ПРОЕКТИРОВАНИЕ И АРХИТЕКТУРА ПРОГРАММНЫХ СИСТЕМ

В В Е Д Е Н И Е.

Совершенствование и развитие промышленности, связи и транспорта тесно связано с интенсификацией производства и развитием работ в области прогрессивных технологий и комплексно-автоматизированных производств.

Важным этапом в проведении крупных мероприятий по совершенствованию технической и технологической базы, а также в использовании новых методов организации производства становится создание различных программных (аппаратных) систем (ПС), основанных на широком применении современного программно-управляемого технологического оборудования, микропроцессорных управляющих, вычислительных средств, средств автоматизации проектно-конструкторских, технологических и планово-производственных работ.

С внедрением ПС практически во всех сферах деятельности человека, с увеличением их сложности и стоимости, требования, предъявляемые к их качеству и эффективности, постоянно возрастают.

ПС при всем их многообразии имеют много общих черт, основу которых составляют процессы сбора, обработки, хранения и представления данных.

В общем случае создание ПС происходит при наличии ограничений двух типов.

Первый тип ограничений характеризует уровень современных знаний о методах постановки задач, принципах построения основных функциональных алгоритмов, методах структурного построения сложных систем и технологии их проектирования.

Второй тип ограничений относится в основном к техническим параметрам средств, на которых предполагается реализовать сложную систему, и к ресурсам, которые могут быть выделены на разработку и эксплуатацию системы.

На современном этапе, в большинстве случаев, качественные и количественные показатели ПС улучшаются за счет уменьшения ограничений второго типа, то есть с помощью использования все время совершенствующейся элементной базы (вычислительная техника, автоматика, радиоэлектроника, средства связи, информатика и др.). Однако отечественный и зарубежный опыт показывает, что большой потенциал повышения эффективности ПС заложен еще на стадии разработки - постановки задачи, сборе исходных данных, формализации предметной области.

На современном уровне развития эта стадия, как правило, выполняется на содержательном уровне, на основе логики, здравого смысла, инженерного опыта и интуиции.

Методы проектирования ПС позволяют формализовать некоторые этапы этой стадии, что дает возможность с использованием уже научных методов (в строгом смысле слова) улучшить качество их разработки.

1. ПОНЯТИЯ И ПРИНЦИПЫ МЕТОДОЛОГИИ ПРОЕКТИРОВАНИЯ.

Название "Методология проектирования" объединяет два понятия: методология и проектирование. Методология вообще есть учение о структуре, логической организации, методах и средствах деятельности. Под проектированием понимается процесс составления описания, необходимого для создания еще не существующего объекта. На этой основе методологию проектирования можно определить как учение о структуре, логической организации, методах и средствах составления описания, необходимого для создания еще не существующего объекта. Несмотря на то, что в ГОСТах указывается, каким образом составляется описание, а именно: путем преобразования первичного описания, оптимизации заданных характеристик объекта или алгоритма его функционирования, устранения некорректности первичного описания и последовательного представления описания на различных языках. Необходимо в определении отразить дополнительно: целенаправленность проектирования на удовлетворение определенной потребности, и особой его

процедуры, предшествующие составлению описания и относящиеся к творческим актам поиска и принятия решений.

С учетом дополнений *под методологией проектирования будем понимать учение о структуре, логической организации, методах и средствах поиска и принятия решений о принципе действия и составе еще не существующего объекта, наилучшим образом удовлетворяющего определенные потребности, а также составление описания, необходимого для его создания в заданных условиях.*

Остановимся на некоторых основных понятиях:

Проектное решение - промежуточное или конечное описание объекта проектирования, необходимое и достаточное для рассмотрения и определения дальнейшего рассмотрения или окончания проектирования.

Алгоритм проектирования - совокупность предписаний, необходимых для выполнения проектирования.

Язык проектирования - язык, предназначенный для представления и преобразования описаний при проектировании.

Проектная процедура - совокупность действий, выполнение которых оканчивается проектным решением.

Сложность современных объектов определяет и сложность задач проектирования. Они не могут быть решены сразу прямым замыканием входной информации на постоянную концептуальную модель действительности, а требуют развернутого во времени сложного информационного поиска.

Множественность путей достижения цели проектирования требует рассмотрения не одного, а многих вариантов технического решения, к каждому из которых применяются определенные методы анализа и оценки.

Современные методы проектирования должны быть ориентированы на широкое использование ЭВМ, не исключая человека при решении наиболее сложных и творческих задач.

Сформулируем **основные задачи методологии** проектирования с учетом приведенных особенностей изучаемых ею методов. Декомпозиция требует логической схемы последовательности действий, наилучшим образом организующей процесс проектирования. Построение такой схемы будем считать первой задачей методологии проектирования. Стремление к широкому использованию ЭВМ требует формализации процедур, а это, в свою очередь, - математической модели как процесса, так и объекта проектирования. Разработка математических моделей составляет вторую, а методы и алгоритмы выполнения проектных процедур - третью задачу методологии.

2. СИСТЕМНЫЙ АНАЛИЗ.

Основным научным методом изучения систем стал системный метод или системный анализ [14,22]. В общем случае, *под системным анализом понимают всестороннее, систематизированное, т.е. построенное на основе определенного набора правил, изучение сложного объекта в целом, вместе со всей совокупностью его сложных внешних и внутренних связей, проводимое для выяснения возможностей улучшения функционирования этого объекта*.

В зависимости от характера используемого набора правил, системный анализ можно выполнять на уровне логики или здравого смысла. Научным методом системный анализ является лишь тогда, когда на всех его этапах используется научный подход, в основу которого берется количественный анализ.

Методологически системный анализ состоит из следующих этапов: постановки задачи, структуризации системы, построения и исследования модели.

Постановка задачи разработки ПС. На этом этапе специалист руководствуется вопросами Что, Где, Когда, Как и Почему. При постановке задачи разработки ПС специалистами, работающими в исследуемой системе, последние, в большинстве случаев, односторонне выхватывают какой-либо один аспект деятельности системы, не учитывают многообразия и взаимосвязи раз-

личных факторов в системе и ее внешней среде. Этап постановки задачи весьма важен для последующей работы, от него существенно зависит, какие будут результаты.

Структуризация системы. Структуризация - второй этап системного анализа. Прежде всего, необходимо определить набор всех элементов, в той или иной степени связанных с поставленной на предыдущем этапе задачей, и разделить их на два класса - исследуемую систему и внешнюю среду. Такое деление существенно зависит от поставленной задачи - при ее изменении меняются границы системы, внешняя среда, а иногда первоначальный набор элементов.

Во внешней среде локализуют в виде подсистем элементы, образующие вертикаль исследуемой системы: вышестоящие, подчиненные и подсистемы одного с нею уровня.

Структуризация самой системы заключается в разбиении ее на подсистемы в соответствии с поставленной задачей. Завершается этап структуризации определением всех существенных связей между нею и системами, выделенными во внешней среде. Тем самым для каждой из выделенных в процессе структуризации систем определяют ее входы и выходы.

Построение и исследование модели ПС. Моделирование - третий этап системного анализа, который используют для изучения и анализа любых сложных систем, процессов и объектов.

Модель - это приближенное, упрощенное представление процесса или объекта. С помощью моделей можно получить характеристики системы или отдельных ее частей значительно проще, быстрее и дешевле, чем при исследовании реальной системы. Естественно, это влечет за собой снижение точности. Таким образом, при моделировании системы мы всегда вынуждены идти на компромисс между простотой модели и обеспечиваемой ею точностью. Модель считают адекватной, если она обеспечивает точность, достаточную для данного исследования.

Завершающим этапом системного анализа является исследование модели. Основным назначением этого этапа является выяснение поведения моделируемого объекта в различных условиях, при разных состояниях самого объекта и внешней среды.

Системное моделирование, прежде всего, связано с многоаспектным исследованием рассматриваемых задач. В общем случае, изменяя число принимаемых во внимание аспектов задачи, можно влиять на качество ее решения с помощью системной модели. Таким образом, многоаспектное исследование порождает противоречие: с одной стороны, полезно увеличивать число аспектов рассмотрения, так как это позволяет в пределе включить все существенные аспекты в системную модель, но с другой, - возможности такого увеличения обычно лимитированы, что вынуждает упускать из виду многие аспекты, в том числе и существенные.

На практике выделяют две стороны системного подхода: **познавательную (дескриптивную) и конструктивную**.

Дескриптивное определение системы в общем виде может быть следующим: *система - это совокупность объектов, свойства которой определяются отношениями между ними*. Дескриптивный подход в качестве познавательного позволяет объяснить функции, выполняемые системой.

Конструктивная формулировка: *система - это конечное множество элементов и отношений между ними, выделяемое из среды в соответствии с конкретной целью в рамках требуемого интервала времени*.

Если дескриптивное описание системы полезно при изучении естественных систем, когда человек пытается объяснить окружающую его действительность, то при создании технических систем для исследователя более важен конструктивный подход, который отвечает на вопрос, как следует строить систему, чтобы она удовлетворяла поставленной цели и обеспечивала требуемые от нее функции.

Познавательный и конструктивный подходы позволяют реализовать две основные проблемы построения системы, а именно: проблему **анализа** и проблему **синтеза** структуры.

Рассмотрим более полное определение и понятие структуры.

Одно из главных направлений в области совершенствования программно-аппаратных комплексов - проектирование структур, определяющих основные свойства и характеристики функционирования систем.

Структура системы отражает строение и внутреннюю форму организации, прочные и относительно устойчивые взаимоотношения и взаимосвязи элементов системы.

В философии имеются понятия формальной, логической и материальной структур системы. При формализации структуры весьма важными моментами являются: классификация структур; выявление класса преобразования структур, инвариантных по отношению к заданной цели; развитие формальных методов анализа и синтеза структур; выбор оптимальных структур.

Под **формальной структурой** обычно понимают совокупность функциональных элементов и отношений между ними, необходимых и достаточных для достижения системой заданных целей.

Материальной структурой называют реальные наполнения формальной структуры. Введение понятия структуры существенно, прежде всего потому, что она определяет общие закономерности состава и взаимосвязей системы.

Основой понятия "система" является наличие связей между объединенными в систему элементами, определяемых некоторой общей закономерностью, правилом или принципом. Элемент, не имеющий хотя бы одной связи с другими, просто не входит в рассматриваемую систему.

Наличие взаимосвязей между элементами определяет особое свойство сложных систем - **организованную сложность**. Система в целом не только обладает иными характеристиками, чем составляющие ее элементы, но качественно отличается от простой суммы составляющих ее частей, имеет свойства, выполняет новые функции, которых нет у ее элемента. Появление таких новых качеств, определяемых свойством целостности системы, иногда называют **эмерджентностью**.

В рамках понятия ПС удобнее всего дать определение "системы" в следующем виде.

Система - целенаправленное множество взаимосвязанных элементов любой природы. Общим свойством, объединяющим элементы в систему, является в данном определении их направленность на достижение цели.

При описании "системы" вводятся понятия:

Внешняя среда - множество существующих вне системы элементов любой природы, оказывающих влияние на систему или находящихся под ее воздействием в условиях рассматриваемой задачи.

Замкнутая система - система, любой элемент которой имеет связи только с элементами самой системы.

Открытой называют систему, у которой, по крайней мере, один элемент имеет связь с внешней средой.

Подсистема - выделенное из системы по определенному правилу или принципу целенаправленное подмножество взаимосвязанных элементов любой природы.

Совокупность связей между элементами системы, отражающую их взаимодействие, называют структурой.

Все подсистемы, полученные непосредственным выделением из одной исходной, относят к подсистемам одного **уровня или ранга**. При дальнейшем делении получаем подсистемы все более низкого уровня. Такое деление называют **иерархией**. Одну и ту же систему можно делить на подсистемы по разному - это зависит от выбранных правил объединения элементов в подсистемы. Наилучшим, очевидно, будет набор правил, который обеспечивает системе в целом наиболее эффективное достижение цели.

3. АКСИОМАТИКА СЛОЖНЫХ СИСТЕМ.

Обобщим ряд положений системного анализа в набор аксиом существования сложных систем:

– Аксиома целостности (эмерджентность). Свойства сложной системы не есть простая сумма свойств подсистем.

– Аксиома автономности. Сложная система подчиняется своему частному закону (метрике) не зависящему от внешней среды. Частные законы имеют модельный характер и действуют только при адекватности модели.
– Аксиома дополнительности. Сложная система во взаимодействии со средой может проявлять различные свойства в различных ситуациях.
– Аксиома действия (реактивности). Реакция системы на воздействие имеет пороговый характер. Начиная с определенного уровня (порогового значения) меняются системные свойства самой системы.
– Аксиома неопределенности (инертности). Чем точнее измерение, тем больше затраты времени, тем больше изменений в самой системе, тем больше ошибки измерения.
– Аксиома выбора. Выбор текущего состояния из множества возможных состояний сложной системы как реакции на внешнее воздействие в зависимости от внутренних критериев целенаправленности не может быть однозначно предсказан.

Сегодня, когда развитие аппаратно-технических средств и формирование новых знаний в области программной индустрии идет огромными темпами, а консервативная структура современных технологий не учитывает эту диалектику развития, учет аксиомы инертности становится как никогда актуальным, а сама аксиома принимает новый вид аксиомы *диалектической инертности*:

Чем больше времени затрачено на разработку программной системы, тем больше качественных изменений (аксиома действия) в окружающей среде (в том числе инструментальных аппаратно-программных средствах, уровне знаний разработчиков, потребностях заказчиков и др.), тем больше готовый продукт не удовлетворяет начальным требованиям.

Таким образом, мы имеем противоречие, возникшее при применении современных технологий разработки ИС. Как правило, акцент делается на учете аксиомы автономности, предполагающей выдвижение ограниченного закона функционирования, строящегося на конкретных, четко классифицированных метриках и формальных моделях представления знаний. При этом технология входит в противоречия с аксиомой диалектической инертности, так как необходим формальный аппарат интегрированный в концепцию самой технологии и учитывающий эволюцию метрик и моделей представления знаний. Это противоречие может быть устранено только на базе создания новой технологии разработки ИС, учитывающей различные факторы, формирующие знания о программном продукте как артефакте подчиняющемся законам системотехники и созданном человеком.

Наиболее существенными чертами сложных систем являются:
– наличие общей задачи и единой цели функционирования для всей системы;
– большое количество взаимодействующих частей или элементов, составляющих систему;
– возможность расчленения на группы наиболее тесно взаимодействующих элементов - подсистемы, имеющие свое специальное назначение и цель функционирования;
– иерархическую структуру связей подсистем и иерархию критериев качества функционирования всей системы;
– сложность поведения системы, связанную со случайным характером внешних воздействий и большим количеством обратных связей внутри системы;
– устойчивость по отношению к внешним и внутренним помехам и наличие самоорганизации и адаптации к различным возмущениям;
– высокую надежность системы в целом, построенной из не абсолютно надежных компонент.

4. ЖИЗНЕННЫЙ ЦИКЛ ИС.

В качестве конструктивного дополнения характеризующего производственную, прагматическую сущность методологии разработки ПС отметим следующее определение: «Современная индустриальная технология проектирования программ включает в себя комплекс мероприятий, руководящих документов и автоматизированных средств, предназначенных для системного ана-

лиза, разработки, отладки, документирования, управления работой специалистов и контроля эксплуатации программ».

Немаловажную роль в определении процесса инженерного проектирования играет структура жизненного цикла программных систем (см. рис.1.), которая позволяет очертить место этого процесса в общей картине эволюции разрабатываемой системы, а также учитывать ряд влияющих внешних факторов, в том числе:

- экономические – оценка рынка (спроса и предложения), учет производственных затрат и т.д.;
- эргономические – учет эстетичного вида и медицинской безопасности интерфейса не только самой программной системы, но и учет товарной привлекательности готового продукта;
- психологические – учет психологии заказчика или покупателя, а также человеческого фактора в лице специалиста(-ов) по предметной области в процессе постановки задачи.

В научной литературе, как правило, процесс разработки ПС носит структурированный характер, выражающийся в разбиении его на стадии и этапы. При этом, большинство подходов к такому разбиению ограничивается манипулированием и последовательной фиксацией названий стадий как элементов из конечного множества терминов: техническое задание, эскизный проект, технический проект, рабочий проект, внедрение и дальнейшей их детализации через описание этапов. Разбиение на этапы также характеризуется индивидуальностью авторских подходов. Можно только выделить наиболее повторяющиеся названия этапов проектирования: системный анализ и проектирование алгоритмов, структурное проектирование, подготовка технологических средств, разработка программ, отладка программ в статике, комплексная динамическая отладка, выпуск машинных носителей и документирование, испытание программных средств и прочее. При этом за основу каждого этапа берутся разные классы моделей представления и, соответственно, разные языки описания этих моделей, и разный формальный аппарат анализа и синтеза (текстовое описание, графические нотации, математические модели и др.).

К сожалению, структуризация стадий и этапов процесса проектирования, выбор моделей представления носит субъективных характер. Из исследований материалов по этому вопросу можно отметить только существование зависимости стадий и этапов от сложности разрабатываемой программной системы.

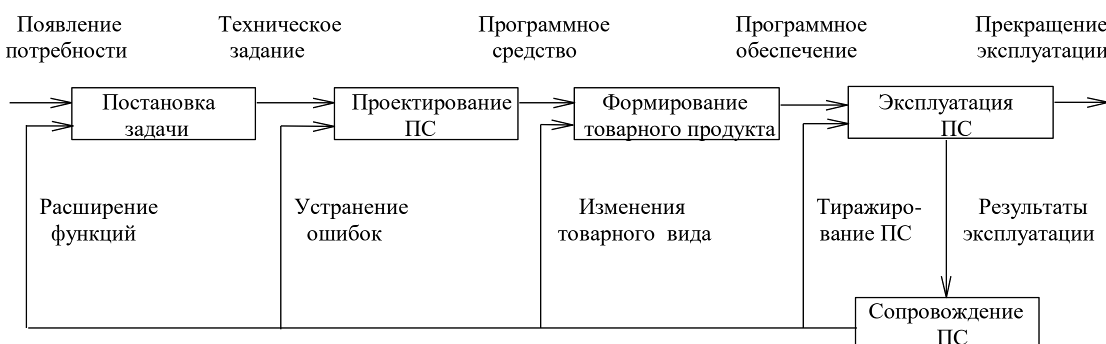


Рис.1. Обобщенная схема этапов жизненного цикла.

5. ДВЕ ПАРАДИГМЫ ПРОЦЕССА РАЗРАБОТКИ ИНФОРМАЦИОННЫХ СИСТЕМ.

Разделим методологию на две парадигмы:

1. Апрагматическая методология – конструктивная часть методологии разработки ПС, которая отвечает на вопрос «что проектируется?» или «что моделируется?». Она описывает продукты проектирования как особого рода системотехнические артефакты. В рамках данной методологии решаются задачи параметрического синтеза и анализ.
2. Прагматическая методология - дескриптивная часть методологии, описывающая процесс разработки ПС, как систематической процедуры, носитель когнитивного начала отвечающего на

вопрос «как проектировать?», «какую модель (модели) представления использовать?». С парадигмой прагматической методологии, как правило, связаны такие исследования как инженерное проектирование, представление моделей знаний, технология разработки.

Концептуальный аналог данных парадигм можно найти у ряда авторов, в том числе по Г. Майерс: «Всякая методология проектирования программного обеспечения состоит из двух частей: набора желательных характеристик результата и руководящих принципов самого процесса проектирования».

При разработке программных систем обычно начинают с определения желательных характеристик (апрагматическая составляющая методологии), а затем конструируют мыслительный процесс (прагматическая составляющая методологии), необходимый для получения нужного результата. Поэтому можно предположить, что о желательных свойствах результата известно больше, чем о самом процессе.

Основным недостатком апрагматической методологии является ее направленность на решение ограниченного круга выделенных технических проблем, связанных с формальным представлением именно объекта разработки как технической системы. При этом игнорируются факторы, оказывающие существенное влияние на процесс разработки ПС как систему действий, в том числе: «поведенческие» изменения субъекта проектирования, качественные изменения технологических средств и другие. Апрагматическая методология охватывает лишь малую толику проблем разработки ПС, рассматривая их через призму формальных аппаратов параметрического синтеза и анализа, как результат технологического действия, элиминированно и аддитивно от других подсистем, оказывающих активное воздействие на весь процесс разработки ПС.

В противовес технической, объектной «приземленности» апрагматической методологии прагматическая парадигма уходит в другую крайность – область теоретических концепций построения системы «идеальных» планов действий. Основной системой исследования является «система технологических действий», при этом объект разработки рассматривается как инертный выходной продукт процесса разработки. Эта инертность инкриминируется объекту путем задания обобщенных, инвариантных к реализации формальных моделей представления объекта разработки.

Таким образом, обе описанные парадигмы выступают довольно часто антагонистическими концепциями позиций и взглядов различных ученых на подходы к технологиям разработки вообще. Однако современная диалектика информационных технологий диктует необходимость создания новой интегрирующей методологии, основанной на единстве дуализма «апрагматическая-прагматическая парадигма», обусловленном концепциями новой технологии верификации программных систем как продукта научно-технической, инженерной деятельности.

Классифицируем существующие подходы к методологии разработки программных систем с позиций высказанных парадигм.

Апрагматическая методология характеризуется частными методологиями. Именно в ней накоплен большой практический опыт, систематизированы и изучены обширные знания. К разряду частных методологий будем относить научные работы, посвященные различным направлениям гносеологии, в том числе распознавание, математическое моделирование, прогнозирование, схемотехника и пр.

Апрагматическую методологию моделирования можно описать как отображение М на множество оценок среза произведения бинарных отношений: множества целей С и множества признаков Р, множества признаков и множества технических решений Х. Тогда целевая функция проектирования запишется как:

$$(\mathbf{M}: (\mathbf{G} * \mathbf{J}(\mathbf{C})) \to \mathbf{E}) \to \mathbf{opt},$$

где **E** - множество оценок, $\mathbf{J} \subset (\mathbf{C} \times \mathbf{P})$, $\mathbf{G} \subset (\mathbf{P} \times \mathbf{X})$.

Исходя из вышеизложенного, задача модельного представления системы может быть представлена следующим образом: определить композицию первичных элементов **S**, выделенных из универсального множества **Sa** на основании **A**, взятых в отношении **R** и объединенных по композиционным правилам **Z** так, что в результате этого объединения появляются системообразую-

щие свойства $\mathbf{Ks_c}$, а также общетехнические свойства $\mathbf{Ks_o}$, следствием чего является возникновение интеграционных свойств $\mathbf{Ks_p}$, не адекватных свойствам первичных элементов и не являющихся линейной комбинацией системообразующих свойств системы $\mathbf{Ks_c}$. При этом оператор **Q** отражает соответствие данному набору интеграционных и системообразующих свойств определенной количественной мере $\mathbf{K_q}$ потребительского качества синтезируемой системы.

Выбор оптимальной организации моделируемой системы осуществляется на основе оценки принятых свойств системы $\mathbf{Ks=Ks_o \cup Ks_c \cup Ks_p \cup Ks_q}$ (общетехнических свойств $\mathbf{Ks_o}$, системообразующих свойств $\mathbf{Ks_c}$, интеграционных потребительских свойств $\mathbf{Ks_p}$ и свойств качества $\mathbf{Ks_q}$).

При апрагматической методологии, как правило, фактор оценки спроектированной модели системы имеет сравнительный характер модельных параметров по отношению к параметрам моделируемого объекта и определяется в виде критерия соответствия:

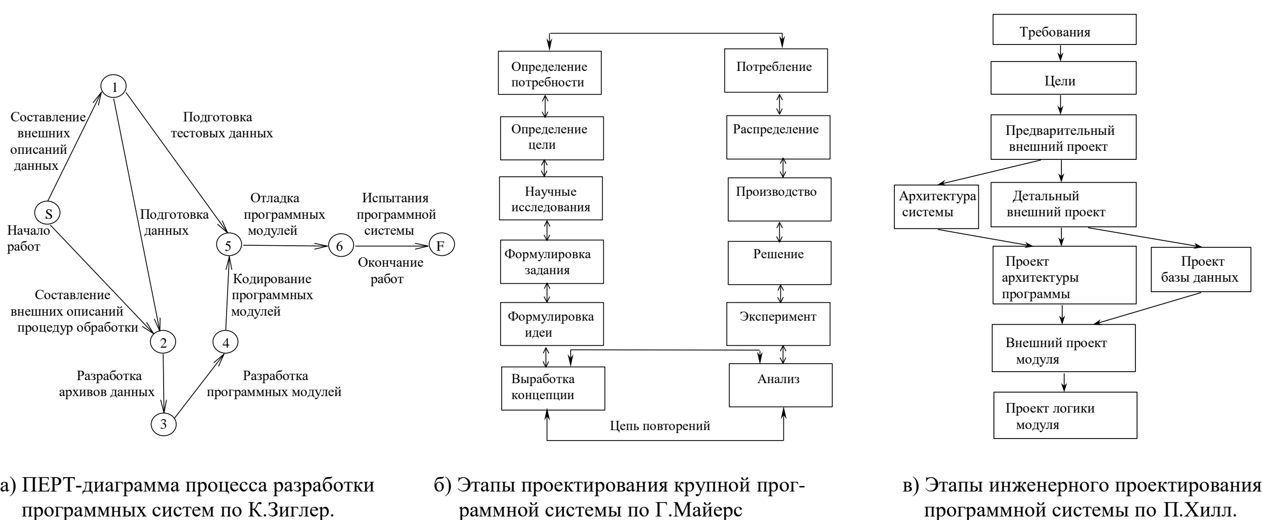
$\mathbf{E_m=Ks^p-Ks^m}$,

где $\mathbf{Ks^p}$ – выходные характеристики реального объекта, $\mathbf{Ks^m}$ – выходные характеристики модели.

Прагматическая методология. Для прагматической методологии вряд ли вообще уместно разрабатывать частные методики, так как присущие им проблемы обнаруживают много общего и могут быть охвачены общей методологией. За основу прагматической методологии положим обобщение некоторых идей, высказанных в ряде научных публикаций.

1) Инженерное проектирование представляется как наука, систематизирующая знания, и где особое внимание акцентируется на этапах создания систем и их взаимосвязи, а методы разработки ориентированы на организацию самого процесса проектирования и связаны с логической организацией идей. На рис.1.1 представлены обобщенные схемы прагматических подходов при разработке программных систем у ряда авторов.

2) Интересна идея трактовать формализацию программных систем как процесс, который кладет начало изменению в искусственной среде. В этом случае целью методологии разработки ПС является уменьшение цикличности и увеличение линейности этого процесса. Существенным методом обеспечения линейности процесса разработки ПС является прогнозирование, позволяющее определить диапазон возможных выходов до их выполнения.



а) ПЕРТ-диаграмма процесса разработки программных систем по К.Зиглер.

б) Этапы проектирования крупной программной системы по Г.Майерс

в) Этапы инженерного проектирования программной системы по П.Хилл.

Рис.1.1. Схемы подходов к прагматическому проектированию.

Исходные предпосылки разработки качественных программных систем.

6. КАЧЕСТВО ПРОГРАММНЫХ СИСТЕМ.

Каждая программа, входящая в систему, должна отвечать таким *требованиям, как правильность, точность, совместимость, надежность, универсальность, защищенность, полезность, эффективность, проверяемость и адаптируемость*. Будем говорить, что программа является

- правильной, если она функционирует в соответствии с техническим заданием. Подразумевается, что техническое задание составлено в четкой форме, позволяющей однозначно судить о том, действительно ли программа отвечает перечисленным в нем требованиям;
- точной, если выдаваемые ею числовые данные имеют допустимые отклонения от аналогичных результатов, полученных с помощью идеальных математических зависимостей;
- совместимой, если она работает должным образом не только автономно, но и как составная часть всей программной системы, осуществляющей обработку информации;
- надежной, если она при всех условиях обеспечивает полную повторяемость результатов.
- универсальной, если она правильно работает при любых допустимых вариантах исходных данных. В ходе разработки программ должны предусматриваться специальные средства защиты от ввода неправильных данных, обеспечивающие целостность системы;
- защищенной, если она сохраняет работоспособность при возникновении сбоев. Это качество особенно важно для программ, предназначенных для решения задач в режиме реального времени. В подобных приложениях отказ оборудования может повлечь катастрофические последствия - например, аварию ракеты или ядерного реактора. Указанными свойствами должны также обладать программы с большим временем выполнения, осуществляющие обработку постоянно хранимых файлов;
- полезной, если задача, которую она решает, представляет практическую ценность;
- эффективной, если объем требуемых для ее работы ресурсов не превышает допустимого предела;
- проверяемой, если ее качества могут быть продемонстрированы на практике. Здесь подразумевается возможность проверки таких свойств программы, как правильность и универсальность. Можно применить формальные математические методы, позволяющие установить, действительно ли программа удовлетворяет техническим условиям и выдает достаточно точные результаты. Однако существует и неформальные способы оценки качества программ, причем иной раз они оказываются более убедительными, чем формальные. Имеются в виду такие неформальные приемы, как прогоны с остановами в контрольных точках, обсуждения результатов с заинтересованными пользователями и др.;
- адаптируемой, если она допускает быструю модификацию с целью приспособления к изменяющимся условиям функционирования. Адаптируемость в значительной степени зависит от конструкции программы, от того, насколько квалифицированно она составлена и полно снабжена документацией.

В какой степени удовлетворяют перечисленные требования, характеризующие качество программ, определяется знаниями и мастерством разработчиков и программистов. Хотя программисты часто пишут небольшие процедуры для своих собственных нужд, все же большинство программ составляется для тех пользователей, которые не обладают должными навыками программирования. На любой вычислительной установке пользователь, прикладной программист, оператор ЭВМ, техник, отвечающий за ввод данных, системный программист - это разные лица, и качество программ сказывается на деятельности каждого из них.

7. СРЕДЫ РАЗРАБОТКИ.

Часто программы создаются коллективами программистов. Некоторые из них занимаются в основном разработкой, некоторые - самим программированием, остальные - составлением документации и испытаниями программы. Если разработка алгоритмов и программирование выполняются разными группами специалистов, за подготовку документации отвечают все группы.

Машинные программы - это своего рода рабочие инструменты, и пользователь нуждается в определенных инструкциях по их применению и правильному обращению с ними. Подобные инструкции должны содержаться в документации, поставляемой вместе с программой. Документация - столь же обязательный продукт процесса программирования, сколь и сама программа. Если программа взаимодействуют с ЭВМ непосредственно, связь ее с людьми обеспечивается в основном с помощью документации.

Программы редко применяются как самостоятельные единицы. Чаще всего они являются элементами более крупных информационных систем, осуществляющих сбор, хранение и обработку данных. Такие системы становятся неотъемлемой частью механизма функционирования предприятий, на которых они эксплуатируются. Прямо или косвенно, они затрагивают деятельность множества людей. Чтобы это воздействие носило положительный характер, программы не просто должны быть полезными, надежными и эффективными, но должны явно обнаруживать эти качества для тех, кто с ними работает.

Тематике разработке информационных систем с позиции планирования и управления сложными и большими проектами, учитывающими специфику коллективов разработчиков, посвящено множество исследований.

Выделим четыре основных среды, на которые необходимо обратить внимание с расчетом на успешное окончание проекта.

1. Среда пользователей. Как процесс проектирования программной системы, так и его конечный продукт должны быть ориентированы на нужды пользователей.

Пользователи будут уверены в эффективности системы, если почувствуют, что в группе, занимающейся ее разработкой, прислушиваются к их пожеланиям, если найдут, что форма входных данных и результатов удобна и близка к привычной, и, наконец, если им будет продемонстрировано, что система должным образом перерабатывает информацию, отобранную по их собственному усмотрению. Если пользователи принимают участие в проекте на стадии разработки, они лучше осведомлены о характеристиках системы и могут внести свою лепту в формирование ее окончательного облика. Если же пользователи привлекаются на этапе испытаний, они получают возможность оценить качества системы еще до начала эксплуатации. Располагая квалифицированно составленной, исчерпывающей документацией, пользователи смогут быть уверены, что система будет работать с полной отдачей.

Пользователями программной системы могут быть служащие административных учреждений, для которых на ЭВМ подготавливаются отчеты и ведомости, или инженеры, выполняющие на машинах научно-технические расчеты. В число пользователей входят также работники из вспомогательного персонала, отвечающие за ввод информации и заполнение бланков входных форм или контролирующие правильность и точность данных, выдаваемых ЭВМ. Все лица, на том или ином этапе принимающие участие в процессе обработки информации, от ее сбора до оформления результатов, еще на стадии проектирования системы должны быть ознакомлены с теми ее особенностями, которые должны приниматься в расчет в их практической деятельности.

2. Среда программного и аппаратного обеспечения. Программа неотделима от вычислительной среды, с которой взаимодействует. Она использует системные программные средства, а те в свою очередь могут пользоваться ее информацией. Программа либо сама создает файлы, либо обрабатывает файлы, сформированные другими программами. Она должна быть построена таким образом, чтобы могла применяться в различных приложениях и обходиться только имеющимися аппаратными ресурсами и средствами программирования. Процесс разработки программы в значительной степени зависит от наличия специализированных языков проектирования, каталогов данных, оптимизирующих компиляторов, генераторов тестовых задач. Существенно проще создать хорошую программу, располагая эффективными вспомогательными средствами.

3. Среда заказчика. Обычно работа по составлению программ начинается в связи с тем, что некоторая организация предлагает создать для нее программную прикладную систему. Официальному заключению договора обычно предшествует выяснение реальной необходимости в такой системе, оценка возможности ее разработки и примерного объема затрат, а также ожидаемого эффекта от ее внедрения. Несмотря на то, что ЭВМ можно спроектировать и запрограмми-

ровать так, что она способна решить любую задачу, которую пользователь может описать в виде формальных правил, определяющих последовательность действий, применение вычислительных машин далеко не всегда оправдано. ЭВМ лучше, чем человек, справляется с трудоемкими задачами, требующими многократного повторения однотипных операций, значительно быстрее и точнее выполняет арифметические вычисления. ЭВМ более эффективно осуществляет поиск и обработку больших массивов информации, состоящих из однородных элементов. В то же время человек лучше, чем машина, может разобраться в том, что и как следует делать, и способен работать с неоднородной информацией. Всякое использование ЭВМ предполагает стандартизацию данных и способов обработки. Эффективная реализация преимуществ ЭВМ возможна лишь в тех случаях, когда необходимо выполнять либо трудоемкие вычисления, либо обработку больших объемов информации.

После завершения этапа предварительных исследований составляется список требований, предъявляемых к системе. В него должны быть включены результаты анализа обстановки, описание выполняемых системой функций и ограничения, которые необходимо учитывать в процессе проектирования. Под обстановкой в данном случае понимается совокупность условий, при которых предполагается эксплуатировать систему. К ним относятся аппаратные и программные ресурсы, предоставляемые системе, внешние условия ее функционирования, а также состав людей и работ, имеющих к ней отношение. Должны быть продуманы изменения в текущей деятельности организации, обусловленные внедрением системы. Возможно, понадобится иная расстановка персонала, придется внести изменения в структуру выполняемых работ. Могут также потребоваться дополнительные вычислительные мощности.

Функциональные требования к системе содержат четкое описание всего того, что она должна делать. Ограничениями в процессе проектирования являются директивные сроки завершения отдельных этапов, имеющиеся в наличии ресурсы, организационные процедуры и мероприятия, обеспечивающие сохранность информации.

Организация-заказчик и группа разработчиков совместно составляют официальный перечень спецификаций, а также договор о порядке проведения проектных работ и приемке системы. Иногда процесс создания системы разбивается на два отдельных этапа, в которых участвуют различные группы специалистов. Первая из них занимается собственно проектированием системы, вторая - ее программной реализацией. В таких случаях договоры заключаются с обеими группами, причем между указанными этапами должен быть предусмотрен определенный промежуток, выделяемый для анализа и обсуждения характеристик системы.

4. Среда разработчиков. Первая из возникающих в проекте проблем, относящаяся одновременно к психологии, социологии и технике информатики, это проблема организации бригады. Два первых аспекта занимают нас здесь лишь в той мере, в какой они испытывают влияние третьего, и тем самым отличаются в программировании от того, чем они могли бы стать в любой другой дисциплине, требующей коллективных усилий.

Какие работы надо распределить? Некоторые из них очевидны: нужны программисты в широком смысле, в котором мы употребляем этот термин ("аналитики", "функциональные аналитики". "проектировщики" и т.д.); прикладные программисты - владеющие современными программными и аппаратными средствами и методикой разработки прикладных программных систем; системные программисты - специалисты в области системных программ (операционных систем, драйверов), владеющие спецификой устройства вычислительной техники на уровне микросхем и обработки прерываний. Вероятно, будет еще руководитель проекта. Заметим по этому поводу, что "технический" руководитель может отличаться от "административного", поскольку имеются две одинаковые опасности доверить техническое руководство проектом "менеджеру", некомпетентному в программировании, или загрузить крупного специалиста по информатике административными вопросами. Менее широко признаны, но все же важны в большом проекте следующие обязанности:

- технический помощник - член бригады, который обладает особым опытом в одной или нескольких конкретных областях: язык программирования, используемое оборудование, система управления базами данных, документирующие системы и т.д. Распределяя определенным обра-

зом свое время, такой специалист поступает в распоряжение своих коллег, чтобы помочь в разрешении встречающихся им технических трудностей. Ясно, что эти обязанности могут делить несколько членов бригады, в частности, если их знания дополняют друг друга или если круг задач не требует полной загрузки программиста; таких технических помощников можно выбирать по очереди среди "программистов". Важным моментом здесь является то, что эта роль должна быть признана официально.

- документалисту поручается составление внутренней и внешней документации проекта. Необходимость документалиста оправдана тремя следующими аксиомами: внутренняя и внешняя документация необходима; ее составление является трудной работой, уровень сложности которой сравним с самим программированием; если она не будет завершена одновременно с построением программы, она никогда не будет завершена. Заметим, что функция документалиста одна из самых трудных: она требует одновременно солидной технической компетенции, "литературных" способностей и склонности к синтезу.

- секретарь бригады программистов освобождает программистов от всех забот, связанных с отладкой и тестированием программ, позволяя тем самым программистам посвятить полностью свое время сложным аспектам программирования.

8. ОСНОВНЫЕ ЭТАПЫ И СТАДИИ ПРОЕКТИРОВАНИЯ.

Стадии разработки определяют наиболее общий состав процедур разработки и требования к документации. Стадии разработки регламентируются ГОСТом и другими нормативными документами.

Выделим четыре стадии разработки: **техническое задание, эскизный проект, технический проект, рабочий проект**. Проектирование ПС на ранних стадиях характеризуется высокой неопределенностью исходных данных и представлений разработчиков о свойствах и функциях создаваемой системы.

Уровни абстрагирования определяют систему понятий (модель абстракции), привлекаемых для описания инженерных решений. Уровни представления определяются в рамках конкретной предметной области, методики моделирования, могут регламентироваться различными стандартами.

Уровень абстрагирования будем сопоставлять с видом моделей абстракций, а переход по уровням абстрагирования – с переходом на новый вид модели абстракций.

Уровни детализации определяют степень детализации элементов и связей компонент моделируемой системы при описании ее на одном уровне абстрагирования. Уровни детализации могут регламентироваться конкретными методиками моделирования.

Под уровнем детализации будем понимать соответствующий иерархический уровень в модели абстракций (модель абстракций – иерархическая структура).

Уровни определенности характеризуют форму описания моделей. Наиболее существенными уровнями определенности являются концептуальный уровень, логический уровень и физический уровень.

- Концептуальный уровень – содержательное описание модели исходя из содержательного процесса управления. Характеризуется неформальными (слабо формализованными) средствами описания инженерных решений.
- Логический уровень – представление моделей системы с использованием типового математического аппарата, на основе которого можно проводить анализ и синтез структур и адекватно отображать с заданной степенью соответствия реальные процессы. Характеризуется формально обоснованными инженерными решениями.
- Физический уровень – описание модели системы на уровне программно-аппаратных средств реализации. Характеризуется практической выполнимостью моделей абстракций на программном уровне.

Таблица 1. План-график. Рекомендуемые этапы и виды работ.
ПТС – программно-технические средства.
ПО – программное обеспечение.

№ пп	Название этапов и видов работ	Длительность выполнения (кал. месяц)	Стоимость (уе)	Материалы и изделия, предоставляемые Заказчику
1.	**Предпроектная подготовка**	**2**	**13182**	
1.1	Разработка технического задания. 1. Сбор материалов для формирования исходных данных для планирования и проектирования ПО. 2. Технико-экономическое обоснование. 3. Обоснование проведения научных исследований. 4. Определение требований к ПО, стадиям, этапам и срокам разработки. 5. Оформление технического задания	1		Документы. • Техническое задание • План-график • Калькуляция
1.2	Системный анализ предметной области и класса проектируемого ПО. 1. Определение целей и назначения ПО; 2. Проектирование и моделирование основных функций и обобщенных алгоритмов. 3. Выбор методов решения задач. 4. Выбор и обоснование критериев эффективности и качества разработки ПО.	1		Документы. • Исходные данные для проектирования. • Сценарии. • Иерархия функций. • Топология АРИС при применении ПО. • Конфигурация ПО и ТС. • Обобщенные алгоритмы. • Список показателей и критериев эффективности и качества
2.	**Эскизный проект**	**2**	**20370**	
2.1	Проектирование архитектуры ПО. 1. Формирование общей структуры ПО и его основных компонент: определение структуры ПО, определение структуры модулей ПО. 2. Распределение ресурсов ТС по функциональным задачам ПО: 3. Оценка производительности ПТС: распределение емкостей накопителей информации и памяти ЭВМ, пропускной способности коммутаторов и каналов связи и пр. 4. Формирование дисциплины взаимодействия процессоров и диспетчеризации вызова программ.	1		Документация. • Покомпонентная спецификация ПО. • Оценка и распределение ресурсов ТС по компонентам ПО. • Инструкции по составлению спецификаций на модули и группы программ. • Методика отладки и комплексирования программ. • Спецификация взаимодействия параллельных задач и диспетчеризация модулей.
2.2	Подготовка технологических средств. 1. Организация базы данных проекта ПО; 2. Выбор и адаптация инструментария и языков программирования, настройка средств трансляции и отладки; 3. Оценка реализуемости данного класса ПО на базе выбранных ТС. 4. Выбор или разработка инструкций по применению технологии проектирования-конструирования ПО.	1		Документы. • Обоснование выбора инструментальных средств программирования. • Структура технологического процесса разработки ПО. • Формальная структура ПТС АРИС «Карты»: информационная, функциональная, потоковая. • Методика конструирования ПО
3.	**Технический проект**	**5**	**81030**	
3.1	Разработка ПО 1. Разработка алгоритмов, спецификаций на модули и группы программ. 2. Конструирование информационного фонда (базы данных) 3. Программирование и трансляция ПО.	3		Документы. • Внешние спецификации модулей • Логика модулей (определение данных, алгоритм, программа) • Структура сопряжений модулей • Глобальная модель данных + локальные модели данных. Изделия • Исходный модуль версии ПО, загрузочные (исполнимые) модули на отдельных магнитных или оптических носителях.
3.2	Отладка программ в статике. 1. Планирование отладки программ. 2. Тестирование программ. 3. Локализация ошибок и корректировка программ. 4. Комплексирование программ.	1		Документы • Методика детерминированного тестирования: тест, исходные данные и эталонные результаты.

3.3	<u>Комплексная динамическая отладка:</u> 1. Выбор средств для имитации абонентов. 2. Разработка программ имитации. 3. Создание программ обработки результатов. 4. Отладка функционирования ПО в реальном масштабе времени. 5. Отладка программы на объектах Заказчика.	1		Документы. • Структура и спецификация модели объекта автоматизации. Изделия • Программные и аппаратурные имитаторы внешней среды. • Средства для контроля и регистрации промежуточных данных, облегчающие обнаружение и локализацию ошибок • Исходный модуль версии ПО, загрузочные (исполнимые) модули на отдельных магнитных или оптических носителях.
4.	**Рабочий проект**	**9**	**169328**	
4.1	<u>Испытания ПО.</u> 1. Разработка, согласование и утверждение программы и методики испытаний ПО: испытания на полноту функционирования; испытания на надежность функционирования и другие характеристики. 2. Обработка результатов испытаний. 3. Разработка акта испытания. 4. Проведение корректировки ПО и программной документации по результатам испытаний.	3		Документы. • Программа испытаний. • Акт испытаний.
4.2	<u>Выпуск машинных носителей и документирование.</u> 1. Разработка инсталляционной версии пакета ПО с приданием ему статуса Программного Продукта (ПП). 2. Изготовление машинных носителей и выпуск тиража.	1		Изделия. • Макет обложки • Инсталляционная версия ПО, загрузочные (исполнимые) модули на отдельных магнитных или оптических носителях. • Тираж ПО в составе: инсталляционная версия (CD), пакет документов (брошюры)
4.3	Разработка и изготовление программной документации. 1. Эксплуатационных документов. 2. Технологических документов. 3. Исследовательских документов.	5		Документы • Руководство пользователя • Руководство программиста • Проектно-конструкторская документация. • **Акт о закрытии договора разработки**
5.	**Внедрение и сопровождение**			Проработка новых юридических документов на внедрение и сопровождение ПО

9. СТРАТЕГИИ ПРОЕКТИРОВАНИЯ.

Под стратегией проектирования часто понимается последовательность этапов, на каждом из которых применяется тот или иной метод проектирования [9]. Применяя термин «стратегия проектирования» [5], мы подразумеваем начальные условия, общую направленность и основные принципы разработки системной архитектуры. Термин методология проектирования мы используем для обозначения понятия, включающего, кроме стратегии проектирования, еще и систематическую процедуру, с помощью которой разработчик описывает, совершенствует и регистрирует свои проектные решения.

С точки зрения направления, в котором ведется разработка проекта системы, различают стратегии "сверху - вниз", "снизу-вверх" , "изнутри - к границам", "от границ - внутрь".

Описание этих стратегий можно найти в литературе относящейся к методам проектирования программных средств [1,5,9,12,25].

В аспекте идеологических основ процесса системного проектирования выделяются:

- стратегия функциональной декомпозиции;
- стратегия, ориентированная на информационные потоки;
- стратегия, ориентированная на структуры данных.

Стратегия функционального синтеза и декомпозиции.

Функциональный аспект стратегии функционального синтеза и декомпозиции отображает основные принципы функционирования, характера физических и информационных процессов, протекающих в объекте, и отражается в принципиальных, функциональных и структурных схемах и сопровождающих их документах.

Функциональная декомпозиция рассматривает систему частных критериев в виде совокупности отдельных функциональных задач. Выявление перечня функциональных задач позволяет сформировать модели функционирования системы и подсистем. На основе декомпозиции удается корректно поставить задачу проектирования как задачу принятия решений с соответствующими правилами выбора конкретного наилучшего проектного варианта. Варианты проекта различных уровней сравниваются по системе частных критериев соответствующего уровня на основе сформулированных моделей функционирования.

Формально декомпозицию можно подразделять на последовательную и параллельную.

При последовательной декомпозиции объект рассмотрения остается одним и тем же, меняется только степень детализации. Процесс поиска оптимальных проектных решений ведется последовательно: сначала на одном уровне детализации, затем на другом. Такой процесс можно организовать и "сверху - вниз" и "снизу-вверх".

При параллельной декомпозиции объект рассмотрения на каждом этапе изменяется, т.е. система разбивается на составляющие подсистемы. Процесс поиска оптимальных проектных решений ведется независимо и параллельно в каждой выделенной подсистеме, затем решения отображаются на уровень системы и там объединяются. Этот процесс также можно организовать как "снизу - вверх", так и "сверху - вниз".

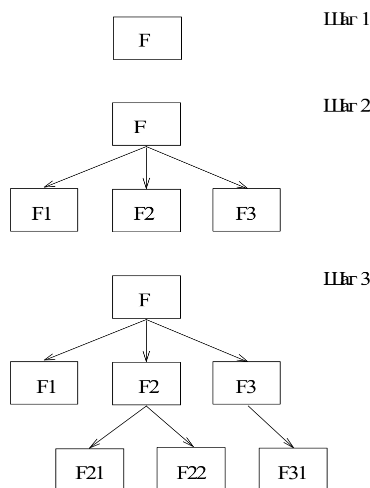


Рис. 2.1. Функциональная декомпозиция по принципу "сверху-вниз".

В рамках схемы функциональной декомпозиции по принципу "сверху - вниз" (см. рис. 2.1.) система первоначально представляется в виде некоторого аморфного объекта, который должен выполнять все функции, указанные в его функциональном описании. Этот набор функций рассматривается как одна сложная функция F; нисходящая функциональная декомпозиция продолжается далее путем разложения F на подфункции, которые в свою очередь, тоже разбиваются на подфункции и т.д.

Одна трудность, связанная с такой декомпозицией, заключается в том, что этот принцип не ведет естественным образом к получению стандартного набора функций на самом нижнем уровне иерархии. Между тем нередко обычные функции самых нижних уровней иерархии могут быть определены заранее.

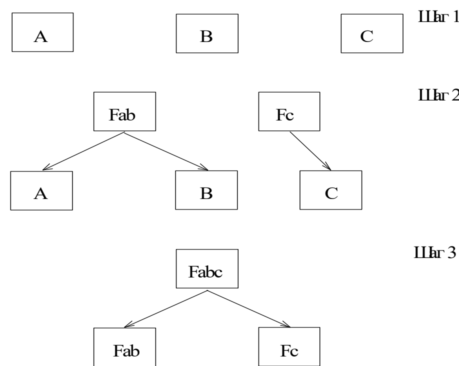


Рис. 2.2. Функциональный синтез по принципу "снизу-вверх".

Такой ситуации соответствует стратегия функционального синтеза по принципу "снизу-вверх" (см. рис. 2.2.). При этой стратегии исходные функции группируются в агрегированные функции все более высокого уровня, пока не будет достигнута вершина системы, которую можно рассматривать как одну сложную функцию.

Функциональная декомпозиция по принципу "изнутри - к границам" (см. рис. 2.3.) представляет собой комбинацию двух рассмотренных ранее стратегий, являясь восходящей стратегией для синтеза "изнутри - вверх" и нисходящей стратегией для декомпозиции "изнутри - вниз".

С нисходящей функциональной декомпозицией, восходящим функциональным синтезом и их вариантами связан целый ряд серьезных проблем. Первая из них заключается в том, что обе стратегии предполагают наличие у системы какой-либо определенной вершины, где управление через некоторый интерфейс передается от системы внешнему автономному объекту. Для системы со структурой вложений свойственно быстрое увеличение числа внешних автономных объектов, таких, как человек-оператор, сеть связи, измерительные устройства и т.п., которые более логично считать связанными с "боковыми гранями", а не с ее вершиной или основанием.

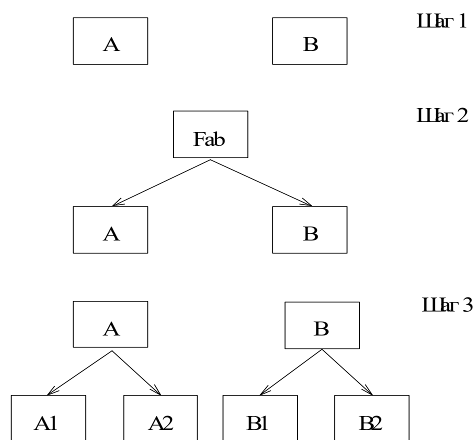


Рис. 2.3. Функциональный синтез и декомпозиция по принципу "изнутри к границам".

Источником другой проблемы является то, что в рамках данных стратегий не принимается во внимание никакая информация: вопросы прохождения данных, совместного их использования и построения вложенных структур оставляются для выяснения после завершения процедур функциональной декомпозиции или синтеза. Потоки информации и структура данных становятся, таким образом, категориями, подчиненными по отношению к задачам управления.

Третья проблема, характерная для стратегий функционального синтеза и декомпозиции, заключается в том, что они сильно затрудняют установление четких критериев оценки получающихся в результате структур. Поэтому разные разработчики будут завершать эту фазу проектирования с совершенно различными результатами функциональной декомпозиции.

Функциональное проектирование представляет наиболее общий подход к описанию систем. Определяются граничные условия и желательные входы и выходы, составляется подроб-

ный перечень функций и операций, которые должны выполняться. Метод в упрощенном виде сводится к составлению блок-схем системы.

Процедура анализа систем состоит в выделении всех возможных следствий из альтернативных систем для выбора из них наилучшей. За такую систему принимается та, которая в большей степени отвечает поставленным целям. При анализе некоторые сведения получаются объективно, т.е. путем сбора опытных данных и распределения частот, другие - субъективно, путем интуитивного восприятия относительных частот.

Стратегия проектирования, ориентированная на структуру данных.

Если при функциональной декомпозиции основное внимание сосредотачивается на структуре управляющей логики программы безотносительно к структуре ее данных, в альтернативном подходе внимание концентрируется прежде всего на структуре данных, в соответствии с которой строится структура управляющей логики. Такой подход может быть назван ориентированным на структуру данных или на модель данных.

Эта стратегия, в том или ином виде, находит отражение наиболее часто в литературе, связанной с разработкой баз данных [6,17].

В структурном плане модель данных характеризуется как совокупность взаимосвязанных понятий и правил порождения структур данных, ограничений целостности (логико-семантических свойств данных), а также операций над данными. Теория моделей данных получила развитие именно в связи с использованием концепций баз данных и проблемами специфических компьютеризованных систем ведения данных - автоматизированных банков данных.

Концепция баз данных связана с проблемами обеспечения логико-семантической и физической целостности, а также эффективности многоцелевого использования данных в условиях непрерывной и естественной эволюции приложений - процессов, оперирующих данными. Конкретная реализация этой концепции приводит к архитектуре многоуровневого представления данных, которая и является отличительным признаком базы данных.

Выделяют три уровня представления данных: концептуальный, внешний и внутренний.

Концептуальный уровень представления данных в форме концептуальной модели данных отражает объективные, не зависящие от конкретного приложения свойства данных, описывающие предметную область.

Внешний уровень представления данных отражает субъективные взгляды приложений на данные. В большинстве практических случаев внешнее представление ограничивается выделением некоторого подмножества концептуального представления.

Внутренний уровень представления данных определяет машинно-ориентированное, физическое представление данных.

Стратегия структурного проектирования, ориентированного на потоки данных.

Стратегия проектирования, о которой будет идти речь в дальнейшем, может быть названа структурным проектированием, ориентированным на потоки данных. В рамках этой стратегии сначала определяют ключевые компоненты потоков информации, циркулирующих в системе, а затем, идентифицируя функции преобразования в узловых точках информационного потока, производят своего рода функциональную декомпозицию; результатом таких действий является граф информационных потоков. Следующий шаг состоит в построении на его основе структурного графа, который описывает структуру управляющей логики системы для реализации нужного информационного потока. Та же стратегия применяется далее к подсистемам, подсистемам подсистем и т.д., и при этом умышленно игнорируются детали структуры данных. На рисунке 2.4 изображены основные элементы представления схем ориентированных на потоки данных, а на рисунке 2.5 представлены основные виды структурных преобразований схем, ориентированных на потоки данных.

Методология проектирования (неформальная):

Шаг 1. Определить в системе наиболее очевидные граничные модули-подсистемы. Центральную часть системы рассмотреть как единый центральный модуль. Приписать модулям функции неформально, только в самом общем виде.

Шаг 2. Определить информационные потоки между модулями и оценить необходимость введения дополнительных внутренних компонентов информационных потоков. Определить внутренние модули, которые целесообразно иметь в системе по соображениям обеспечения требуемых функций и модульной конструкции.

а) Данные б) Функциональные элементы в) Направление потока данных

Рис. 2.4. Базовые элементы представления схем, ориентированных на потоки данных

Рис. 2.5. Структурные преобразования схем, ориентированных на потоки данных.

Шаг 3. Уточнить представление о модулях и данных. С той степенью подробности, какая возможна на данной стадии, приписать модулям конкретные функции, используя перечень технических требований к системе. Снова уточнить информационный потоковый граф с учетом произведенного распределения функций. Провести критический анализ проектных решений по системе на данной стадии и при необходимости внести изменения.

Шаг 4. Разработать один или несколько структурных графов, определяя возможные варианты архитектуры системы в виде пакетов и задач.

Шаг 5. Описать в деталях интерфейсы системы, в том числе подробности, касающиеся типов и структур данных, и опять провести критический анализ.

Шаг 6. Если приходится иметь дело с большой и сложной системой, то может оказаться необходимым рекурсивное применение описанного метода для разработки проектов по подсистемам.

10. СПЕЦИФИКАЦИИ.

Спецификация - задание для программиста, написанное постановщиком задачи (системным аналитиком). Спецификация - это точное, однозначное, недвусмысленное описание. Написать спецификацию - значит, в первую очередь, подобрать точные понятия, адекватные задаче.

Спецификация обращена прежде всего к человеку, ориентирована на человека и, в принципе, может быть предназначена только человеку.

Понятие спецификации относительно и многоаспектно [7,15]. В принципе, спецификация должна быть наиболее прямым и естественным описанием задачи. Оно должно быть понятно, с одной стороны, заказчику или потребителю программы, а с другой - программисту. Таким образом, есть задача и есть разные категории людей, заинтересованных в ее решении с помощью ЭВМ, с разным запасом знаний, разными психологическими, интеллектуальными, профессиональными и производственными установками, опытом в программировании и т.д. Все это - аспекты, которые нужно учитывать, разрабатывая и используя спецификации.

Спецификация - это точное, однозначное, недвусмысленное описание. Написать спецификацию - значит, прежде всего, подобрать точные понятия, адекватные задаче. Это еще один аспект, в котором проявляется тесная связь спецификаций программ с математикой.

Разрабатывая, выбирая или оценивая тот или иной способ организации понятийных средств, особенно требуется выразительность, богатство и гибкость, но, в отличие от программ, не обязательна их непосредственная эффективная машинная реализация. Программа должна быть формализована до мельчайших деталей, часто несущественных и произвольных с человеческой точки зрения, чтобы она стала приемлемой для машины. Тогда как степень и способ формализации спецификации должны максимально облегчать ее написание и понимание человеком.

Основными чертами или свойствами спецификаций, как особого рода описаний, являются: полнота, точность и понятность.

Полнота спецификаций означает, что ничто существенное из задачи в ней не упущено, не забыто. Понятие полноты спецификации, конечно, неформальное и неформализуемое, но интуитивно ясное и содержательно очень важное. Полнота достигается после возможно, неоднократных обсуждений вариантов спецификаций, составленных системным аналитиком, с заказчиком и другими заинтересованными лицами; внесения изменений и дополнений, демонстраций выполняемой модели, составленной по спецификациям.

Точность спецификации называют также формальностью или однозначностью, причем требования к формализации фактически варьируют в широком диапазоне: от полностью формализованного описания до слегка формализованного. Описания на естественном языке обычно считаются неудовлетворительными, но какие формальные средства или средства достижения точности следует использовать в спецификациях - вопрос дискуссионный. Суть дела не в том, чтобы использовать в описаниях формулы и символические обозначения. Главное - построить, определить понятия, используемые в спецификациях как математические объекты. Таким образом, точность отождествляют с "математичностью". Степень формализации описаний в математике варьируется в зависимости от цели, которая ставится перед описанием и от его потенциальных читателей.

Обсудим теперь третье основное свойство спецификации - **понятийность**, называемое также **ясностью** или читабельностью. В принципе спецификация должна быть более понятным описанием задачи, чем программа, ее должно быть легче написать и легче прочесть. Однако в этом пункте особенно проявляется относительность понятия спецификации. Спецификация должна быть одновременно и понятной, и точной, а точность, как уже говорилось, требует привлечения тех или иных математических средств.

Отмечают возможность машинного выполнения спецификаций, хотя бы и очень неэффективного. Спецификации с таким свойством называют выполнимыми спецификациями. Их можно использовать в роли действующих макетов, или прототипов программ. Действующий макет позволяет посмотреть на конкретных примерах, какие результаты будет давать макетируемая программа, и экспериментировать с макетом, не вникая глубоко в его смысл.

Процесс составления спецификаций концентрируется вокруг взаимодействия системы и пользователя; остальные процессы проектирования касаются внутренней структуры программного обеспечения. По этой причине, а также вследствие трудностей в подготовке высококачественных спецификаций проверка их правильности приобретает исключительное значение.

На этом этапе должна быть занята активная позиция в отношении ошибок. Цель всякого процесса проверки правильности или тестирования - найти как можно больше ошибок, а не показать, что спецификации не содержат ошибок. Важно также проверить правильность спецификаций, пока еще они имеют вид набросков, поскольку часто, как только документ приобретает "законченную", печатную форму, возникает психологический барьер, препятствующий внесению изменений.

Контроль по правилу "n плюс-минус один". Специалисты уровня n-1 ищут ошибки перевода, а специалисты уровня n+1 - неясности, которые могут привести к ошибкам в последующих процессах. Предварительные спецификации оцениваются теми, кто отвечает за подготовку целей, разработку архитектуры и детальное проектирование. Спецификации следует сопоставить с целями, рассматривая каждую цель и анализируя, насколько адекватно она отражена в спецификациях.

Контроль со стороны пользователя. Крайне важно добиться того, чтобы пользователь просмотрел и одобрил предварительные и детальные спецификации. Если по какой-то причине это не может быть сделано (например, в независимом от пользователя проекте), тогда должна быть сформирована специальная группа в самой организации-разработчике, которая призвана исполнять роль "адвоката дьявола"; в ее задачу будет входить оценка спецификаций с точки зрения пользователя.

Таблицы решений. Если в спецификациях используются таблицы решений, тогда для их проверки на полноту (поиск всех пропущенных условий) и неоднозначность (идентичные условия приводят к разным выходным данным или преобразованиям) могут быть применены **"механические" методы**.

Ручная имитация. Эффективный прием оценки детальных спецификаций - подготовить тесты и затем воспользоваться спецификациями для имитации поведения системы. Этот процесс оценки проектного документа методом "выполнения" его на тестовых данных часто называется сквозным контролем (или прослеживанием).

Имитация за терминалом. Более изощренный метод сквозного контроля - игра в "человека, спрятанного в пустой машине". Вместо того чтобы просто читать список тестов, как при ручной имитации, человек садится за терминал и вводит тестовые данные и изучает результаты точно так, как если бы терминал был подключен к настоящей программной системе. Для имитации системы еще один участник проверки, вооружившись спецификациями, садится за другой терминал. Небольшая программа связывает эти терминалы, передавая входные данные с терминала "пользователя" на терминал "имитатора", а выходные данные - обратно по командам "имитатора". Последний действует так же, как при ручной имитации, т.е. изучает данные, поступающие с терминала "пользователя", ищет по спецификациям соответствующий результат, прослеживает преобразование системы и посылает результат на другой терминал "пользователя". Этот метод очень ценен, поскольку он помогает обнаружить в спецификациях все недочеты по части психологических факторов.

Функциональные диаграммы. Метод предполагает анализ семантического содержания спецификаций и перевод их на язык логических отношений между входными данными (ситуациями) и выходными данными и преобразованиями (эффектами), представленных в форме логической диаграммы ("и-или" - графа), называемой функциональной диаграммой. Диаграмма снабжается примечаниями в виде синтаксических правил и ограничений внешней среды и затем преоб-

разуется в таблицу решений с ограниченным входом. Каждый столбец таблицы соответствует будущему тесту.

Спецификации в виде графических схем.

11. ГРАФ-ДИАГРАММЫ

Для демонстрации связей, существующих между отдельными компонентами системы, используются различные графические схемы. Некоторые из них, такие как граф-диаграммы, отображают главным образом прохождение потоков данных между процессами. Другие, в частности, функциональные схемы, выделяют моменты, связанные с хранением данных и используемыми для этого носителями. Существуют также схемы, в которых основное внимание уделяется взаимодействию процессов.

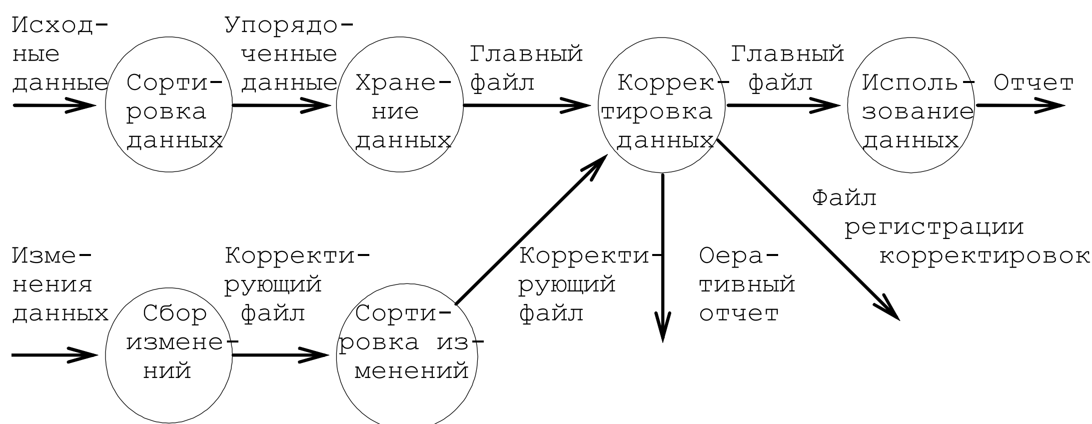


Рис. 11.1. Граф-диаграмма системы сопровождения данных.

Граф-диаграммы. Иногда называемые графами потоков данных. Каждый кружок на такой диаграмме отображает некоторое преобразование данных. Потоки данных отмечаются стрелками. Этот тип схем можно использовать как на системном уровне для описания внешних входов и выходов программ, так и при проектировании самих программ для описания перемещений данных между отдельными модулями. На рисунке 3.1 представлен пример граф-диаграммы системы сопровождения данных.

12. ДИАГРАММЫ ВАРНЬЕ-ОРРА

Диаграммы Варнье-Орра. На диаграмме Варнье-Орра в иерархической структуре системы выделяются ее элементарные составные части, которые снабжаются контурными изображениями носителей информации. Сначала система разделяется на ряд отдельных процессов. На следующем уровне иерархии указываются потоки данных для каждого процесса. Затем перечисляются наборы данных и, наконец, - соответствующие носители информации. Последние обозначаются с помощью стандартных условных изображений, применяемых на функциональных схемах. Направления потоков данных отмечаются стрелками, проведенными между наборами данных и физическими носителями информации. Наборы данных, используемые одновременно в нескольких процессах, связаны между собой и имеют одинаковые имена. На рисунке 12.1 представлена диаграмма Варнье-Орра для системы сопровождения данных.

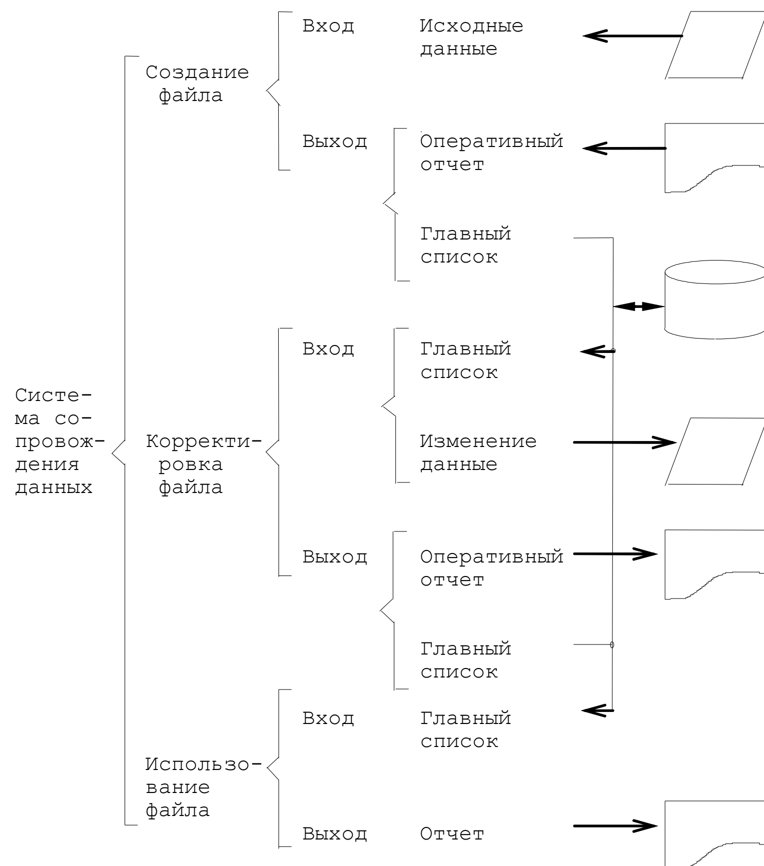


Рис. 12.1 - Диаграмма Варнье-Орра для системы сопровождения данных.

13. ФУНКЦИОНАЛЬНЫЕ СХЕМЫ.

Функциональные схемы. Функциональная схема системы состоит из одного или нескольких прямоугольных блоков, содержащих названия программ. Эти блоки соединяются входящими в них стрелками с источниками и исходящими из них стрелками - с приемниками данных. Источники и приемники изображаются в виде блоков, очертания которых напоминают определенные физические носители информации (некоторые блоки представлены на рис. 13.1). В каждом блоке записано имя программы или набора данных, иногда оно дополняется информацией, раскрывающей назначение блока. Основное внимание в схемах этого типа уделяется описанию потоков данных в системе и используемых наборов данных. На рисунке 13.2. изображен пример отображения фрагмента системы корректировки главного файла в виде функциональной схемы.

Все рассмотренные выше типы схем рассчитаны на описание потоков данных в программно-управляемых системах, в которых только программы могут инициализировать или прекращать генерацию потоков данных. Однако в ПС, для которых характерна работа в режиме реального времени, некоторые из системных функций управляются не столько программами, сколько самими данными, т.е. в таких системах данные приводят в действие или заставляют прекращаться определенные процессы. В одно и то же время в активном состоянии могут находиться несколько процессов.

Рис. 13.1. Блоки представления функциональных схем.

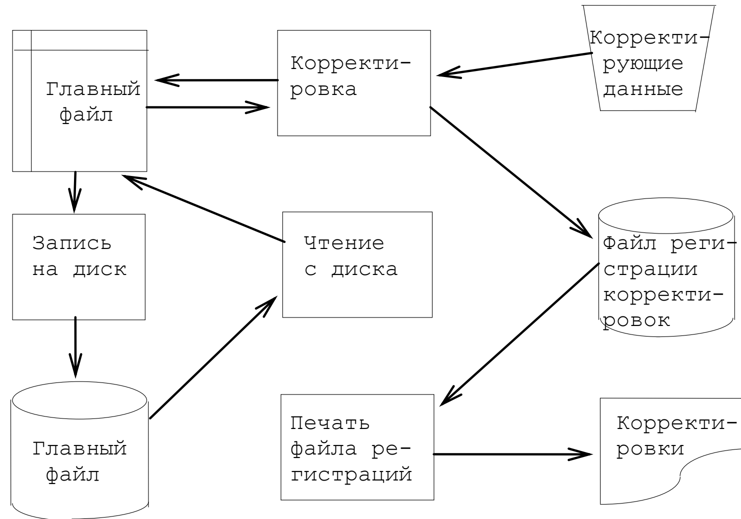


Рис. 13.2. Пример функциональной схемы «Корректировка главного файла»

14. ПЕРТ-ДИАГРАММЫ.

ПЕРТ-диаграммы. На функциональных схемах нельзя показать порядок взаимодействия программ. Для этого удобнее использовать ПЕРТ-диаграммы. На ПЕРТ-диаграмме не указываются наборы или потоки данных. Она отображает связи по управлению, существующие в системе, а также координацию выполняемых действий. Каждая стрелка соответствует определенной операции, а каждый кружок - событию, под которым понимается завершение одной или нескольких операций и переход к другим. По содержанию эти символы прямо противоположны аналогичным обозначениям на граф-диаграммах (см. рис. 14.1).

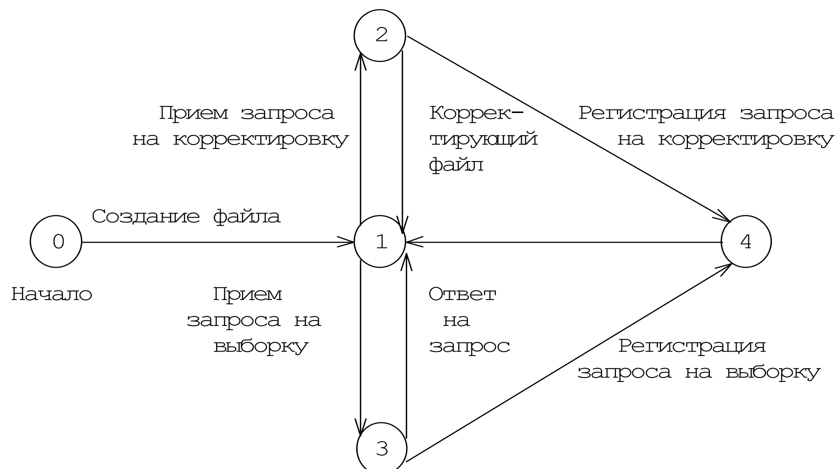


Рис. 14.1. ПЕРТ-диаграмма интерактивной системы сопровождения файлов.

15. СЕТИ ПЕТРИ.

Сети Петри. Диаграммы, называемые сетями Петри, используются в качестве моделей, которые описывают движение потоков данных в сетях, допускающих частичное или полное переключение потоков из одних магистралей в другие. Такая ситуация характерна для интерактивной системы корректировки - выборки данных, в которой данные могут проходить через программы, одновременная работа которых не допускается. Сети Петри позволяют исследовать как потоки данных, так и динамику передач управления в системе. Для этого строится несколько схем, отражающих последовательные состояния сети, из которых видно, как происходит перемещение точек управления вдоль потоков данных. Следующие друг за другом изображения сети Петри отличаются лишь расположением указанных точек (см. пример на рис. 15.1).

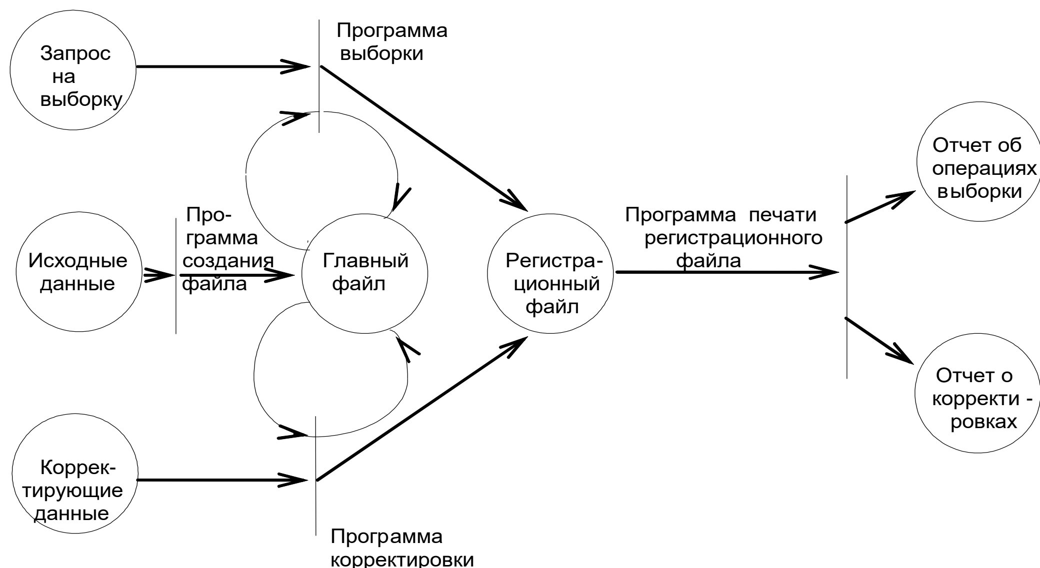


Рис. 15.1. Сеть Петри для интерактивной системы сопровождения файлов.

16. СХЕМЫ HIPO.

Схемы HIPO. Использование схем HIPO характерно для той стадии проектирования, когда системные аналитики уже могут приступать к разработке программ и данных. Эти схемы, определяя основные функции каждой программы и перечень основных элементов данных, не конкретизируют способы организации данных, иерархическую структуру подпрограмм и выбор алгоритмов обработки. На этапе разработки программ схемы HIPO могут применяться в качестве средства описания функций, реализуемых программой, и циркулирующих внутри нее потоков данных. На рисунке 16.1 представлена схема HIPO для программы корректировки файла.

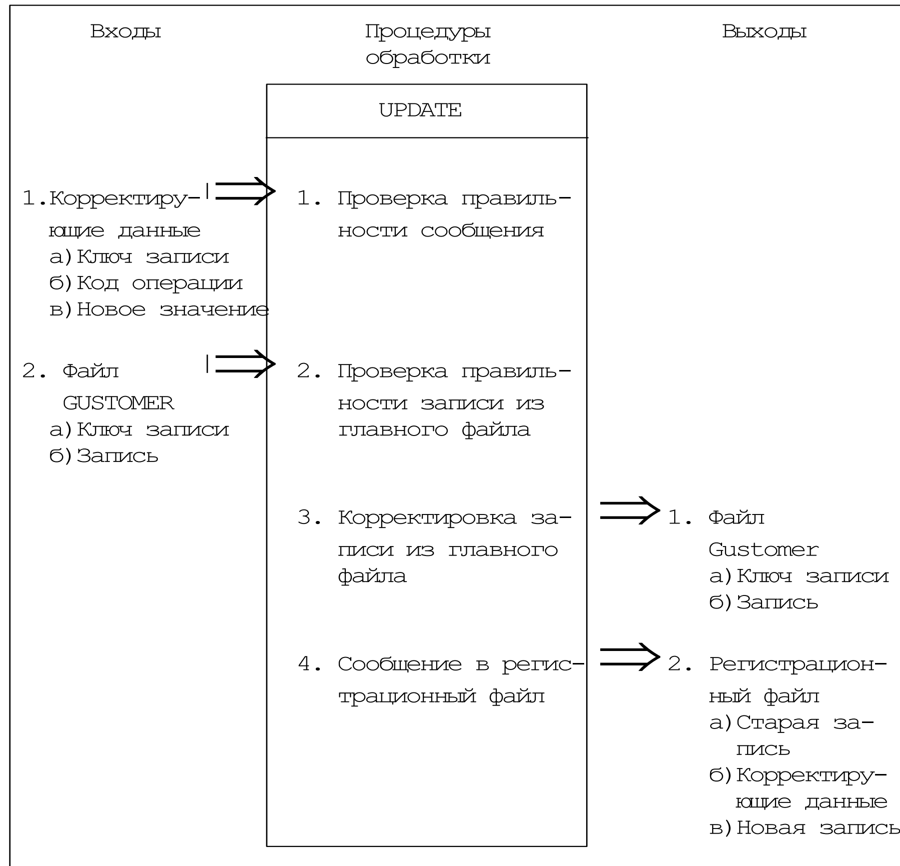


Рис. 16.1. Схема HIPO для программы корректировки файла GUSTOMER.

17. БЛОК-СХЕМЫ.

На рисунке 17.1 показаны стандартные и нестандартные символы для изображения структурных схем в виде блок-схем. Их можно использовать для представления организации программы так же, как и для передач управления. Прокомментируем представленные символы.

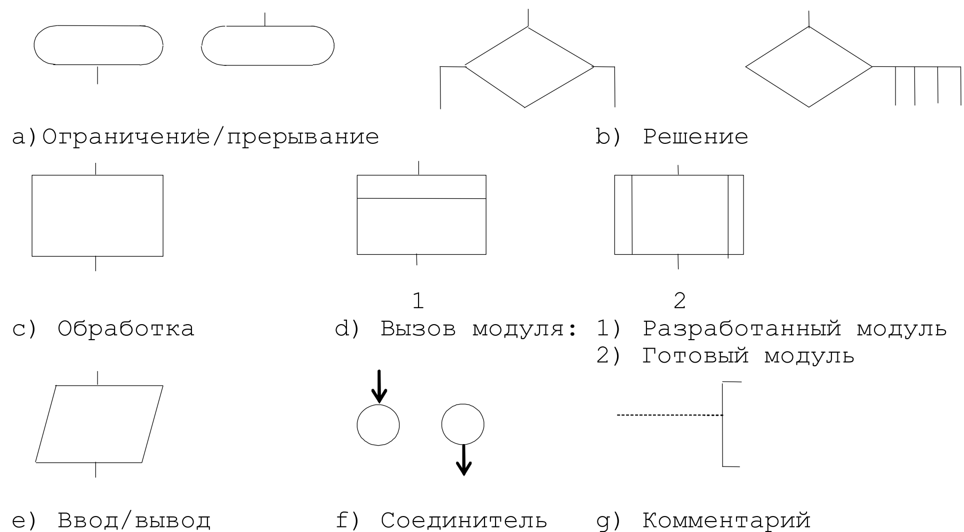


Рис. 17.1. Символы блок-схем.

Блок ограничения/прерывания. Этот символ предназначен для обозначения входов в структурную схему, а также для указания всех выходов из нее. Каждая структурная схема должна начинаться и заканчиваться символом ограничения.

Блок решения. Этот символ используется для обозначения переходов управления по условию. Для каждого блока решения должны быть указаны: вопрос, решение, условия или сравнение, которые он определяет. Стрелки, выходящие из этого блока, должны быть помечены соответствующими ответами так, чтобы были учтены все возможные ответы.

Блок обработки. Этот символ применяется для обозначения одного или нескольких операторов, изменяющих значение, форму представления или размещения информации. Для улучшения наглядности схемы несколько отдельных блоков обработки могут быть объединены в один блок.

Блок вызова модуля. Этот модуль используется для обращений к модулям или подпрограммам. Вертикальные линии обозначают обращение к внешним модулям обработки, горизонтальная линия - данный блок представлен в документации отдельной структурной схемой.

Блок ввода/вывода. Этот символ используется для обозначения операций ввода/вывода информации. Отдельным логическим устройствам или отдельным функциям обмена должны соответствовать отдельные блоки. В каждом блоке указывается тип устройства или файла, тип информации, участвующий в обмене, а также вид операции обмена.

Соединители. Эти символы используются в том случае, если структурная схема должна быть разделена на части или не умещается на одном листе. Применение соединителей не должно нарушать структурности при изображении схем.

Блок комментария. Этот символ позволяет включать в структурные схемы пояснения к функциональным блокам. Частое использование комментариев нежелательно: оно усложняет структурную схему.

Схемы передач управления. Для изображения передач управления в программном модуле обычно используются структурные схемы программ. Структурное программирование и его влияние на применение основных управляющих конструкций способствовали тому, что стандартные символы схем были дополнены новыми символами и были разработаны новые типы схем. В частности, схемы Насси-Шнейдермана представляют для программиста средство описания вложенных управляющих структур.

Структурные схемы можно применять на любом уровне абстракции. Основная тенденция в использовании структурных схем в настоящее время - не указание последовательности операций, а группирование символов, выражающих базовые конструкции: **следование, выбор, повторение**. На рис. 17.2 показаны схемы этих управляющих конструкций.

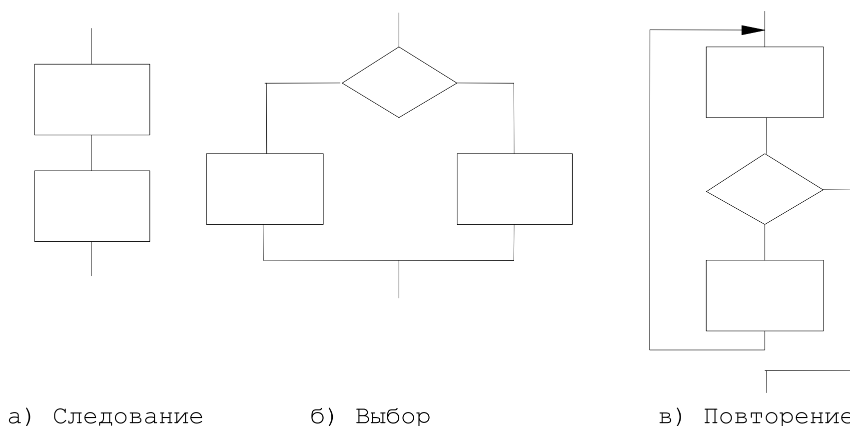


Рис. 17.2. Базовые управляющие конструкции.

18. СХЕМЫ НАССИ-ШНЕЙДЕРМАНА.

Схемы Насси-Шнейдермана. Способ изображения модуля с помощью схем Насси-Шнейдермана представляет собой попытку использования требований структурного программирования (см. ниже) в структурных схемах модулей. Он позволяет изображать схему передач управления не с помощью явного указания линий переходов по управлению, а с помощью представления вложенности структур. Некоторые из используемых в этом способе символов соответствуют символам структурных схем. Эти символы показаны на рисунке 18.1. Каждый блок имеет форму прямоугольника и может быть вписан в любой внутренний прямоугольник любого другого блока. Блоки помечаются тем же способом, что и блоки структурных схем, т.е. с использованием предложений на естественном языке или с помощью математических нотаций. Если использовать символы схем Насси-Шнейдермана одновременно с дополнительными символами структурных схем для изображения множественных выходов и обработки прерываний, то представление рассматриваемого модуля может быть упрощенно.

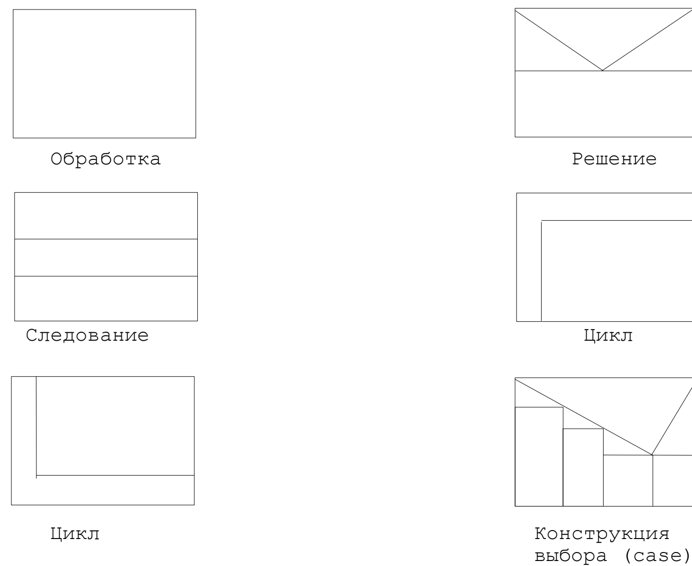


Рис. 18.1. Базовые конструкции схем Насси-Шнейдермана.

19. СИНТАКСИЧЕСКИЕ ДИАГРАММЫ.

Синтаксические диаграммы. Так как грамматические правила просты и их немного, то для описания грамматических правил используют синтаксические диаграммы. Идея синтаксической диаграммы состоит в том, что надо войти в нее слева и пройти по ней до правого края. Синтаксические диаграммы, как правило, используются для описания синтаксиса операторов языков программирования при их представлении. На рис. 3.11 представлена синтаксическая диаграмма для оператора - цикла с заданными границами PASCAL ("FOR").

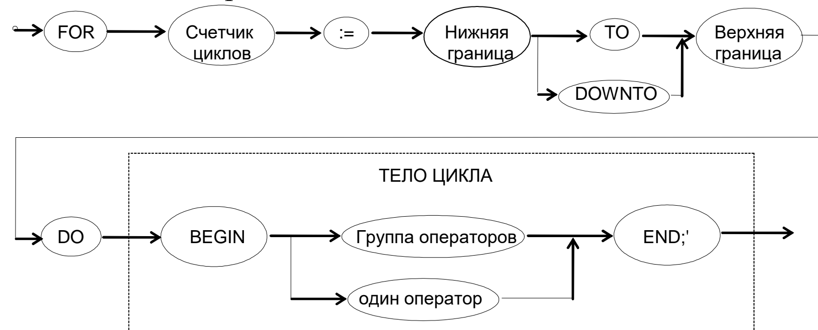


Рис. 3.11. Синтаксическая диаграмма оператора "FOR".

20. ТАБЛИЦЫ РЕШЕНИЙ

Таблицы решений. Метод проектирования с помощью таблиц решений заключается в перечислении вариантов управляющих решений, принимаемых на основе анализа данных. Поскольку в этих таблицах перечисляются все возможные сочетания данных, существует гарантия того, что учитываются все необходимые решения. Таблицы решений обычно состоят из двух частей. Верхняя часть используется для определения условий, а нижняя - для действий. Левая часть таблицы содержит описание условий и действий, а правая часть - соответствующую ситуацию. Рисунок 20.1 демонстрирует возможность использования таблицы решений для формализации задачи светофорного регулирования.

Условия	Правила				
1: КРАСНЫЙ	1	1	0	0	иначе
2: ЖЕЛТЫЙ	0	1	1	0	
3: ЗЕЛЕНЫЙ	0	0	1	1	
Действия					
1: СТОП	X		X		
2: ДВИГАЙСЯ ВПЕРЕД				X	
3: ПОДГОТОВКА К ДВИЖЕНИЮ		X			
4: ПРОПУСТИ С ПРАВА И ДВИГАЙСЯ					X

Рис. 20.1. Таблица решений для формализации задачи светового регулирования.

Вопросы, на которые следует ответить в структуре управления, перечислены в столбце условий. Действия, выполняемые в зависимости от ответов, указаны в столбце действий. Затем рассматриваются все возможные комбинации ответов "да" и "нет". Если какая-либо комбинация невозможна, она может быть опущена. Крестами отмечены действия, необходимые для каждого набора условий. Порядок расположения условий не должен влиять на порядок их проверки. Однако действия могут быть записаны в порядке их выполнения.

Таблицы решений могут быть использованы для проектирования структуры управления модулями в иерархической схеме. Их также можно преобразовать в двоичные деревья решений и принять как основу для проектирования любого модуля, использующего решения.

21. СТРУКТУРНЫЕ ПРЕОБРАЗОВАНИЯ. ПРОСТЫЕ ПРЕОБРАЗОВАНИЯ.

Простые преобразования. Простые преобразования схем передач управления связаны со сверсткой и разверткой: а - линейных участков; б - условий; в, г -циклов (рис. 21.1).

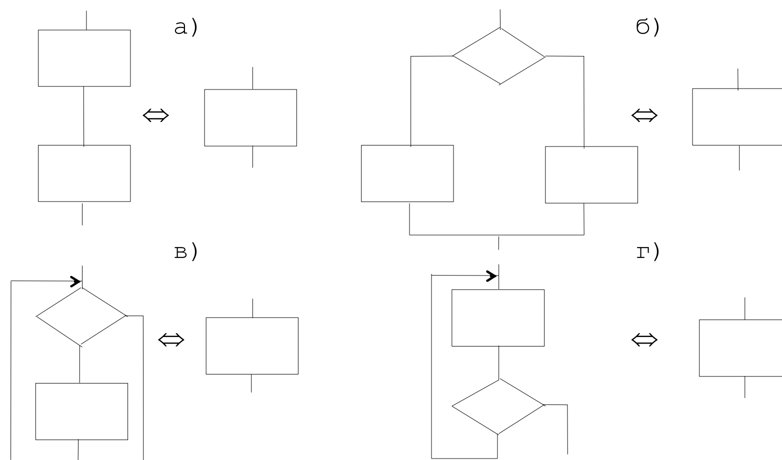


Рис. 21.1. Простые преобразования схем передач управления.

22. СТРУКТУРНЫЕ ПРЕОБРАЗОВАНИЯ. ДУБЛИРОВАНИЕ ЭЛЕМЕНТОВ.

Дублирование элементов. Данное преобразование позволяет привести схему передачи управления к структурированному виду введением в нее по определенным правилам дополнительных элементов, эквивалентных уже имеющимся.

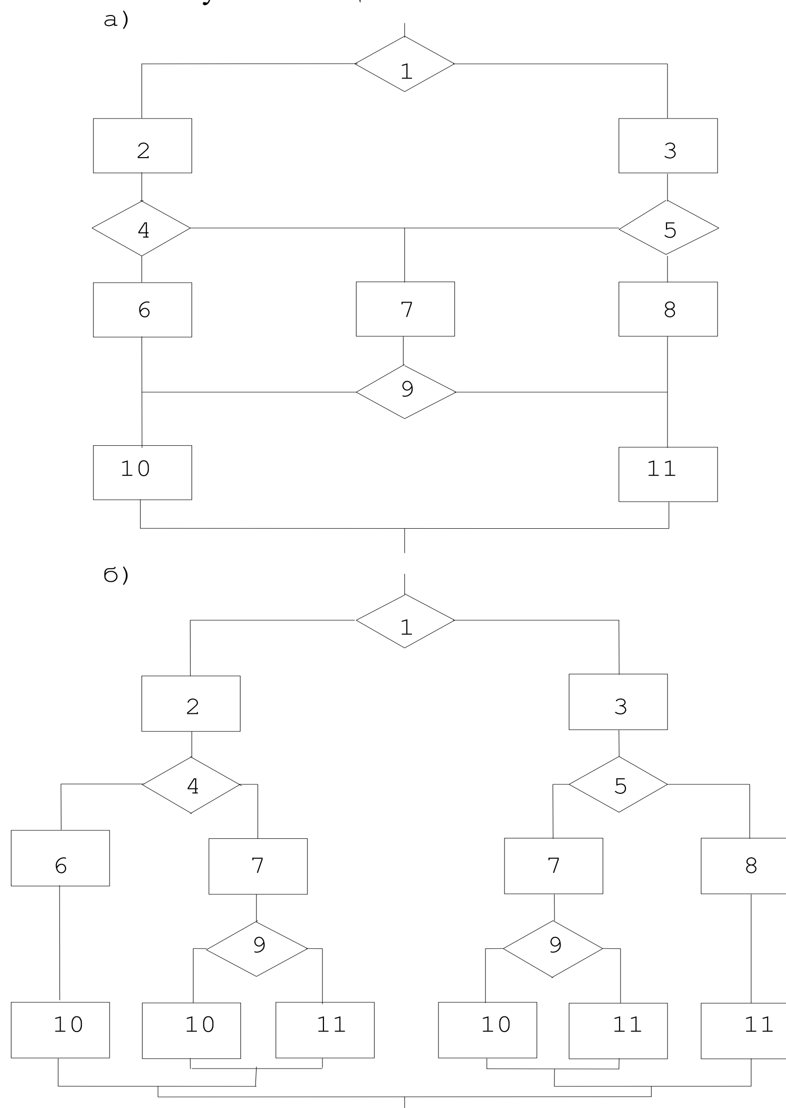


Рис. 22.1. Преобразование схемы путем дублирования элементов.

Пусть имеется структура, представленная на рисунке 22.1,а . В целом эта схема имеет один вход и один выход. Однако стремление использовать блоки 7,9,10 и 12 в ветвях, начинающихся в блоках 4 и 5, привело к запутанным связям по управлению. Дублируя соответствующим образом блоки 7,9,10,11 можно привести исходную схему к структурированному виду. При дублировании, строя очередной путь после разветвления, каждый раз вводят необходимые блоки, не обращая внимания на то, что они уже введены на альтернативных участках других путей. Каждый дублированный элемент, по существу, имеет собственное имя, однако в функциональном отношении эквивалентен исходному. На рис. 22.1,б изображена преобразованная исходная схема.

23. СТРУКТУРНЫЕ ПРЕОБРАЗОВАНИЯ. ВВЕДЕНИЕ ПЕРЕМЕННОЙ СОСТОЯНИЯ.

Введение переменной состояния. Второй подход преобразования управляющей структуры основывается на введении переменной состояния.

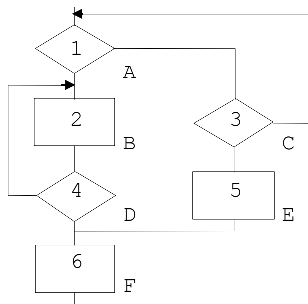


Рис. 23.1. Фрагмент схемы передачи управления, требующий приведения к структурированному виду.

Пример на рис. 23.2 иллюстрирует преобразование схемы 23.1 передачи управления с циклами и метками к структурированному виду на основе введения переменной состояния.

Процесс преобразования состоит из пяти шагов:

1. Каждому блоку схемы приписывается номер. Причем 0 - последний исполняемый элемент.

2. Вводится новая переменная, принимающая значение в диапазоне 0..n, где n - число блоков в схеме передачи управления.

3. Вводятся n операций присвоения значений введенной переменной состояния. С каждым блоком связывается одна (для логического блока по количеству выходов) операция, в которой значение переменной становится равным номеру очередного исполняемого блока.

4. Вводятся n операций анализа переменной состояния, причем, если значение переменной состояния равно m ($m<n$), то управление передается блоку с номером m.

5. Строится новая управляющая структура в виде цикла с вложенными в него операциями анализа переменной состояния и исполнения блоков исходной структуры с добавлением элементов присвоения значений переменной состояния.

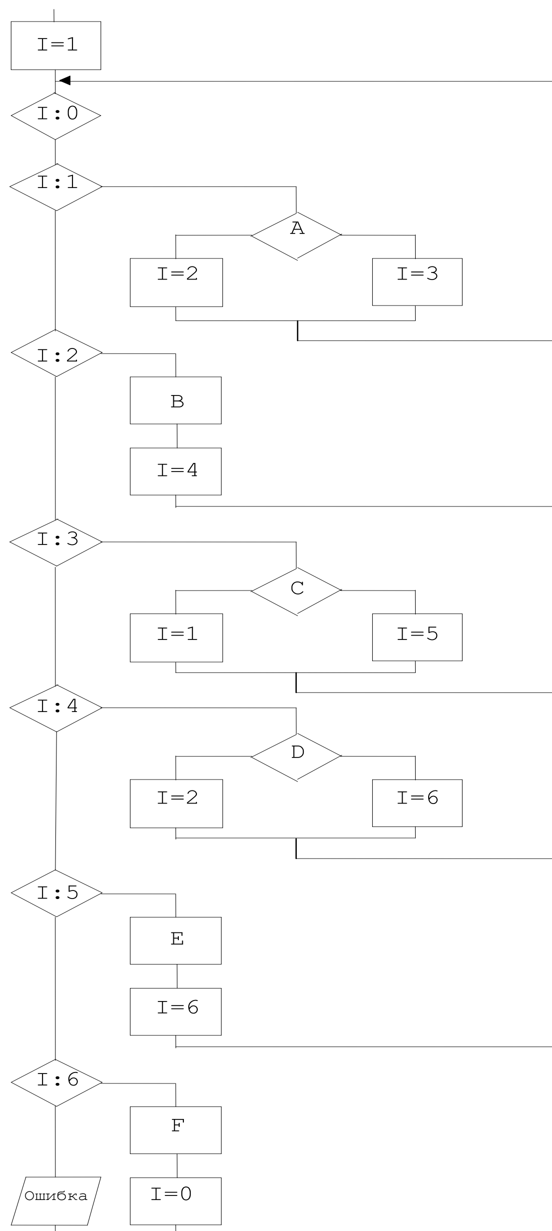


Рис. 23.2. Преобразование схемы передачи управления на основе введения переменной состояния.

24. АРХИТЕКТУРА ПРОГРАММНОЙ СИСТЕМЫ.

Архитектура системы — принципиальная организация системы, воплощенная в ее элементах, их взаимоотношениях друг с другом и со средой, а также принципы, направляющие ее проектирование и эволюцию.

Архитектура программного обеспечения (англ. *software architecture*) — это структура программы или вычислительной системы, которая включает программные компоненты, видимые снаружи свойства этих компонентов, а также отношения между ними. Этот термин также относится к документированию архитектуры программного обеспечения. Документирование архитектуры ПО упрощает процесс коммуникации между разработчиками и заинтересованными лицами, позволяет зафиксировать принятые на ранних этапах проектирования решения о высокоуровневом дизайне системы и позволяет использовать компоненты этого дизайна и шаблоны повторно в других проектах.

Разработка архитектуры - это процесс разбиения большой системы на более мелкие части. Для обозначения этих частей придумано множество названий: программы, компоненты, подсистемы и уровни абстракции.

Процесс разработки архитектуры - шаг, необходимый при создании программы. Если описывают программную систему, то следующий шаг проектирования - разработка архитектуры, а за ним - проектирование структуры программы. Если же спецификации описывают программу, разработка архитектуры не обязательна. Очевидно, немедленно возникает проблема, как различать системы и программы. К сожалению, точных определений этих понятий нет. Словарь здесь бесполезен, потому что все объекты, от операционной системы до подпрограммы и цикла, удовлетворяют определению системы.

Мы можем сказать, что программная система представляет собой набор решений множества различных, но связанных между собой задач, и далее положиться на интуицию в случаях, когда надо отличить систему от программы. Так, операционная система, система резервирования авиабилетов или система управления базами данных - все это примеры систем, и поэтому они должны пройти процесс разработки архитектуры. Программа редактирования текстов, компилятор, программы "картотека" являются примерами отдельных программ, и в них этап разработки архитектуры отсутствует.

Всякая методология проектирования программного обеспечения состоит из двух частей: набора желательных характеристик результата и руководящих принципов самого процесса проектирования. При разработке методологии проектирования обычно начинают с определения желательных характеристик, а затем конструируют мыслительный процесс, необходимый для получения нужного результата. Поэтому можно предположить, что о желательных свойствах результата известно больше, чем о самом процессе. Сам процесс проектирования - процедура, все еще во многом зависящая от конкретной ситуации. При проектировании логики модуля желательный результат - структурная программа, и один из мыслительных процессов называется "пошаговая детализация".

Если попытаться выделить упоминавшиеся две части методологии проектировании в разработке архитектуры, станет ясно, что это - пока наименее понятный процесс перевода. Здесь не только не определен соответствующий мыслительный процесс, но даже и желательные характеристики структуры системы только начинают осмысливаться.

Уровни абстракции. Концепция уровней абстракции была предложена Э. Дейкстра [5]. Система разбивается на различные иерархические упорядоченные части, называемые уровнями, удовлетворяющие определенным проектировочным критериям. Каждый уровень является группой тесно связанных модулей. Идея уровней призвана минимизировать сложность системы за счет такого их определения, которое обеспечивает высокую степень независимости уровней друг от друга. Это достигается благодаря тому, что свойства определенных объектов такой системы (ресурсы, представление данных и т.п.) упрятываются внутрь каждого уровня, что позволяет рассматривать каждый уровень как "абстракцию" этих объектов.

Хотя современное определение уровней абстракции само довольно абстрактно, некоторые свойства уровней начинают проясняться.

1. На каждом уровне абсолютно ничего не известно о свойствах (и даже существовании) более высоких уровней. Это - фундаментальное свойство уровней абстракции, существенно сокращающее число связей между частями программной системы и их предположений относительно свойств друг друга, благодаря чему уменьшается сложность системы.

2. На каждом уровне ничего не известно о внутреннем строении других уровней. Связь между уровнями осуществляется только через жесткие, заранее определенные сопряжения.

3. Каждый уровень представляет собой группу модулей (раздельно компилируемых программ). Некоторые из этих модулей являются внутренними для уровня, т.е. недоступны для других уровней. Имена остальных модулей, вообще говоря, известны на следующем, более высоком уровне. Эти имена представляют собой сопряжение с этим уровнем.

4. Каждый уровень располагает определенными ресурсами и либо скрывает их от других уровней, либо предоставляет другим уровням некоторые их абстракции. Например, в системе

управления файлами один из уровней может содержать физические файлы, скрывая их организацию от остальной части системы. Другие уровни могут владеть такими ресурсам, как каталоги, словари данных, механизмы защиты.

5. Каждый уровень может также обеспечивать некоторую абстракцию данных в системе. В системе управления данными один из уровней может представлять файл как древовидную структуру, изолируя тем самым действительную физическую структуру файла от всех остальных уровней.

6. Предположения, которые на каждом уровне делаются относительно других уровней, должны быть минимальны. Эти предположения могут принимать вид соотношений, которые должны соблюдаться перед выполнением функции, либо относятся к представлению данных или факторов внешней среды.

7. Связи между уровнями ограничены явными аргументами, передаваемыми с одного уровня на другой. Недопустимо совместное использование глобальных данных несколькими уровнями.

8. Каждый уровень должен иметь высокую прочность и слабое сцепление.

Хотя подход на основе уровней абстракции остается пока довольно туманным, эти идеи кажутся полезными при определении общей структуры больших систем.

Метод портов. Второй метод проектирования систем основан на идее подсистемы, управляемой портом, или очередью. В отличие от метода уровней абстракции, где внимание уделяется в первую очередь правильному распределению функций между уровнями иерархии в системе, здесь основное внимание обращено на методы связи между подсистемами, компонентами или программами.

Механизмы, обеспечивающие взаимосвязь между частями системы, можно разбить на три группы. Механизмы первой группы, такие, как операторы GO TO, прерывание, передают управление от одной части другой, не передавая явно никаких данных. Механизмы типа процедуры или подпрограммы относятся ко второй группе; они передают от одной части системы другой и управление, и данные. К третьей группе относятся механизмы портов - это методы передачи данных без передачи управления.

При использовании механизма портов система делится на несколько подсистем, каждая из которых имеет один или несколько входных портов (одну или несколько входных очередей). Порт - это просто текущий список входных сообщений (списков параметров) для подсистемы. Каждая подсистема рассматривается как асинхронный процесс, т.е. все подсистемы работают параллельно. Если одна из них хочет передать некоторые данные другой, она посылает их во входной порт этой другой подсистеме. Если подсистема готова обрабатывать какие-то данные, она читает их из одного из своих портов. Используемые для такой связи две операции называются ПОСЛАТЬ и ПОЛУЧИТЬ.

Механизм портов увеличивает независимость подсистем, в значительной степени освобождая их от временной зависимости друг от друга. Более того, подсистемы не должны даже знать о взаимном расположении. Только механизм посылки-получения (почта) должен знать расположение каждой подсистемы. Механизм портов имеет также некоторые преимущества при тестировании, отладке и проверке, поскольку благодаря ей в центре всех взаимодействий подсистем оказывается основной механизм связи.

Механизм посылки-получения сам удовлетворяет определению уровня абстракции, поскольку он обеспечивает абстракцию (понятие порта), обладает ресурсами (физические порты) и скрывает от остальной части системы информацию (физические характеристики портов и физическое расположение всех подсистем). Поэтому, естественно, эти две концепции можно использовать вместе, определяя каждую подсистему, управляемую методом портов, как уровень абстракции.

К сожалению, нынешнее состояние обоих подходов - метода уровня абстракций и метода портов - таково, что оба они описывают желаемые свойства системы, но не сам процесс ее проек-

тирования. Тем не менее оба подхода уже сегодня полезны тем, что дают разработчику некоторые руководящие принципы проектирования программных систем.

Поскольку представления о процессе разработки архитектуры пока еще довольно расплывчаты, это же во многом относится и к проверке его правильности. Можно указать лишь два достижения в этом направлении - метод "n плюс-минус один" (см. выше) и метод сквозного контроля.

Для проведения процесса сквозного контроля, сначала, нужно разбить тесты, затем мысленно проследить за их выполнением, стремясь найти не полностью определенные или несогласованные сопряжения, отсутствующие или недоопределенные функции.

25. МОДУЛЬНОСТЬ

Следующий процесс проектирования программного обеспечения - проектирование структуры программы. Он включает определение всех модулей программы, их иерархии и сопряжений между ними. Если разрабатывается отдельная программа, исходными данными для этого процесса будут детальные спецификации, если же система - детальные спецификации и архитектура системы. В этом последнем случае рассматриваемый процесс полной системы.

Традиционный метод борьбы со сложностью - принцип "разделяй и властвуй", часто называемый "модуляризацией". На практике, однако, этот подход часто не приводит к ожидаемому уменьшению сложности. Можно указать три причины подобной неудачи:

1. Модули выполняют слишком много связанных, но различных функций - это делает их логику запутанной.
2. При проектировании остались не выявленными общие функции, вследствие чего они рассредоточены (и по-разному реализованы) в разных модулях.
3. Модули взаимодействуют посредством совместно используемых или общих данных самым неожиданным образом.

Чтобы уменьшить сложность ПС, нужно разбить ее на множество небольших, в высокой степени независимых модулей. Довольно высокой степени независимости можно достичь с помощью двух методов оптимизации: усилением внутренних связей в каждом модуле и ослаблением взаимосвязи между модулями. Если рассматривать ПС как набор предложений, связанных между собой некоторыми отношениями (как по выполняемым функциям так и по обрабатываемым данным), то основное, что требуется, это догадаться, как распределить предложения по отдельным "ящикам" (модулям) так, чтобы предложения внутри каждого модуля были тесно связаны, а связь между любой парой предложений в разных модулях была минимальной.

Модульность. Понятие модульность носит универсальный характер и применяется при проектировании информационных систем, при проектировании вычислительной техники, при разработке программного обеспечения. Основной смысл разбиения системы на модули заключается в том, чтобы локализовать и изолировать влияние возмущающих воздействий или изменений. Существуют различные типы модульности в зависимости от того, какого рода возмущения или изменения рассматриваются.

Прочность модулей. Модульность, которая облегчает внесение изменений в систему, можно охарактеризовать как гибкость. Этот тип модульности имеет место в тех случаях, когда система спроектирована так, что изменение одного из требований приводит к необходимости корректировки лишь небольшого числа модулей (желательно только одного). Если модульная структура системы такова, что ее отдельные модули могут быть реализованы различными разработчиками практически независимо друг от друга, то имеет место конструктивная модульность. Модульность, при которой в системе локализуется влияние различных событий, происходящих в реальном масштабе времени, называется событийной модульностью. Если внесение изменений в аппаратуру не требует модификации программного обеспечения, то это свойство называется про-

зрачностью. И наконец, упомянем о функциональной модульности, обеспечивающей обозримость системы. В этом случае система разбивается на легко обозримые части с четко определенным набором функций. Необходимость в разбиении системы на модули по этому принципу может появляться даже тогда, когда другие критерии этого не требуют.

Степень модульности можно определить по двум критериям - **прочности (связности) и сцеплению**. Каждому из этих критериев соответствует некоторое разбиение на классы, позволяющее оценивать модульность системы количественно. Типы связности и сцепления, перечисленные ниже, приведены для того, чтобы дать представление о понятиях, ассоциируемых с критериями модульности.

Связность модулей. Связность модуля определяется как мера независимости его частей. Чем выше связность модуля, тем лучше результат проектирования. Для обозначения связности используется также понятие силы связности модуля. Типы связности модулей приведены в таблице 5.1.

Название и оценки связности у разных авторов различаются, но незначительно.

Модуль с функциональной связностью не может быть разбит на два других модуля, имеющих связность того же типа. Модуль управления обработкой пакетов имеет функциональную связность.

Таблица 5.1.

Связность	Сила связности
Функциональная	10 (сильная связность)
Последовательная	9
Коммуникативная	7
Процедурная	5
Временная	3
Логическая	1
По совпадению	0

Модуль, который может быть разбит только на исток, преобразователь и сток, также имеет функциональную связность. Модуль, имеющий последовательную связность, может быть разбит на последовательные части, выполняющие независимые функции, но совместно реализующие единственную функцию. Если один и тот же модуль используется для оценки, а затем для обработки данных, то он имеет последовательную связность. Если модуль составлен из независимых модулей, разделяющих структуру данных, он имеет коммуникативную связность. Общая структура данных является основой его организации как единого модуля. Если модуль спроектирован так, чтобы упростить работу со сложной структурой данных, изолировать эту структуру, он имеет коммуникативную связность. Такой модуль предназначен для выполнения нескольких различных и независимо используемых функций. Модули высшего уровня иерархической структуры должны иметь функциональную или последовательную связность. Если модули имеют процедурную, временную, логическую или случайную связность, это свидетельствует о недостаточно продуманном их планировании. Процедурная связность обнаруживается в модуле, управляющие конструкции которого организованы так, как изображены на структурной схеме программы. Такая структура модуля может возникнуть при расчленении длинной программы на части в соответствии с передачами управления, но без определения какого-либо функционального базиса при выборе разделительных точек. Процедурная связность может появиться при группировании альтернативных частей программы.

Модуль, содержащий части функционально не связанные, но необходимые в один и тот же момент обработки, имеет временную связность или связность по классу. Связность такого типа имеет место в тех случаях, когда все множество требуемых в момент входа в программу функций выполняется независимым модулем активации. Если в модуле объединены операторы только по

признаку их функционального подобия, а для его настройки применяется алгоритм переключения, такой модуль имеет логическую связность, поскольку его части ничем не связаны, а имеют лишь небольшое сходство между собой. Если операторы модуля объединяются произвольным образом, такой модуль имеет связность по совпадению.

Сцепление модулей. Сцепление модулей представляет собой меру относительной независимости модулей, которая определяет их читабельность и сохранность. Независимые модули могут быть модифицированы без переделки каких-либо других модулей. Слабое сцепление более желательно, так как это означает высокий уровень их независимости. Модули являются полностью независимыми, если каждый из них не содержит о другом никакой информации. Чем больше информации о других модулях используется в них, тем менее они независимы и тем менее теснее сцеплены. Чем очевиднее взаимодействие двух связанных друг с другом модулей, тем проще определить необходимую корректировку одного модуля, зависящую от изменений, производимых в другом. Большая изоляция и непосредственное взаимодействие модулей приводит к трудностям в определении границ изменений одного модуля, которое устраняли бы неизбежные ошибки в другом. Ниже в таблице 5.2 приведены меры сцепления модулей.

Модули сцепления по данным, если они имеют общие единицы, которые передаются от одного к другому как параметры, представляющие собой простые элементы данных, то есть вызывающий модуль "знает" только имя вызываемого модуля, а также типы и значения некоторых его переменных. Изменения в структуре данных в одном из модулей не влияют на другой. Кроме того, модули с этим типом сцепления не имеют общих областей данных или неявных параметров. Меньшая степень сцепления возможна только в том случае, если модули не вызывают друг друга или не обрабатывают одну и ту же информацию.

Таблица 5.2.

Сцепление	Степень сцепления модулей
Независимое	0 (слабое сцепление)
По данным	1
По образцу	3
По общей области	4
По управлению	5
По внешним ссылкам	7
По кодам	9 (сильное сцепление)

Модули сцеплены по образцу, если параметры содержат структуры данных. Недостатком такого сцепления является то, что оба модуля должны знать о внутренней структуре данных.

Модули сцеплены по общей области, если они разделяют одну и ту же глобальную структуру данных.

Модули имеют сцепление по управлению, если какой-либо из них управляет решениями внутри другого с помощью передачи флагов, переключателей или кодов, предназначенных для выполнения функций управления, то есть один из модулей знает о внутренних функциях другого.

Говорят, что модуль предсказуем, если его работа обусловлена только одними параметрами.

Модуль сцеплен по внешним ссылкам, если у него есть доступ к данным в другом модуле через внешнюю точку входа.

Модули имеют сцепление по кодам, если коды их команд перемежаются друг с другом.

26. ПРОЕКТИРОВАНИЕ МОДУЛЯ.

Этап проектирования и программирования каждого модуля - заключительные в общем цикле проектирования. На этих этапах выполняются процессы внешнего проектирования модуля (т.е. разработки сопряжений каждого модуля) и проектирование логики модуля (т.е. ряд шагов, включающих определение данных, выбор алгоритма, разработку логики и собственно программирование). Для многих эти процессы олицетворяют сущность программирования; однако после прочтения первых глав этого пособия должно быть ясно, что эти два процесса - лишь малая часть полного цикла разработки программного обеспечения**.**

Проектирование модуля. Первый шаг при проектировании модуля состоит в определении его внешних характеристик. Эта информация выражается в виде спецификаций модуля, которые содержат все сведения, необходимые вызывающим его модулям, и ничего больше. В частности, спецификации модуля не должны содержать никакой информации о логике модуля или о внутреннем представлении данных. Кроме того, спецификации не должны включать каких бы то ни было ссылок на вызывающие модули или на контексты, в которых этот модуль используется.

Спецификации модуля должны содержать сведения следующих шести типов:

1. Имя модуля. Указывается имя, применяемое для вызова модуля.

2. Функция. Дается определение функции или функций, выполняемых модулем.

3. Список параметров. Определяется число и порядок параметров, передаваемых модулю.

4. Входные параметры. Дается точное описание всех входных параметров. Сюда включается определение формата, размеров, атрибутов, единиц измерения и допустимых диапазонов значений всех входных параметров.

5. Выходные параметры. Дается точное описание всех данных, возвращаемых модулем. Сюда должно входить определение формата, размеров, атрибутов, единиц измерения и допустимых диапазонов значений всех выходных данных. Должна быть описана функциональная связь между входными и выходными данными, т.е. следует показать, какие выходные данные какими входными порождаются. Должны быть также определены выходные данные, порождаемые модулями в случае, когда входные данные не годятся. Для того, чтобы можно было считать модуль специфицированным полностью, должно быть определено его поведение при любых входных условиях.

6. Внешние эффекты. Дается описание всех внешних для программы или системы событий, происходящих при работе модуля. Примерами внешних эффектов являются печать сообщения, чтение запроса с терминала, чтение файла заказов, вывод сообщения об ошибке.

Важно отделить спецификации модуля от другой документации (например, описание его логики), потому что изменение логики может никак не повлиять на вызывающие модули, а изменение спецификаций обычно требует изменить вызывающие модули.

Проектирование логики модуля. Последним в длинной цепи процессов проектирования программного обеспечения является процесс проектирования и собственно программирования (кодирования) внутренней логики каждого модуля. Очень часто идея тщательного планирования здесь отбрасывается, и программист разрабатывает модуль более или менее хаотично. Однако процесс разработки модуля может и должен быть тщательно спланирован. Следующие 11 шагов составляют набросок дисциплинированного подхода к проектированию модуля.

1. Выберите язык. Выбор языка обычно диктуется требованиями контракта или принятыми в организации стандартами. Хотя выбор языка и включен сюда, на самом деле язык должен быть выбран в начальный период работы над проектом, поскольку он влияет на планирование всей работы над проектом.

2. Спроектируйте спецификации модуля. Это процесс определения внешних характеристик каждого модуля, о котором шла речь в предыдущем разделе.

3. Проверьте правильность спецификаций. Правильность спецификаций каждого модуля должна быть проверена сравнением их с информацией о сопряжениях, полученной при проекти-

ровании структуры программы, и анализом их всеми программистами, разрабатывающими вызывающие модули.

4. Выберите алгоритм и структуры данных. Жизненно важным шагом в процессе проектирования логики является выбор алгоритма и структуры данных. Сегодня лишь немногие алгоритмы создаются впервые; огромное их число уже было изобретено, и весьма вероятно, что уже имеется один или несколько алгоритмов, вполне устраивающих проектировщика. Вместо того, чтобы тратить время, заново изобретая алгоритмы и структуры данных, лучше поискать готовые решения.

5. Напишите первое и последнее предложение. Следующий шаг - написать начальный оператор PROCEDURE и конечный - END будущего модуля.

6. Объявите все данные из сопряжения. Следующий шаг состоит в написании тех предложений программы, которые определяют или объявляют все переменные для сопряжения создаваемого модуля.

7. Объявите остальные данные. Напишите предложения, которые определяют или объявляют все другие необходимые переменные. Поскольку трудно предсказать все переменные, которые понадобятся, этот шаг часто перекрывается со следующим.

8. Детализируйте текст программы. Следующий шаг итеративный, он предполагает последовательную детализацию логики модуля, начиная с достаточно высокого уровня абстракции и кончая готовым текстом программы. На этом шаге используются методы пошаговой детализации и структурного программирования, о которых говорится в следующем разделе.

9. Отшлифуйте текст программы. Теперь модуль нужно отшлифовать для достижения ясности и снабдить его дополнительными комментариями, отвечающими на вопросы, которые могут возникнуть при чтении программы.

10. Проверьте правильность программы. Вручную проверяется правильность модуля.

11. Компилируйте модуль. Последний шаг - компиляция модуля. Этот шаг отмечает переход от проектирования к тестированию; компиляцией по существу, начинается тестирование программного обеспечения.

27. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ.

Концепция, называемая структурным программирование, оказала настолько значительное влияние на разработку программного обеспечения, что она, вероятно, войдет в историю как одно из крупнейших достижений в технологии программирования (наряду с концепциями подпрограммы и языка высокого уровня). Однако, хотя самое общее и довольно смутное представление об этой концепции имеется почти у всех, общепринятого четкого ее определения нет. Вот почему здесь не дается ни строгого определения структурного программирования, ни всестороннего анализа сути дела. Вместо этого рассматриваются **основные свойства структурных программ**, а также некоторые интересные моменты, которые часто остаются без внимания.

Определим структурное программирование как программирование, ориентированное на общение с людьми, а не с машиной. Чтобы соответствовать этому определению, структурная программа должна удовлетворять следующим основным требованиям:

1. Текст программы представляет собой композицию трех основных элементов: последовательное соединение (следование), условное предложение (развилка) и повторение (цикл).

2 Употребления GO TO избегают всюду, где это возможно (если ограничения по объему и времени выполнения не критично, то везде). Наихудшим применением GO TO считается переход на оператор, расположенный выше (раньше) в тексте программы.

3. Отказаться по возможности от оператора ELSE. Легко показать, что конструкция ELSE обычно не является необходимой, поскольку ELSE эквивалентно IF (not <логич.выр.>) THEN. Конструкция ELSE необходима только в той редкой ситуации, когда конструкция THEN изменяет одну из переменных в условии. Например, фрагмент программы (PASCAL)

IF (<логич.выр.) THEN

```
    BEGIN
     <опер1>;
    END
  ELSE
    BEGIN
     <опер2>;
    END;
```

можно было переписать так:

```
IF (<логич.выр.>) THEN
    BEGIN
     <опер1>;
    END;
IF (NOT <логич.выр.>) THEN
    BEGIN
     <опер2>;
    END;
```

3. Программа написана в приемлемом стиле (см. ниже).

4. Текст программы напечатан с правильными сдвигами, так что разрывы в последовательности выполнения легко прослеживаются (например, для предложения BEGIN легко найти предложение, заканчивающее группу, без труда устанавливается соответствие между конструкциями THEN и ELSE и т.д.).

5. Каждый модуль имеет ровно один вход и один выход.

6. Текст программы физически разбит на части, чтобы облегчить чтение. Выполняемые предложения каждого модуля должны умещаться на одной странице печатающего устройства.

7. Программа представляет собой простое и ясное решение задачи.

Эти требования четко выражают цели структурного программирования: писать программы минимальной сложности, заставить программиста мыслить ясно, облегчать восприятие программы.

Конструкции структурного программирования.

Структурные программы составлены из четырех основных строительных конструкций, изображенных на рис. 3.13 и конструкции "выбор" (см. рис. 6.1). Часто говорят о трех строительных конструкциях, потому что в большинстве языков программирования имеется только один из двух циклов и нет конструкции "выбор" (CASE), или она примитивна.

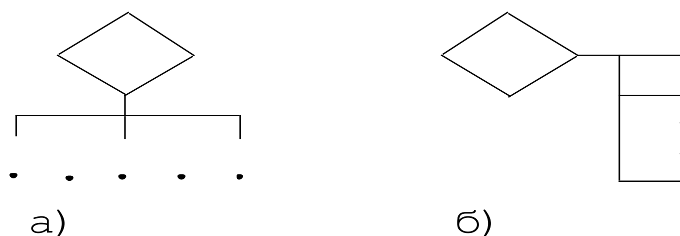


Рис. 6.1. Два варианта конструкции "выбор".

С чисто функциональной точки зрения, можно довольствоваться только следующими конструкциями: последовательность, цикл с предусловием, выбор.

Действительно, все виды циклов преобразуются друг в друга. А если учесть, что цикл с предусловием ("сначала подумай, потом сделай") является более понятным с точки зрения чело-

вического восприятия по отношению к циклу с постусловием ("сначала сделал, потом подумал") то и очевиден выбор в пользу первого.

В свою очередь, конструкция "альтернатива" является упрощением от конструкции "выбор" и если в языке программирования оператор "выбор" имеет возможность анализировать сложные логические выражения, то тогда очевидна избыточность оператора "альтернатива".

Пошаговая детализация.

Структурное программирование до сих пор было у нас представлено как свойство или оценка окончательного текста программы. Необходимо добавить еще один ключевой элемент: методологию, или особенности мыслительного процесса, управляющего проектированием модуля для получения структурной программы. Этот мыслительной процесс, который мы будем сейчас рассматривать, называется пошаговой детализацией и был первоначально предложен Дейкстрой, а затем Виртом.

Пошаговая детализация представляет собой простой процесс, предполагающий первоначальное выражение логики модуля в терминах гипотетического языка "очень высокого уровня" с последующей детализацией каждого предложения в терминах языка более низкого уровня, до тех пор, пока, наконец, не будет достигнут уровень используемого языка программирования. На протяжении всего процесса логика выражается основными конструкциями структурного программирования.

Стиль программирования.

Центральное понятие, связанное со структурным программированием, - стиль программирования, т.е. манера, в которой программист (правильно или неправильно) использует естественный язык. Именно плюсы и минусы стиля программирования обычно оказываются основной причиной таких, например, суждений: "Я видел программу, очень красиво размещенную на странице и без единого GOTO, но при этом абсолютно непостижимую, и другую, совершенно понятную программу, в которой было несколько GOTO".

Ясность программы.

Одна из главнейших задач программиста должна состоять в том, чтобы писать на исходном языке программы, предназначенные для аудитории, состоящей в первую очередь из людей, а не машин. Это требует уделения большего внимания ясности, простоте и доступности текста за счет игнорирования менее важных критериев, например, краткости или машинной эффективности. Выполнение следующих правил помогает писать ясные программы.

Используйте осмысленные имена. Нет ничего хуже программы с именами I, I1, I12, RII, заполненной комментариями, разъясняющими смысл и назначение этих имен. Простой прием - использование более длинных содержательных имен - значительно облегчает чтение программы и сводит к минимуму количество необходимых комментариев.

Избегайте сходных имен. Иногда встречаются программы, в которых используются такие имена переменных, как VALUE и VALUES с отличием в одно-два знака. Это усложняет чтение программы. Выбирая осмысленные имена, старайтесь, чтобы они были как можно менее похожи между собой.

Если в идентификаторах используются цифры, помещайте их только в конце. Некоторые цифры (0,1,2,5) легко спутать с буквами O,I,Z,S.

Никогда не используйте в качестве идентификатора ключевые слова.

Избегайте промежуточных переменных. Хотя в большинстве программ некоторое количество промежуточных переменных необходимо, никогда не создавайте лишних. Например, операторы

X1:=A(1);

X:=X1+1/X1;   лучше заменить на один оператор
X:=A(1)+1/A(1);

Во избежание неоднозначности употребляйте скобки. Порядок выполнения (приоритет) арифметических операций в разных языках программирования всегда является источником путаницы.

Располагайте только один оператор на строке. Размещение нескольких операторов на одной физической строке противоречит правилу структурированного программирования, требующего сдвигать оператор по строке в соответствии с уровнем его вложенности.

Не изменяйте значение цикла в теле цикла. Изменение параметра цикла внутри цикла усложняет и понимание цикла, и доказательство того, что он не является "бесконечным".

Использование языка.

Вторая важная характеристика стиля программирования - способ, которым программист отбирает для употребления (или отбраковывает) возможности языка программирования. Общее правило здесь состоит в том, чтобы понять и использовать все возможности языка, но остерегаться плохо продуманных его особенностей и зависящих от реализации трюков.

Изучите и активно используйте возможности языка.

28. ПРОВЕРКА ПРАВИЛЬНОСТИ ПРОГРАММ.

Программу нельзя использовать до тех пор, пока не будет уверенности в ее надежности. Надежность - это свойство программы, более строгое, чем корректность, поскольку программа может быть корректной, но не быть надежной. Программа является корректной, если удовлетворяет внешним спецификациям, т.е. выдает ожидаемые ответы на определенные комбинации значений входных данных. Программа является надежной, если она корректна, приемлемо реагирует на неточные входные данные и удовлетворительно функционирует в необычных условиях.

В процессе создания программы программист старается предвидеть все возможные ситуации и написать программу так, чтобы она реагировала на них вполне удовлетворительно. Этап тестирования является последней попыткой определить надежность и корректность программы. Проверка надежности включает в себя просмотр проектной документации и текста программы, анализ текста программы, тестирование и, наконец, демонстрацию заказчику того, что программа работает надежно.

Все принципы и методы разработки надежного программного обеспечения можно разбить на четыре группы:

1. Предупреждение ошибок.
2. Обнаружение ошибок.
3. Исправление ошибок.
4. Обеспечение устойчивости к ошибкам.

Предупреждение ошибок. К этой группе относятся принципы и методы, цель которых - не допустить появление ошибок в готовой программе. Большинство методов концентрируется на отдельных процессах перевода и направлено на предупреждение ошибок в этих процессах (упрощение программ, достижение большей точности при переводе, немедленное обнаружение и устранение ошибок).

Обнаружение ошибок. Если предполагать, что в программном обеспечении какие-то ошибки все же будут, то лучшая стратегия в этом случае - включить средства обнаружения ошибок в само программное обеспечение. Немедленное обнаружение имеет два преимущества: можно минимизировать как влияние ошибки, так и последующие затруднения для человека, которому придется извлекать информацию об этой ошибке, находить ее место и исправлять.

Исправление ошибок. После того, как ошибка обнаружена, либо она сама, либо ее последствия должны быть исправлены программным обеспечением. Некоторые устройства способны

обнаружить неисправные компоненты и перейти к использованию резервных. Другой метод - восстановление информации (например, при сбое питания).

Устойчивость к ошибкам. Методы этой группы ставят своей целью обеспечит функционирование программной системы при наличии в ней ошибок. Они разбиваются на три подгруппы: динамическая избыточность (методы голосования, резервных копий); методы отступления (когда необходимо корректно закончить работу - например, закрыть базу данных); изоляция ошибок (основная идея - не дать последствиям ошибки выйти за пределы как можно меньшей части системы программного обеспечения, так, чтобы если ошибка возникнет, то не вся система оказалась бы неработоспособной).

29. ТЕСТИРОВАНИЕ, ДОКАЗАТЕЛЬСТВО, КОНТРОЛЬ, ИСПЫТАНИЕ и др.

Тестирование - процесс выполнения программы с намерением найти ошибки. Если Ваша цель - показать отсутствие ошибок, Вы их найдете не слишком много. Если же Ваша цель - показать наличие ошибок, Вы найдете значительную их часть.

Доказательство - попытка найти ошибки в программе безотносительно к внешней для программы среде. Большинство методов доказательства предполагает формулировку утверждений о поведении программы и затем вывод и доказательство математических теорем о правильности программы. Доказательства могут рассматриваться как форма тестирования, хотя они и не предполагают прямого выполнения программы.

Контроль - попытка найти ошибки, выполняя программу в тестовой, или моделируемой, среде.

Испытание - попытка найти ошибки, выполняя программу в заданной реальной среде.

Аттестация - авторитетное подтверждение правильности программы. При тестировании с целью аттестации выполняется сравнение с некоторым заранее определенным стандартом.

Отладка - не является разновидностью тестирования. Хотя слова "отладка" и "тестирование" часто используются как синонимы, под ними подразумеваются разные виды деятельности. Тестирование - деятельность, направленная на обнаружение ошибок; отладка направлена на установление точной природы известной ошибки, а затем - на исправление этой ошибки. Эти два вида деятельности связаны - результаты тестирования являются исходными данными для отладки.

Тестирование модуля, или автономное тестирование - контроль отдельного программного модуля, обычно в изолированной среде. Тестирование модуля иногда включает также математическое доказательство.

Тестирование сопряжений - контроль сопряжений между частями системы (модулями, компонентами, подсистемами).

Тестирование внешних функций - контроль внешнего поведения системы, определенного внешними спецификациями.

Комплексное тестирование - контроль и испытание системы по отношению к исходным целям. Комплексное тестирование является процессом испытания, если выполняется в среде реальной, жизненной.

Тестирование приемлемости - проверка соответствия программы требованиям пользователя.

Тестирование настройки - проверка соответствия каждого конкретного варианта установки системы с целью выявить любые ошибки, возникшие в процессе настройки системы.

30. БАЗОВЫЕ ПРАВИЛА ТЕСТИРОВАНИЯ.

Обсудим некоторые из важнейших аксиом тестирования. они приведены в настоящем разделе и являются фундаментальными принципами тестирования.

Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы. Поскольку невозможно показать, что программа

не имеет ошибок и, значит, все такие попытки бесплодны, процесс тестирования должен представлять собой попытки обнаружить а программе прежде не найденные ошибки.

Одна из самых сложных проблем при тестировании - решить, когда нужно закончить. Как уже говорилось, исчерпывающее тестирование (т.е. испытание всех входных значений) невозможно. Таким образом, при тестировании мы сталкиваемся с экономической проблемой: как выбрать конечное число тестов, которое дает максимальную отдачу (вероятность обнаружения ошибок) для данных затрат. Известно слишком много случаев, когда написанные тесты имели крайне малую вероятность обнаружения новых ошибок, в то время как довольно очевидные хорошие тесты оставались незамеченными.

Невозможно тестировать свою собственную программу. Ни один программист не должен пытаться тестировать свою собственную программу. Тестирование должно быть в высшей степени разрушительным процессом, но имеются глубокие психологические причины, по которым программист не может относится к своей программе как разрушитель.

Необходимая часть всякого теста - описание ожидаемых выходных данных или результатов. Одна из самых распространенных ошибок при тестировании состоит в том, что результаты каждого теста не прогнозируются до его выполнения. Ожидаемые результаты нужно определять заранее, чтобы не возникла ситуация, когда "глаз видит то, что хочет увидеть". Чтобы совсем исключить такую возможность, лучше разрабатывать самопроверяющиеся тесты, либо пользоваться инструментами тестирования, способными автоматически сверять ожидаемые и фактические результаты.

Избегайте невоспроизводимых тестов, не тестируйте "с лету". В условиях диалога программист слишком часто выполняет тестирование "с лету", т.е., сидя за терминалом, задает конкретные значения и выполняет программу, чтобы посмотреть, что получится. Это -неряшливая и нежелательная форма тестирования. Основной ее недостаток в том, что такие тесты мимолетны; они исчезают по окончании их выполнения. Никогда не используйте тестов, которые тут же выбрасываются.

Готовьте тесты как для правильных, так и для неправильных входных данных. Многие программисты ориентируются в своих тестах на "разумные" условия на входе, забывая о последствиях появления непредусмотренных или ошибочных входных данных. Однако многие ошибки, которые потом неожиданно обнаруживаются в работающих программах, проявляются вследствии никак не предусмотренных действий пользователя программы. Тесты, представляющие неожиданные или неправильные входные данные, часто лучше обнаруживают ошибки, чем "правильные" тесты.

Детально изучите результаты каждого теста. Самые изощренные тесты ничего не стоят, если их результаты удостаиваются лишь беглого взгляда. Тестирование программы означает большее, нежели выполнение достаточного количества тестов; оно также предполагает изучение результатов каждого теста.

По мере того как число ошибок, обнаруженных в некоторой компоненте программного обеспечения увеличивается, растет также относительная вероятность существования в ней необнаруженных ошибок. Этот феномен наблюдался во многих системах. Его понимание способно повысить качество тестирования, обеспечивая обратную связь между результатами прогона тестов и их проектированием. Если конкретная часть системы окажется при тестировании полной ошибок, для нее следует подготовить дополнительные тесты.

Поручайте тестирование самым способным программистам. Тестирование, и в особенности проектирование тестов, - этап в разработке программного обеспечения, требующий особенно творческого подхода. К сожалению, во многих организациях на тестирование смотрят совсем не так. Его часто считают рутинной, нетворческой работой. Однако практика показывает, что проектирование тестов требует даже больше творчества, чем разработка архитектуры и проектирование программного обеспечения.

Считайте тестируемость ключевой задачей Вашей разработки. Хотя "тестируемость" и не фигурировала явно в "проектных" главах, сложность тестирования программы напрямую зависит от ее структуры и качества проектирования. Несмотря на то, что эта связь осознана еще недоста-

точно глубоко, можно утверждать, что многие характеристики хорошего проекта (например, небольшие, в значительной степени независимые модули и независимые подсистемы), улучшают и тестируемость программы.

Никогда не изменяйте программу, чтобы облегчить ее тестируемость. Часто возникает соблазн изменить программу, чтобы было легче ее тестировать. Например, программист, тестируя модуль, содержащий цикл, который должен повторяться 100 раз, меняет его так, чтобы цикл повторялся только 10 раз.

31. ОТЛАДКА.

Рекомендуемый подход к методам отладки аналогичен особенностям проектирования и включает в себя следующие этапы:

1. Поймите задачу. Многие программисты начинают процесс отладки бессистемно, пропуская жизненно важный этап детального анализа имеющихся данных. Первым делом нужно тщательно исследовать, что в программе выполнено правильно, а что - неправильно, чтобы выработать одну или несколько гипотез о природе ошибки. Одна из самых распространенных причин затруднений при отладке - не учтен какой-нибудь существенный фактор в выходных данных программы. Важно исследовать данные в поисках противоречий гипотезе (например, ошибка возникает только в каждой второй записи), потому что это поведет к уточнению гипотезы или, возможно, покажет, что имеется не одна причина ошибки.

2. Разработайте план. Следующий шаг - построить одну или несколько гипотез об ошибке и разработать план проверки этих гипотез.

3. Выполните план. Следуя своему плану, пытайтесь доказать гипотезу. Если план включает несколько шагов, нужно проверить каждый.

4. Проверьте решение. Если кажется, что точное местоположение ошибки обнаружено, необходимо выполнить еще несколько проверок, прежде чем пытаться исправить ошибку. Проанализируйте, может ли предполагаемая ошибка давать в точности известные симптомы. Убедитесь, что найденная причина полностью объясняет все симптомы, а не только их часть. Проверьте, не вызовет ли ее исправление новой ошибки.

Главная причина затруднений при отладке - такая психологическая установка, когда разум видит то, что он ожидает увидеть, а совсем не то, что имеет место в действительности. Один из способов преодоления такой установки - скептицизм в отношении всего, что Вы изучаете, в особенности комментариев и документации. Опытные специалисты по отладке, изучая модуль, часто закрывают комментарии, поскольку комментарии нередко описывают, что программа делает, по мнению ее создателя. Обратный просмотр (чтение программы в обратном направлении) - еще один полезный тактический прием, поскольку он помогает по-новому взглянуть на алгоритм.

Еще одна трудность при отладке - такое состояние, когда все идеи зашли в тупик и найти местоположение ошибки кажется просто невозможно. Это означает, что Вы либо смотрите не туда, куда нужно, и следует еще раз изучить симптомы и построить новую гипотезу, либо подозрения правильные, но разум уже не способен заметить ошибку. Если кажется, что именно так и есть , то лучший принцип - "утро вечера мудренее". Переключите внимание на другую деятельность, и пусть над задачей работает Ваше подсознание. Многие программисты признают, что самые трудные свои задачи они решают во время бритья или по дороге на работу.

Когда Вы найдете и проверите ошибку и убедитесь в том, что нашли ее правильно, не забудьте о том, что вероятность других ошибок в этой части программы теперь выше. Изучите программу в окрестности найденной ошибки в поисках новых неприятностей. Проверьте, не была ли сделана такая же ошибка в других местах программы.

Исследования методов отладки вначале концентрировались на сравнении отладки в пакетном и диалоговом режимах, причем большинство исследований приходило к выводу, что диалоговый режим предпочтительнее. Однако более поздняя работа показала, что, вероятно, наилучший способ отладки - просто читать программу и изо всех сил стараться вникнуть в алгоритм, хотя это требует усердия и собранности.

Важно подчеркнуть, что многие из методов проектирования помогают и в процессе отладки, такие методы, как структурное программирование и хороший стиль программирования не только уменьшают исходное количество ошибок, но и облегчают отладку, делая программу более простой для понимания.

После того, как точно установлено, где находится ошибка, надо ее исправить. Самая большая трудность на этом шаге - суметь охватить проблему целиком; самая распространенная неприятность - устранить только некоторые симптомы ошибки. Избегайте "экспериментальных" исправлений; они показывают, что Вы еще недостаточно подготовлены к отладке этой программы, поскольку не понимаете ее.

В деле исправления ошибок очень важно понимать, что оно возвращает нас назад, к стадии проектирования. Обидно, если после завершения хорошо организованного проектирования весь его строгий порядок нарушается, когда вносятся поправки. Исправления должны выполняться по крайней мере так же строго, как первоначальное выполнение программы. Если необходимо, следует обновить документацию, поправки должны проходить сквозной структурный контроль или другие формы контрольного чтения программы. Ни одна поправка не "мала" настолько, чтобы не нуждаться в тестировании.

По самой своей природе исправления всегда имеют некоторое отрицательное влияние на структуру программы и легкость ее чтения. Тот факт, что они делаются в условиях жесткого давления, усиливает это влияние. Опыт показывает, что при исправлении довольно высока вероятность внесения в программу новой ошибки (обычно от 20 до 50%). Из этого следует, что отладка должна выполняться лучшими программистами проекта.

Изучение процесса отладки.

Один из лучших способов повысить надежность программного обеспечения в нынешних или в будущих проектах - очевидный, но часто упускаемый из виду процесс обучения на сделанных ошибках. Каждую ошибку следует внимательно изучить, чтобы понять, почему она возникла и что должно было бы сделано, чтобы ее предотвратить или обнаружить раньше. Редко можно встретить программиста или организацию, которые выполняли бы такой полезный анализ, а когда он проводится, то обычно имеет поверхностный характер и сводится, например, к классификации ошибок: ошибки проектирования, логические ошибки, ошибки сопряжения или другие, не имеющие особого смысла категории.

Нужно уделять время изучению природы каждой обнаруженной ошибки. Необходимо подчеркнуть, что анализ ошибок должен быть в значительной мере качественным и не сводиться просто к упражнению в количественном подсчете. Чтобы понять причины , лежащие в основе ошибок, и усовершенствовать процессы проектирования и тестирования, нужно ответить на следующие вопросы:

1. Почему возникла именно такая ошибка? В ответе должны быть указаны как первоисточник, так и непосредственный источник ошибки. Например, ошибка могла быть сделана при программировании конкретного модуля, но в ее основе могла лежать неоднозначность в спецификациях или исправление другой ошибки.

2. Как и когда ошибка была обнаружена? Поскольку мы только что добились значительного успеха, почему бы нам не воспользоваться приобретенным опытом?

3. Почему эта ошибка не была обнаружена при проектировании, контроле или на предыдущей фазе тестирования?

4. Что следовало сделать при проектировании или тестировании, чтобы предупредить появление этой ошибки или обнаружить ее раньше?

Собирать эту информацию нужно не только для того, чтобы учиться на ошибках. Официально отчетность об ошибках и об их исправлении необходима и для того, чтобы гарантировать, что обнаруженные ошибки в работающих или тестируемых системах не упущены и что исправления выполнены в соответствии с принятыми нормами.

Другой способ обучения на ошибках в программном обеспечении - учиться на опыте других организаций. К сожалению, это не жизненный вариант, поскольку даже в лучшие времена научного книгоиздания, такие материалы если и встречались то крайне редко.

СТАНДАРТИЗАЦИЯ И УНИФИКАЦИЯ

Стандартизация и унификация в области информационного обеспечения направлена на достижение максимальной упорядоченности на всех уровнях описания данных, на достижение единообразия и непротиворечивости такого описания, что обеспечивает в конечном итоге возможность эффективной автоматизированной обработки информации, создания интегрированных систем управления и организации их информационного взаимодействия.

32. SADT-ТЕХНОЛОГИЯ СТРУКТУРНОГО АНАЛИЗА И ПРОЕКТИРОВАНИЯ.

SADT (Structured Analysis and Design Technique) — одна из самых известных спецификаций анализа и проектирования систем, введенная в 1973 г. Россом (Ross). Министерством обороны США, которое было инициатором разработки стандарта **IDEF**0 как подмножества SADT. Это, наряду с растущей автоматизированной поддержкой, сделало ее более доступной и простой в употреблении.

С точки зрения SADT модель может основываться либо на функциях системы, либо на ее предметах (планах, данных, оборудовании, информации и т.д.). Соответствующие модели принято называть активностными моделями и моделями данных. Активностная модель представляет с нужной степенью подробности систему активностей, которые в свою очередь отражают свои взаимоотношения через предметы системы. Модели данных дуальны к активностным моделям и представляют собой подробное описание предметов системы, связанных системными активностями. Полная методология SADT заключается в построении моделей обеих типов для более точного описания сложной системы. Однако, в настоящее время широкое применение нашли только активностные модели.

Основным рабочим элементом при моделировании является диаграмма. Модель SADT объединяет и организует диаграммы в иерархические древовидные структуры. В состав диаграммы входят блоки, изображающие активности моделируемой системы, и дуги, связывающие блоки вместе и изображающие взаимодействия и взаимосвязи между блоками. SADT требует, чтобы в диаграмме было 3-6 блоков: в этих пределах диаграммы и модели удобны для чтения, понимания и использования.

Блоки на диаграммах изображаются прямоугольниками и сопровождаются текстами на естественном языке, описывающими активности. В отличие от других методов структурного анализа в SADT каждая сторона блока имеет вполне определенное особое назначение: левая сторона блока предназначена для Входов, верхняя - для Управления, правая — для Выходов, нижняя — для Исполнителей. Такое обозначение отражает определенные принципы активности: Входы преобразуются в Выходы, Управления ограничивают или предписывают условия выполнения, Исполнители описывают, за счет чего выполняются преобразования.

Дуги в SADT представляют наборы предметов и маркируются текстами на естественном языке. Предметы могут состоять с активностями в четырех возможных отношениях: Вход, Выход, Управление, Исполнитель. Каждое из этих отношений изображается дугой, связанной с определенной стороной блока — стороны блока чисто графически сортируют предметы, изображаемые дугами. Входные дуги изображают предметы, используемые и преобразуемые активностями. Управляющие дуги обычно изображают информацию, управляющую действиями активностей. Выходные дуги изображают предметы, в которые преобразуются входы. Исполнительские дуги отражают реализацию активностей.

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 активностей и автоматически организуют эти активности в иерархию модели.

Взаимовлияние блоков может выражаться либо в Выходе к другой активности для дальнейшего преобразования, либо в выработке управляющей информации, предписывающей, что именно должна делать другая активность. 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. Отношение Входа возникает, когда Выход одного блока становится Входом для блока с меньшим доминированием. Обратные связи более сложны, поскольку они отражают итерацию или рекурсию — Выходы из одной активности влияют на будущее выполнение других функций, что впоследствии влияет на исходную активность. Управленческая Обратная Связь возникает, когда Выход некоторого блока влияет на блок с большим доминированием, а отношение Входной Обратной Связи имеет место, когда Выход одного блока становится Входом другого блока с большим доминированием. Отношения Выход-Исполнитель встречаются нечасто и представляют особый интерес. Они отражают ситуацию, при которой Выход одной активности становится средством достижения цели другой активностью.

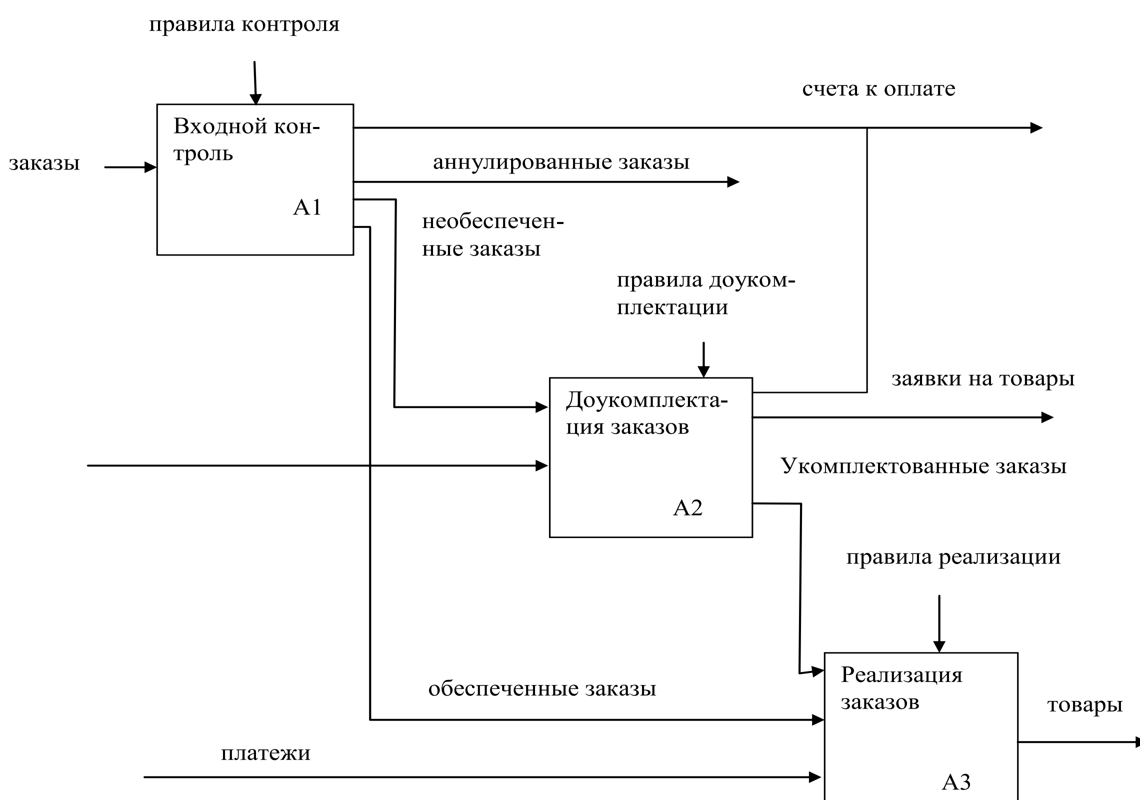


Рис. 9.1: Пример SADT-диаграммы.

Пример SADT-диаграммы, моделирующей деятельность компании, занимающейся распределением товаров по заказам, приведен на рис. 9.1.

Дуги SADT, как правило, изображают наборы предметов, поэтому они могут разветвляться и соединяться вместе различным образом. Разветвления дуги означают, что часть ее содержимого (или весь набор предметов) может появиться в каждом ответвлении дуги. Дуга всегда помечается до разветвления, чтобы дать название всему набору. 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. Слияние дуг указывает, что содержимое каждой ветви участвует в формировании после слияния объединенной дуги. После слияния дуга всегда помечается для указания нового набора, кроме того, каждая ветвь перед слиянием может помечаться в соответствии со следующими правилами: считается, что непомеченные ветви содержат все предметы, указанные в общей метке после слияния; каждая метка ветви уточняет, что именно содержит эта ветвь.

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.

33. UML-ПРОЕКТИРОВАНИЕ.

В начале 90-х годов прошлого столетия возник огромный интерес к парадигме объектов и смежным технологиям. Все это сопровождалось чрезмерным и сбивающим с толку количеством методов проектирования объектно-ориентированных ПС и языков спецификаций процесса моделирования. Они вызывали непонимание и излишнюю фрагментацию рынка. Разработчики программного обеспечения должны были делать выбор между внутренне несовместимыми языками, инструментальными средствами, методами и поставщиками.

По этой причине, когда Rational Software предложила инициативу Unified Modeling Language (UML) (под руководством Гради Буча (Grady Booch), Ивара Якобсона (Ivar Jacobson) и Джима Рамбо (Jim Rumbaugh)) реакция была немедленной и позитивной. Целью проекта было не предложение чего-то нового, а (через совместную работу лидеров передовой технической мысли) объединение наилучших функциональных возможностей различных объектно-ориентированных подходов в один, независимый от производителей язык моделирования и систему обозначений. По этой причине UML очень быстро стал широко практикуемым стандартом. После его принятия организацией Object Management Group в 1996 он стал утвержденным промышленным стандартом.

Язык UML изначально не задумывался в качестве инструментального средства для создания какой-то действующей модели [3]. Чаще всего он служит лишь «ручным» инструментом для выражения концептуальных взглядов. Однако если создать достаточно детальное описание приложения, то такое описание вполне может быть скомпилировано в работоспособную программу.

UML помог сформировать всеобщее понимание значения моделирования при работе со сложным программным обеспечением. Пока он используется на практике только узкой прослойкой разработчиков, которые мыслят стратегически и которые способны видеть общую картин. Тем не менее UML завоевывает все большую популярность среди поставщиков инструментальных средств. Помимо Rational новую спецификацию поддерживают такие компании, как Microsoft, Sun Microsystems, IBM, Oracle, Borland Software и Compuware, входящие в состав консорциума, разрабатывающего язык моделирования. В каждую из наиболее популярных интегрированных инструментальных сред первого уровня встроены определенные возможности моделирования.

Следует также отметить, что, необходимо обращать внимание на поддерживаемые версии UML и на особенности его реализации, т.к. некоторые программные продукты обладают собственными особенностями представления диаграмм, или не полностью следуют стандарту.

Хорошо вписывающийся в условия нового мира, язык UML был специально разработан для распределенной, параллельной и связанной среды. Он основан на объектах. Объекты распределены - каждый поддерживает свое собственное состояние, отличное от других. Объекты параллельны - каждый из них потенциально может действовать параллельно с другими. Объекты связаны - каждый из них может отправлять другим сообщения через сеть соединений. Язык UML не

привязан к какой-либо отдельной платформе или языку программирования, поэтому он хорошо подходит для соединения сетей различных систем. Он разрабатывался с учетом гибкости и поэтому способен адаптироваться к возникающим новым проблемам.

Используя UML участники проекта могут эффективно взаимодействовать друг с другом. Продуманное моделирование в различных его проявлениях предоставляет следующие возможности [3,16,23]:

- однозначное описание функционального поведения системы с помощью прецедентов и сценариев;
- спецификация и анализ технических особенностей системы с помощью моделей;
- ранний акцент на построение гибкой и надежной архитектуры.
- выявление и исключение на ранних стадиях проекта ошибок проектирования

UML состоит из трех частей:

- основные конструкции языка,
- правила их взаимодействия
- некоторые общие для всего языка механизмы.

UML всякое языковое средство, он предоставляет словарь и правила комбинирования слов в этом словаре. В данном случае словарь и правила фокусируются на концептуальном и физическом представлениях системы.

Словарь UML включает три вида основных конструкций:

- сущности - абстракции, являющиеся основными элементами модели;
- отношения - связи между сущностями;
- диаграммы, группирующие представляющие интерес множества сущностей и отношений.

Язык диктует, как создать и прочитать модель, однако не содержит никаких рекомендаций о том, какую модель системы необходимо создать. Это выходит за рамки UML и является прерогативой процесса разработки программного обеспечения.

UML можно использовать и без конкретной методологии, поскольку он не зависит от процесса и какой бы вариант процесса не был бы применен, вы можете использовать диаграммы для документирования принятых в ходе разработки решений и отображения создаваемых моделей.

Использование языка UML основывается на следующих общих принципах моделирования:

- абстрагирование - в модель следует включать только те элементы проектируемой системы, которые имеют непосредственное отношение к выполнению ей своих функций или своего целевого предназначения. Другие элементы опускаются, чтобы не усложнять процесс анализа и исследования модели;
- многомодельность - никакая единственная модель не может с достаточной степенью точности описать различные аспекты системы. Допускается описывать систему некоторым числом взаимосвязанных представлений, каждое из которых отражает определенный аспект её поведения или структуры;
- иерархическое построение - при описании системы используются различные уровни абстрагирования и детализации в рамках фиксированных представлений. При этом первое представление системы описывает её в наиболее общих чертах и является представлением концептуального уровня, а последующие уровни раскрывают различные аспекты системы с возрастающей степенью детализации вплоть до физического уровня. Модель физического уровня в языке UML отражает компонентный состав проектируемой системы с точки зрения ее реализации на аппаратурной и программной платформах конкретных производителей.

34. ЕДИНАЯ СИСТЕМА ПРОГРАММНОЙ ДОКУМЕНТАЦИИ.

Единая система программной документации (ЕСПД) — комплекс государственных стандартов, устанавливающих взаимоувязанные правила разработки, оформления и обращения программ и программной документации.

В стандартах ЕСПД устанавливают требования, регламентирующие разработку, сопровождение, изготовление и эксплуатацию программ, что обеспечивает возможность:

- унификации программных изделий для взаимного обмена программами и применения ранее разработанных программ в новых разработках;
- снижения трудоемкости и повышения эффективности разработки, сопровождения, изготовления и эксплуатации программных изделий;
- автоматизации изготовления и хранения программной документации.

Сопровождение программы включает анализ функционирования, развитие и совершенствование программы, а также внесение изменений в нее с целью устранения ошибок.

Перечень стандартов, входящих в ЕСПД

- ГОСТ 19.001-77 ЕСПД. Общие положения.
- ГОСТ 19.002-80 ЕСПД. Схемы алгоритмов и программ. Правила выполнения. (Заменен на ГОСТ 19.701-90).
- ГОСТ 19.003-80 ЕСПД. Схемы алгоритмов и программ. Обозначения условные графические. (Заменен на ГОСТ 19.701-90).
- ГОСТ 19.004-80 ЕСПД. Термины и определения. (Заменен на ГОСТ 19781-90).
- ГОСТ 19.005-85 ЕСПД. Р-схемы алгоритмов и программ. Обозначения условные графические и правила выполнения.
- ГОСТ 19.101-77 ЕСПД. Виды программ и программных документов.
- ГОСТ 19.102-77 ЕСПД. Стадии разработки.
- ГОСТ 19.103-77 ЕСПД. Обозначение программ и программных документов.
- ГОСТ 19.104-78 ЕСПД. Основные надписи.
- ГОСТ 19.105-78 ЕСПД. Общие требования к программным документам.
- ГОСТ 19.106-78 ЕСПД. Требования к программным документам, выполненным печатным способом.
- ГОСТ 19.201-78 ЕСПД. Техническое задание. Требования к содержанию и оформлению.
- ГОСТ 19.202-78 ЕСПД. Спецификация. Требования к содержанию и оформлению.
- ГОСТ 19.301-79 ЕСПД. Программа и методика испытаний. Требования к содержанию и оформлению.
- ГОСТ 19.401-78 ЕСПД. Текст программы. Требования к содержанию и оформлению.
- ГОСТ 19.402-78 ЕСПД. Описание программы.
- ГОСТ 19.403-79 ЕСПД. Ведомость держателей подлинников.
- ГОСТ 19.404-79 ЕСПД. Пояснительная записка. Требования к содержанию и оформлению.
- ГОСТ 19.501-78 ЕСПД. Формуляр. Требования к содержанию и оформлению.
- ГОСТ 19.502-78 ЕСПД. Общее описание. Требования к содержанию и оформлению.
- ГОСТ 19.503-79 ЕСПД. Руководство системного программиста. Требования к содержанию и оформлению.
- ГОСТ 19.504-79 ЕСПД. Руководство программиста. Требования к содержанию и оформлению.
- ГОСТ 19.505-79 ЕСПД. Руководство оператора. Требования к содержанию и оформлению.
- ГОСТ 19.506-79 ЕСПД. Описание языка. Требования к содержанию и оформлению.
- ГОСТ 19.507-79 ЕСПД. Ведомость эксплуатационных документов.
- ГОСТ 19.508-79 ЕСПД. Руководство по техническому обслуживанию. Требования к содержанию и оформлению.
- ГОСТ 19.601-78 ЕСПД. Общие правила дублирования, учета и хранения.
- ГОСТ 19.602-78 ЕСПД. Правила дублирования, учета и хранения программных документов, выполненных печатным способом.

- ГОСТ 19.603-78 ЕСПД. Общие правила внесения изменений.
- ГОСТ 19.604-78 ЕСПД. Правила внесения изменений в программные документы, выполненные печатным способом.
- ГОСТ 19.701-90 (ИСО 5807-85) ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. (Взамен ГОСТ 19.002-80, ГОСТ 19.003-80).

Последний из списка ЕСПД ГОСТ 19.701-90 (ИСО 5807-85) - распространяется на условные обозначения (символы) в схемах алгоритмов, программ, данных и систем и устанавливает правила выполнения схем, используемых для отображения различных видов задач обработки данных и средств их решения в Российской Федерации и ряде стран СНГ. По существу этот стандарт вводит язык спецификаций, основанный на парадигме функциональных схем и схем передач управления структурного описания программных систем.