Учебно-методическое пособие посвящено изучению основ анализа требований к программному обеспечению информационных систем мониторинга технических объектов.

Рассмотрены вопросы обеспечения жизненного цикла программных продуктов информационных систем, представлены модели создания информационных систем, их особенности, достоинства и недостатки.

Рассмотрены основы структурного и функционально–модульного проектирования ИС. Приведены принципы структурного подхода, последовательность построения функциональной модели ИС. Рассмотрены модели SADT, DFD, workflow diagramming, их состав, схемы построения иерархических систем. Рассмотрены требования к программному обеспечению информационной системы и рассмотрен процесс анализа требований к программному обеспечению информационной системы, а также формирования системных, пользовательских, функциональных и нефункциональных требований, документирования требований, оценки их выполнимости.

# Оглавление

## Введение

Инженерия программного обеспечения – сравнительно молодая научная дисциплина. Термин software engineering был впервые предложен в 1968 году на конференции, посвященной так называемому кризису программного обеспечения. Этот кризис был вызван появлением мощной (по меркам того времени) вычислительной техники третьего поколения. Новая техника позволяла воплотить в жизнь не реализуемые ранее программные приложения. В результате программное обеспечение достигло размеров и уровня сложности, намного превышающих аналогичные показатели у программных систем, реализованных на вычислительной технике предыдущих поколений.

Целью инженерии программного обеспечения является эффективное создание программных систем. Программное обеспечение абстрактно и нематериально. Оно не имеет физической природы, отвергает физические законы и не подвергается обработке производственными процессами. Такой упрощенный взгляд на ПО показывает, что не существует физических ограничений на потенциальные возможности программных систем. С другой стороны, отсутствие материального наполнения порой делает ПО чрезвычайно сложным и, следовательно, трудным для понимания "объектом".

Оказалось, что неформальный подход, применявшийся ранее к построению программных систем, недостаточен для разработки больших систем. На реализацию крупных программных проектов иногда уходили многие годы. Стоимость таких проектов многократно возрастала по сравнению с первоначальными расчетами, сами программные системы получались ненадежными, сложными в эксплуатации и сопровождении. Разработка программного обеспечения оказалась в кризисе. Стоимость аппаратных средств постепенно снижалась, тогда как стоимость ПО стремительно возрастала. Возникла необходимость в новых технологиях и методах управления комплексными сложными проектами разработки больших программных систем.

Такие методы составили часть инженерии программного обеспечения и в настоящее время широко используются, хотя, конечно же, не являются универсальными. Сохраняется много проблем в процессе разработки сложных программных систем, на решение которых затрачивается много времени и средств.

## Моделирование систем

В процессе формализации требований к системе и на этапе проектирования система рассматривается как совокупность компонентов и взаимосвязей между ними. Для этого используются модели системной архитектуры, которые в графическом виде предоставляют всю организацию системы, т.е. ее компоненты и взаимосвязи между ними.

Архитектура системы обычно представляется в виде блочной диаграммы (блок-схемы), где блоки соответствуют основным подсистемам, а существующие связи между подсистемами обозначаются линиями со

стрелками, соединяющими отдельные блоки диаграммы. Связи могут соответствовать потокам данных, последовательности включения подсистем в работу или каким-либо другим типам зависимости.

На этом уровне детализации система разбивается на отдельные подсистемы. Каждая подсистема, в свою очередь, может быть представлена как декомпозиция своих функциональных компонентов. Это такие компоненты подсистемы, которые, исходя из предназначения подсистемы, выполняют какую-либо одну функцию. В противоположность этому подсистема обычно выполняет несколько функций. Конечно, декомпозицию подсистем (и самой системы) можно проводить по другим признакам, например конструктивным или технологическим.

## Процесс создания программного обеспечения

Создание ПО – это совокупность процессов, приводящих к созданию программного продукта. Эти процессы основываются главным образом на технологиях инженерии программного обеспечения. Существует четыре фундаментальных процесса (более подробно описанных далее в книге), которые присущи любому проекту создания ПО.

1. Разработка спецификации требований на программное обеспечение. Требования определяют функциональные характеристики системы и обязательны для выполнения.
2. Создание программного обеспечения. Разработка и создание ПО согласно спецификации на него.
3. Аттестация программного обеспечения. Созданное ПО должно пройти аттестацию для подтверждения соответствия требованиям заказчика.
4. Совершенствование (модернизация) программного обеспечения. ПО должно быть таким, чтобы его можно было модернизировать согласно измененным требованиям потребителя.

При выполнении разнообразных программных проектов эти процессы могут быть организованы различными способами и описаны на разных уровнях детализации. Длительность реализации этих процессов также далеко не всегда одинакова. И вообще, различные организации, занимающиеся производством ПО, если использовать неподходящий процесс, это может привести к снижению качества и функциональности разрабатываемого программного продукта.

## Модель процесса создания ПО

Такая модель представляет собой упрощенное описание процесса создания ПО – последовательность практических этапов, необходимых для разработки создаваемого программного продукта. Подобные модели,

несмотря на их разнообразие, служат абстрактным представлением реального процесса создания ПО. Модели могут отображать процессы, которые являются частью технологического процесса создания ПО, компоненты программных продуктов и действия людей, участвующих в создании ПО.

1. Модель последовательности работ. Показывает последовательность этапов, выполняемых в процессе создания ПО, включая начало и завершение каждого этапа, а также зависимость между выполнением этапов. Этапы в этой модели соответствуют определенным работам, выполняемым разработчиками ПО.
2. Модели потоков данных и процессов. В них процесс создания ПО представляется в виде множества активностей (процессов), в ходе реализации которых выполняются преобразования определенных данных. Например, на вход активности (процесса) создания спецификации ПО поступают определенные данные, на выходе этой активности получают данные, которые поступают на вход активности, соответствующей проектированию ПО, и т.д. Активность в такой модели часто является процессом более низкого порядка, чем этапы работ в модели предыдущего типа. Преобразования данных при реализации активностей могут выполнять как разработчики ПО, так и компьютеры.
3. Ролевая модель. Модель этого типа представляет роли людей, включенных в процесс создания ПО, и действия, выполняемые ими в этих ролях.

Существует также большое количество разнообразных моделей процесса разработки программного обеспечения (т.е. подходов к процессу разработки).

1. Каскадный подход. Весь процесс создания ПО разбивается на отдельные этапы: формирование требований к ПО, проектирование и разработка программного продукта, его тестирование и т.д. Переход к следующему этапу осуществляется только после того, как полностью завершаются работы на предыдущем.
2. Эволюционный подход. Здесь последовательно перемежаются этапы формирования требований, разработки ПО и его аттестации. Первоначальная программная система быстро разрабатывается на основе некоторых абстрактных (общих) требований. Затем они уточняются и детализируются в соответствии с требованиями заказчика. Далее система дорабатывается и аттестуется в соответствии с новыми уточненными требованиями. Такая последовательность действий может повториться несколько раз.
3. Формальные преобразования. Основан на разработке формальной математической спецификации программной системы и преобразовании этой спецификации посредством специальных математических методов в программы. Такое преобразование удовлетворяет условию "сохранения корректности". Это означает, что полученная программа будет в точности соответствовать разработанной спецификации.

4. Сборка программного продукта из ранее созданных компонентов. Предполагается, что отдельные составные части программной системы уже существуют, т.е. созданы ранее. В этом случае технологический процесс создания ПО основное внимание уделяет интеграции отдельных компонентов в общее целое, а не созданию этих компонентов.

Каскадная и эволюционная модели разработки широко используются на практике. Модель формальной разработки систем успешно применялась во многих проектах, но количество организаций-разработчиков, постоянно использующих этот метод, невелико.

## I. Модели жизненного цикла программного обеспечения

В основе создания ПО лежит понятие жизненного цикла (ЖЦ). ЖЦ ПО – это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.

Основным нормативным документом, регламентирующим ЖЦ ПО, является международный стандарт ISO/IEC 12207 (ISO – International Organization of Standardization – Международная организация по стандартизации, IEC – International Electrotechnical Commission – Международная комиссия по электротехнике). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

В соответствии со стандартом ISO/IEC 12207 все процессы ЖЦ ПО разделены на три группы.

Это следующие группы:

1. Основные процессы:
   - приобретение,
   - поставка,
   - разработка,
   - эксплуатация,
   - сопровождение.
2. Вспомогательные процессы, обеспечивающих выполнение основных процессов:
   - документирование,
   - управление конфигурацией,
   - обеспечение качества,
   - верификация, аттестация,
   - совместная оценка, аудит,
   - разрешение проблем.
3. Организационные процессы:
   - управление,
   - создание инфраструктуры,
   - усовершенствование,

- обучение.

Стандарт ISO/IEC 12207 не предлагает конкретную модель ЖЦ ПО и методы разработки.

Существует множество моделей ЖЦ, каждая из которых определяет структуру, последовательность выполнения и взаимосвязи процессов, этапов, действий и задач на протяжении жизненного цикла, а также критерии перехода от этапа к этапу. Наибольшее распространение в настоящее время получили следующие модели ЖЦ.

## Каскадная модель

Это первая модель процесса создания ПО, порожденная моделями других инженерных процессов. Эту модель также иногда называют моделью жизненного цикла программного обеспечения*. Основные принципиальные этапы (стадии) этой модели отражают все базовые виды деятельности, необходимые для создания ПО:

* Жизненный цикл программного обеспечения - это совокупность процессов, протекающих в период от момента принятия решения о создании ПО до его полного вывода из эксплуатации. Таким образом, "жизненный цикл ПО" является более широким понятием, чем модель процесса создания ПО. Вместе с тем каскадную модель можно рассматривать как одну из моделей жизненного цикла ПО. - Прим. ред.

1. Анализ и формирование требований. Путем консультаций с заказчиком ПО определяются функциональные возможности, ограничения и цели создаваемой программной системы.
2. Проектирование системы и программного обеспечения. Процесс проектирования системы разбивает системные требования, на требования, предъявляемые к аппаратным средствам, и требования к программному обеспечению системы. Разрабатывается общая архитектура системы. Проектирование ПО предполагает определение и описание основных программных компонентов и их взаимосвязей.
3. Кодирование и тестирование программных модулей. На этой стадии архитектура ПО реализуется в виде множества программ или программных модулей. Тестирование каждого модуля включает проверку его соответствия требованиям к данному модулю.
4. Сборка и тестирование системы. Отдельные программы и программные модули интегрируются и тестируются в виде целостной системы. Проверяется, соответствует ли система своей спецификации.
5. Эксплуатация и сопровождение системы. Обычно (хотя и не всегда) это самая длительная фаза жизненного цикла ПО. Система инсталлируется, и начинается период ее эксплуатации. Сопровождение системы включает исправление ошибок, которые не были обнаружены на более ранних этапах жизненного цикла, совершенствование системных компонентов и "подгонку" функциональных возможностей системы к новым требованиям.

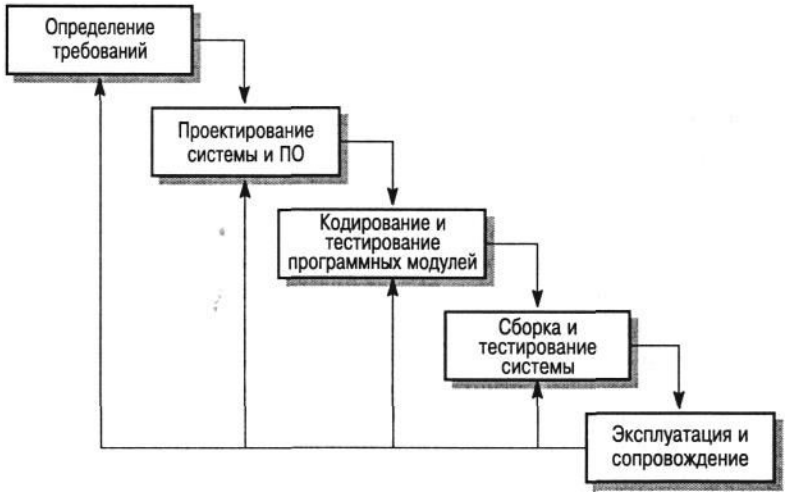



Рис. 3.1. Жизненный цикл программного обеспечения

Как правило результат каждого этапа должен утверждаться документально (это как бы сигнал об окончании этапа). Тогда следующий этап не может начаться до завершения предыдущего. Однако на практике этапы могут перекрываться с постоянным перетеканием информации от одного этапа к другому. Например, на этапе проектирования может возникнуть необходимость уточнить системные требования либо на этапе кодирования могут выявиться проблемы, которые можно решить лишь на этапе проектирования, и т.д. Процесс создания ПО нельзя описать простой линейной моделью, так как он неизбежно содержит последовательность повторяющихся процессов.

Поскольку на каждом этапе проводятся определенные работы и оформляется сопутствующая документация, повторение этапов приводит к повторным работам и значительным расходам. Поэтому после небольшого числа повторений обычно "замораживается" часть этапов создания ПО, например этап определения требований, но продолжается выполнение последующих этапов. Возникающие проблемы, решение которых требует возврата к "замороженным" этапам, игнорируются либо делаются попытки решить их программно. "Замораживание" этапа определения требований может привести к тому, что разработанная система не будет удовлетворять всем требованиям заказчика. Это также может привести к появлению плохо структурированной системы, если упущения этапа проектирования исправляются только с помощью программистских хитростей.

Последний этап жизненного цикла ПО (эксплуатация и сопровождение) – это "полноценное" использование программной системы. На этом этапе могут обнаружиться ошибки, допущенные, например, на первом этапе формирования требований. Могут также проявиться ошибки проектирования и кодирования, что может потребовать определения новых функциональных возможностей системы. С другой стороны, система постоянно должна оставаться работоспособной. Внесение необходимых изменений в программную систему может потребовать повторения некоторых или даже всех этапов процесса создания ПО.

К недостаткам каскадной модели можно отнести негибкое разбиение процесса создания ПО на отдельные фиксированные этапы. В этой модели определяющие систему решения принимаются на ранних этапах и затем их трудно отменить или изменить, особенно это относится к формированию системных требований. Поэтому каскадная модель применяется тогда, когда требования формализованы достаточно четко и корректно. Вместе с тем каскадная модель хорошо отражает практику создания ПО. Технологии

создания ПО, основанные на данной модели, используются повсеместно, в частности для разработки систем, входящих в состав больших инженерных проектов. (В каскадной модели создания ПО определение требований осуществляется совместно с заказчиком до начала проектирования системы, точно так же системная архитектура должна быть создана до начала непосредственной реализации (кодирования) системы. Изменения в требованиях, сделанные на этапе написания кода, ведут к необходимости выполнения повторных работ по проектированию и кодированию системы.)

К достоинствам каскадной модели можно отнести простоту управления процессом создания ПО (в рамках данной модели), а также наличие отдельных этапов проектирования и реализации, что приводит к созданию вполне работоспособных систем, в которых учтены все изменения в спецификации, сделанные уже во время самого процесса разработки ПО.

### Эволюционная модель разработки

Эта модель основана на следующей идее: разрабатывается первоначальная версия программного продукта, которая передается на испытание пользователям, затем она дорабатывается с учетом мнения пользователей, получается промежуточная версия продукта, которая также проходит "испытание пользователем", снова дорабатывается и так несколько раз, пока не будет получен необходимый программный продукт (рис. 3.2). Отличительной чертой данной модели является то, что процессы специфицирования, разработки и аттестации ПО выполняются параллельно при постоянном обмене информацией между ними.

Различают два подхода к реализации эволюционного метода разработки.

1. Подход пробных разработок. Здесь большую роль играет постоянная работа с заказчиком (или пользователями) для того, чтобы определить полную систему требований к ПО, необходимую для разработки конечной версии продукта. В рамках этого подхода вначале разрабатываются те части системы, которые очевидны или хорошо специфицированы. Система эволюционирует (дорабатывается) путем добавления новых средств по мере их предложения заказчиком.
2. Прототипирование. Здесь целью процесса эволюционной разработки ПО является поэтапное уточнение требований заказчика и, следовательно, получение законченной спецификации, определяющей разрабатываемую систему. Прототип* обычно строится для экспериментирования с той частью требований заказчика, которые сформированы нечетко или с внутренними противоречиями.

* Под прототипом обычно понимается действующий программный модуль, реализующий отдельные функции создаваемого ПО.

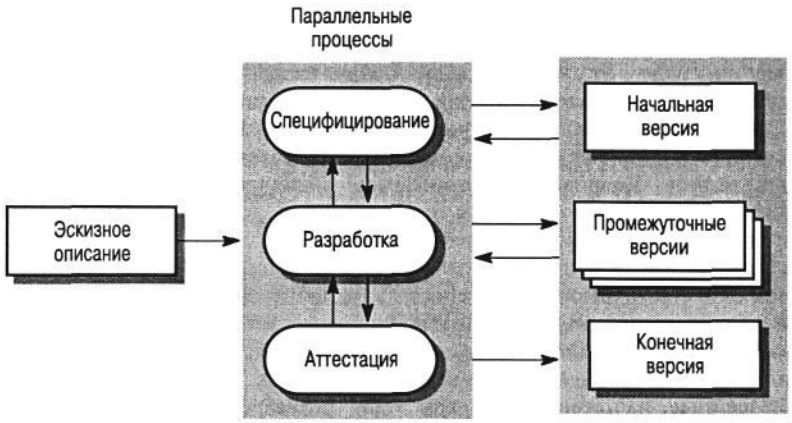



Рис. 3.2. Эволюционная модель разработки

Эволюционный подход часто более эффективен, чем подход, построенный на основе каскадной модели, особенно если требования заказчика могут меняться в процессе разработки системы. Достоинством процесса создания ПО, построенного на основе эволюционного подхода, является то, что здесь спецификация может разрабатываться постепенно, по мере того как заказчик (или пользователи) осознает и сформулирует те задачи, которые должно решать программное обеспечение. Вместе с тем данный подход имеет и некоторые недостатки.

1. Многие этапы процесса создания ПО не документированы. Менеджерам проекта создания ПО необходимо регулярно документально отслеживать выполнение работ. Но если система разрабатывается быстро, то экономически не выгодно документировать каждую версию системы.

2. Система часто получается плохо структурированной. Постоянные изменения в требованиях приводят к ошибкам и упущениям в структуре ПО. Со временем внесение изменений в систему становится все более сложным и затратным.

3. Часто требуются специальные средства и технологии разработки ПО. Это вызвано необходимостью быстрой разработки версий программного продукта, что может привести к несовместимости некоторых применяемых средств и технологий, что, в свою очередь, требует наличия в команде разработчиков специалистов высокого уровня.

В отличие от каскадной, в эволюционной модели можно отложить принятие окончательных решений о спецификации и структуре системы, однако это может привести к созданию плохо структурированной системы, которая будет также трудна в сопровождении.

Считается, что эволюционный подход наиболее приемлем для разработки небольших программных систем (до 100 000 строк кода) и систем среднего размера (до 500 000 строк кода) с относительно коротким сроком жизни. На больших долгоживущих системах слишком заметно проявляются недостатки этого подхода.

Для таких систем считается приемлемым смешанный подход к созданию ПО, который вобрал бы в себя лучшие черты каскадной и эволюционной моделей разработки.

При таком смешанном подходе для прояснения "темных мест" в системной спецификации можно использовать прототипирование. Часть системных компонентов, для которых полностью определены требования, может создаваться

на основе каскадной модели. Другие системные компоненты, которые трудно поддаются специфицированию, например пользовательский интерфейс, могут разрабатываться с использованием прототипирования.

## Формальная разработка систем

Этот подход к созданию ПО имеет много черт, сходных с каскадной моделью, но построен на основе формальных математических преобразований системной спецификации в исполняемую программу. Процесс создания программного обеспечения в соответствии с этим подходом показан на рис. 3.3. Здесь для упрощения не указаны обратные связи между этапами процесса.

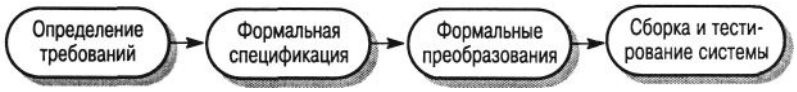



Рис. 3.3. Модель формальной разработки ПО

Между данным подходом и каскадной моделью существуют следующие кардинальные отличия.

1. Здесь спецификация системных требований имеет вид детализированной формальной спецификации, записанной с помощью специальной математической нотации.
2. Процессы проектирования, написания программного кода и тестирования системных модулей заменяются процессом, в котором формальная спецификация путем последовательных формальных преобразований трансформируется в исполняемую программу. Этот процесс показан на рис. 3.4.

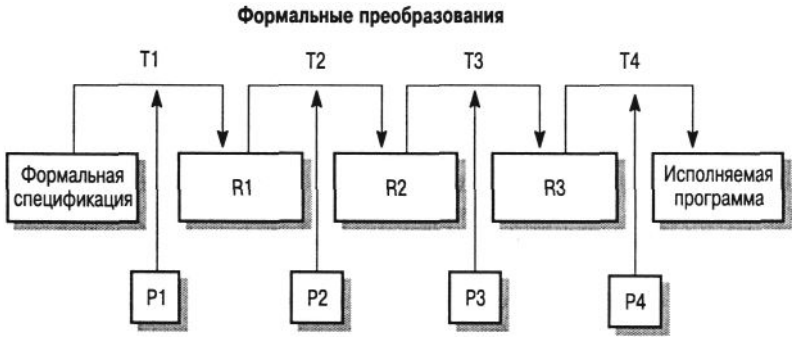



Рис. 3.4. Процесс формальных преобразований

В процессе преобразования формальное математическое представление системы последовательно и математически корректно трансформируется в программный код, постепенно все более детализированный. Эти преобразования выполняются до тех пор, пока все позиции формальной спецификации не будут трансформированы в эквивалентную программу. Преобразования выполняются математически корректно – здесь не существует проблемы проверки соответствия спецификации и программы.

Преимущество метода формальных преобразований, которое заключается в точном соответствии конечной программы спецификации, обеспечивается тем, что дистанция между последовательными преобразованиями значительно меньше дистанции между спецификацией и программой. Доказательство корректности программного кода для больших масштабируемых систем обычно очень длительно, а часто просто не выполнимо. В этом отношении метод формальных преобразований, состоящий из последовательности небольших формальных шагов, весьма привлекателен. Однако выбор для применения соответствующих формальных преобразований сложен и неочевиден.

Наиболее известным примером метода формальных преобразований является метод "чистой комнаты" (Cleanroom), разработанный компанией IBM [239, 310, 219, 284]. Этот метод предполагает пошаговую разработку ПО, когда на каждом шаге применяется формальные преобразования. Это позволяет отказаться от тестирования отдельных программных модулей, а тестирование всей системы происходит после ее сборки. Этот подход более подробно рассмотрен в главе 19.

Все эти методы имеют несколько "врожденных" недостатков, а стоимость разработки ПО с помощью этих методов не намного отличается от стоимости разработок другими методами. Методы формальных преобразований обычно применяют для разработки систем, которые должны удовлетворять строгим требованиям надежности, безотказности и безопасности, так как они гарантируют соответствие созданных систем их спецификациям.

Кроме разработки указанного типа систем, методы формальных преобразований не нашли широкого применения, поскольку требуют специальных знаний и опыта использования. Кроме того, для большинства систем эти методы не дают существенного выигрыша в стоимости или качестве по сравнению с другими методами разработки ПО. Основная причина заключается в том, что функционирование большинства систем с трудом поддается описанию методом формальных спецификаций, – при создании большинства программных систем большая часть усилий разработчиков уходит именно на создание спецификаций.

## Разработка ПО на основе ранее созданных компонентов

В большинстве программных проектов применяется повторное использование некоторых программных модулей. Это обычно случается там, где разработчики проекта знают о ранее созданных программных продуктах, в составе которых есть компоненты, приблизительно удовлетворяющие требованиям разрабатываемых компонентов. Эти компоненты модифицируются в соответствии с новыми требованиями и затем включается в состав новой системы. В эволюционной модели разработки, описанной в разделе 3.1.2, для ускорения процесса создания ПО повторное использование ранее созданных компонентов применяется достаточно часто.

Неформальное решение о повторном использовании ранее созданных программных компонентов обычно принимается независимо от общего процесса создания ПО. Вместе с тем на протяжении нескольких последних лет все более широко применяется подход к созданию ПО, основанный именно на повторном использовании ранее созданных программных модулей.

Этот подход основан на наличии большой базы существующих программных компонентов, которые можно интегрировать в создаваемую новую систему. Часто такими компонентами являются свободно продаваемые на рынке программные продукты, которые можно использовать для выполнения определенных специальных функций, таких как форматирование текста, числовые вычисления и т.п. Общая модель процесса разработки ПО с повторным использованием ранее созданных компонентов показана на рис. 3.5.

В этом подходе начальный этап специфицирования требований и этап аттестации такие же, как и в других моделях процесса создания ПО. А этапы, расположенные между ними, имеют следующий смысл.

1. Анализ компонентов. Имея спецификацию требований, на этом этапе осуществляется поиск компонентов, которые могли бы удовлетворить сформулированным требованиям. Обычно невозможно точно сопоставить функции, реализуемые готовыми компонентами, и функции, определенные спецификацией требований.
2. Модификация требований. На этой стадии анализируются требования с учетом информации о компонентах, полученной на предыдущем этапе. Требования модифицируются таким образом, чтобы максимально использовать возможности отобранных компонентов. Если изменение требований невозможно, повторно выполняется анализ компонентов для того, чтобы найти какое-либо альтернативное решение.
3. Проектирование системы. На данном этапе проектируется структура системы либо модифицируется существующая структура повторно используемой системы. Проектирование должно учитывать отобранные программные компоненты и строить структуру в соответствии с их функциональными возможностями. Если некоторые готовые программные компоненты недоступны, проектируется новое ПО.
4. Разработка и сборка системы. Это этап непосредственного создания системы. В рамках рассматриваемого подхода сборка системы является скорее частью разработки системы, чем отдельным этапом.

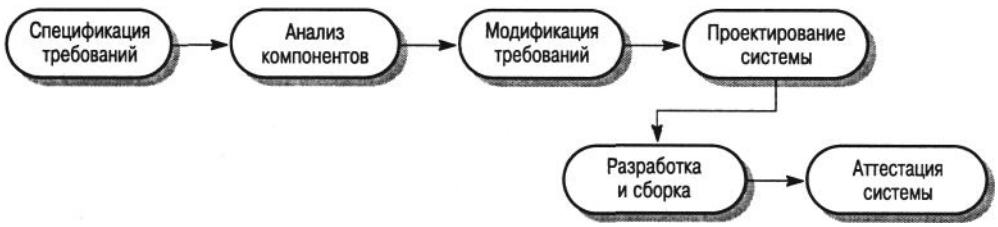



Рис. 3.5. Разработка ПО с повторным использованием ранее созданных компонентов

Основные достоинства описываемой модели процесса разработки ПО с повторным использованием ранее созданных компонентов заключаются в том, что сокращается количество непосредственно разрабатываемых

компонентов и уменьшается общая стоимость создаваемой системы. Вместе с тем при использовании этого подхода неизбежны компромиссы при определении требований; это может привести к тому, что законченная система не будет удовлетворять всем требованиям заказчика. Кроме того, при проведении модернизации системы (т.е. при создании ее новой версии) отсутствует возможность влиять на появление новых версий компонентов, используемых в системе, что значительно затрудняет сам процесс модернизации.

## Итерационные модели разработки ПО

Описанные модели процесса создания ПО имеют свои достоинства и недостатки. При создании больших систем, как правило, приходится использовать различные подходы к разработке разных частей системы, т.е. в целом к разработке системы применяются гибридные (смешанные) модели. Поэтому важную роль играет возможность выполнять отдельные процессы разработки подсистем и весь процесс создания ПО итерационно, когда в ответ на изменения требований повторно выполняются определенные этапы создания системы (чаще всего этапы проектирования и кодирования).

В этом разделе я представлю две гибридные итерационные модели, сочетающие несколько различных подходов к разработке ПО и разработанные специально для поддержки итерационного способа создания ПО.

1. Модель пошаговой разработки, где процессы специфицирования требований, проектирования и написания кода разбиваются на последовательность небольших шагов, которые ведут к созданию ПО.
2. Спиральная модель разработки, в которой весь процесс создания ПО, от начального эскиза системы до ее конечной реализации, разворачивается по спирали.

Существенным отличием итерационных моделей является то, что здесь процесс разработки спецификации протекает параллельно с разработкой самой программной системы. Более того, в модели пошаговой разработки полную системную спецификацию можно получить только после завершения самого последнего шага процесса создания ПО. Очевидно, что такой подход входит в противоречие с моделью приобретения ПО, когда полная системная спецификация является составной частью контракта на разработку системы. Поэтому, чтобы применять итерационные модели для разработки больших систем, которые заказываются "серьезными" организациями, например государственными агентствами, необходимо менять форму контракта, на что такие организации идут с большим трудом.

## Модель пошаговой разработки

Модель пошаговой разработки находится где-то между каскадной и эволюционной моделями и объединяет их достоинства. Эта модель (рис. 3.6) была предложена Миллсом (Mills, [240]) как попытка уменьшить количество повторно выполняемых работ в процессе создания ПО и увеличить для заказчика временной период окончательного принятия решения обо всех деталях системных требований.

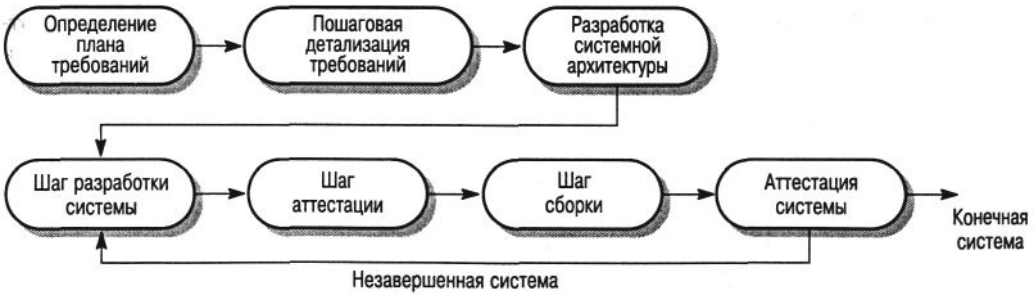



Рис. 3.6. Модель пошаговой разработки

В процессе пошаговой разработки заказчик сначала в общих чертах определяет те сервисы (функциональные возможности), которые должны присутствовать у создаваемой системы. При этом устанавливаются приоритеты, т.е. определяется, какие сервисы более важны, а какие – менее. Также определяется количество шагов разработки, причем на каждом шаге должен быть получен системный компонент, реализующий определенное подмножество системных функций. Распределение реализации системных сервисов по шагам разработки зависит от их приоритетов. Сервисы с более высокими приоритетами реализуются первыми.

Последовательность шагов разработки определяется заранее до начала их выполнения. На первых шагах детализируются требования для сервисов, затем для их реализации (на последующих шагах) используется один из подходящих способов разработки ПО. В ходе их реализации анализируются и детализируются требования для компонентов, которые будут разрабатываться на более поздних шагах, причем изменение требований для тех компонентов, которые уже находятся в процессе разработки, не допускается.

После завершения шага разработки получаем программный компонент, который передается заказчику для интегрирования в подсистему, реализующую определенный системный сервис. Заказчик может экспериментировать с готовыми подсистемами и компонентами для того, чтобы уточнить требования, предъявляемые к следующим версиям уже готовых компонентов или к компонентам, разрабатываемым на последующих шагах. По завершении очередного шага разработки полученный компонент интегрируется с ранее произведенными компонентами; таким образом, после каждого шага разработки система приобретает все большую функциональную завершенность. Общесистемные функции в этом процессе могут реализоваться сразу или постепенно, по мере разработки необходимых компонентов.

В описываемой модели не предполагается, что на каждом шаге используется один и тот же подход к процессу разработки компонентов. Если создаваемый компонент имеет хорошо разработанную спецификацию, то для его создания можно применить каскадную модель. Если же требования определены нечетко, можно использовать эволюционную модель разработки.

Процесс пошаговой разработки имеет целый ряд достоинств.

1. Заказчику нет необходимости ждать полного завершения разработки системы, чтобы получить о ней представление. Компоненты, полученные на первых шагах разработки, удовлетворяют наиболее критическим требованиям (так как имеют наибольший приоритет) и их можно оценить на самой ранней стадии создания системы.
2. Заказчик может использовать компоненты, полученные на первых шагах разработки, как прототипы и провести с ними эксперименты для уточнения требований к тем компонентам, которые будут разрабатываться позднее.
3. Данный подход уменьшает риск общесистемных ошибок. Хотя в разработке отдельных компонентов возможны ошибки, но эти компоненты должны пройти соответствующее тестирование и аттестацию, прежде чем их передадут заказчику.
4. Поскольку системные сервисы с высоким приоритетом разрабатываются первыми, а все последующие компоненты интегрируются с ними, неизбежно получается так, что наиболее важные подсистемы подвергаются более тщательному всестороннему тестированию и проверке. Это значительно снижает вероятность программных ошибок в особо важных частях системы.

Вместе с тем при реализации пошаговой разработки могут возникнуть определенные проблемы. Компоненты, получаемые на каждом шаге разработки, имеют относительно небольшой размер (обычно не более 20 000 строк кода), но должны реализовать какую-либо системную функцию. Отобразить множество системных требований к компонентам нужного размера довольно сложно. Более того, многие системы должны обладать набором базовых системных свойств, которые реализуются совместно различными частями системы. Поскольку требования детально не определены до тех пор, пока не будут разработаны все компоненты, бывает весьма сложно распределить общесистемные функции по компонентам.

В настоящее время предложен метод так называемого экстремального программирования (extreme programming), который устраняет некоторые недостатки метода пошаговой разработки. Этот метод основан на пошаговой разработке малых программных компонентов, реализующих небольшие функциональные требования, постоянном вовлечении заказчика в процесс разработки и обезличенном программировании.

### Спиральная модель разработки

Спиральная модель процесса создания программного обеспечения (рис. 3.7) в настоящее время получила широкую известность и популярность. В отличие от рассмотренных ранее моделей, где процесс создания ПО представлен в виде последовательности отдельных процессов с возможной обратной связью между ними, здесь процесс разработки представлен в виде спирали. Каждый виток спирали соответствует одной стадии (итерации) процесса создания ПО. Так, самый внутренний виток спирали соответствует стадии принятия решения о создании ПО, на следующем витке определяются

системные требования, далее следует стадия (виток спирали) проектирования системы и т.д.

Каждый виток спирали разбит на четыре сектора.

1. Определение целей. Определяются цели каждой итерации проекта. Кроме того, устанавливаются ограничения на процесс создания ПО и на сам программный продукт, уточняются планы производства компонентов. Определяются проектные риски (например, риск превышения сроков или риск превышения стоимости проекта). В зависимости от "проявленных" рисков, могут планироваться альтернативные стратегии разработки ПО.
2. Оценка и разрешение рисков. Для каждого определенного проектного риска проводится его детальный анализ. Планируются мероприятия для уменьшения (разрешения) рисков. Например, если существует риск, что системные требования определены неверно, планируется разработать прототип системы.
3. Разработка и тестирование. После оценки рисков выбирается модель процесса создания системы. Например, если доминируют риски, связанные с разработкой интерфейсов, наиболее подходящей будет эволюционная модель разработки ПО с прототипированием. Если основные риски связаны с соответствием системы и спецификации, скорее всего, следует применить модель формальных преобразований. Каскадная модель может быть применена в том случае, если основные риски определены как ошибки, которые могут проявиться на этапе сборки системы.
4. Планирование. Здесь пересматривается проект и принимается решение о том, начинать ли следующий виток спирали. Если принимается решение о продолжении проекта, разрабатывается план на следующую стадию проекта.

Существенное отличие спиральной модели от других моделей процесса создания ПО заключается в точном определении и оценивании рисков. Если говорить неформально, то риск – это те неприятности, которые могут случиться в процессе разработки системы. Например, если при написании программного кода используется новый язык программирования, то риск может заключаться в том, что компилятор этого языка может быть ненадежным или что результирующий код может быть не достаточно эффективным. Риски могут также заключаться в превышении сроков или стоимости проекта. Таким образом, уменьшение (разрешение) рисков – важный элемент управления системным проектом.

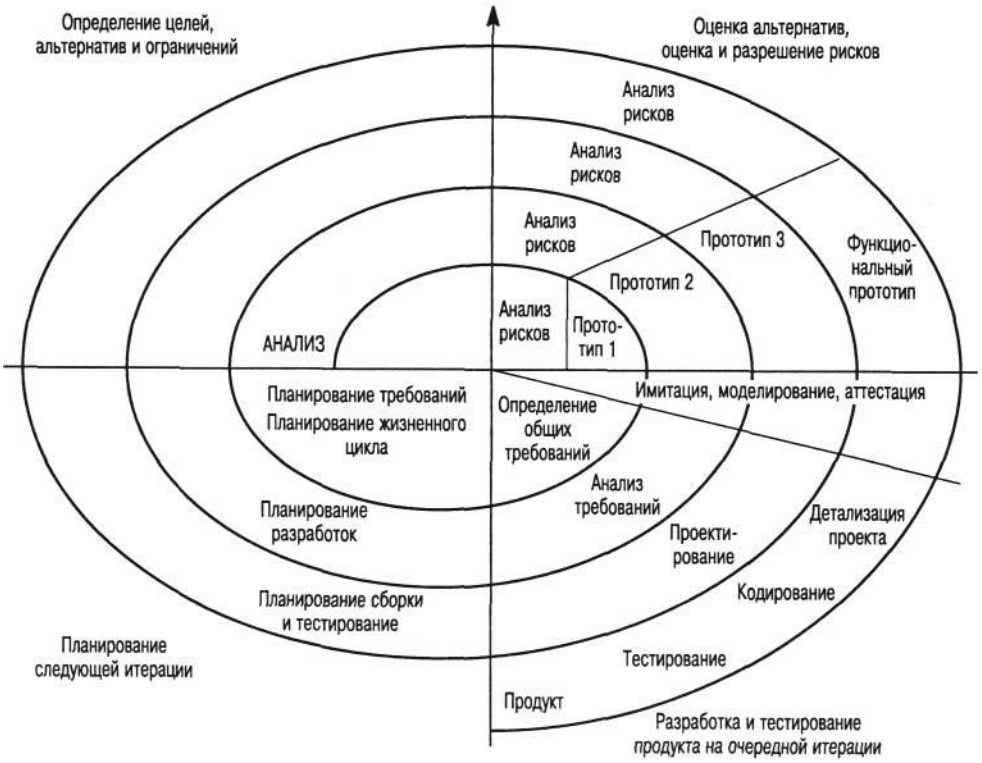



Рис. 3.7. Спиральная модель создания ПО (© 1988 IEEE)

Первая итерация создания ПО в спиральной модели начинается с тщательной проработки системных показателей (целей системы), таких как эксплуатационные показатели и функциональные возможности системы. Конечно, альтернативных путей достижения этих показателей или целей можно сформировать бесконечно много. Но каждая альтернатива должна оценивать стоимость достижения каждой сформулированной цели. Результаты анализа возможных альтернатив служат источником оценки проектного риска. Это происходит на следующей стадии спиральной модели, где для оценки рисков используются более детальный анализ альтернатив, прототипирование, имитационное моделирование и т.п. С учетом полученных оценок рисков выбирается тот или иной подход к разработке системных компонентов, далее он реализуется, затем осуществляется планирование следующего этапа процесса создания ПО.

В спиральной модели нет фиксированных этапов, таких как разработка спецификации или проектирование. Эта модель может включать в себя любые другие модели разработки систем. Например, на одном витке спирали может использоваться протипирование для более четкого определения требований (и, следовательно, для уменьшения соответствующих рисков). Но на следующем витке может применяться каскадная модель. Если требования четко сформулированы, может применяться метод формальных преобразований.

**Контрольные вопросы**

1. Что такое жизненный цикл программного обеспечения?
2. Перечислите основные этапы классической каскадной модели жизненного цикла программного обеспечения (ЖЦ ПО).
3. Укажите основные достоинства и недостатки классической каскадной модели ЖЦ ПО.

4. Укажите основные отличия реального процесса создания ПО в рамках каскадной модели от классической каскадной модели ЖЦ ПО.

5. Перечислите основные положения инкрементной модели ЖЦ ПО.

6. Укажите особенности реализации модели экстремального программирования.

7. Приведите основные положения спиральной модели ЖЦ ПО.

8. Перечислите основные достоинства и недостатки спиральной модели ЖЦ ПО.

9. Укажите особенности реализации методология быстрой разработки приложений.

## II Основные принципы структурного анализа

Анализ основных моделей жизненного цикла ПО позволяет выделить ключевые процессы – **структурный анализ и проектирование,** выполняемые на начальном этапе создания ПО ИС, от которых в целом зависит успешное его создание.

**Структурный анализ –** метод исследования системы, заключающийся в последовательной ее детализации на основе определенного представления об этой системе (например, с позиций выполняемых ею функций и обмена информационными и управляющими потоками), и формировании описания системы в виде иерархической многоуровневой структуры из элементов выбранного представления.

Для метода характерно:

- разбиение на уровни абстракции с ограничением числа элементов на каждом из уровней иерархии представления (обычно от 3 до 6–7);
- ограниченный контекст, включающий лишь существенные на каждом уровне детали;
- двойственность данных и операций над ними;
- использование строгих формальных правил записи;
- последовательное приближение к конечному результату.

В ходе структурного анализа определяются все задачи и функции системы и формируются требования к ИС. Именно на этой стадии закладывается фундамент успеха всего проекта. Известно множество неудачных реализаций из–за неполноты и неточностей в определении требований к системе.

Список требований к разрабатываемой системе должен включать:

1. Совокупность условий, при которых предполагается эксплуатировать будущую систему (аппаратные и программные ресурсы, предоставляемые системе; внешние условия ее функционирования; состав людей и работ, имеющих к ней отношение),
2. Описание выполняемых системой функций,
3. Ограничения в процессе разработки (директивные сроки завершения отдельных этапов, имеющиеся ресурсы, организационные процедуры и мероприятия, обеспечивающие защиту информации).

Целью анализа является преобразование общих, неясных знаний о требованиях к будущей системе в точные (по возможности) определения. На этом этапе определяются:

1. Архитектура системы, ее функции, внешние условия, распределение функций между аппаратурой и ПО,
2. Интерфейсы и распределение функций между человеком и системой,
3. Требования к программным и информационным компонентам ПО, необходимые аппаратные ресурсы, требования к БД, физические характеристики компонентов ПО, их интерфейсы.

Анализ требований разрабатываемой системы является важнейшим среди всех этапов ЖЦ. Он оказывает существенное влияние на все последующие этапы (являясь в то же время наименее изученным и понятным процессом).

На этом этапе:

- необходимо понять, что предполагается сделать;
- документально зафиксировать полученную информацию, т.к. если требования не зафиксированы и не сделаны доступными для участников проекта, то можно считать, что они не существуют.

Язык, на котором формулируются требования, должен быть достаточно прост и понятен заказчику.

Во многих аспектах системный анализ является наиболее трудной частью разработки. Проблемы, с которыми сталкивается разработчик, взаимосвязаны, что и является одной из главных причин их трудной разрешимости:

1. Разработчику сложно получить исчерпывающую информацию для оценки требований к системе с точки зрения заказчика,
2. Заказчик, в свою очередь, не имеет достаточной информации о проблеме обработки данных, чтобы судить, что является выполнимым, а что нет,
3. Разработчик сталкивается с чрезмерным количеством подробных сведений о предметной области и о новой системе,
4. Спецификация системы из-за объема и технических терминов часто непонятна для заказчика,
5. В случае понятности спецификации для заказчика, она будет являться недостаточной для проектировщиков и программистов, создающих систему.

Применение известных аналитических методов снимает некоторые из проблем анализа, однако эти проблемы могут быть существенно облегчены за счет применения современных структурных методов, среди которых центральное место занимают методологии структурного анализа.

Сущность структурного подхода к разработке ИС, заключается в ее декомпозиции (разбиении) на автоматизируемые функции: система разбивается на функциональные подсистемы, которые в свою очередь делятся на подфункции, подразделяемые на задачи и так далее. Процесс разбиения продолжается вплоть до конкретных процедур. При этом автоматизируемая система сохраняет целостное представление. Подсистемы базируются на ряде общих принципов, часть из которых регламентирует организацию работ на начальных этапах ЖЦ, а часть используется при выработке рекомендаций по организации работ. Все наиболее распространенные методологии структурного подхода базируются на ряде общих принципов. В качестве двух базовых принципов используются следующие:

- *принцип "разделяй и властвуй"* – принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;
- *принцип иерархического упорядочивания* – принцип организации составных частей общей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне. Этот принцип декларирует, что устройство этих частей также существенно для понимания, а ясность и понимание проблемы резко повышается при организации ее частей в древовидные иерархические структуры.

Выделение двух базовых принципов не означает, что остальные принципы являются второстепенными, поскольку игнорирование любого из них может привести к непредсказуемым последствиям (в том числе и к провалу всего проекта). Основными из этих принципов являются следующие:

- *принцип абстрагирования* – заключается в выделении существенных с некоторых позиций аспектов системы и отвлечение от несущественных с целью представления проблемы в простом общем виде;
- *принцип формализации* – заключается в необходимости строгого методического подхода к решению проблемы;
- *принцип непротиворечивости* – заключается в обоснованности и согласованности элементов;
- *принцип ограниченной видимости*, который заключается в игнорировании несущественной на конкретном этапе информации, когда каждая часть системы знает только необходимую ей информацию,
- *принцип концептуальной общности* заключается в следовании единой методологии на всех этапах ЖЦ ПО, в данном случае последовательность процессов: структурный анализ; структурное проектирование; структурное программирование; структурное требование необходимой полноты заключается в контроле присутствия лишних элементов и их исключении из части системы,
- *принцип логической независимости* заключается в необходимости акцента на логическое проектирование для обеспечения независимости от физического проектирования,
- *принцип независимости данных* заключается в том, что модели данных должны быть проанализированы и спроектированы независимо от процессов их логической обработки, а также от их физической структуры и распределения,
- *принцип структурирования данных* заключается в том, что данные должны быть структурированы и иерархически организованы,
- *принцип доступа конечного пользователя* заключается в том, что пользователь должен иметь средства доступа к базе данных, которые он может использовать непосредственно (без программирования).

Соблюдение указанных принципов необходимо при организации работ на начальных этапах ЖЦ независимо от типа разрабатываемого ПО и используемых при этом методологий. Руководствуясь всеми принципами в комплексе, можно на более ранних стадиях разработки понять, что будет представлять из себя создаваемая система, обнаружить недоработки, что, в свою очередь, облегчит работы на последующих этапах ЖЦ и понизит стоимость разработки.

В структурном анализе используются в основном две группы средств, иллюстрирующих функции, выполняемые системой и отношения между данными.

*Первая группа* – модели, реализуемые в виде диаграмм, иллюстрирующие функции, которые система должна выполнять, и связи между этими функциями. На практике наиболее часто используют:

- SADT (Structured Analysis and Design Technique) модели и соответствующие функциональные диаграммы;
- DFD (Data Flow Diagrams) диаграммы потоков данных.

*Вторая группа* – модели, реализуемые в виде диаграмм, моделирующие данные и их отношения.

Подавляющее большинство разработчиков ПО использует ERD (Entity–Relationship Diagrams) диаграммы "сущность-связь".

**Контрольные вопросы**

1. Что такое структурный анализ?
2. В чем состоит суть метода структурного анализа?
3. Что характерно для метода структурного анализа?
4. Какие основные требования предъявляются к системе?
5. Что определяется на этапе структурного анализа?
6. Перечислите и охарактеризуйте основные принципы структурного анализа.
7. Перечислите и охарактеризуйте вспомогательные принципы структурного анализа.

## III. Построение структурно-функциональной модели объекта.

Методология SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. При этом решается задача анализа объекта с позиции компьютеризации, а сама модель используется для определения требований к ПО ИС и проектирования ИС. На основе SADT разработана методология IDEF0 (Icam DEFinition) анализа и проектирования ПО ИС.

Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Основные элементы этой методологии основываются на принципах структурного анализа и включают в себя:

- графическое представление блочного моделирования. В SADT-диаграммах функция объекта отображается в виде блока, а интерфейсы входа/выхода функций представляются дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описываются посредством интерфейсных дуг, которые определяют, когда и каким образом функции выполняются и управляются;
- строгость и точность выполнения правил построения модели. Выполнение правил SADT требует достаточной строгости и точности, не накладывая в то же время чрезмерных ограничений на действия разработчика.

Правила SADT включают:

- ограничение количества блоков (не более 3–6 блоков) на каждом уровне декомпозиции, что опирается на общие требования структурного подхода и обосновывается психологическими особенностями человека в части обработки информации;
- связность диаграмм, заключающаяся в построении системы нумерации блоков;
- уникальность меток и наименований, реализуемая требованием отсутствия повторяющихся имен в диаграммах;
- синтаксические правила для графики, т.е. блоков и дуг;
- разделение входов данных и управлений, которое базируется на правиле определения роли данных в реализации той или иной функции, т.е. будут данные служить в качестве обрабатываемой информации или они используются только для управления обработкой.
- отделение объекта анализа от функции, т.е. исключение влияния текущего состояния объекта на разрабатываемую функциональную модель.

Методология SADT может использоваться для моделирования широкого круга систем и *определения требований и функций*, а затем для *разработки системы, которая удовлетворяет этим требованиям и реализует эти функции. Дл*я уже существующих систем SADT может быть использована для анализа функций, выполняемых системой, а также для указания механизмов, посредством которых они осуществляются.

*Состав функциональной модели*

Основным рабочим элементом при моделировании является диаграмма. Модель SADT объединяет и организует диаграммы в иерархические древовидные структуры, при этом, чем выше уровень диаграммы, тем она менее детализирована. В состав диаграммы входят блоки, изображающие функции моделируемой системы, и дуги, связывающие блоки вместе и изображающие взаимодействия и взаимосвязи между блоками. Вместо одной громоздкой модели используются несколько небольших взаимосвязанных моделей, значения которых взаимодополняют друг друга, делая понятной структуризацию сложного объекта.

Блоки на диаграммах изображаются прямоугольниками и сопровождаются текстами на естественном языке, описывающими функции.

В отличие от других методов структурного анализа в SADT каждая сторона блока имеет вполне определенное особое назначение (рис.3.1):

- левая сторона блока предназначена для **Входов** данных, которые обрабатываются функциональным блоком (преобразовываются, претерпевают количественные или качественные изменения),
- правая сторона блока предназначена для Выходов данных, которые были получены после обработки в функциональном блоке,
- верхняя сторона блока предназначена для Управления. Это тоже данные, но которые используются для управления обработкой в функциональном блоке. Они, как правило, не претерпевают количественных или качественных изменений,
- нижняя сторона блока предназначена для Исполнителей, – элементов системы, программной или аппаратной составляющих ее, которые реализуют или в рамках которых реализуется функциональный блок.

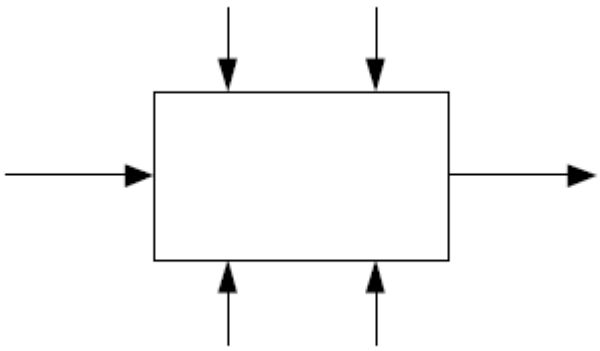


Рисунок 3.1 – Функциональный блок и интерфейсные дуги

Такое обозначение отражает определенные принципы описания системы с позиции функций, когда входная информация после обработки преобразуются в выходную, управляющие данные ограничивают или предписывают условия выполнения тех или иных подфункций (алгоритмов) в функциональном блоке, а исполнители описывают исполнительные механизмы (подпрограмма, модуль, подсистема, оператор, элементы

аппаратной части системы и т.д.), в составе какого реального компонента ИС и за счет чего выполняются преобразования в функциональном блоке.

Таким образом, дуги в SADT представляют поименованные, т.е. маркированные текстами на естественном языке, потоки данных, которые могут представлять в рамках модели различные предметы или элементы системы (объекта) (рис. 3.2).


Рисунок 3.2 – Пример представления функционального блока и интерфейсных дуг

Блоки на диаграмме размещаются по ступенчатой схеме в соответствии с их доминированием, которое понимается как влияние, оказываемое одним блоком на другие. Кроме того, блоки должны быть пронумерованы, например, в соответствии с их доминированием. Номера блоков служат однозначными идентификаторами для функций и формируют функциональную иерархическую структуру или иерархию модели. Взаимовлияние блоков может выражаться либо в пересылке Выходных данных к другому функциональному блоку для дальнейшего преобразования, либо в выработке управляющей информации, предписывающей, что именно должен делать другой функциональный блок.

Таким образом, **диаграммы SADT являются предписывающими диаграммами**, описывающими как преобразования между Входом и Выходом, так и предписывающие правила этих преобразований.

В SADT требуются только пять типов взаимосвязей между блоками для описания их отношений: **Управление, Вход, Управленческая Обратная Связь, Входная Обратная Связь, Выход-Исполнитель**.

Отношения Управления и Входа являются простейшими, поскольку они отражают интуитивно очевидные прямые воздействия.

Отношение Управления возникает тогда, когда Выход одного блока непосредственно влияет на блок с меньшим доминированием (Рис. 3.3).

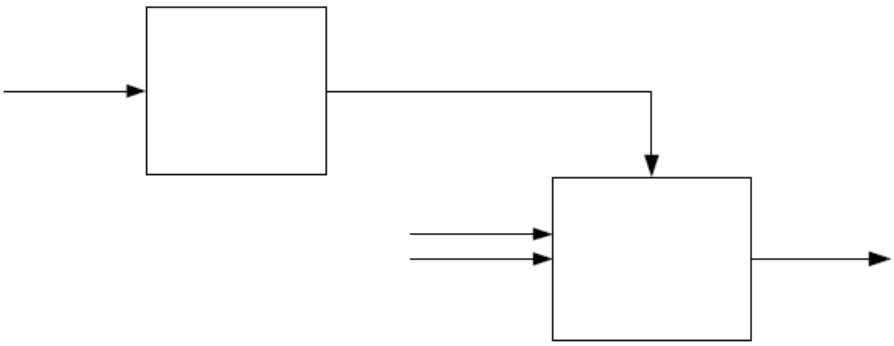


Рисунок 3.3 – Схема отношения Управление – Вход

Отношение Входа возникает, когда Выход одного блока становится Входом для блока с меньшим доминированием (Рис. 3.4).

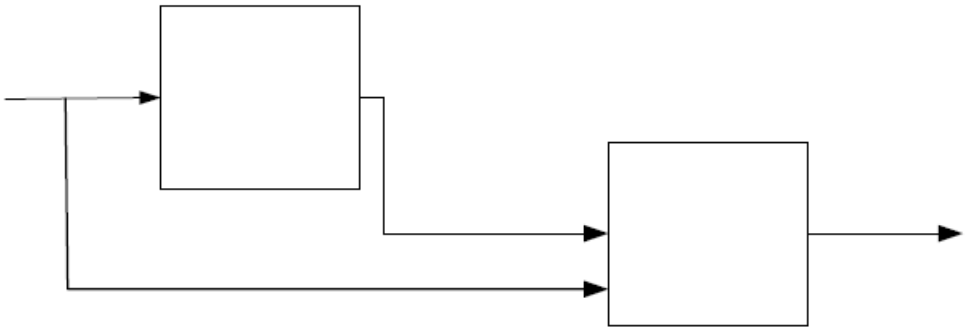


Рисунок 3.4 – Схема отношения Вход

Обратные связи более сложны, поскольку они отражают итерацию или рекурсию. Выходы из одного функционального блока влияют на будущее выполнение других функций, что впоследствии влияет на исходный функциональный блок.

Управленческая Обратная Связь возникает, когда Выход некоторого блока влияет на блок с большим доминированием (Рис.3.5).

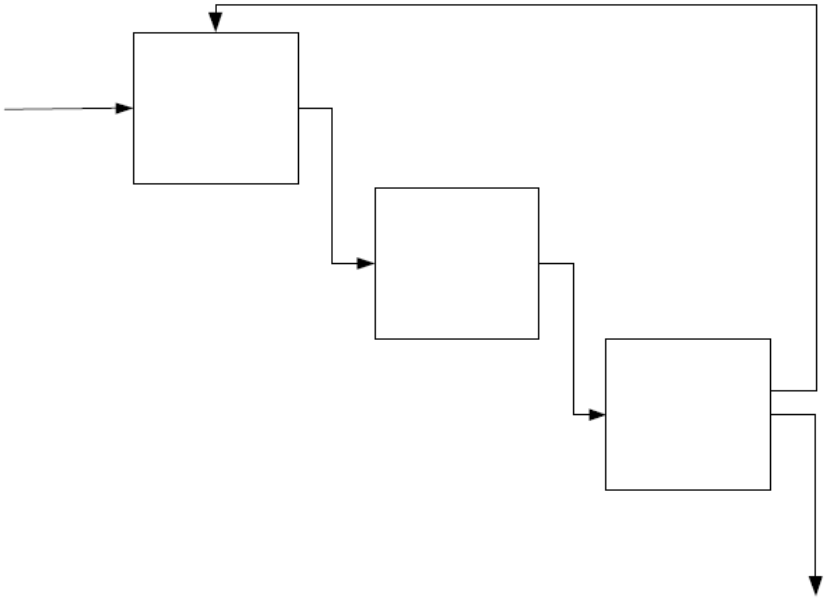


Рисунок 3.5 – Схема отношения Управленческая Обратная Связь

Отношение Входной Обратной Связи имеет место, когда Выход одного блока становится Входом другого блока с большим доминированием (Рис.3.6).

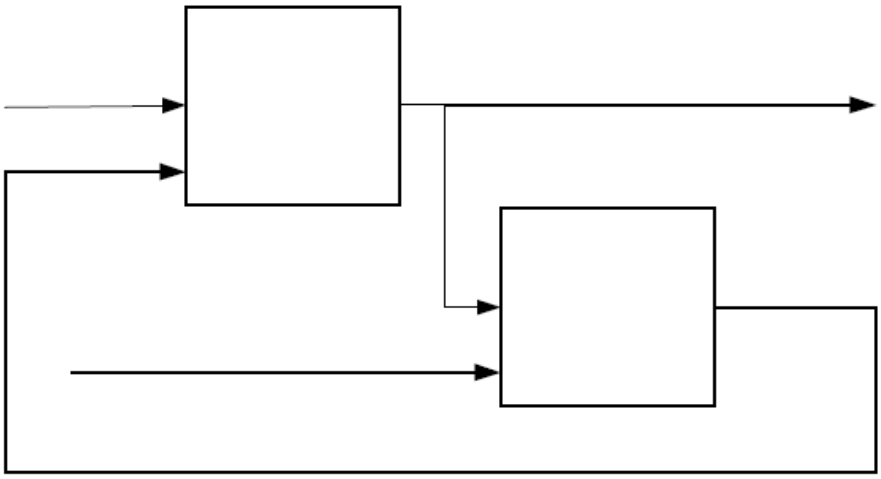


Рисунок 3.6 – Схема отношения Входная Обратная Связь

Отношения Выход-Исполнитель встречаются нечасто и представляют особый интерес. Они отражают ситуацию, при которой Выход одного функционального блока становится средством достижения цели другого функционального блока (Рис.3.7).

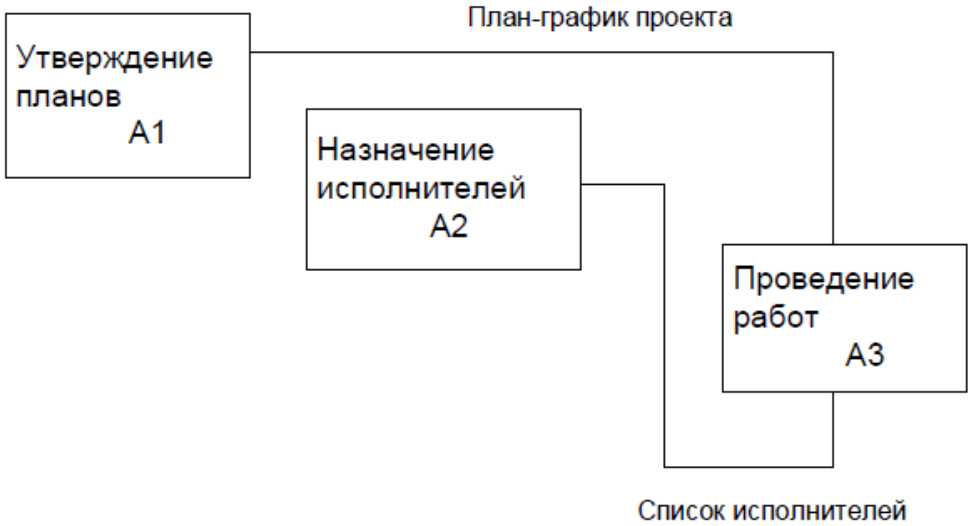



Рисунок 3.7 – Схема отношения Выход– Исполнитель

Дуги SADT, как правило, изображают потоки данных, поэтому они могут разветвляться и соединяться вместе различным образом. Разветвления дуги означают, что часть ее содержимого (или весь поток) может появиться в каждом ответвлении дуги. Дуга всегда помечается до разветвления, чтобы дать название всему набору (Рис.3.8).

Кроме того, каждая ветвь дуги может быть помечена в соответствии со следующими правилами:

- считается, что непомеченная ветвь содержит все предметы, указанные в метке перед разветвлением;
- каждая метка ветви уточняет, что именно содержит эта ветвь.


Рисунок 3.8 – Разветвление данных

Слияние дуг указывает, что содержимое каждой ветви участвует в формировании после слияния объединенной дуги. После слияния дуга всегда помечается для указания нового потока (Рис. 3.9).

Кроме того, каждая ветвь перед слиянием может помечаться в соответствии со следующими правилами:

 считается, что непомеченные ветви содержат все данные, указанные в общей метке после слияния;

 каждая метка ветви уточняет, что именно содержит эта ветвь.

Рисунок 3.9 – Слияние данных

Результатом применения методологии SADT является модель, которая состоит из:

- диаграмм,
- фрагментов текстов,
- глоссария, имеющего ссылки друг на друга.

Диаграммы – главные компоненты модели, все функции ИС и интерфейсы на них представлены как блоки и дуги.

**Одной из наиболее важных особенностей методологи SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель**. На рисунке 3.10, где приведены четыре диаграммы и их взаимосвязи, показана структура SADT-модели. Каждый компонент модели может быть декомпозирован на другой диаграмме. Каждая диаграмма иллюстрирует "внутреннее строение" блока на родительской диаграмме.

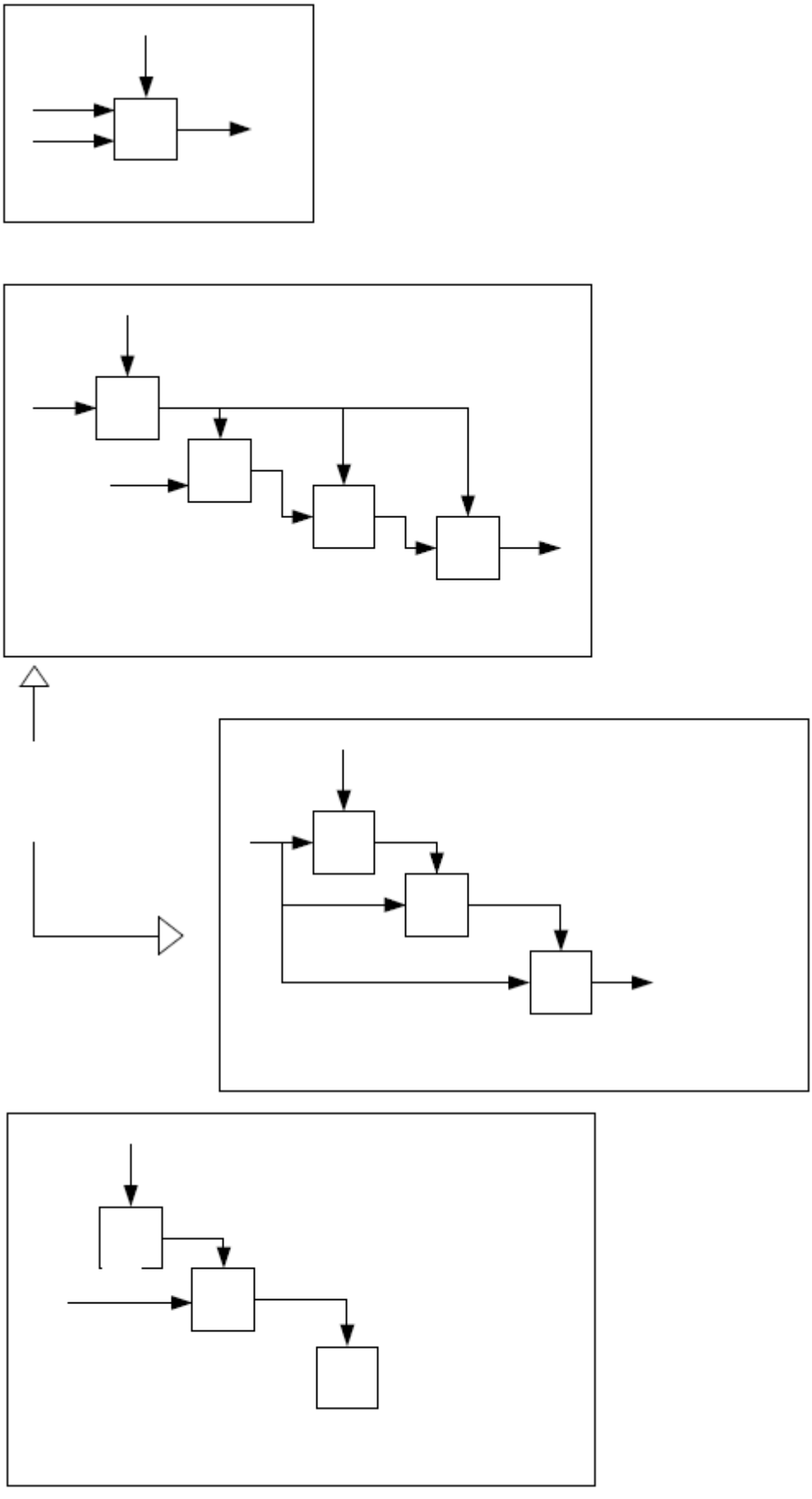


Рис. 3.10 – Структура SADT-модели. Декомпозиция диаграмм

*Иерархия диаграмм*

Построение SADT-модели начинается с представления всей системы в виде простейшей компоненты – одного блока и дуг, изображающих интерфейсы с функциями вне системы.

Поскольку единственный блок представляет всю систему как единое целое, имя, указанное в блоке, является общим. Это верно и для интерфейсных дуг – они также представляют полный набор внешних интерфейсов системы в целом.

Затем блок, который представляет систему в качестве единого модуля, детализируется на другой диаграмме с помощью нескольких блоков, соединенных интерфейсными дугами. Эти блоки представляют основные подфункции исходной функции. Данная декомпозиция выявляет полный набор подфункций, каждая из которых представлена как блок, границы которого определены интерфейсными дугами. Каждая из этих подфункций может быть декомпозирована подобным образом для более детального представления.

Во всех случаях каждая подфункция может содержать только те элементы, которые входят в исходную функцию.

Кроме того, модель не может опустить какие-либо элементы, т.е., как уже отмечалось, родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить, и из него не может быть ничего удалено.

Некоторые дуги присоединены к блокам диаграммы обоими концами, у других же один конец остается неприсоединенным.

Неприсоединенные дуги соответствуют входам, управлениям и выходам родительского блока. Источник или получатель этих пограничных дуг может быть обнаружен только на родительской диаграмме. Неприсоединенные концы должны соответствовать дугам на исходной диаграмме. Все граничные дуги должны продолжаться на родительской диаграмме, чтобы она была полной и непротиворечивой.

Модель SADT представляет собой серию диаграмм с сопроводительной документацией, разбивающих сложный объект на составные части, которые представлены в виде блоков. Детали каждого из основных блоков показаны в виде блоков на других диаграммах. Каждая детальная диаграмма является декомпозицией блока из более общей диаграммы. На каждом шаге декомпозиции более общая диаграмма называется родительской для более детальной диаграммы.

Дуги, входящие в блок и выходящие из него на диаграмме верхнего уровня, являются точно теми же самыми, что и дуги, входящие в диаграмму нижнего уровня и выходящие из нее, потому что блок и диаграмма представляют одну и ту же часть системы.

На рисунках 3.12 – 3.13 дополнительно схематично представлены различные варианты выполнения функций и соединения дуг с блоками.

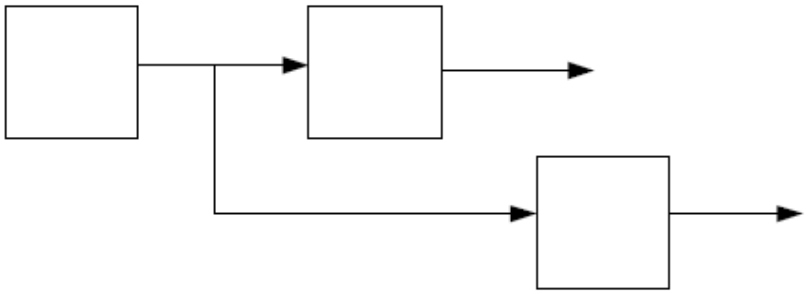


Рисунок 3.12 – Одновременное выполнение

На SADT–диаграммах не указаны явно ни последовательность, ни время. Обратные связи, итерации, продолжающиеся процессы и перекрывающиеся (по времени) функции могут быть изображены с помощью дуг. Обратные связи могут выступать в виде комментариев, замечаний, исправлений и т.д. (рисунок 3.13).

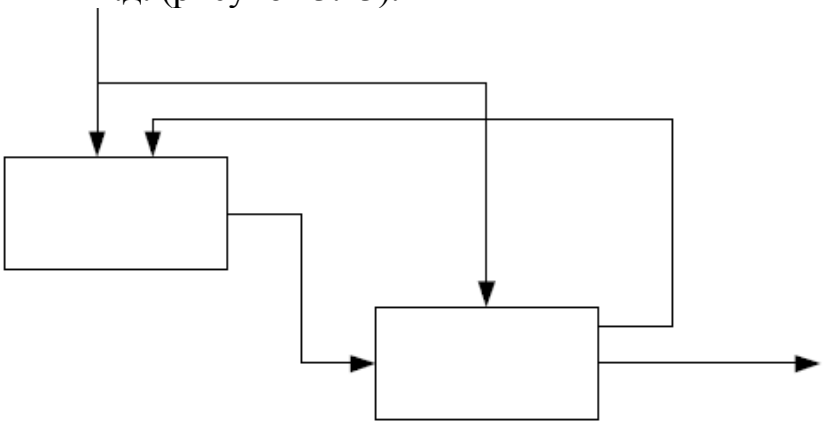


Рисунок 3.13 – Пример обратной святи

Как было отмечено, механизмы (дуги с нижней стороны) показывают средства, с помощью которых осуществляется выполнение функций. Механизм может быть человеком, компьютером или любым другим устройством, которое помогает выполнять данную функцию (рисунок 3.14).

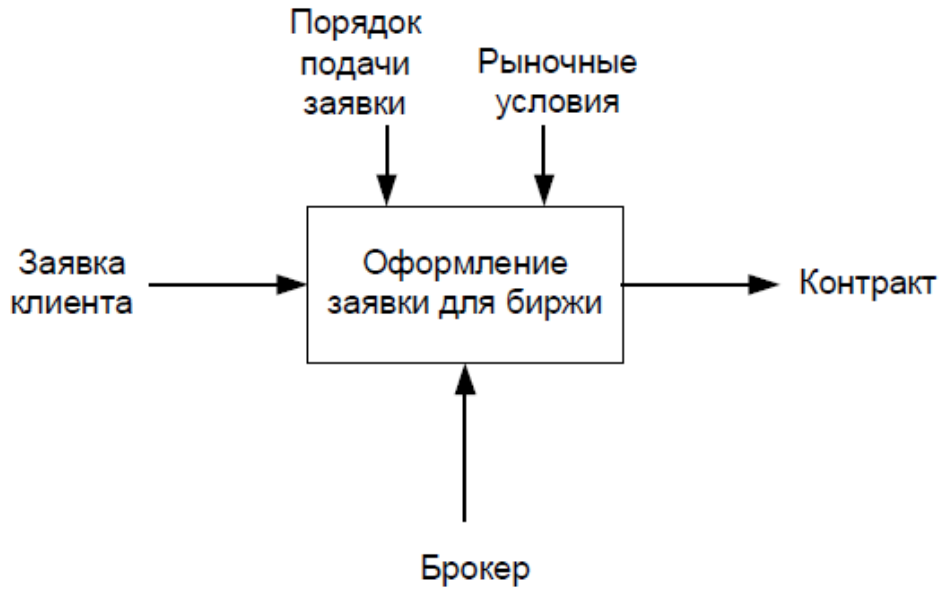



Рисунок 3.14 – Пример исполнителя (механизма)

Каждый блок на диаграмме имеет свой номер. Блок любой диаграммы может быть далее описан диаграммой нижнего уровня, которая, в свою очередь, может быть далее детализирована с помощью необходимого числа диаграмм. Таким образом, формируется иерархия диаграмм.

Для того, чтобы указать положение любой диаграммы или блока в иерархии, используются номера диаграмм. Например, A21 является диаграммой, которая детализирует блок 1 на диаграмме A2. Аналогично, A2 детализирует блок 2 на диаграмме A0, которая является самой верхней диаграммой модели. На рисунке 3.15 показано типичное дерево диаграмм.


Рисунок 3.15 – Иерархия диаграмм

При создании модели одна и та же диаграмма чертится несколько раз, что создает различные ее варианты. Чтобы различать различные версии одной и той же диаграммы, в SADT используется схема контроля конфигурации диаграмм, основанная на их номерах. Если диаграмма замещает более старый вариант, то предыдущий номер помещается в скобках для указания связи с предыдущей работой.

Пример SADT-диаграммы, моделирующей деятельность компании, занимающейся распределением товаров по заказам, приведен на рис. 3.16.

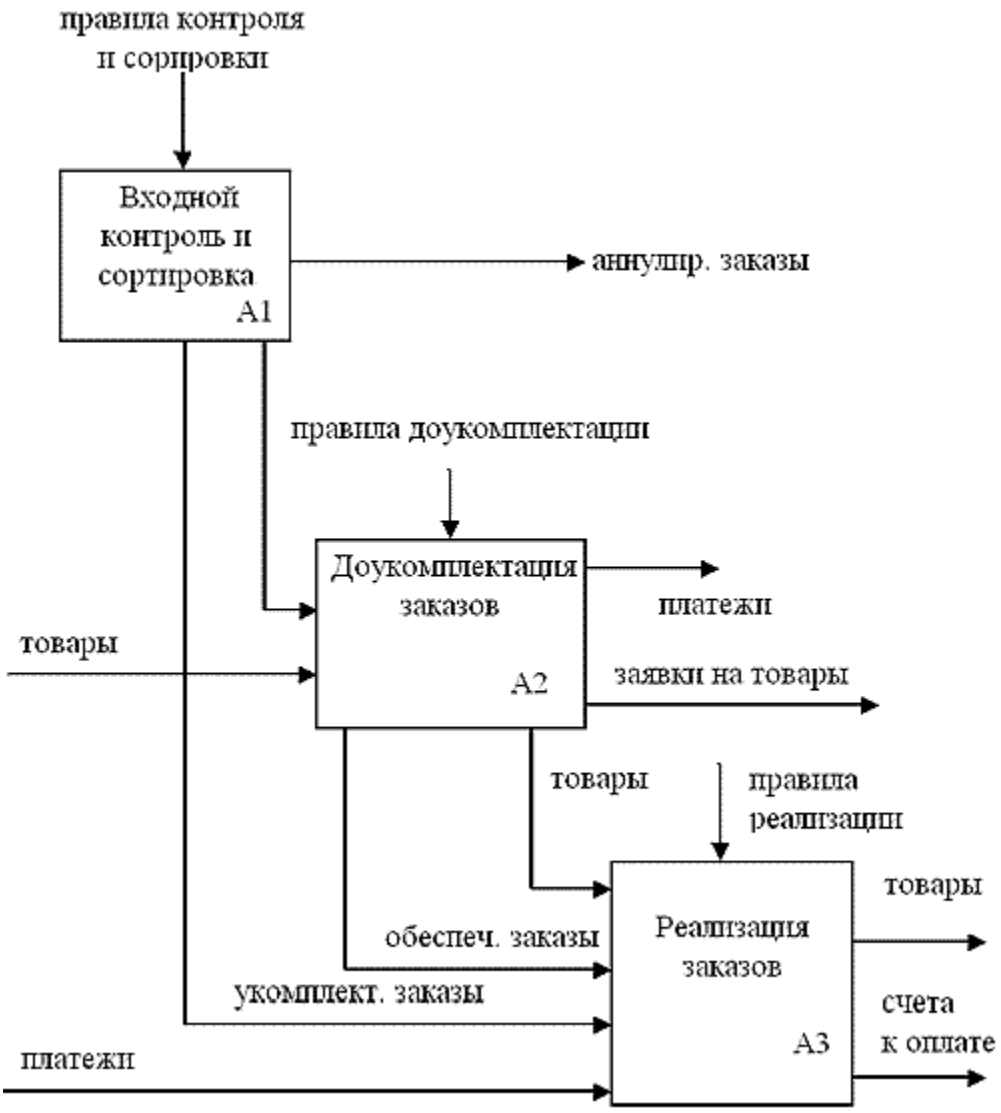



Рисунок 3.16 – Пример SADT–диаграммы, моделирующей деятельность компании, занимающейся распределением товаров по заказам

**SADT, как и другие методологии проектирования, целесообразно использовать на ранних этапах ЖЦ: для понимания системы до ее воплощения**. SADT позволяет сократить дорогостоящие ошибки на ранних этапах создания системы, улучшить контакт между пользователями и разработчиками, сгладить переход от анализа к проектированию. Несомненное достоинство SADT заключается в том, что она легко отражает такие характеристики как управление, обратная связь и исполнители.

**Контрольные вопросы**

1. Перечислите основные принципы и правила SADT.
2. Приведите и опишите элементы диаграммы SADT (блок и интерфейсная дуга).
3. Опишите характер взаимосвязи между блоками типа «Вход» и приведите пример элемента диаграммы.
4. Опишите характер взаимосвязи между блоками типа «Управление» и приведите пример элемента диаграммы.

5. Опишите характер взаимосвязи между блоками типа «Управленческая Обратная Связь» и приведите пример элемента диаграммы.

6. Опишите характер взаимосвязи между блоками типа «Входная Обратная Связь» и приведите пример элемента диаграммы.

7. Опишите характер взаимосвязи между блоками типа «Выход–Исполнитель» и приведите пример элемента диаграммы.

8. Опишите правила разветвления и слияния потоков данных и приведите примеры элементов диаграмм.

9. Опишите соотношения между родительской и дочерними диаграммами в модели SADT.

10. Приведите пример декомпозиции диаграмм.

11. Перечислите особенности построения иерархии диаграмм в модели SADT.

## IV. Модель DFD.

4.1. Построение диаграмм потоков данных.

**Моделирование потоков данных** (процессов)

В основе методики лежит построение модели анализируемого объекта и, соответственно, разрабатываемой системы с позиций движения, распределения, обработки и представления данных. Модель системы определяется как иерархия диаграмм потоков данных (DFD), описывающих процесс преобразования информации от ее ввода в систему до выдачи пользователю. Модель содержит два уровня представления:

1. Первый уровень представления составляют диаграммы верхних уровней иерархии (контекстные диаграммы), которые определяют основные процессы или подсистемы ИС с внешними входами и выходами.

2. Второй уровень представления составляют собственно диаграммы потоков данных и процессов их обработки. Они детализируют контекстные диаграммы. Такая декомпозиция продолжается, создавая многоуровневую иерархию диаграмм, до тех пор, пока не будет достигнут такой уровень декомпозиции, на котором процесс становятся элементарными и детализировать их далее невозможно. На этом уровне декомпозиции выполняются миниспецификации.

Основными компонентами диаграмм потоков данных являются:

- внешние сущности;
- системы/подсистемы;
- процессы;
- накопители данных;
- потоки данных.

Внешние сущности, характеризующие источники информации, порождают потоки данных, которые переносят информацию к подсистемам или процессам. Те в свою очередь преобразуют информацию и порождают новые потоки данных, которые переносят информацию к другим процессам

или подсистемам, накопителям данных или потребителям информации – внешним сущностям.

**Внешняя сущность**

*Внешняя сущность* представляет собой материальный предмет или физическое лицо, представляющее собой источник или приемник информации, например, оператор, внешняя система и др. Определение некоторого объекта или системы в качестве внешней сущности указывает на то, что она находится за пределами границ анализируемой ИС. В процессе анализа некоторые внешние сущности могут быть перенесены внутрь диаграммы анализируемой ИС, если это необходимо, или, наоборот, часть процессов ИС может быть вынесена за пределы диаграммы и представлена как внешняя сущность.

Внешняя сущность обозначается квадратом (рисунок 4.1).

Рис. 4.1 – Внешняя сущность

**Системы и подсистемы**

При формировании контекстных диаграмм для сложной ИС на первом уровне представления часто используют такие элементы, как система и подсистема.

Система или подсистема на контекстной диаграмме изображается следующим образом (рисунок 4.2).

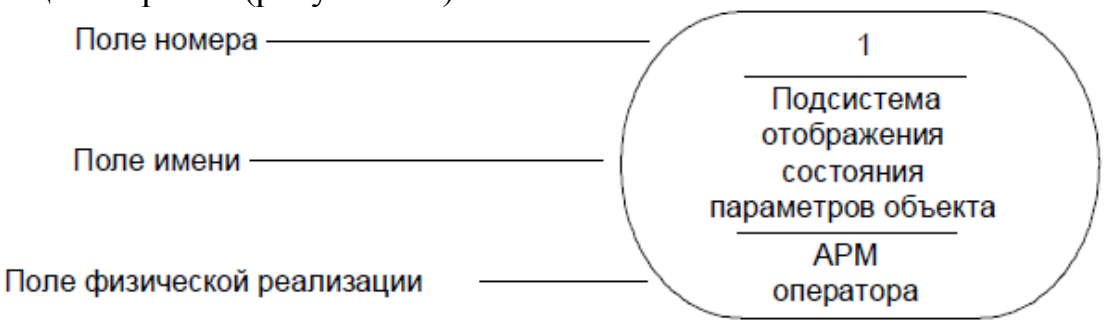



Рисунок 4.2 – Подсистема

Номер подсистемы служит для ее идентификации. В поле имени вводится наименование подсистемы в виде предложения с подлежащим и соответствующими определениями и дополнениями.

**Процессы**

*Процесс* характеризует конкретное преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом. Например, преобразование входного потока данных о значениях сигналов с датчиков в выходной поток вычисленных физических величин (давление, температура). Изображение процесса на диаграмме потоков данных показано на рисунке 4.3.

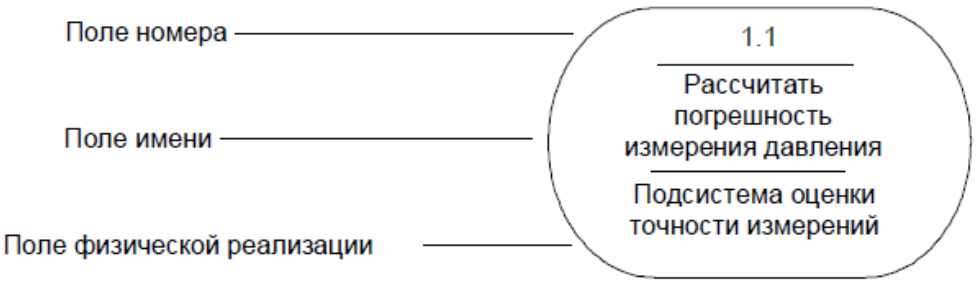



Рисунок 4.3 – Процесс

Номер процесса служит для его идентификации. В поле имени вводится наименование процесса в виде предложения с активным глаголом в неопределенной форме (вычислить, рассчитать, проверить, определить, создать, получить), за которым следуют существительные в винительном падеже, например:

- «Рассчитать давление на объекте»;
- «Вычислить среднее значение температуры».

Фраза, характеризующая процесс, должна недвусмысленным образом определять конкретную операцию или совокупность операций, которым подвергается входящий поток данных.

Информация в поле физической реализации показывает, какое подразделение организации, программа или аппаратное устройство выполняет данный процесс.

**Накопители данных**

*Накопитель данных* представляет собой абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь, причем способы помещения и извлечения могут быть любыми.

Накопитель данных может быть реализован физически в виде базы данных, файла, таблицы и т.д. На диаграмме потоков данных накопитель данных изображается, как показано на рисунке 4.4.

Рисунок 4.4 – Накопитель данных

Пример:


Накопитель данных идентифицируется буквой "D" и произвольным числом. Имя накопителя выбирается из соображения наибольшей информативности для проектировщика. Накопитель данных в общем случае является прообразом будущей базы данных и описание хранящихся в нем данных должно быть увязано с информационной моделью.

**Поток данных**

*Поток данных* определяет информацию, передаваемую через некоторое соединение от источника к приемнику. Реальный поток данных может быть информацией, передаваемой по линии связи между датчиком и микроконтроллером, по кабелю между двумя вычислительными устройствами и т.д.

На диаграмме поток данных изображается в виде линии со стрелкой, которая показывает направление потока (рисунок 4.5). Каждый поток данных должен иметь имя, отражающее его содержание.

| | |
|---|---|
| Объект, над которым выполняется действие. Может быть информационным (логическим) или управляющим. (Управляющие потоки обозначаются пунктирной линией со стрелкой). | Имя объекта |

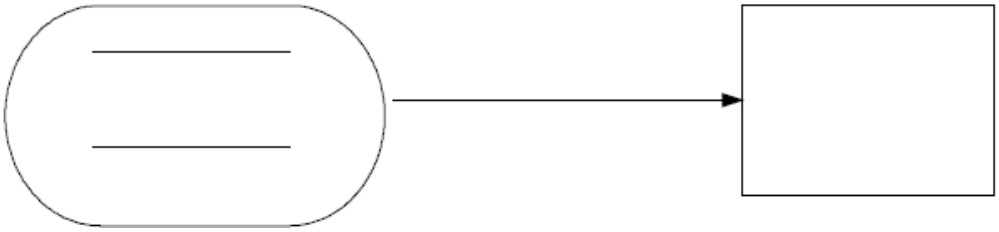


Рисунок 4.5 – Поток данных

Как уже было сказано ранее, анализ объекта и описание ИС, его автоматизирующей, с помощью модели DFD осуществляется в виде иерархии диаграмм потоков данных.

### 4.2. Построение иерархии диаграмм потоков данных

Первым уровенем представления модели при построении иерархии DFD является построение контекстных диаграмм. Модель простой ИС может быть представлена контекстной диаграммой в виде одной системы как единого целого. Модель в этом случае может выглядеть как контекстная диаграмма со звездообразной топологией, в центре которой находится так называемый главный процесс, соединенный с приемниками и источниками информации, посредством которых с системой взаимодействуют пользователи и другие внешние системы (рис. 4.6).


Рисунок 4.6 – Поток данных

Если же для сложной системы ограничиться единственной контекстной диаграммой, то она будет содержать слишком большое количество источников и приемников информации, которые трудно расположить на листе бумаги, и кроме того, единственный главный процесс не раскрывает структуры распределенной системы.

Признаками сложности ИС могут быть:

- наличие большого количества внешних сущностей (десять и более);
- распределенная природа системы;
- многофункциональность системы с уже сложившейся или выявленной группировкой функций в отдельные подсистемы.

Для сложных ИС строится иерархия контекстных диаграмм. Для модели сложной ИС контекстная диаграмма верхнего уровня содержит не единственный главный процесс, а набор подсистем, соединенных потоками данных. Далее эта диаграмма последовательно декомпозируется на ряд подсистем нижнего уровня и в итоге может быть представлена в виде иерархии контекстных диаграмм. При этом контекстные диаграммы следующего уровня детализируют контекст и структуру. Иерархия контекстных диаграмм определяет взаимодействие основных функциональных подсистем проектируемой ИС как между собой, так и с внешними входными и выходными потоками данных и внешними объектами (источниками и приемниками информации), с которыми взаимодействует ИС.

Разработка контекстных диаграмм решает проблему строгого определения функциональной структуры ИС на самой ранней стадии ее проектирования, что особенно важно для сложных многофункциональных систем, в разработке которых участвуют разные организации и коллективы разработчиков.

В заключении полученную иерархию контекстных диаграмм проверяют на полноту исходных данных об объектах системы и изолированность объектов (отсутствие информационных связей с другими объектами).

Второй уровень представления модели – уровень диаграмм потоков данных и процессов их обработки.

Каждая подсистема, которая присутствует на контекстных диаграммах, детализируется при помощи DFD –диаграмм и, как правило, представляется в виде процессов, связанных между собой потоками данных.

Каждый процесс на диаграмме, в свою очередь, может быть детализирован при помощи DFD –диаграмм или, если дальнейшая детализация на подпроцессы невозможна или нецелесообразна, представлен в виде спецификации (т.е. в виде текстового описания).

При детализации должны выполняться следующие правила [2]:

- правило балансировки – означает, что при детализации подсистемы или процесса детализирующая диаграмма в качестве внешних источников/приемников данных может иметь только те компоненты (подсистемы, процессы, внешние сущности, накопители данных), с которыми

имеет информационную связь детализируемая подсистема или процесс на родительской диаграмме;

- правило нумерации – означает, что при детализации процессов должна поддерживаться их иерархическая нумерация. Например, процессы, детализирующие процесс с номером 12, получают номера 12.1, 12.2, 12.3.

Спецификация является конечной вершиной иерархии DFD. Решение о завершении детализации процесса и использовании спецификации принимается исходя из следующих критериев:

- наличия у процесса относительно небольшого количества входных и выходных потоков данных (2–3 потока);
- возможности описания преобразования данных процессом в виде последовательного алгоритма;
- выполнения процессом единственной логической функции преобразования входной информации в выходную;
- возможности и описания логики процесса при помощи спецификации небольшого объема (не более 20–30 строк).

Спецификация (описание логики процесса) должна формулировать его основные функции таким образом, чтобы в дальнейшем специалист, выполняющий реализацию проекта, смог выполнить их или разработать соответствующую программу.

Важно отметить, что до построения иерархии DFD-диаграмм для детализации процессов необходимо определить содержание всех потоков и накопителей данных, которое описывается при помощи структур данных.

Структуры данных конструируются из элементов данных и могут содержать альтернативы, условные вхождения и итерации.

Условное вхождение означает, что данный компонент может отсутствовать в структуре.

Альтернатива означает, что в структуру может входить один из перечисленных элементов.

Итерация означает вхождение любого числа элементов в указанном диапазоне.

Для каждого элемента данных может указываться его тип (непрерывные или дискретные данные). Для непрерывных данных может указываться единица измерения, диапазон значений, точность представления и форма физического кодирования. Для дискретных данных может указываться таблица допустимых значений.

После построения законченной модели системы ее необходимо верифицировать, то есть проверить на полноту и согласованность. В полной модели все ее объекты (подсистемы, процессы, потоки данных) должны быть подробно описаны и детализированы. Выявленные недетализированные объекты следует подчинить следующему правилу: для всех потоков данных и накопителей данных должно выполняться правило сохранения

информации: все поступающие куда–либо данные должны быть считаны, а все считываемые данные должны быть записаны.

**Контрольные вопросы**

1. Охарактеризуйте уровни представления модели диаграмм потоков данных (DFD).
2. Перечислите основные элементы DFD.
3. Опишите и приведите пример элемента DFD внешняя сущность.
4. Опишите и приведите пример элемента DFD система/подсистема.
5. Опишите и приведите пример элемента DFD процесс.
6. Опишите и приведите пример элемента DFD накопитель данных.
7. Опишите и приведите пример элемента DFD поток данных.
8. Приведите основные правила построения иерархии контекстных диаграмм.
9. Приведите основные правила и особенности построения иерархии диаграмм потоков данных.
10. Приведите пример контекстной диаграммы.
11. Приведите пример иерархии диаграмм потоков данных.

## V. Методика "сущность-связь" построения структур баз данных

Указав накопитель данных в построенной DFD-модели, который в общем случае является прообразом будущей базы данных, в последующем нужно перейти к проектированию БД, которая является одними из важнейших элементов информационной системы. Самый ответственный момент в создании базы данных – это построение ее структуры. Если структура базы данных построена правильно, то это положительно отразится на характеристике всей БД, особенно на надежности и скорости выборки данных.

Рассмотрим методику создания структурного каркаса базы данных, т.е. ключевого звена в проектировании информационных систем.

Для начала вспомним термины, которые приняты при проектировании БД.

***База данных*** – некоторый упорядоченный банк информации, предназначенный для хранения необходимых данных. С***истема управления базой данных (СУБД)*** – программный комплекс, отвечающий за работу с базой данных, как-то: ее сортировка, извлечение информации по запросу пользователя и некоторые другие действия.

***Поле*** – самая маленькая единица хранения информации, из которой строятся другие более крупные единицы данных – ***записи***. В поле обычно хранится один атрибут для описания некоего объекта, информация о котором хранится в базе данных.

***Запись*** – набор полей, объединенных в единую структуру, на значение которой – хранить информацию об экземпляре интересующего нас объекта. Иногда в литературе запись называют кортежем.

***Таблица*** – набор записей, в который входит информация обо всех экземплярах объекта. Обычно каждая таблица сохраняется в виде отдельного файла.

На рис. 5.1 представлена таблица БД.

**Один столбец соответствует одному полю** — *Поле записи*

| | **Имя 1** | **Имя 2** | **Имя 3** | **Имя 4** |
|---|---|---|---|---|
| **Одна строка соответствует одной записи** | *ФИО* | *Начислено* | *Удержано* | *Сумма* |
| | Иванов В.А. | 8900 | 1200 | 7700 |
| *Запись БД* | Дроздов И.И. | 12300 | 2000 | 10300 |
| | Жариков А.А. | 10870 | 1340 | 9530 |
| | … | … | … | … |

Рисунок 5.1 – Таблица базы данных

В качестве хорошего примера таблицы в базе данных можно привести ведомость выдачи зарплаты, на которой удобно демонстрировать принцип работы СУБД. За сквозной пример можно принять систему подсчета зарплаты.

Для начала примем следующее ***техническое задание***:

**создать систему хранения и вычисления заработной платы с учетом различных премий, надбавок и налогов.**

В простейшем случае мы можем создать таблицу, состоящую из четырех полей:

"Фамилия", "Начислено", "Удержано", "Сумма", и такой "плоской" базы данных нам бы вполне могло хватить. Но на нашем гипотетическом предприятии несколько тысяч работников, и если теперь спроецировать полученную таблицу на реальный мир, то становится ясно, что она не свободна от недостатков. Так, например, у каждого работника может быть много различных начислений, равно как и налогов. Исходя из этого, мы вынуждены делать новую запись на каждое начисление и удержание. Результат не замедлит сказаться: файл, в котором хранится наша таблица, настолько увеличился, что найти в нем теперь нужную запись становится весьма не просто. А постоянное дублирование информации в поле "Фамилия" просто выводит из себя. А поле "Сумма"? Из какой записи его считывать? Из первой, второй или может последней? Придется применить другой подход для решения нашей задачи. Первое, что приходит на ум, это преобразование нашей плоской базы данных в реляционную.

Реляционная база данных представляет собой набор таблиц, которые содержат всю необходимую информацию. Чтобы преобразовать старую базу данных в реляционную БД, необходимо произвести нормализацию, то есть разбиение универсальной таблицы на несколько простых. Для правильной нормализации можно использовать методику "сущность-связь".

**Сущности** – это объекты, которыми можно оперировать при помощи СУБД и которые интересуют разработчика.

**Связи** – показывают, как эти сущности взаимодействуют между собой.

Например, для БД «ведомость выдачи зарплаты» условие: **"Работник** *получает* **Деньги",** с использованием понятий сущностей и связей будет определено "**Работник**" и "**Деньги**" – сущности, а "*получает*" – связь, описывающая их взаимодействие( связи зачастую выглядят как глаголы с предлогами или без).

Определяя сущности, необходимо также выделить те их атрибуты, которые будут использованы в базе данных.

**Атрибуты** – это свойства, которыми сущности обладают. Так, для сущности "Работник" нас будут интересовать такие атрибуты, как "Фамилия Имя Отчество" ("ФИО"), "Табельный номер" и "Должность".

Правильность структуры базы данных будет во многом зависеть от того, как подробно и корректно будут описаны сущности, связи между ними и атрибуты сущностей.

Следующее, что необходимо определить – каковы степени связей, которые мы выделили.

**Степень связи** – абстрактная характеристика, показывающая, сколько элементов одной сущности связано со сколькими элементами другой сущности в рамках решения конкретной задачи.

Степень связи может принимать значения:

- **Один к одному – 1 : 1** – один элемент одной сущности связан только с одним элементом другой сущности;
- **Один к многим – 1 : М** – один элемент одной сущности связан со многими элементами другой сущности;
- **Многие к одному – М : 1** – многие элементы одной сущности связаны с одним элементом другой сущности;
- **Многие ко многим – М : М** – многие элементы одной сущности связаны с многими элементами другой сущности.

Например, степень связи будет 1:1, то есть один работник может получить лишь одну сумму денег, и наоборот, запланированная для этого человека сумма денег может быть получена только им одним.

Кроме степени связи в проектировании БД есть, еще одна важная характеристика, называемая классом принадлежности.

**Класс принадлежности** – характеристика, определяющая, обязательно или нет все элементы сущности должны быть связаны с элементами другой сущности. Классы принадлежностей:

- Обязательный – (**О**);
- Необязательный – (**N**)

Например, можно рассуждать следующим образом: поскольку человек делает работу, он должен получить зарплату, значит, все элементы сущности «Работник» связаны с сущностью "Деньги". Из этого следует, что класс принадлежности сущности «Работник» обязательный – **О**.

Для сущности "Деньги" класс принадлежности также будет обязательный – **О**, поскольку каждая сумма, планируемая к выдаче, принадлежит какому–то работнику.

Можно представить все вышесказанное в виде диаграммы (см. рис. 5.2*)*.

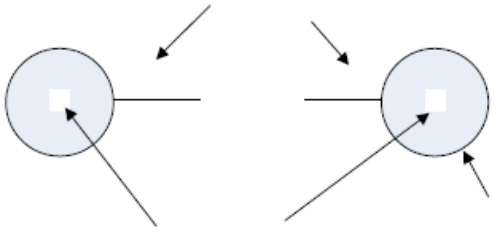


Рисунок 5.2 – Диаграмма для конструкции **"Работник** *получает* **Деньги"**

Прежде чем построить полную диаграмму базы данных, давайте зададимся вопросом: а что будет, если несколько работников носят одну и ту же фамилию? Очевидно, что ИС может ошибиться и предоставить нам данные о другом работнике.

Для того, чтобы различать записи с одинаковыми атрибутами, используют понятие *ключа*.

*Ключ(первичный ключ)* помогает однозначно выделить именно ту запись в БД, которая нам нужна. Для этого в таблице выбирается одно или несколько полей, по содержимому которого (которых) можно с полной уверенностью установить, ту ли запись мы нашли.

*Ключ* из нескольких полей называют *композитным,* то есть составным. Нахождение *ключа* для таблицы – дело довольно трудное.

Поле "ФИО" может содержать одну и ту же фамилию в разных записях, если есть однофамильцы, следовательно, оно *не может* служить *ключом*. А вот поле "Табельный номер", без сомнения, подходит на эту роль.

В самом деле: на каком предприятии разные работники могут иметь одинаковые табельные номера?

Теперь ИС будет сортировать записи этой таблицы по степени возрастания этих номеров.

Если вам неудобен такой порядок и вы предпочитаете сортировку по фамилиям, то можете задать альтернативный ключ сортировки, называемый *вторичным индексом*. Таким ключом может служить поле "ФИО".

Используя вторичный индекс, ИС может извлекать данные в алфавитном порядке, не меняя физического расположения записей в таблице.

Вернемся к диаграмме. Усложним задачу. Пусть сумма выплаты работнику будет вычисляться из нескольких сумм, каждая из которых считается за каждую выполненную работу отдельно. Видов работ также может быть несколько.

Таким образом, мы имеем четыре сущности:
**"Работник"** – информация о сотруднике,
**"Виды работ"** – типовые работы, выполняемые на предприятии, и расценки,
**"Выполненные работы"** – список закрытых нарядов,
**"Баланс"** – заработанная сумма, налоги и итоговая сумма для получения.
Атрибуты сущностей:
**"Работник":** Фамилия, Имя, Отчество, Табельный номер, Должность.
**"Виды работ" :** Наименование, Стоимость.
**"Выполненные работы"**: Наименование вида, номер закрытого наряда, объект.
**"Баланс":** Сумма выплат.
Детализируя связи для диаграммы, получаем следующие возможные конструкции:
**1. Работник** *выполняет* разные **Виды работ.**
**2. Выполненные работы** *состоят из* разных **Видов работ.**
**3. Работник** *участвовал в* **Выполнении работ (Выполненные работы).**
**4. Работник** *имеет* **Баланс выплаты.**
**5. Выполненные работы** *составляют* **Баланс.**

Необходимо анализировать возможные конструкции на предмет избыточности. Для того чтобы по возможности уменьшить количество сущностей.

Выражения 4 и 5 мы отбрасываем, поскольку все суммы денег мы можем легко вычислить из расценок на работы и закрытых нарядов:

- каждая отдельная сумма равна расценке на работу, умноженной на количество единиц выполненной работы;
- общая сумма складывается из полученных отдельных сумм;
- налог начисляется с общей суммы; сумма к выдаче получается из общей заработанной суммы за вычетом налогов.

В таком случае от сущности "**Баланс**" мы также можем отказаться, тем более что итоговые суммы для ведомости выдачи зарплаты мы считаем один раз и не имеет смысла хранить конечный результат, если в базе есть все компоненты для его получения.

Тогда структура диаграммы будет выглядеть следующим образом (см. рис. 5.3)

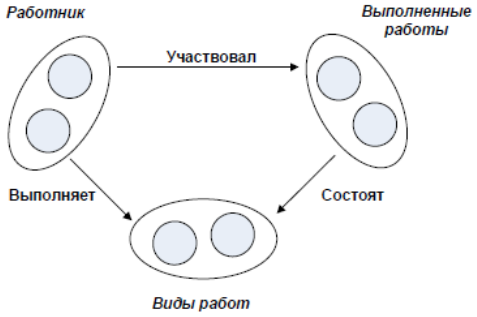



Рисунок 5.3 – Структура диаграммы

Определим степень связи и класс принадлежности каждой конструкции на диаграмме.

Анализ конструкции **Работник** *выполняет* разные **Виды работ.**

1. Определение класса принадлежности:

- каждый работник должен выполнять какой–то вид работы, следовательно, здесь сущность "**Работник**" имеет обязательный класс принадлежности – **О**.
- работники предприятия должны выполнять весь спектр видов работ (имеется в виду нормальное предприятие). Следовательно, сущность «**Виды работ**» имеет также обязательный класс принадлежности – **О**.

2. Определение степени связи:

- каждый работник может выполнять несколько видов работ;
- каждый вид работы может быть выполнен несколькими работниками.

Следовательно, конструкция имеет степень связи **"многие ко многим" (M:M).**

Анализ конструкции **Выполненные работы** *состоят из* разных **Видов работ.**

1. Определение класса принадлежности:

- каждая выполненная работа должна быть определенного вида, следовательно, здесь сущность **Выполненные работы** имеет обязательный класс принадлежности – **О**.
- некоторые виды работ могут в текущем месяце не выполняться совсем, что дает сущности "**Вид работ**" в данной конструкции имеет необязательный класс принадлежности – **N**.

2. Определение степени связи:

- многие выполненные работы могут быть одного вида;

- каждая выполненная работа может быть только одного вида.

Следовательно, конструкция имеет степень связи **"многие к одному" (M:1).**

Анализ конструкции **Работник** *участвовал в* **Выполнении работ (Выполненные работы).**

1. Определение класса принадлежности:

- каждый работник должен участвовать в выполнении работ, следовательно, здесь сущность **Работник** имеет обязательный класс принадлежности – **О**.
- каждая выполненная работа имеет своего исполнителя, то есть сущность **Выполненные работы** в данной конструкции имеет обязательный класс принадлежности – **О**.

2. Определение степени связи:

- один работник может участвовать в выполнении множества работ;
- одна выполненная работа выполняется одним работником.

Следовательно, конструкция имеет степень связи **"один ко многим" (1:M).**

Итоговая диаграммы будет выглядеть следующим образом (см. рис. 5.4)

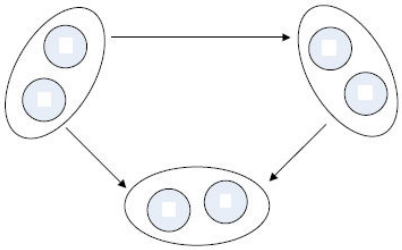


Рисунок 5.4 – Итоговая диаграмма

Зафиксируем атрибуты каждой сущности (табл. 5.1).

Таблица 5.1 – Атрибуты сущностей диаграммы

| Сущность | Атрибуты |
|---|---|
| Работник | Табельный номер,<br>ФИО,<br>Отдел |
| Виды работ | Название работы,<br>Расценка |
| Выполненные работы | Номер наряда,<br>Количество выполненных единиц работы,<br>Сумма за работу |

Для построения таблиц БД на основе диаграмм «сущность – связь» имеются правила:

**Правило №1**

Если имеется связь степени 1:1 и класс принадлежности обеих сущностей обязательный, то требуется всего одна таблица. Первичным ключом этой таблицы может быть любой из ключей сущностей.

**Правило №2**

Если имеется связь степени 1:1 и класс принадлежности одной сущности является обязательным, а другой – необязательным, то требуются две таблицы. Для каждой сущности необходимо создать свою таблицу, первичным ключом которой будет ключ этой сущности. Кроме того, первичный ключ сущности с необязательным классом принадлежности должен быть добавлен в качестве атрибута в таблицу, созданную для сущности с обязательным классом принадлежности.

**Правило №3**

Если имеется связь степени 1:1 и класс принадлежности обеих сущностей является необязательным, то требуются три таблицы. Для каждой сущности необходимо создать свою таблицу, первичным ключом которой будет ключ этой сущности. Кроме того, связующая таблица должна содержать в себе ключи других таблиц в качестве атрибутов.

**Правило №4**

Если имеется связь степени 1 :M и класс принадлежности M–связанной сущности является обязательным, то требуются две таблицы. Для каждой сущности необходимо создать свою таблицу, первичным ключом которой

будет ключ этой сущности. Кроме того, первичный ключ 1–связанной сущности должен быть добавлен в качестве атрибута М–связанной таблицы.

**Правило №5**

Если имеется связь степени 1:М и класс принадлежности М–связанной сущности является необязательным, то требуются три таблицы. Для каждой сущности необходимо создать свою таблицу, первичным ключом которой будет ключ этой сущности. Кроме того, связующая таблица должна содержать в себе ключи других таблиц в качестве ее атрибутов.

**Правило №6**

Если имеется связь степени М:М, то требуются три таблицы. Для каждой сущности необходимо создать свою таблицу, первичным ключом которой будет ключ этой сущности. Кроме того, связующая таблица должна содержать в себе ключи других таблиц в качестве ее атрибутов.

**Правило №7**

При наличии многосторонней связи необходимо создать свою таблицу для каждой сущности, первичным ключом которой будет ключ этой сущности. Кроме того, необходимо создать еще одну таблицу связи, которая будет содержать первичные ключи остальных таблиц в качестве своих атрибутов. То есть для п–сторонней связи необходимо создать п+1 таблиц.

Воспользуемся правилами для построения структуры БД для нашего случая.

Из рис. 5.4 явствует, что мы имеем трехстороннюю связь, которая подпадает под правило 7 преобразования сущностей в таблицы. В соответствии с этим правилом нужно создать четыре таблицы по одной для каждой сущности и дополнительную таблицу связи.

- 1 таблица: **Работник** ('Табельный номер", "ФИО", "Отдел").
- 2 таблица: **Виды работ** ("Название работы", "Расценка").
- 3 таблица: **Выполненные работы** ("Наряд №", "Количество единиц работы", "Сумма").
- 4 таблица: **Таблица связи** ("Табельный номер", "Название работы", "Наряд №").Здесь имя перед скобками означает название таблицы. Названия в скобках соответствуют будущим полям таблиц. Подчеркиванием показаны поля, которые выбраны в качестве первичного индекса (*ключа*) таблицы.

Правила моделирования СУБД:

- желательно делать ключевым первое поле в таблице, так как некоторые СУБД требуют этого;
- записывайте свои размышления на бумаге, это поможет обнаружить многие дефекты;
- не храните в базе те данные, которы могут быть вычислены из других полей.

**Контрольные вопросы**

1. Дайте определение, что такое база данных, СУБД, запись, поле записи, таблица базы данных?
2. Дайте определение сущности, связи, степени связи, класса принадлежности?
3. Какие значения может принимать степень связи и класс принадлежности?
4. Как строится структура диаграммы «сущность – связь»?
5. Как анализируется конструкция диаграммы «сущность – связь»?
6. Сформулируйте правила для построения таблиц БД на основе диаграмм «сущность – связь».

## VI. Метод описания процессов WORKFLOW DIAGRAMMING

Наличие в диаграммах ***DFD*** элементов для описания источников, приемников и хранилищ данных позволяет более эффективно и наглядно описать процесс обработки информации. Однако для описания логики взаимодействия информационных потоков более подходит ***IDEF3***, называемая также ***workflow diagramming*** – методологией моделирования, использующая графическое описание информационных потоков, взаимоотношений между процессами обработки информации и объектов, являющихся частью этих процессов.

Диаграммы ***Workflow*** могут быть использованы в моделировании бизнес-процессов для анализа завершенности процедур обработки информации. С их помощью можно описывать сценарии действий сотрудников организации, например последовательность обработки заказа или события, которые необходимо обработать за конечное время. Каждый сценарий сопровождается описанием процесса и может быть использован для документирования каждой функции.

***IDEF3*** – это метод, основная цель которого, – дать возможность аналитикам описать ситуацию, когда процессы выполняются в определенной последовательности, а также описать объекты, участвующие совместно в одном процессе.

Техника описания набора данных ***IDEF3*** является частью структурного анализа. ***IDEF3,*** в отличие от некоторых методик описаний процессов, не ограничивает разработчика чрезмерно жесткими рамками синтаксиса, что может привести к созданию неполных или противоречивых моделей.

***IDEF3*** также может быть использован в качестве метода создания процессов.

***IDEF3*** дополняет ***SADT (***то есть ***IDEF0)*** и содержит все необходимое для построения моделей, которые в дальнейшем могут быть использованы для имитационного анализа.

Каждая работа в ***IDEF3*** описывает какой–либо сценарий бизнес–процесса и может являться составляющей другой работы. Поскольку сценарий описывает цель и рамки модели, важно, чтобы работы именовались отглагольным существительным, обозначающим процесс действия, или фразой, содержащей такое существительное.

Точка зрения на модель должна быть задокументирована. Обычно это точка зрения человека, ответственного за работу в целом. Также необходимо задокументировать цель модели – те вопросы, на которые призвана ответить модель.

**Диаграммы.** Диаграмма является основной единицей описания в ***IDEF3***. Важно правильно построить диаграммы, поскольку они предназначены для чтения другими людьми.

**Единицы работы – *Unit of Work (UOW). UOW***, также называемые работами (***activity***) и являются центральными компонентами модели.

В ***IDEF3*** работы изображаются прямоугольниками с прямыми углами и имеют имя, выраженное отглагольным существительным, одиночным или в составе фразы, обозначающим процесс действия и номер (идентификатор).


Рисунок 6.1 – Изображение работы на диаграмме ***IDEF3***

Другое имя существительное в составе той же фразы обычно отображает основной выход (результат) работы (например, *"Изготовление изделия")*. Часто имя существительное в модель может уточняться и редактироваться. Идентификатор работы присваивается при создании и *не меняется никогда*.

Даже если работа будет удалена, ее идентификатор не будет вновь использоваться для других работ.

Обычно номер работы состоит из номера родительской работы и порядкового номера на текущей диаграмме.

**Связи.** Связи показывают взаимоотношения работ. Все связи в ***IDEF3*** однонаправлены и могут быть направлены куда угодно, но обычно диаграммы ***IDEF3*** стараются построить так, чтобы связи были направлены слева направо. В ***IDEF3*** различают три типа стрелок, изображающих связи:

*Старшая (Precedence)* **–** сплошная линия, связывающая единицы работ (***UOW***). Рисуется слева направо или сверху вниз. Показывает, что работа–источник должна закончиться прежде, чем работа–цель начнется (рис. 6.2).

Рисунок 6.2 – Изображение связи *Старшая* на диаграмме ***IDEF3***

*Отношения (Relational Link)* – пунктирная линия, использующаяся для. изображения связей между единицами работ (***UOW***) а также между единицами работ и объектами ссылок (рис. 6.3).

Рисунок 6.3 – Изображение связи *Отношения* на диаграмме ***IDEF3***

*Потоки объектов (Object Flow)* – стрелка с двумя наконечниками, применяется для описания того факта, что объект используется в двух или более единицах работы, например, когда объект порождается в одной работе и используется в другой (рис 6.4).

Рисунок 6.4 – Изображение связи *Потоки объектов* на диаграмме ***IDEF3***

**Старшая связь и поток объектов. Поскольку с**таршая связь показывает, что работа–источник заканчивается ранее, чем начинается работа–цель, то часто результатом работы–источника становится объект,

необходимый для запуска работы–цели. В этом случае стрелку, обозначающую объект, изображают с двойным наконечником. Имя стрелки должно ясно идентифицировать отображаемый объект. Поток объектов имеет ту же семантику, что и старшая стрелка.

Отношение показывает, что стрелка является альтернативой старшей стрелке или потоку объектов в смысле задания последовательности выполнения работ – работа–источник необязательно должна закончиться, прежде чем работа–цель начнется. Более того, работа–цель может закончиться прежде, чем закончится работа–источник.

**Перекрестки (Junction).** Окончание одной работы может служить сигналом к началу нескольких работ, или же одна работа для своего запуска может ожидать окончания нескольких работ. Перекрестки используются для отображения логики взаимодействия стрелок при слиянии и разветвлении или для отображения множества событий, которые могут или должны быть завершены перед началом следующей работы.

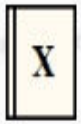
Различают перекрестки:

- для слияния *(Fan–in Junction)* стрелок*;*
- для разветвления *(Fan–out Junction)* стрелок.

Перекресток не может использоваться одновременно для слияния и для разветвления.

Расшифровки каждого типа перекрестков приведены в табл. 6.1.

Таблица 6.1. Типы перекрестков

| Обозначение | Наименование | Смысл в случае слияния стрелок (Fan-in Junction) | Смысл в случае разветвления стрелок (Fan-out Junction) |
|---|---|---|---|
|  | Асинхронное «И» (Asynchronous AND) | Все предшествующие процессы должны быть завершены | Все следующие процессы должны быть запущены |
|  | Синхронное «И» (Synchronous AND) | Все предшествующие процессы завершены одновременно | Все следующие процессы запускаются одновременно |
|  | Асинхронное «ИЛИ» (Asynchronous OR) | Один или несколько предшествующих процессов должны быть завершены | Один или несколько следующих процессов должны быть запущены |
|  | Синхронное «ИЛИ» (Synchronous OR) | Один или несколько предшествующих процессов завершены одновременно | Один или несколько следующих процессов запускаются одновременно |
|  | Исключающее «ИЛИ» XOR (Exclusive OR) | Только один предшествующий процесс завершен | Только один следующий процесс запускается |

Все перекрестки на диаграмме нумеруются, каждый номер имеет префикс ***J***. Свойства перекрестка можно редактировать. В отличие от ***IDEF0*** и ***DFD*** в ***IDEF3*** стрелки могут сливаться и разветвляться только через перекрестки.

**Объект ссылки.** Объект ссылки в ***IDEF3*** выражает некую идею, концепцию или данные, которые нельзя связать со стрелкой, перекрестком или работой (рис. 6.5).

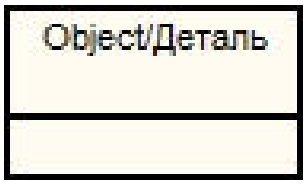



Рисунок 6.5 – Изображение *Объекта ссылки* на диаграмме ***IDEF3***

Объект ссылки изображается в виде прямоугольника, похожего на прямоугольник работы.

Объекты ссылки должны быть связаны с единицами работ или перекрестками пунктирными линиями.

Официальная спецификация ***IDEF3*** различает три стиля объектов ссылок:

- безусловные (***unconditional***),
- синхронные (***synchronous***),
- асинхронные (***asynchronous***).

При внесении объектов ссылок помимо имени следует указывать тип объекта ссылки. Типы объектов ссылок приведены в табл. 6.2.

Таблица 6.2. Типы объектов ссылок.

| Тип объекта ссылки | Цель описания |
|---|---|
| **OBJECT** | Описывает участие важного объекта в работе |
| **GOTO** | Инструмент циклического перехода (в повторяющейся последовательности работ), возможно на текущей диаграмме, но не обязательно. Если все работы цикла присутствуют на текущей диаграмме, цикл может также изображаться стрелкой, возвращающейся на стартовую работу. GOTO может ссылаться на перекресток |
| **UOB (Unit of behavior)** | Применятся, когда необходимо подчеркнуть множественное использование какой-либо работы, но без цикла. Например, работа "Контроль качества" может быть использована в процессе "Изготовления изделия" несколько раз, после каждой единичной операции. Обычно этот тип ссылки не используется для модели- |

| | |
|---|---|
| | рования автоматически запускающихся работ |
| **NOTE** | Используется для документирования важной информации, относящейся к каким-либо графическим объектам на диаграмме. NOTE является альтернативой внесению текстового объекта в диаграмму |
| **ELAB (Elaboration)** | Используется для усовершенствования графиков или их более детального описания. Обычно употребляется для детального описания разветвления и слияния стрелок на перекрестках |

**Декомпозиция работ.**

В ***IDEF3*** декомпозиция используется для детализации работ.

Методология ***IDEF3*** позволяет декомпозировать работу многократно, т. е. работа может иметь множество дочерних работ. Это позволяет в одной модели описать альтернативные потоки.

Возможность множественной декомпозиции предъявляет дополнительные требования к нумерации работ. Так, номер работы состоит из номера родительской работы, версии декомпозиции и собственного номера работы на текущей диаграмме.

Процесс декомпозиции диаграмм ***IDEF3***, включает взаимодействие разработчика и заказчика программного продукта, являющегося экспертом рассматриваемой предметной области.

На первом этапе формируется описание сценария, области и точки зрения. Перед проведением анализа должен быть подготовлены документированные версии сценариев и границы модели для того, чтобы заказчик мог понять цели декомпозиции.

Кроме того, если точка зрения разработчика отличается от точки зрения заказчика, она должна быть особенно тщательно документирована. Возможно, что заказчик самостоятельно не сможет передать необходимую информацию. В этом случае разработчик должен приготовить список вопросов для проведения опроса заказчика.

Затем определяются работы и объекты. Обычно заказчик предметной области передает разработчику текстовое описание сценария. В дополнение к этому может существовать документация, описывающая интересующие процессы. Из всей этой информации разработчик должен составить список кандидатов на работы (отглагольные существительные, обозначающие процесс, одиночные или в составе фразы) и кандидатов на объекты (существительные, обозначающие результат выполнения работы), которые необходимы для перечисленных в списке работ.

В некоторых случаях целесообразно создать графическую модель для представления ее заказчику, который хорошо знает предметную область. Графическая модель может быть также создана после сеанса сбора информации для того, чтобы детали форматирования диаграммы не противоречили мнению заказчиков. Поскольку разные фрагменты модели

***IDEF3*** могут быть созданы разными группами разработчиков в разное время, ***IDEF3*** поддерживает простую схему нумерации работ в рамках всей модели.

Разные разработчики оперируют разными диапазонами номеров, работая при этом независимо.

На следующем этапе проверяется последовательность и согласование диаграмм. Если диаграмма создается после проведения опроса заказчиков, то разработчик должен принять некоторые решения, относящиеся к иерархии диаграмм, например, сколько деталей включать в одну диаграмму. Если последовательность и согласование диаграмм неочевидны, может быть проведена еще одна экспертиза предметной области для детализации и уточнения информации.

Важно различать подразумевающее согласование (согласование, которое подразумевается в отсутствие связей) и ясное согласование (согласование, ясно изложенное в мнении заказчика).

***IDEF3*** позволяет внести информацию в модель различными способами. Например, логика взаимодействия может быть отображена графически в виде комбинации перекрестков. Та же информация может быть отображена в виде объекта ссылки типа ***ELAB (Elaboration).*** Это позволяет разработчику вносить информацию в удобном в данный момент времени виде. Важно учитывать, что модели могут быть реорганизованы, например, для их представления в более презентабельном виде. Выбор формата для презентации часто имеет важное значение для организации модели, поскольку комбинация перекрестков занимает значительное место на диаграмме и использование иерархии перекрестков затрудняет расположение работ на диаграмме.

Для более полного соответствия построенной с помощь ***IDEF1, IDEF2*** и ***IDEF3*** модели реальной предметной области часто используют имитационное моделирование.

***Имитационное моделирование*** – это метод, позволяющий строить модели, учитывающие время выполнения функций.

Полученную модель можно проанализировать во времени и получить статистику происходящих процессов так, как это было бы в реальности. В имитационной модели изменения процессов и данных ассоциируются с событиями. Временной анализ модели заключается в последовательном переходе от одного события к другому.

Обычно имитационные модели строятся для поиска оптимального решения в условиях ограничения по ресурсам, когда другие математические модели оказываются слишком сложными.

Связь между имитационными моделями и моделями процессов заключается в возможности преобразования модели процессов в неполную имитационную модель. Имитационная модель дает больше информации для анализа системы, в свою очередь результаты такого анализа могут стать причиной модификации модели.

Имитационная модель включает следующие основные элементы:

1. Источники и цели (***Sourses*** и ***Destinations***). Источники – это элементы, от которых в модель поступает информация или объекты. По смыслу они близки к понятиям "внешняя ссылка" на ***DFD***–диаграммах или "объект ссылки" на диаграммах ***IDEF3***. Скорость поступления данных или объектов от источника обычно задается статистической функцией. Цель – это устройство для приема информации или объектов.

2. Очереди (***Queues***). Понятие очереди близко к понятию хранилища данных на ***DFD***–диаграммах – это место, где объекты ожидают обработки. Времена обработки объектов (производительность) в разных работах могут быть разными. В результате перед некоторыми работами могут накапливаться объекты, ожидающие своей очереди. Часто целью имитационного моделирования является минимизация количества объектов в очередях. Тип очереди в имитационной модели может быть конкретизирован. Очередь может быть похожа на стек – пришедшие последними в очередь объекты первыми отправляются на дальнейшую обработку. Альтернативой стеку может быть последовательная обработка, когда первыми на дальнейшую обработку отправляются объекты, пришедшие первыми. Могут быть заданы и более сложные алгоритмы обработки очереди.

3. Оборудование (***Facilities***). Оборудование – это аналог работ в модели процессов. В имитационной модели может быть задана производительность оборудования.

Контрольные вопросы.

1. Что такое имитационное моделирование?
2. Основные элементы имитационного моделирования.
3. Определение «Декомпозиция работ».
4. Стили объектов ссылок IDEF3.
5. Перекрестки.
6. Старшая связь и поток объектов.
7. Связи
8. Определение IDEF3.
9. Что такое workflow diagramming?

## VII. Анализ требований к информационной системе

### *7.1.* Требования к программному обеспечению информационной системы

#### 7.1.1. Общие требования к информационной системе

Исторически сложился ряд требований к информационным системам.

Требования эти таковы:

- Системность;
- Комплексность;
- Модульность;
- Открытость;

- Адаптивность;
- Надежность;
- Безопасность;
- Мобильность;
- Простота в изучении;
- Поддержка внедрения и сопровождения со стороны разработчика.

Рассмотрим эти требования подробнее.

ИС должна отвечать требованиям *комплексности* и *системности*. Она должна охватывать все уровни управления техническим объектом. Функционирующий объект ИС можно представить в виде информационно-логической модели, состоящей из узлов и связей между ними. Такая модель должна охватывать все аспекты функционирования объекта, должна быть логически обоснована и направлена на выявление механизмов достижения основной цели, что и подразумевает требование системности.

*Достаточно эффективное решение* этой задачи возможно только на базе строгого учета максимально возможного обоснованного множества параметров и возможности многокритериальных анализа, оптимизации и прогнозирования – то есть комплексности системы. Информация в такой модели носит распределенный характер и может быть достаточно строго структурирована на каждом узле и в каждом потоке.

Узлы и потоки могут быть условно сгруппированы в подсистемы, что выдвигает еще одно важное требование к ИС – *модульность* построения. Это требование очень важно с точки зрения внедрения системы, поскольку позволяет распараллелить, облегчить и, соответственно, ускорить процесс инсталляции, подготовки персонала и запуска системы в промышленную эксплуатацию.

Поскольку ни одна реальная система не может быть исчерпывающе полной и в процессе эксплуатации может возникнуть необходимость в дополнениях, следующим определяющим требованием является *открытость*. Любой объект существует не в замкнутом пространстве, а в меняющемся окружении других объектов, систем и факторов.

ИС должна обладать свойством *адаптивности*, то есть гибко настраиваться на взаимодействие с другими системами.

Кроме средств настройки система должна обладать и средствами развития – инструментарием, при помощи которого программисты и наиболее квалифицированные пользователи могли бы самостоятельно создавать необходимые им компоненты, которые встраивались бы в ИС.

Поэтому одним из важнейших требований к системе является надежность ее функционирования, подразумевающая непрерывность функционирования системы в целом даже в условиях частичного выхода из строя отдельных ее элементов. Чрезвычайно большое значение для любой информационной системы, содержащей большое количество информации, имеет *безопасность* и включает в себя несколько аспектов:

*Защита данных от потери*. Это требование реализуется, в основном, на организационном, аппаратном и системном уровнях.

*Сохранение целостности и непротиворечивости данных.*

Прикладная система должна отслеживать изменения во взаимозависимых документах и обеспечивать управление версиями и поколениями наборов данных.

*Предотвращение несанкционированного доступа к данным внутри системы.* Эти задачи решаются комплексно как организационными мероприятиями, так и на уровне операционных и прикладных систем.

*Предотвращение несанкционированного доступа к данным извне.* Решение этой части проблемы ложится в основном на аппаратную и операционную среду функционирования ИС и требует ряда административно-организационных мероприятий.

Темпы развития предприятия рост требований к производительности и ресурсам информационной системы может потребовать перехода на более производительную программно-аппаратную платформу, т.е. необходимо выполнение требования *мобильности.*

*Простота в изучении* – это требование, включающее в себя не только наличие интуитивно понятного интерфейса программ, но и наличие подробной и хорошо структурированной документации, возможности обучения персонала на специализированных.

*Поддержка разработчика*.

*Сопровождение*. Сопровождение включает в себя выезд специалиста на объект заказчика для устранения последствий аварийных ситуаций, техническое обучение на объекте заказчика, методическую и практическую помощь при необходимости внести изменения в систему, не носящие характер радикальной реструктуризации или новой разработки. Подразумевается также установка новых релизов программного обеспечения, получаемого от разработчика бесплатно силами уполномоченной разработчиком сопровождающей организации или силами самого разработчика.

### 7.1.2. Функциональные и нефункциональные требования

Требования к программной системе часто классифицируются как функциональные, нефункциональные и требования предметной области.

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

2. *Нефункциональные требования.* Описывают характеристики системы и ее окружения, а не поведение системы. Здесь также может быть приведен перечень ограничений, накладываемых на действия и функции, выполняемые системой. Они включают временные ограничения, ограничения на процесс разработки системы, стандарты и т.д.

3. *Требования предметной области.* Характеризуют ту предметную область, где будет эксплуатироваться система. Эти требования могут быть функциональными и нефункциональными. В действительности четкой границы между этими типами требований не существует. Например, пользовательские

требования, касающиеся безопасности системы, можно отнести к нефункциональным. Однако при более детальном рассмотрении такое требование можно отнести к функциональным, поскольку оно порождает необходимость включения в систему средства авторизации пользователя. Поэтому, рассматривая далее эти виды требований, мы должны всегда помнить, что данная классификация в значительной степени искусственна.

Эти требования описывают поведение системы и сервисы (функции), которые она выполняет, и зависят от типа разрабатываемой системы и от потребностей пользователей. Если функциональные требования оформлены как пользовательские, они, как правило, описывают системы в обобщенном виде. В противоположность этому функциональные требования, оформленные как системные, описывают систему максимально подробно, включая ее входные и выходные данные, исключения и т.д.

*Функциональные требования*

Функциональные требования для программных систем могут быть описаны разными способами.

На практике для больших и сложных систем крайне трудно разработать комплексную и непротиворечивую спецификацию функциональных требований. Причина кроется частично в сложности самой разрабатываемой системы, а частично – в несогласованных опорных точках зрения на то, что должна делать система. Эта несогласованность может не проявиться на этапе первоначального формулирования требований – для ее выявления необходим более глубокий анализ спецификации.

Когда несогласованность системных функций проявится на каком-либо этапе жизненного цикла программы, в системную спецификацию придется внести соответствующие изменения.

*Нефункциональные требования*

Как следует из названия, нефункциональные требования не связаны непосредственно с функциями, выполняемыми системой. Они связаны с такими интегрированными свойствами системы, как надежность, время ответа или размер системы.

Кроме того, нефункциональные требования могут определять ограничения на систему, например на пропускную способность устройств ввода–вывода, или форматы данных, используемых в системном интерфейсе.

Многие нефункциональные требования относятся к системе в целом, а не к отдельным ее средствам. Это означает, что они более значимы и критичны, чем отдельные функциональные требования. Ошибка, допущенная в функциональном требовании, может снизить качество системы, ошибка в нефункциональных требованиях может сделать систему неработоспособной. Вместе с тем нефункциональные требования могут относиться не только к самой программной системе: одни могут относиться к технологическому процессу создания ПО, другие – содержать перечень стандартов качества, накладываемых на процесс разработки. Кроме того, в спецификации нефункциональных требований может быть указано, что проектирование системы должно выполняться только определенными CASE-средствами, и приведено описание процесса проектирования, которому необходимо следовать.

Нефункциональные требования отображают пользовательские потребности; при этом они основываются на бюджетных ограничениях, учитывают организационные возможности компании-разработчика и возможность взаимодействия разрабатываемой системы с другими программными и вычислительными системами, а также такие внешние факторы, как правила техники безопасности, законодательство о защите интеллектуальной собственности и т.н. На рис. 7.1 показана классификация нефункциональных требований.

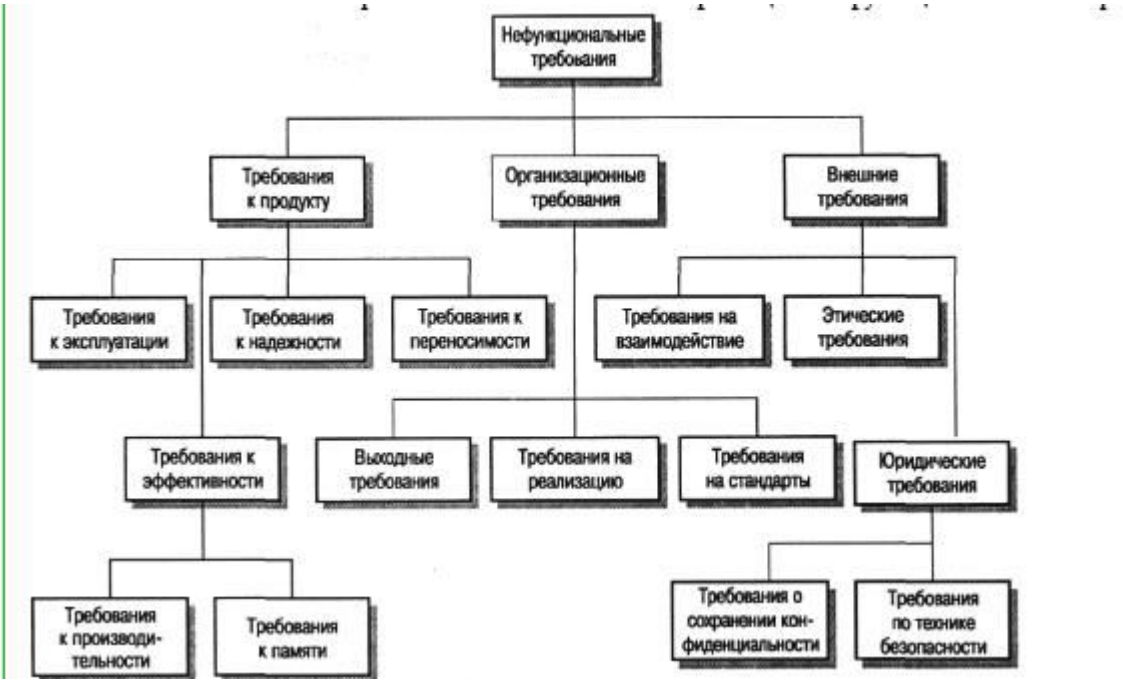


Рисунок 7.1 – Классификация нефункциональных требований

Все нефункциональные требования, показанные на рис. 7.1 разбиты на три большие группы.

1. *Требования к продукту.* Описывают эксплуатационные свойства программного продукта.

Сюда относятся требования к производительности системы, объему необходимой памяти, надежности (определяет частоту возможных сбоев в системе), переносимости системы на разные компьютерные платформы и удобству эксплуатации.

2. *Организационные требования.* Отображают политику и организационные процедуры заказчика и разработчика ПО. Они включают стандарты разработки программного продукта, требования к реализации ПО (т.е. к языку программирования и методам проектирования), выходные требования, которые определяют сроки изготовления программного продукта, и сопутствующую документацию.

3. *Внешние требования.* Учитывают факторы, внешние по отношению к разрабатываемой системе и процессу ее разработки. Они включают требования, определяющие взаимодействие данной системы с другими системами, юридические требования, следование которым гарантирует, что система будет разрабатываться и функционировать в рамках существующего законодательства, а также этические требования. Последние должны гарантировать, что система будет приемлемой для пользователей или заказчика.

Проблема нефункциональных требований состоит в том, что их выполнение трудно проверить. Часто они пишутся для того, чтобы отобразить общие цели заказчика системы, такие, как простота эксплуатации, возможность восстановления после сбоев или быстрый ответ на запросы пользователя. Реализация подобных требований может оказаться сложной

для системных разработчиков, поскольку они нечетко сформулированы и открывают простор для различных толкований.

В идеале нефункциональные требования должны выражаться через количественные показатели, которые можно объективно измерить.

На практике выразить нефункциональные требования с помощью количественных показателей весьма затруднительно. Часто заказчик ПО не может оформить свое видение будущей системы посредством требований, выраженных количественными показателями. Либо некоторые системные требования, например удобство сопровождения, вообще нельзя выразить через количественные показатели. Кроме того, затраты на объективное измерение количественных нефункциональных требований могут оказаться крайне высоки. Поэтому часто документ, определяющий требования к системе, содержит описание системных цепей совместно с четко сформулированными требованиями. Эти системные цели, поскольку они отражают представления (и приоритеты) заказчика о будущей системе. Вместе с тем заказчик должен понимать, что его системные цели могут трактоваться различными способами и их невозможно объективно проконтролиовать. В таблице7.1 ведены показатели, с помощью которых можно специфицировать нефункциональные системные свойства.

Таблица 7.1. Количественные показатели для нефункциональных требований

| **Показатель** | **Единицы измерения** |
|---|---|
| Скорость | Количество выполненных транзакций в секунду;<br>время реакции на действия пользователя;<br>время обновления экрана |
| Размер | Килобайты;<br>количество модулей памяти |
| Простота эксплуатации | Время обучения персонала;<br>количество статей в справочной системе |
| Надежность | Средняя продолжительность времени между двумя последовательными проявлениями ошибок в системе;<br>вероятность выхода системы из строя;<br>коэффициент готовности системы |
| Устойчивость к сбоям | Время восстановления системы после сбоя;<br>процент событий, приводящих к сбоям;<br>вероятность порчи данных при сбоях |
| Переносимость | Процент машинно-зависимых операторов;<br>количество машинно-зависимых подсистем |

Нефункциональные требования часто вступают в конфликт с другими требованиями, предъявляемыми системе. Если одновременное выполнение этих требований невозможно, то следует отказаться от одного из требований.

### 7.1.3. Пользовательские требования

Пользовательские требования к системе должны описывать функциональные и нефункциональные системные требования так, чтобы они

были понятны даже пользователю, не имеющему специальных технических знаний. Эти требования должны определять только внешнее поведение системы, избегая по возможности определения структурные характеристик системы. Пользовательские требования должны быть написаны естественным языком с использованием простых таблиц, а также наглядных и понятных диаграмм. Вместе с тем при описании требований на естественном языке могут возникнуть различные проблемы.

1. *Отсутствие четкости изложения.* Иногда нелегко изложить какую–либо мысль естественным языком четко и недвусмысленно, не сделав при этом текст многословным и трудночитаемым.

2. *Смешение требований.* В пользовательских требованиях отсутствует четкое разделение на функциональные и нефункциональные требования, на системные цели и проектную информацию.

3. *Объединение требований.* Несколько различных требований и системе могут описываться как единое пользовательское требование. В документе, содержащем требования к системе, желательно отделять пользовательские требования от более детализированных системных требований. Иначе неподготовленный читатель пользовательских требований может потонуть в технических подробностях, понимание которых требует определенных профессиональных знаний. Чтобы свести к минимуму неясности при написании пользовательских требований, рекомендуется придерживаться приведенных ниже правил.

1. Разработайте стандартную форму для записи пользовательских требований и неукоснительно ее придерживайтесь. Стандартная форма записи уменьшает неясности в формулировке требований и позволяет легко их проверить. Рекомендуется включать в форму записи требования не только саму его формулировку, но его обоснование и ссылку на более детализированную спецификацию требований.

2. Делайте различие между обязательными и описательными требованиями. Здесь обязательным требованием является наличие средства добавления новых структурных элементов, описательным – описание последовательности действий пользователя. Описательное требование не является абсолютно необходимым для реализации данного пользовательского требования и при необходимости может быть изменено. Используйте разные начертания шрифта {полужирное и курсив) для выделения ключевых частей требования.

4. Избегайте по возможности компьютерного жаргона. Это не исключает использования технических терминов той предметной области, для которой разрабатывается программное обеспечение.

### 7.1.4. Системные требования. Документирование требований

Системные требования – это более детализированное описание пользовательских требований. Они обычно служат основой для заключения контракта на разработку программной системы и поэтому должны представлять максимально полную спецификацию системы в целом.

Системные требования также используются в качестве отправной точки на этапе проектирования системы.

Спецификация системных требований может строиться на основе различных системных моделей, таких, как объектная модель или модель потоков данных.

Спецификации системных требований часто пишутся естественным языком. Но использование естественного языка может породить определенные проблемы при написании детализированной спецификации. Применение естественного языка подразумевает, что те, кто пишет спецификацию, и те, кто се читает, одни и те же слова и выражения понимают одинаково. Однако на самом деле это не так, поскольку естественному языку присуща определенная размытость понятий. Вследствие этого одно и то же требование может трактоваться разными людьми по-разному.

Чтобы избежать подобных проблем, разработаны методы описания требований, которые структурируют спецификацию и уменьшают размытость определений. Эти методы представлены в табл. 7.2.

Таблица 7.2 – Способы записи спецификаций требований

| Система записи | Описание |
|---|---|
| Структурированный естественный язык | Использование стандартных форм и шаблонов для написания спецификации |
| Языки описания программ | Использование специальных структурированных языков, подобных языкам программирования, где спецификация требований строится на основе выбранной операционной модели системы |
| Графические нотации | Графический язык, использующий для описания функциональных требований диаграммы и блок-схемы, дополненные текстовыми пояснениями. Наиболее известный пример такого графического языка — диаграммы структурного анализа и проектирования ПО (SADT) |
| Математические спецификации | Это системы нотаций, основанные на математических концепциях, таких, как теория конечных автоматов или теория множеств. Это формализованная однозначная и лишенная дву-смысленности запись системных требований. Однако многие заказчики |

Структурированный язык спецификации – это сокращенная форма естественного языка, предназначенная для написания спецификации требований. Достоинством такого подхода к написанию спецификаций является то, что он сохраняет выразительность и понятность естественного языка и вместе с тем формализует описание требований.

Структурированность языка проявляется в использовании специальной терминологии, а также шаблонов для описания системных требований. Структурированный язык может включать языковые конструкции, взятые из языков программирования. Для описания системных требований часто

разрабатываются специальные формы и шаблоны. Они должны учитывать, на основе чего строится спецификация: на основе объектов, управляемых системой, на основе функций, выполняемых системой, или на основе событий, обрабатываемых системой.

Стандартные формы, используемые для специфицирования функциональных требований, должны содержать следующую информацию.

1. Описание функции или объекта.
2. Описание входных данных и их источники
3. Описание выходных данных с указанием пункта их назначения.
4. Указание, что необходимо для выполнения функции.
5. Если это спецификация функции, необходимо описание предварительных условий (предусловий), которые должны выполняться перед вызовом функции, и описание заключительного условия (постусловия), которое должно быть выполнено после завершения выполнения функции.
6. Описание побочных эффектов (если они есть).

Использование структурированного языка снимает некоторые проблемы, присущие спецификациям, написанным естественным языком, поскольку снижает "вариабельность" спецификации и более эффективно ее структурирует. Вместе с тем некоторая "размытость" определений и описаний в спецификации остается. Альтернативой использованию структурированного естественного языка может служить специальный язык описания спецификаций, который полностью снимает проблему нечеткости описания требований. Но с другой стороны, неспециалист найдет такую спецификацию трудной для чтения и понимания.

Подавляющее большинство разрабатываемых программных систем должны взаимодействовать с другими, уже существующими системами. Для того чтобы новая система могла работать совместно с другими системами, необходимо специфицировать интерфейсы этих систем. Такие спецификации создаются на раннем этапе разработки систем и включаются (обычно в виде приложения) в спецификацию системных требований. Различают три типа специфицируемых интерфейсов.

1. ***Процедурные интерфейсы***, когда существующие подсистемы предлагают набор сервисов, доступных посредством вызываемой интерфейсной процедуры.
2. ***Структуры (интерфейсные форматы) данных***, которые пересылаются от одной подсистемы к другой. Для описания этого типа интерфейса наиболее подходят диаграммы "сущность–связь".
3. ***Специальные представления данных***, например в виде упорядоченной последовательности двоичных разрядов.

***Документирование системных требований***

Документ, содержащий требования, также называемый спецификацией системных требований, – это официальное предписание для разработчиков программной системы. Он содержит пользовательские требования и детализированное описание системных требований. В некоторых случаях пользовательские и системные требования могут не различаться, выступая совместно в виде однородного описания системы. В других случаях

пользовательские требования приводятся во введении документа – спецификации. Если общее количество требований велико, детализированные системные требования могут быть представлены в виде отдельного документа.

Сформулируем условия, которым должна соответствовать спецификация программной системы.

• Описывать только внешнее поведение системы.

• Указывать ограничения, накладываемые на процесс реализации системы.

• Предусматривать возможность внесения изменений в спецификацию.

• Служить справочным средством в процессе сопровождения системы.

• Отображать весь жизненный цикл системы.

• Предусматривать реакцию системы и группы сопровождения на непредвиденные (нештатные) ситуации.

Многие организации разработали собственные стандарты документирования спецификации. Наиболее известный стандарт разработан IEEE и называется IEEE/ANSI 830–1993.

Данный стандарт предполагает следующую структуру спецификации.

1. Введение
1.1. Цели документа
1.2. Назначение программного продукта
1.3. Определения, акронимы и аббревиатуры
1.4. Список литературы и других источников
1.5. Обзор спецификации
2. **Общее описание**
2.1. Описание программного продукта
2.2. Функции программного продукта
2.3. Пользовательские характеристики
2.4. Общие ограничения
2.5. Обоснования, предположения и допущения

***Спецификация требований*** охватывает функциональные, нефункциональные и интерфейсные требования. Это наиболее значимая часть документа, но вследствие крайне широкого диапазона возможных требований, предъявляемых программным системам, в стандарте не определена структура этого раздела. Здесь могут быть документированы:

- внешние интерфейсы,
- функциональные возможности системы,
- требования, определяющие логическую структуру баз данных,
- ограничения, накладываемые на структуру системы,
- интеграционные свойства системы все ее качественные характеристики.

В табл. 7.3 описаны возможные разделы спецификации, построенной на основании стандарта IEEE.

Конечно, информация, включаемая в спецификацию, зависит от типа разрабатываемого программного обеспечения и от выбранной технологии разработки. Для больших документов необходимо делать оглавление и подробные указатели для поиска необходимой информации.

Таблица 7.3. Структура спецификации требований

| Раздел | Описание |
|---|---|
| Предисловие | Здесь определяется круг лиц, на которых рассчитан данный документ. Описываются предыдущие версии разрабатываемого программного продукта, а также изменения, внесенные в каждую версию. Дается обоснование для создания новой версии |
| Введение | Здесь более развернуто обосновывается необходимость создания системы. Кратко перечисляются системные функции и объясняется, как система будет работать совместно с другими системами. Должно быть показано, как разработка системы "вписывается" в общую бизнес-стратегию компании, заказывающей программный |
| Глоссарий | Дается описание технических терминов, используемых в документе. Здесь не делается каких-либо предположений об уровне знаний или практическом опыте читателя документа |
| Пользовательские требования | Описываются сервисы, предоставляемые пользователям, и нефункциональные системные требования. Это описание может быть сделано на естественном языке с использованием диаграмм, блок-схем и других форм записи, понятных заказчику программной системы. Здесь также должны быть приведены стандарты на программный продукт и процесс его разработки |
| Системная архитектура | Здесь приводится высокоуровневое представление возможной системной архитектуры с указанием, как распределены системные функции по компонентам системы. Обязательно должны быть выделены повторно используемые (т.е. уже существующие) компо- |
| Системные требования | Подробно описываются функциональные и нефункциональные требования. Если необходимо, |

| | |
|---|---|
| | нефункциональные требования дополняются описанием интерфейсов других систем |
| Системные модели | Здесь представлено несколько системных моделей, показывающих взаимоотношения между системными компонентами и между системой и ее окружением. Это могут быть объектные модели, модели потоков данных или модели данных |
| Эволюция системы | Приводятся основные предположения и допущения, на которых базируется система, а также ожидаемые (прогнозируемые) изменения в аппаратных средствах, в потребностях пользователей и т.п. |
| Приложения | Здесь приводится специализированная информация, относящаяся к разрабатываемой системе, например описание аппаратных средств или базы данных, с которыми должна работать система. При описании аппаратных средств необходимо показать минимальную и оптимальную конфигурации, при которых может работать программная система. Описание базы данных должно отображать логическую структуру данных, с которыми будет работать система, и |
| Указатели | В документе возможно использование различных указателей. Это может быть обычный алфавитный указатель, указатель диаграмм или указатель системных функций |

## 7.2. Разработка требований к программному обеспечению информационной системы

### 7.2.1. Анализ возможности выполнения

Разработка требований – это процесс, включающий мероприятия, необходимые для создания и утверждения документа, содержащего спецификацию системных требований.

Различают четыре основных этапа процесса разработки требований:

- анализ технической осуществимости создания системы,
- формирование и анализ требований,
- специфицирование требований,
- создание соответствующей документации, а также аттестация этих требований.

На рис. 7.2 показаны взаимосвязи между этими этапами и документы, сопровождающие каждый этап процесса разработки системных требований.


Рисунок 7.2 – Процесс разработки требований.

Для новых программных систем процесс разработки требований должен начинаться с анализа осуществимости. Началом такого анализа является общее описание системы и ее назначения, а результатом анализа – отчет, в котором должна быть четкая рекомендация, продолжать или нет процесс разработки требований проектируемой системы. При проведении анализа осуществимости решаются следующие вопросы:

1. Отвечает ли система поставленным целям заказчика и разработчика?
2. Можно ли реализовать систему, используя существующие на данный момент технологии и не выходя за пределы заданной стоимости?
3. Можно ли объединить систему с другими системами, которые уже эксплуатируются?

Критическим является вопрос, будет ли система соответствовать целям. Если система не соответствует этим целям, она не представляет никакой ценности.

Выполнение анализа осуществимости включает сбор и анализ информации о будущей системе и написание соответствующего отчета. Сначала следует определить, какая именно информация необходима, чтобы ответить на поставленные выше вопросы.

После обработки собранной информации готовится отчет по анализу осуществимости создания системы. В нем должны быть даны рекомендации относительно продолжения разработки системы. Могут быть предложены изменения бюджета и графика работ по созданию системы или предъявлены более высокие требования к системе

### 7.2.2. Формирование и анализ требований

После выполнения анализа осуществимости следующим этапом процесса разработки требований является формирование и анализ требований. На этом этапе команда разработчиков ПО работает с заказчиком и конечными пользователями системы для выяснения области применения, описания системных сервисов, определения режимов работы системы и её характеристик выполнения, аппаратных ограничений и т.д.

В процесс формирования требований могут быть вовлечены люди разных профессий. В нем принимают участие конечные пользователи, которые будут работать с системой, инженеры, которые разрабатывают и эксплуатируют подобные системы, менеджеры, специалисты по предметной области, где будет эксплуатироваться система.

Процесс формирования и анализа требований достаточно сложен по ряду причин.

1. Лица, участвующие в формировании требований, часто не знают конкретно, чего они хотят от компьютерной системы, за исключением наиболее общих положений; им трудно сформулировать, что они ожидают от системы; они могут предъявлять нереальные требования, так как не подозревают, какова стоимость их реализации.

2. Лица, участвующие в формировании требований, выражают в этих требованиях собственные точки зрения, основываясь па личном опыте работы.

3. Лица, участвующие в формировании требований, имеют различные предпочтения и могут выражать их разными способами. Разработчики должны определить все потенциальные источники требований и выделить общие и противоречивые требования.

4. На требования к системе могут влиять политические факторы. Они могут исходить от руководителей, которые предъявляют требования только для того, чтобы усилить свое влияние в организации.

5. Экономическая обстановка, в которой происходит формирование требований, неизбежно будет меняться в ходе выполнения этого процесса. Следовательно, и важность отдельных требований может изменяться.

Новые требования могут быть выдвинуты новым лицом, с которым первоначально не консультировались. Обобщенная модель процесса формирования и анализа требований показана на рис. 7.3. Каждая организация использует собственный вариант этой модели, зависящий от "местных" факторов: опыта работы коллектива разработчиков, типа разрабатываемой системы, используемых стандартов и т.д.


Рисунок 7.3 – Процесс формирования и анализа требований

Процесс формирования и анализа требований проходит через ряд этапов:

1. *Анализ предметной области.* Аналитики должны изучить предметную область, где будет эксплуатироваться система.

2. *Сбор требований.* Это процесс взаимодействия с лицами, формирующими требования. Во время этого процесса продолжается анализ предметной области.

3. *Классификация требований.* На этом этапе бесформенный набор требований преобразуется в логически связанные группы требований.

4. *Разрешение противоречий.* Без сомнения, требования многочисленных лиц, занятых в процессе формирования требований, будут противоречивыми. На этом этапе определяются и разрешаются противоречия такого рода.

*5. Назначение приоритетов.* В любом наборе требований одни из них будут более важны, чем другие. На этом этапе совместно с лицами, формирующими требования, определяются наиболее важные требования.

*6. Проверка требований.* На этом этапе определяется их полнота, последовательность и непротиворечивость.

Таким образом, процесс формирования и анализа требований циклический, с обратной связью от одного этапа к другому. Цикл начинается с анализа предметной области и заканчивается проверкой требований. Понимание требований предметной области увеличивается в каждом цикле процесса формирования требований.

### 7.2.3. Согласование требований

Согласование (аттестация) должна продемонстрировать, что требования действительно определяют ту систему, которую хочет иметь заказчик. Проверка требований важна, так как ошибки в спецификации требований могут привести к переделке системы и большим затратам, если будут обнаружены во время процесса разработки системы или после введения ее в эксплуатацию. Стоимость внесения в систему изменений, необходимых для устранения ошибок в требованиях, рамного выше, чем исправление ошибок проектирования или кодирования. Причина в том, что изменение требований обычно влечет за собой значительные изменения в системе, после внесения которых она должна пройти повторное тестирование.

Во время процесса аттестации должны быть выполнены различные типы проверок документации требований.

1. *Проверка правильности требований.* Пользователь может считать, что система необходима для выполнения некоторых определенных функций. Однако дальнейшие размышления и анализ могут привести к необходимости введения дополнительных или новых функций. Системы предназначены для разных пользователей с различными потребностями, и потому набор требований будет представлять собой некоторый компромисс между требованиями пользователей

2. *Проверка. на непротиворечивость.* Спецификация требований не должна содержать противоречий. Это означает, что в требованиях не должно быть противоречащих друг другу ограничений или различных описаний одной и той же системной функции.

3. *Проверка на полноту.* Спецификация требований должна содержать требования, которые определяют все системные функции и ограничения, налагаемые на систему.

4. *Проверка на выполнимость.* На основе знания существующих технологий требования должны быть проверены на возможность их реального выполнения. Здесь также проверяются возможности финансирования и график разработки системы.

Существует ряд методов аттестации требований, которые можно использовать совместно или каждый в отдельности.

1. *Обзор требований.* Требования системно анализируются рецензентами.

2. *Прототипирование.* На этом этапе прототип системы демонстрируется конечным пользователям и заказчику. Они могут экспериментировать с этим прототипом, чтобы убедиться, что он отвечает их потребностям.

3. *Генерация тестовых сценариев.* В идеале требования должны быть такими, чтобы их реализацию можно было протестировать. Если тесты для требований разрабатываются как часть процесса аттестации, то часто это позволяет обнаружить проблемы в спецификации. Если такие тесты сложно или невозможно разработать, то обычно это означает, что требования трудно выполнить и поэтому необходимо их пересмотреть.

4. *Автоматизированный анализ непротиворечивости.*

Если требования представлены в виде структурных или формальных системных моделей, можно использовать инструментальные CASE-средства для проверки непротиворечивости моделей. Этот процесс показан на рис. 7.4. Для автоматизированной проверки непротиворечивости необходимо построить базу данных требований и затем проверить все требования в этой базе данных. Анализатор требований готовит отчет обо всех обнаруженных противоречиях.

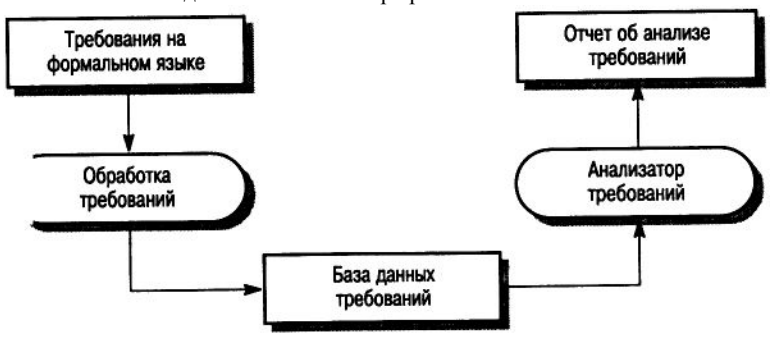



Рисунок 7.4 – Автоматизированный анализ непротиворечивости требований

Трудности аттестации требований нельзя недооценивать. Продемонстрировать, что все требования отвечают потребностям пользователя, очень трудно. Пользователи должны представить систему в действии и вообразить, как эта система впишется в их работу. Это трудно представить даже квалифицированным специалистам, не говоря уже о пользователях системы. В результате аттестации требований редко обнаруживаются все проблемы системной спецификации, поэтому в ней неизбежны изменения даже после согласования документа.

## *Управление требованиями*

Требования к большим системам ПО неизбежно будут изменяться в процессе их разработки. Причины этого многочисленны и разнообразны. Одной из причин является то, что во время процесса создания ПО понимание разработчиками поставленных перед ними задач будет неизбежно меняться, что вызывает необходимость возвращения к требованиям. Кроме того, для больших программных систем, которые приходят на смену действующим, должна быть обеспечена преемственность. Хотя проблемы в работе со старой системой известны, трудно предсказать, какой эффект "улучшенная" система даст для организации. Если конечные пользователи имеют опыт работы с подобной системой, новые требования появляются по ряду причин.

1. Большие системы обычно имеют многообразный контингент пользователей. Разные пользователи имеют различные требования и приоритеты, которые могут быть противоречивыми или несовместимыми. Окончательный вариант системных требований представляет неизбежный компромисс между ними, который часто принимается только на заключительном этапе разработки системы.

2. Заказчики системы и ее пользователи– редко одни и те же люди. Заказчики формулируют требования, руководствуясь своими организационными и бюджетными ограничениями. Они могут входить в противоречие с требованиями конечных пользователей. Деловая среда и техническое окружение системы изменяются, что должно найти отражение в системе. Например, может быть закуплено новое оборудование, может появиться необходимость сопряжения системы с другими системами, деловые приоритеты организации могут измениться, будут введены новые законодательство и стандарты и т.д. Изменения в аппаратных средствах особенно затрагивают нефункциональные системные требования.

Управление требованиями – это процесс управления изменениями системных требований. Процесс управления требованиями выполняется совместно с другими процессами разработки требований. Начало этого процесса планируется на то же время, когда начинается процесс первоначального формирования требований, непосредственно процесс управления требованиями должен начаться сразу после того, как черновая версия спецификации требований будет готова.

При формировании требований основное внимание сосредоточено на возможностях создаваемого ПО, целях и других системах организации. После формирования требований достигается более глубокое понимание потребностей пользователей, вследствие чего может возникнуть необходимость в изменении ранее сформулированных требований. Измененные требования отсылаются заказчику с объяснением причины сделанных изменений.

Создание большой системы может занять несколько лет. За это время окружение и требования к системе, несомненно, изменятся, что также должно найти отражение в измененных требованиях.

С точки зрения разработки требования можно разделить на два класса.

1. *Постоянные требования.* Это относительно стабильные требования, которые исходят из основной деятельности организации и касаются непосредственно предметной области, где будет эксплуатироваться система.

2. *Изменяемые требования.* Эти требования отображают изменения, сделанные во время разработки системы или после ввода ее в эксплуатацию.

Таблица 7.4. Классификация изменяемых требований

| Тип требований | Описание |
|---|---|
| Непостоянные требования | Требования, которые изменяются из-за изменений в окружении системы |
| Неожиданно возникающие требования | Требования, которые появляются во время разработки системы. В процессе проектирования может возникнуть необходимость добавления новых требований |
| Непрямые требования | Требования, которые являются результатом внедрения компьютерной системы, способной изменить организационные процессы и показать новые способы работы, которые приведут к новым системным требованиям |
| Вторичные требования | Требования, которые зависят от особенностей данной системы или от проблем организации |

**Контрольные вопросы**

1. Охарактеризуйте общие требования к информационной системе из нескольких.
2. Из каких уровней состоит архитектура информационной системы?
3. Что такое функциональные, нефункциональные требования и требования предметной области
4. Как формируются пользовательские требования?
5. Какова роль системных требований?
6. Опишите особенности документирования требований.
7. Что представляет из себя спецификация требований?
8. Как осуществляется формирование требований?
9. Каковы основные этапы анализа требований?
10. Как осуществляется согласование и управление требованиями?