вызове. Когда реально происходит вызов подпрограммы, на место формальных параметров подставляются фактические параметры того же типа.

Образование библиотек подпрограмм происходило, как правило, по функциональному признаку. То есть, в одну библиотеку помещались подпрограммы, выполняющие близкую по составу работу. Например, библиотека подпрограмм, работающих с каким-то устройством. Библиотеки имели существенный недостаток. Результаты их работы могли быть неверны при подаче на вход не корректных данных. Однако данные располагались в программе за пределами библиотеки и могли меняться произвольно. Появилась реальная потребность объединить данные и код, который их обрабатывает. Эта задача была решена в **модульном программировании.** Модуль, в отличие от библиотеки подпрограмм, мог содержать не только подпрограммы, но и данные.

Данные, располагаемые в модуле, были глобальными и для модуля и для программы, использующей модуль. Для увеличения безопасности данных модуль разделили на несколько зон. Одна из зон обеспечивала взаимодействие между модулем и внешним программные обеспечением. Это так называемая зона интерфейса. Другая зона отвечала за реализацию модуля и была недоступна для внешнего программного обеспечения.

Соответственно, критичные к произвольным изменениям данные помещались в зону реализации и были недоступны для программы и других модулей. Про эти данные можно сказать, что они инкапсулированы модулем.

В интерфейсной зоне оставались, как правило, только те данные, с которыми активно взаимодействовало внешнее программное обеспечение. Нельзя сказать, что эти данные можно было произвольно изменять без ущерба для работоспособности программы. Практически все данные были инкапсулированы, то есть помещены в зону реализации, а для взаимодействия с ними в интерфейсную зону помещались объявления специальных интерфейсных программ. Только они могли изменять значения переменных внутри модуля или возвращать значения этих переменных.

Перенос данных в область реализации поставил еще одну проблему: данные необходимо было инициализировать перед началом работы и освобождать перед окончанием работы модуля. Например, перед началом работы модуля могла потребоваться динамически распределяемая область памяти, а в конце требовалось вернуть эту область памяти системе. Это привело к появлению в модулях областей инициализации, которые вызывались до начала работы модуля, и областей, которые автоматически вызывались перед тем, как модуль завершит работу.

Если теперь сложить все элементы полученной мозаики воедино, то получится знакомая картина: все подпрограммы модуля собраны вместе по функциональному признаку и работают над одними данными; модуль имеет области кода, которые вызываются при инициализации и окончании работы модуля. Такой модуль можно назвать прообразом **объекта.**

Мы вплотную подошли к возможности реализации объектной технологии. Фактически осталось только ввести поддержку полиморфизма и наследования, что сегодня, как правило, реализуется в рамках компиляторов. Важно понимать, что объектная технология, являясь прямым продолжением технологии структурной разработки программ, открыла новые принципиально иные подходы к проектированию сложного программного обеспечения.

Объектно-ориентированное программирование

Абстракция

Абстрагирование — это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Соответственно, абстракция — это набор всех таких характеристик.

Инкапсуляция

Инкапсуляция — это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

Класс

Класс является описываемой на языке терминологии (пространства имён) исходного кода моделью ещё не существующей сущности (объекта). Фактически он описывает устройство объекта, являясь своего рода чертежом. Говорят, что объект — это экземпляр класса. При этом в некоторых исполняющих системах класс также может представляться некоторым объектом при выполнении программы посредством динамической идентификации типа данных. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области.

Наследование

Наследование — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом.

Объект

Сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса или копирования прототипа (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

Полиморфизм

Полиморфизм — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Прототип

Прототип — это объект-образец, по образу и подобию которого создаются другие объекты. Объекты-копии могут сохранять связь с родительским объектом, автоматически наследуя изменения в прототипе; эта особенность определяется в рамках конкретного языка.

Аспектно-ориентированное программирование

Аспектно-ориентированное программи́рование (АОП) — парадигма программирования, основанная на идее разделения функциональности для улучшения разбиения программы на модули.

Основные понятия АОП:

Аспект — модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определённых некоторым срезом.

Совет — средство оформления кода, который должен быть вызван из точки соединения. Совет может быть выполнен до, после или вместо точки соединения.

Точка соединения — точка в выполняемой программе, где следует применить совет. Многие реализации АОП позволяют использовать вызовы методов и обращения к полям объекта в качестве точек соединения.

Срез — набор точек соединения. Срез определяет, подходит ли данная точка соединения к данному совету. Самые удобные реализации АОП используют для определения срезов синтаксис основного языка (например, в AspectJ применяются Java-сигнатуры) и позволяют их повторное использование с помощью переименования и комбинирования.

Внедрение — изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код. Обычно реализуется с помощью некоторого метаобъектного протокола (англ. metaobject protocol, MOP).

Концептуальные положения объектных моделей

При рассмотрении объектных моделей обычно выделяют **три основных концептуальных положения: инкапсуляция, полиморфизм и наследование**. Описание концептуальных положений создает представление о том, какие возможности заложены в данную технологию, на какие горизонты эта технология может вывести процесс разработки программного обеспечения.

Сложные системы.

Объектная технология открывает возможность построения гораздо более сложных систем и программных комплексов, чем допускала технология структурного программирования. Разница в подходах к разработке принципиальна и она базируется именно на высокой сложности современного программного обеспечения. Как результат применение структурной технологии ведет к слишком продолжительному циклу разработки, сложности в модернизации и управлении, сокращенному жизненному циклу и повышенной стоимости.

Гради Буч в своей книге «Объектно-ориентированный анализ и проектирование с примерами приложений на C++» со ссылкой на Брукса отмечает, что сложность вызывается четырьмя основными причинами:

1. сложностью реальной предметной области,
2. трудностью управления процессом разработки,
3. необходимостью обеспечить достаточную гибкость программы,
4. неудовлетворительными способами описания поведения больших дискретных систем.

Далее Буч выделяет пять признаков сложных систем:

- Сложные системы часто являются иерархичными и состоят из связанных подсистем.
- Выбор, какие компоненты в данной системе считаются элементарными, как правило на усмотрение исследователя.
- Внутрикомпонентная связь обычно сильнее, чем связь между компонентами.
- Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.

- Любая работающая сложная система является результатом развития работавшей более простой системы.

Сложная система, как было отмечено ранее, отличается сложностью управления, то есть сложностью связей между компонентами. Каждый уровень управления обладает собственной логикой, которая не зависит от логики управления других уровней. Взаимодействие между уровнями управления в любой сложной системе основано на декларированном интерфейсе каждого уровня. Создание интерфейсов и связей на каждом уровне управления представляет собой отдельную задачу и требует отдельных проектных решений. Создание и развитие каждого уровня может и должно вестись параллельно с созданием и развитием других уровней. Развитие любого уровня начинается с декларации его интерфейса и только после определения интерфейса и увязки его с интерфейсами смежных уровней управления, можно переходить к реализации всех уровней. Интерфейс представляет собой формальную спецификацию возможностей данного логического уровня. Этот тезис переформулирует положение о трудностях, связанных с процессом разработки. Трудность разработки зависит от того, насколько точно определены основные логические уровни и того, насколько хорошо формализованы интерфейсы всех уровней.

Умение выполнять декомпозицию системы адекватно умению превращать сложную систему в простую. Так как система находится в постоянном развитии, здесь не применимы методы, используемые для разработки программы как законченного продукта. Развитие системы должно происходить непрерывно от экспериментальной модели до моделей, находящихся в эксплуатации.

2. Принципы ООП

2.1. Класс, объект

Любая сущность — объект

В настоящем объектно-ориентированном языке все элементы так называемой *предметной области* (problem domain) выражаются через концепцию *объектов.* Как вы уже, наверное, поняли, объекты — это центральная идея объектно-ориентированного программирования. Многие из нас, обдумывая какую-то проблему, вряд ли оперируют понятиями "структура", "пакет данных", "вызов функций" и "указатели", ведь привычнее применять понятие "объекты". Возьмем такой пример.

Допустим, вы создаете приложение для выписки счета-фактуры, в котором нужно подсчитать сумму по всем позициям. Какая из двух формулировок понятней с точки зрения пользователя?

- **Не объектно-ориентированный подход** Заголовок счета-фактуры представляет структуру данных, к которой я получу доступ. В эту структуру войдет также дважды связанный список структур, содержащих описание и стоимость каждой позиции. Поэтому для получения общего итога по счету мне потребуется объявить переменную с именем наподобие *totallnvoiceAmount* и инициализировать ее нулем, получить указатель на головную структуру счета, получить указатель на начало связанного списка, а затем "пробежать" по всему этому списку. Просматривая структуру для каждой позиции, я буду брать оттуда переменную-член, где находится итог для данной позиции, и прибавлять его к *totallnvoiceAmount.*
- **Объектно-ориентированный подход** У меня будет объект "счет-фактура", и ему я отправлю сообщение с запросом на получение общей суммы. Мне не важно, как информация хранится внутри объекта, как это было в предыдущем случае. Я

общаюсь с объектом естественным образом, запрашивая у него информацию посредством сообщений. (Группа сообщений, которую объект в состоянии обработать, называется *интерфейсом* объекта. Чуть ниже я объясню, почему в объектно-ориентированном подходе вместо термина "реализация" правильней употреблять термин "интерфейс".)

Очевидно, что объектно-ориентированный подход естественнее и ближе к тому способу рассуждений, которым многие из нас руководствуются при решении задач. Во втором варианте объект "счет-фактура", наверно, просматривает в цикле *совокупность*(collection) объектов, представляющих данные по каждой позиции, посылая им запросы на получение суммы по данной позиции. Но если требуется получить только общий итог, то *вам все равно, как это реализовано,* так как одним из основных принципов объектно-ориентированного программирования является *инкапсуляция* (encapsulation). Инкапсуляция — это свойство объекта скрывать свои внутренние данные и методы, представляя наружу только интерфейс, через который осуществляется программный доступ к самым важным элементам объекта. Как объект выполняет задачу, не имеет значения, главное, чтобы он справлялся со своей работой. Имея в своем распоряжении интерфейс объекта, вы заставляете объект выполнять нужную вам работу. (Ниже я остановлюсь на понятиях "инкапсуляция" и "интерфейс".) Здесь важно отметить, что разработка и написание программ моделирования реальных объектов предметной области облегчается тем, что представить поведение таких объектов довольно просто.

Заметьте: во втором подходе от объекта требовалось, чтобы он произвел нужную вам работу, т. е. подсчитал общий итог. В отличие от структуры, в объект по определению входят не только данные, но и методы их обработки. Это значит, что при работе с некоторой проблемной областью можно не только создать нужные структуры данных, но и решить, какие методы связать с данным объектом, чтобы объект стал полностью инкапсулированной частью функциональности системы.

2.2. Инкапсуляция

Инкапсуляция подразумевает объединение и защиту кода и данных в некоторой сущности: объекте или модуле. Для чего вообще нужна инкапсуляция? Для того, чтобы защитить реализацию класса. Если не обеспечивается должная защита, то разработчик, использующий данный класс, имеет возможность рассчитывать на определенную реализацию. В этом случае теряется возможность простой модификации не только данного класса, но и всей системы в целом. Предположим, что существует возможность обращаться к любому оператору подпрограммы из внешнего программного обеспечения. Это нарушает инкапсуляцию кода программы, но одновременно утрачивается и возможность модификации самой подпрограммы. Нельзя изменить реализацию по той причине, что какое-то внешнее ПО могло обращаться к одному из изменяемых операторов.

Кроме того, инкапсуляция прямо связана с требованием повышения надежности систем, иначе объекты могут произвольно менять данные других объектов.

Третьим достоинством инкапсуляции является упрощенное восприятие объекта его пользователем. Пользователь видит только те свойства объекта, которые сможет использовать, абстрагируясь от сложности реализации.

2.3. Наследование

Наследование это передача подклассу свойств, структуры и поведения суперкласса. Однако передача свойств и структуры это механизм, а не цель. Реальных целей несколько. К ним относится, например, упорядочивание типов сущностей, то есть классификация сущностей в системе. Действительно, не так просто разобраться в ньюансах, отличающих один тип сущности от другого типа, если не представлять иерархию их развития, ее идеологическую основу.

Благодаря классификации становится возможным введение и широкое практическое использование абстракции. Абстракция дает возможность исключить из рассмотрения малозначительные детали и сосредоточиться на том, что действительно важно на данном уровне рассмотрения. Как правило, абстракция используется для определения интерфейса, который позже реализуется и детализируется в подклассах. Например, можно реализовать абстрактное предприятие и определить необходимый интерфейс, который позволит взаимодействовать предприятию с внешним миром: другими предприятиями, государственными учреждениями и т.п. При этом неважно, чем будет заниматься это предприятие.

Абстракцию также полезно использовать для задания поведения класса. Здесь под поведением понимается последовательность действий по обработке какого-то сообщения (или реакция класса на некоторое воздействие). Подклассы наследуют реализацию данного алгоритма, что приводит не только к значительной экономии кода за счет многократного использования, но и позволяет при необходимости менять поведение множества подклассов простой заменой алгоритма суперкласса.

Итак, наследование позволяет классифицировать используемые программные сущности, вводить высокоуровневые абстракции, передавать свойства и структуру суперкласса своим подклассам, что позволяет существенно экономить код. Наконец, наследование определяет направление развития классов, что существенно облегчает разработку иерархии классов, отражающей частный, но важный случай взаимосвязи между сущностями реальной предметной области.

Полиморфизм подразумевает много форм реализации. Полиморфизм имеет большое значение для объектной технологии, так как он позволяет устанавливать аналогии между методами различных классов.

Например, для классов фигур можно использовать методы Скрыть(Hide), Отобразить (Draw). Несмотря на то, что все фигуры различны, данный шаблон поведения будет корректно работать для любой из этих фигур благодаря полиморфизму.

Обычно полиморфизм рассматривают как составную часть механизма наследования. Считается, что полиморфными могут быть только свойства (методы) подклассов. Но это узкая трактовка. Например, летать могут птицы, насекомые, механические аппараты. Означает ли это, что они имеют общего летающего предка? Конечно, нет.

Основное достоинство полиморфизма заключается в том, что он позволяет различным классам иметь семантически однородные свойства. Благодаря этому становится возможным однотипно взаимодействовать с различными классами.

Полиморфизм реализуется через наследование и интерфейсы.

3. Критика ООП

- Более низкое быстродействие.
- Не панацея от всех бед – сложность предметной области остается.

- Лучше моделирует сущности, хуже моделирует процессы, отсюда распространение функционального программирования.

4. Этапы проектирования ИС

К проектированию ИС непосредственное отношение имеют два направления деятельности:

1) проектирование ИС конкретных организаций на базе готовых программных и аппаратных компонентов;

2) проектирование компонентов ИС и инструментальных средств, ориентированных на многократное применение при разработке многих конкретных автоматизированных систем.

Сущность первого направления можно охарактеризовать словами *«системная интеграция»* (другое близкое понятие имеет название *консалтинг).* Разработчик ИС должен быть специалистом в области системотехники, хорошо знать соответствующие международные стандарты, состояние и тенденции развития информационных технологий и программных продуктов, владеть инструментальными средствами разработки приложений (CASE-средствами) и быть готовым к восприятию и анализу автоматизируемых процессов в сотрудничестве со специалистами-прикладниками.

Существует ряд фирм, специализирующихся на разработке проектов ИС (например, Price Waterhouse, ЛАНИТ, LUXOFT и др.)

Второе направление в большей мере относится к области разработки математического и программного обеспечения для реализации функций ИС - моделей, методов, алгоритмов, программ на базе знания системотехники, методов анализа и синтеза проектных решений, технологий программирования, операционных систем и т.п. Существует ряд общеизвестных технологий (методик) проектирования ПО ИС, среди которых, прежде всего, следует назвать компонентно-ориентированную разработку - технологию индустриальной разработки программных систем.

В России действует государственный стандарт на стадии создания автоматизированных систем ГОСТ 34.601-90. Существует и международный стандарт на стадии жизненного цикла программной продукции (ISO 12207:1995). Как собственно ИС, так и компоненты ИС являются сложными системами, и при их проектировании нужно использовать один из стилей проектирования:

- *нисходящее (Top-of-Design),* четкая реализация нисходящего проектирования приводит к *спиральной модели* разработки ПО, на каждом витке спирали блоки предыдущего уровня детализируются, используются обратные связи (альтернативой является так называемая *каскадная модель,* относящаяся к поочередной реализации частей системы);
- *восходящее (Bottom-of-Design);*
- *эволюционное (Middle-of-Design).*

Чаще всего применяют нисходящий стиль блочно-иерархического проектирования.

Рассмотрим этапы нисходящего проектирования ИС.

Верхний уровень проектирования ИС часто называют *концептуальным* проектированием. Концептуальное проектирование выполняют в процессе предпроектных исследований, формулировки ТЗ, разработки эскизного проекта.

Предпроектные исследования проводят путем анализа деятельности предприятия (компании, учреждения, офиса), на котором создает или модернизируется ИС. При этом нужно получить, ответы на вопросы:

1. Что устраивает в существующей технологии?
2. Что можно улучшить?
3. Кому это нужно и, следовательно, каков будет эффект?

Перед обследованием формируются и в процессе его проведения уточняются цели обследования - определение возможностей и ресурсов для повышения эффективности функционирования предприятия на основе автоматизации процессов управления, проектирования, документооборота и т.п. Содержание обследования - выявление структуры предприятия, выполняемых функций, информационных потоков, имеющихся опыта *и* средств автоматизации. Обследование проводят системные аналитики (системные интеграторы) совместно с представителями организации-заказчика.

На основе анализа результатов обследования строят модель, отражающую деятельность предприятия на данный момент (до реорганизации). Такую модель называют «*As Is (как есть).* Далее разрабатывают исходную концепцию ИС. Эта концепция включает в себя предложения по изменению структуры предприятия, взаимодействию подразделений, информационным потокам, что выражается в модели «То Ве» (Как должно быть).

Результаты анализа конкретизируются в ТЗ на создание ИС. В ТЗ указывают потоки входной информации, типы выходных документов и предоставляемых услуг, уровень защиты информации, требования к производительности (пропускной способности) и т.п. ТЗ направляют заказчику для обсуждения и окончательного согласования.

Эскизный проект (техническое предложение) предоставляют в виде проектной документации, описывающей архитектуру системы, структуру ее подсистем, состав модулей. Здесь же содержатся предложения по выбору базовых программно-аппаратных средств, которые должны учитывать прогноз развития предприятия.

В отношении аппаратных средств и особенно ПО такой выбор чаще всего есть выбор фирмы-поставщика необходимых средств (или, по крайней мере, базового ПО), так как правильная совместная работа программ разных фирм достигается с большим трудом. В проекте может быть предложено несколько вариантов выбора. При анализе выясняются возможности покрытия автоматизи\руемых функций имеющимися программными продуктами и, следовательно, объемы работ по разработке оригинального ПО. Подобный анализ необходим для предварительной оценки временных и материальных затрат на автоматизацию. Учет ресурсных ограничений позволяет уточнить достижимые масштабы автоматизации, подразделить проектирование ИС на работы первой, второй очереди и т. д.

После принятия эскизного проекта разрабатывают *прототип* ИС, представляющий собой набор программ, эмулирующих работу готовой системы. Благодаря прототипированию можно не только разработчикам, но и будущим пользователям ИС увидеть контуры и особенности системы и, следовательно, заблаговременно внести коррективы в проект.

Как на этапе предпроектных исследований, так и на последующих этапах целесообразно придерживаться определенной дисциплины фиксации и представления получаемых результатов, основанной на той или иной методике формализации спецификаций. Формализация нужна для однозначного понимания исполнителями и заказчиком требований, ограничений и принимаемых решений.

При концептуальном проектировании применяют ряд спецификаций, среди которых центральное место занимают модели преобразования, хранения и передачи информации. Модели, полученные в процессе обследования предприятия, являются моделями его функционирования. В процессе разработки ИС модели, как правило, претерпевают существенные изменения (переход от «As Is» к «To Be») и в окончательном виде модель *«To Be»* рассматривают *в* качестве модели проектируемой ИС.

Различают функциональные, информационные, поведенческие и структурные модели.

- **Функциональная модель** системы описывает совокупность выполняемых системой функций.
- ***Информационная*** **модель** отражает структуры данных – их состав и взаимосвязи.
- **Поведенческая модель** описывает информационные процессы (динамику функционирования), в ней фигурируют такие категории, как состояние системы, событие, переход из одного состояния в другое, условия перехода, последовательность событий, осуществляется привязка ко времени.
- ***Структурная*** **модель** характеризует построение системы – состав подсистем, их взаимосвязи.

Содержанием последующих этапов нисходящего проектирования (согласно ГОСТ *34.601-90,* это стадия разработки технического проекта, рабочей документации, ввода в действие) являются уточнение перечней приобретаемого оборудования и готовых программных продуктов, построение системной среды, детальное инфологическое проектирование баз данных и их первоначальное наполнение, разработка собственного оригинального ПО, которая, в свою очередь, делится на ряд этапов нисходящего проектирования. Эти работы составляют содержание *рабочего проектирования.* После этого следуют закупка и инсталляция программно-аппаратных средств, внедрение и опытная эксплуатация системы.

Особое место в ряду проектных задач занимает разработка проекта корпоративной вычислительной сети, поскольку ТО ИС имеет сетевую структуру. Если территориально ИС располагается в одном здании или в нескольких близко расположенных зданиях, то корпоративная сеть может быть выполнена в виде совокупности нескольких локальных подсетей, связанных опорной локальной сетью. Кроме выбора типов подсетей, связных протоколов и коммутационного оборудования приходится решать задачи распределения узлов по подсетям, выделения серверов, выбора сетевого ПО, определения способа управления данными в выбранной схеме распределенных вычислений и т. п.

В случае, если ИС располагается в удаленных друг от друга пунктах, в частности, расположенных в разных городах, то прежде всего рассматривается возможность использования услуг Internet. Возникающие при этом проблемы связаны с обеспечением информационной безопасности и надежности доставки сообщений.

Источники

1. А.С. Усов – Письмо 1. Стр. 1 - 3.
2. А.С. Усов – Письмо 2. Стр. 4 – 8.
3. Норенков – Основы автоматизированного проектирования. Стр. 33-36
4. Веб-библиотека. Статья «Любая сущность - объект»

   http://www.weblibrary.biz/c-sharp/princypy/obekt