МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение
Высшего профессионального образования
«Оренбургский государственный университет»

Кафедра математического обеспечения информационных систем

И.В. ВЛАЦКАЯ, Н.А. ЗАЕЛЬСКАЯ, Н.С. ШАМСУДИНОВА

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
К ЛАБОРАТОРНЫМ И САМОСТОЯТЕЛЬНЫМ РАБОТАМ СТУДЕНТОВ

Рекомендовано Редакционно-издательским советом
Государственного образовательного учреждения
высшего профессионального образования
«Оренбургский государственный университет»

Оренбург 2008

УДК 004.4(076.5)
ББК 32.973.26-018.2я73
В – 58

Рецензент
кандидат педагогических наук, доцент Шухман А.Е.

Влацкая И.В.

В – 58 Технология разработки программного обеспечения: методические указания к лабораторным и самостоятельным работам студентов/И.В. Влацкая, Н.А. Заельская, Н.С. Шамсудинова. – Оренбург ГОУОГУ, 2008.– 58 с.

Методические указания предназначены для выполнения лабораторных работ по курсу «Технология разработки программного обеспечения» и могут быть использованы в самостоятельной работе студентов.

ББК 32.973.26-018.2я73

Содержание

Введение

Современная индустрия программного обеспечения характеризуется очень высокой степенью конкуренции. Создание программного обеспечения для персональных компьютеров за последние годы превратилось в важную и мощную сферу промышленности. Развитие программного обеспечения, предназначенного для широкого круга пользователей, происходит уже не в состязании индивидуальных программистов, а в процессе ожесточенной конкурентной борьбы между фирмами-производителями программного обеспечения.

При разработке программ основной задачей является обеспечение их успеха, т.е. необходимо, чтобы программы обладали следующими качествами:

- функциональность программы, т.е. полнота удовлетворения ею потребностей пользователя;
- наглядный, удобный, интуитивно понятный и привычный пользователю интерфейс (т.е. способ взаимодействия программы с пользователем);
- простота освоения программы даже начинающими пользователями, для чего используются информативные подсказки, встроенные справочники и подробная документация;
- надежность программы, т.е. устойчивость ее к ошибкам пользователя,
- отказам оборудования и т.д., и разумные ее действия в этих ситуациях.

В методическом указании излагаются современные технологии разработки программного обеспечения. Предлагаемое вниманию методическое указание может рассматриваться в качестве основного источника для выполнения лабораторных работ по курсу «Технология разработки программного обеспечения» для студентов, обучающихся по направлению 010503 – «Математическое обеспечение и администрирование информационных систем».

Методические указания могут быть рекомендованы также более широкому кругу читателей, интересующихся вопросами качества и надежности при создании программных продуктов, методами и организацией процесса проектирования, современным уровнем решения этих задач.

В методическом указании по дисциплине ТРПО содержатся основные теоретические положения по курсу:

- Характеристики качества и надежности программного продукта.
- Этапы жизненного цикла программного обеспечения.
- Декомпозиция задачи. Структурный и модульный подход к проектированию.
- Характеристика программного модуля. Потоки данных и процессы.
- Методы тестирования и виды отладки программного средства.

По каждой из рассмотренных глав студентами выполняется лабораторная работа. Целью выполнения лабораторных работ является изучение теоретической части материала и приобретение практических навыков в рассматриваемой области.

1 Основные характеристики качества и надежности программного продукта

1.1 Качество

Качество (quality) программного средства - это совокупность его черт и характеристик, которые влияют на его способность удовлетворять заданные потребности пользователей. Задача оценки качества программного средства опирается на необходимость формирования системы показателей, характеризующих качество функционирования программного средства, с учетом технологических возможностей разработчика. Для конкретных программных средств доминирующие критерии качества выделяются и определяются при проектировании требованиями технического задания и задачами функционирования программного средства. Программы как объекты проектированной разработки характеризуются следующими показателями:

1) проблемно - ориентированная область применения, а также техническое и социальное назначение программного комплекса;
2) конкретный тип решаемых функциональных задач с достаточно определенной областью применения;
3) объем и сложность совокупности программ и базы данных, решающих единую целевую задачу данного типа;
4) необходимый состав и требуемые значения характеристик качества функционирования программы и величина допустимого ущерба из-за недостаточного качества;
5) степень связи решаемых задач с реальным масштабом времени или допустимой длительностью ожидания решения задачи;
6) прогнозируемые значения длительности эксплуатации и перспективы создания множества версий программ;
7) предполагаемый тираж производства и применения комплекса программ;
8) степень необходимой документированности программ;

Качество программного средства в среде пользователей может отличаться от качества в среде разработчиков, поскольку некоторые функции могут быть невидимыми пользователю и не использованными им.

Пользователь оценивает только те атрибуты программного средства, которые видимы и полезны в реальном применении, поэтому к дефектам программных комплексов следует относить не только прямые потери при их применении, но и избыточные свойства, которые не нужны пользователю, но потребовали дополнительных затрат.

Спецификация качества определяет основные ориентиры (цели), которые на всех этапах разработки ПС так или иначе влияют при принятии различных решений на выбор подходящего варианта. Однако каждый из *примитивов качества* имеет свои особенности такого влияния, тем самым, обеспечение его наличия в ПС может потребовать своих подходов и методов разработки ПС или

отдельных его частей. Кроме того, *противоречивость критериев качества ПС*, а также и выражающих их примитивов качества: хорошее обеспечение одного какого-либо примитива качества ПС может существенно затруднить или сделать невозможным обеспечение некоторых других из этих примитивов. Поэтому существенная часть процесса обеспечения качества ПС состоит из поиска приемлемых компромиссов. Эти компромиссы частично должны быть определены уже в спецификации качества ПС: модель качества ПС должна конкретизировать требуемую степень присутствия в ПС каждого из примитивов качества и определять приоритеты достижения этих степеней.

Обеспечение качества осуществляется в каждом технологическом процессе: принятые в нем решения в той или иной степени оказывают влияние на качество ПС в целом. В частности и потому, что значительная часть примитивов качества связана не столько со свойствами программ, входящих в ПС, сколько со свойствами документации. В силу отмеченной *противоречивости примитивов качества*.

Весьма важно придерживаться выбранных приоритетов в их обеспечении. При этом следует придерживаться двух общих принципов:

сначала необходимо обеспечить требуемую функциональность и надежность ПС, а затем уже доводить остальные критерии качества до приемлемого уровня их присутствия в ПС;

нет никакой необходимости и, может быть, даже вредно добиваться более высокого уровня присутствия в ПС какого-либо примитива качества, чем тот, который определен в спецификации качества ПС.

1.2 Схема угроз качеству программных средств и методы их предотвращения

Рисунок 1 – Схема угроз качеству программных средств

Различия между ожидаемым и полученным результатами функционирования программы могут быть следствием ошибок, не только в созданных программах и данных, но и системных ошибок в первичных требованиях спецификаций, являвшихся исходной базой данных для создания программного средства.

1.3 Надежность

Надежность (reliability) программного средства - это его способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью.

Альтернативой правильного ПС является надежное ПС. При этом под *отказом в ПС* понимают проявление в нем ошибки. Таким образом, надежное ПС не исключает наличия в нем ошибок - важно лишь, чтобы эти ошибки при практическом применении этого ПС в заданных условиях проявлялись достаточно редко. Убедиться, что ПС обладает таким свойством можно при его испытании путем тестирования, а также при практическом применении. Таким образом, фактически мы можем разрабатывать лишь надежные, а не правильные ПС.

ГОСТ 13377 - 75 надежность определяется как средство объекта выполнять заданные функции, сохраняя во времени значения установленных эксплуатационных показателей в заданных пределах, соответствующих заданным режимом и условием использования технического обслуживания, ремонта, хранения и транспортирования. Таким образом, надежность является внутренним свойством системы, заложенным при ее создании и проявляющемся во времени при функционировании и эксплуатации. Свойство надежности изучается теорией надежности, которая является системой определенных идей, математических моделей и методов, направленных на решение проблем прогнозирования (или предсказания), оценки и оптимизации различных показателей надежности. При применении понятия надежности к программному средству следует учитывать особенности и отличия этих объектов от традиционных технических систем, а именно не для всех видов программ можно говорить о надежности. Это имеет смысл по отношению к программным средствам, функционирующим в реальном времени и непосредственно взаимодействующим с внешней средой. Доминирующими факторами, определяющими надежность программ, являются дефекты, ошибки проектирования и разработки, второстепенное значение имеют физическое разрушение программных компонент при внешних воздействиях.

Относительно редкое разрушение программных средств и необходимость их физической замены приводит к принципиальному изменению понятий сбоя и отказа программ.

Для повышения надежности программного средства особое значение имеют методы автоматического сокращения, длительности восстановления и преобразование отказов в кратковременные сбои путем введения в программное средство временной программной и информационной избыточности.

Непредсказуемость места, времени и вероятности проявления дефектов и ошибок, а также редкое обнаружение их при реальной эксплуатации достаточно надежных программных средств, все это не позволяет эффективно использовать традиционные методы априорного расчета показателей надежности.

В международном стандарте ISO 9126:1991г. формализованы основные показатели качества и надежности.

ПС может обладать различной степенью надежности. Как измерять эту степень? Так же как в технике, степень надежности можно характеризовать вероятностью работы ПС без отказа в течение определенного периода времени. Однако в силу специфических особенностей ПС определение этой вероятности наталкивается на ряд трудностей по сравнению с решением этой задачи в технике. При оценке степени надежности ПС следует также учитывать последствия каждого отказа. Некоторые ошибки в ПС могут вызывать лишь некоторые неудобства при его применении, тогда как другие ошибки могут иметь катастрофические последствия, например, угрожать человеческой жизни. Поэтому для оценки надежности ПС иногда используют дополнительные показатели, учитывающие стоимость (вред) для пользователя каждого отказа.

Характеристики и субхарактеристики качества программных средств по стандарту ISO 9126:1991 представлены на рисунке 2.

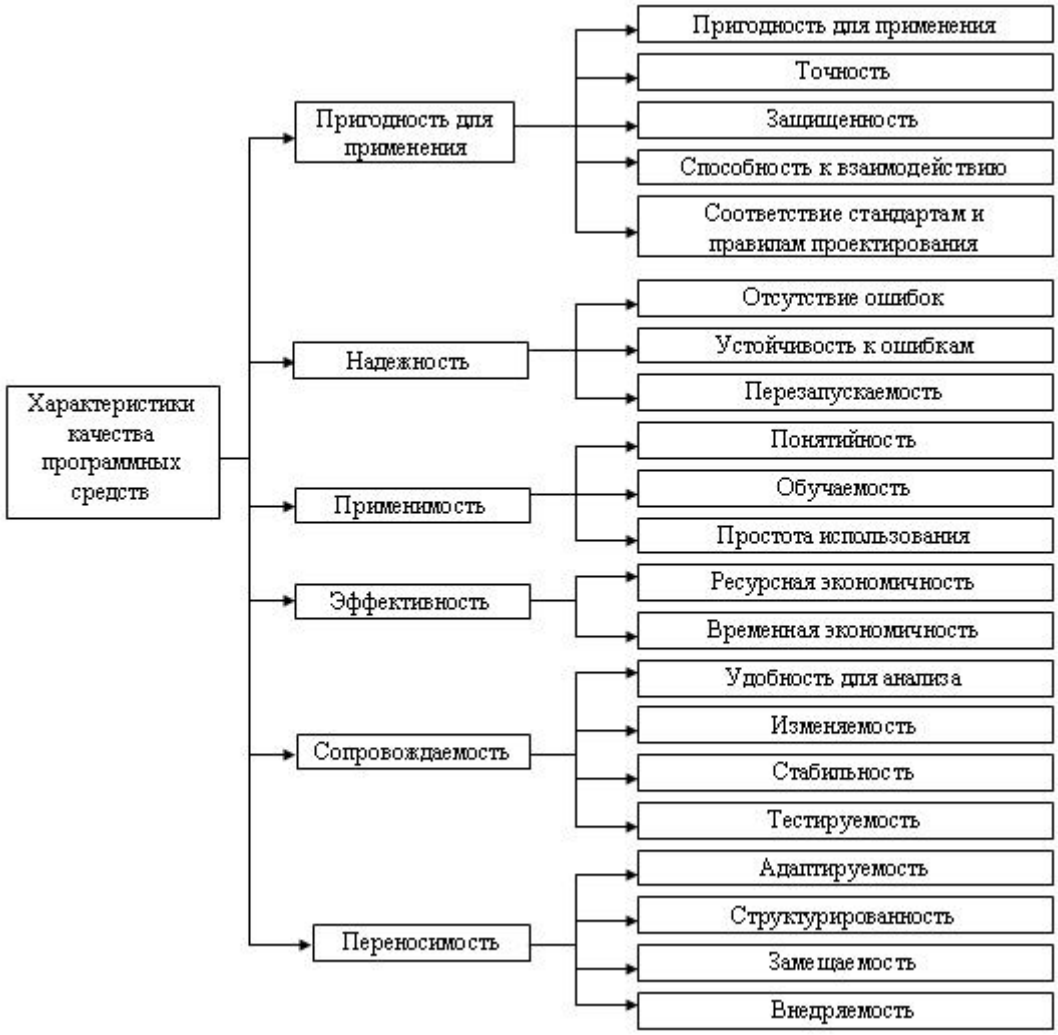

Рисунок 2 – Характеристики качества программных средств

2 Рассмотрение этапов жизненного цикла программного обеспечения

2.1 Понятие жизненного цикла

Одним из базовых понятий методологии проектирования программного обеспечения (ПО) является понятие ее жизненного цикла (ЖЦ ПО).

ЖЦ ПО (*software life cycle*) *- это непрерывный процесс, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации*

Жизненный цикл охватывает довольно сложный процесс создания и использования ПО. Этот процесс может быть организован по-разному для разных классов ПО и в зависимости от особенностей коллектива разработчиков.

В настоящее время можно выделить 5 основных подходов к организации процесса создания и использования ПО.

1. *Водопадный подход*. При таком подходе разработка ПО состоит из цепочки этапов. На каждом этапе создаются документы, используемые на последующем этапе. В исходном документе фиксируются требования к ПО. В конце этой цепочки создаются программы, включаемые в ПО.
2. *Исследовательское программирование*. Этот подход предполагает быструю (насколько это возможно) реализацию рабочих версий программ ПО, выполняющих лишь в первом приближении требуемые функции. После экспериментального применения реализованных программ производится их модификация с целью сделать их более полезными для пользователей. Этот процесс повторяется до тех пор, пока ПО не будет достаточно приемлемо для пользователей. Такой подход применялся на ранних этапах развития программирования, когда технологии программирования не придавали большого значения (использовалась интуитивная технология). В настоящее время этот подход применяется для разработки таких ПО, для которых пользователи не могут точно сформулировать требования (например, для разработки систем искусственного интеллекта).
3. *Прототипирование*. Этот подход моделирует начальную фазу исследовательского программирования вплоть до создания рабочих версий программ, предназначенных для проведения экспериментов с целью установить требования к ПО. В дальнейшем должна последовать разработка ПО по установленным требованиям в рамках какого-либо другого подхода (например, водопадного).
4. *Формальные преобразования.* Этот подход включает разработку формальных спецификаций ПО и превращение их в программы путем корректных преобразований. На этом подходе базируется компьютерная технология (CASE-технология) разработки ПО.

5. *Сборочное программирование.* Этот подход предполагает, что ПО конструируется, главным образом, из компонент, которые уже существуют. Должно быть некоторое хранилище (библиотека) таких компонент, каждая из которых может многократно использоваться в разных ПО. Такие компоненты называются *повторно используемыми* (*reusable*). Процесс разработки ПО при данном подходе состоит скорее из сборки программ из компонент, чем из их программирования.

2.2 Каскадная модель ЖЦ

В изначально существовавших однородных ПО каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся *каскадный способ*. Его основной характеристикой является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем (рисунок 3). Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

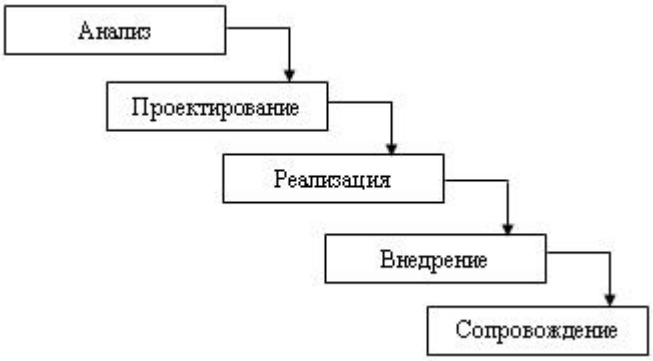

Рисунок 3 – Каскадная схема разработки ПО

Однако в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимал следующий вид (рисунок 4):

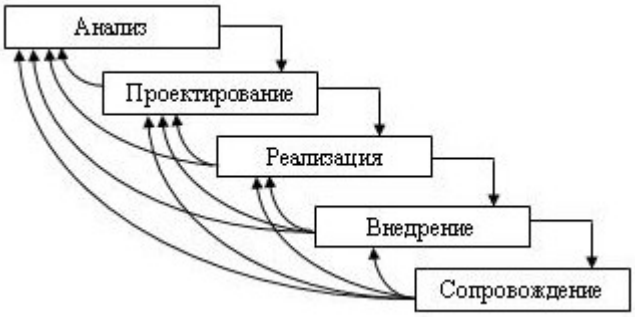

Рисунок 4 – Реальный процесс разработки ПО по каскадной схеме

Рассмотрим основные этапы каскадного ЖЦ. Всем этим этапам сопутствуют процессы документирования и управления (*management*) ПО.

Этап *анализа* ПО включает процессы, приводящие к созданию некоторого документа, который мы будем называть *внешним описанием* (*requirements document*) *ПО*. Этот документ является описанием поведения ПО с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества. Внешнее описание ПО начинается с анализа и определения требований к ПО со стороны пользователей (заказчика), а также включает процессы спецификации этих требований.

Проектирование (*designing*) ПО охватывает процессы: разработку архитектуры ПО, разработку структур программ ПО и их детальную спецификацию.

Этапы проектирования и реализации (кодирования) часто перекрываются, иногда довольно сильно. Это означает, что кодирование некоторых частей программного средства может быть начато до завершения этапа конструирования.

Этап *реализации можно разделить на еще два этапа: кодирование и аттестацию. Кодирование* (*coding*) ПО включает процессы создания текстов программ на языках программирование, их отладку с тестированием ПО. На этапе *аттестации* (*acceptance*) ПО производится оценка качества ПО. Если эта оценка оказывается приемлемой для практического использования ПО, то разработка ПО считается законченной. Это обычно оформляется в виде некоторого документа, фиксирующего решение комиссии, проводящей аттестацию ПО.

Программное изделие (ПИ) – экземпляр или копия разработанного ПО.

Изготовление ПИ – это процесс генерации и/или воспроизведения (снятия копии) программ и программных документов ПО с целью их поставки пользователю для применения по назначению.

Внедрение ПИ – это совокупность работ по обеспечению изготовления требуемого количества ПИ в установленные сроки. Стадия внедрения ПИ в жизненном цикле ПО является, по существу, вырожденной (не существенной), так как представляет рутинную работу, которая может быть выполнена автоматически и без ошибок. Этим она принципиально отличается от стадии производства различной техники. В связи с этим в литературе эту стадию, как правило, не включают в жизненный цикл ПО.

Сопровождение (*maintenance*) ПО – это процесс сбора информации о качестве ПО в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях.

2.3 Спиральная модель ЖЦ

Спиральная модель ЖЦ (рисунок 5), делает упор на начальные этапы ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка

спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации.

Рисунок 5 – Спиральная модель ЖЦ

2.4 Методология разработки ПО RAD

Одним из возможных подходов к разработке ПО в рамках спиральной модели ЖЦ является получившая в последнее время широкое распространение методология быстрой разработки приложений RAD (Rapid Application Development). Под этим термином обычно понимается процесс разработки ПО, содержащий три элемента:

небольшую команду программистов (от 2 до 10 человек);

короткий, но тщательно проработанный производственный график (от 2 до 6 мес.);

повторяющийся цикл, при котором разработчики, по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные через взаимодействие с заказчиком.

Команда разработчиков должна представлять из себя группу профессионалов, имеющих опыт в анализе, проектировании, генерации кода и тестировании ПО с использованием CASE-средств. Члены коллектива должны также уметь трансформировать в рабочие прототипы предложения конечных пользователей.

Жизненный цикл ПО по методологии RAD состоит из четырех фаз:

фаза анализа и планирования требований;

фаза проектирования;

фаза построения;

фаза внедрения.

На фазе *анализа и планирования требований* пользователи системы определяют функции, которые она должна выполнять, выделяют наиболее приоритетные из них, требующие проработки в первую очередь, описывают информа-

ционные потребности. Определение требований выполняется в основном силами пользователей под руководством специалистов-разработчиков. Ограничивается масштаб проекта, определяются временные рамки для каждой из последующих фаз. Кроме того, определяется сама возможность реализации данного проекта в установленных рамках финансирования, на данных аппаратных средствах и т.п. Результатом данной фазы должны быть список и приоритетность функций будущей ПО, предварительные функциональные и информационные модели ПО.

На фазе *проектирования* часть пользователей принимает участие в техническом проектировании системы под руководством специалистов-разработчиков. CASE-средства используются для быстрого получения работающих прототипов приложений. Пользователи, непосредственно взаимодействуя с ними, уточняют и дополняют требования к системе, которые не были выявлены на предыдущей фазе. Более подробно рассматриваются процессы системы. Анализируется и, при необходимости, корректируется функциональная модель. Каждый процесс рассматривается детально. При необходимости для каждого элементарного процесса создается частичный прототип: экран, диалог, отчет, устраняющий неясности или неоднозначности. Определяются требования разграничения доступа к данным. На этой же фазе происходит определение набора необходимой документации.

После детального определения состава процессов оценивается количество функциональных элементов разрабатываемой системы и принимается решение о разделении ПО на подсистемы, поддающиеся реализации одной командой разработчиков за приемлемое для RAD-проектов время – порядка 60 – 90 дней. С использованием CASE-средств проект распределяется между различными командами (делится функциональная модель). Результатом данной фазы должны быть:

общая информационная модель системы;

функциональные модели системы в целом и подсистем, реализуемых отдельными командами разработчиков;

точно определенные с помощью CASE-средства интерфейсы между автономно разрабатываемыми подсистемами;

построенные прототипы экранов, отчетов, диалогов.

На фазе *построения* разработчики производят итеративное построение реальной системы на основе полученных в предыдущей фазе моделей, а также требований нефункционального характера. Программный код частично формируется при помощи автоматических генераторов, получающих информацию непосредственно из репозитория CASE-средств. Конечные пользователи на этой фазе оценивают получаемые результаты и вносят коррективы, если в процессе разработки система перестает удовлетворять определенным ранее требованиям. Тестирование системы осуществляется непосредственно в процессе разработки.

После окончания работ каждой отдельной команды разработчиков производится постепенная интеграция данной части системы с остальными, формируется полный программный код, выполняется тестирование совместной рабо-

ты данной части приложения с остальными, а затем тестирование системы в целом. Завершается физическое проектирование системы:

определяется необходимость распределения данных;
производится анализ использования данных;
производится физическое проектирование базы данных;
определяются требования к аппаратным ресурсам;
определяются способы увеличения производительности;
завершается разработка документации проекта.

Результатом фазы является готовая система, удовлетворяющая всем согласованным требованиям.

На фазе *внедрения* производится обучение пользователей, организационные изменения и параллельно с внедрением новой системы осуществляется работа с существующей системой (до полного внедрения новой).

Приведенная схема разработки ПО не является абсолютной. Возможны различные варианты, зависящие, например, от начальных условий, в которых ведется разработка: разрабатывается совершенно новая система; уже было проведено обследование предприятия и существует модель его деятельности; на предприятии уже существует некоторая ПО, которая может быть использована в качестве начального прототипа или должна быть интегрирована с разрабатываемой.

Оценка размера приложений производится на основе так называемых функциональных элементов (экраны, сообщения, отчеты, файлы и т.п.) Подобная метрика не зависит от языка программирования, на котором ведется разработка. Размер приложения, которое может быть выполнено по методологии RAD, для хорошо отлаженной среды разработки ПО с максимальным повторным использованием программных компонентов, определяется следующим образом:

< 1000 функциональных элементов	один человек
1000-4000 функциональных элементов	одна команда разработчиков
> 4000 функциональных элементов	4000 функциональных элементов на одну команду разработчиков

2.5 Проблемы межу пользователем и программистом и способы их преодоления

В заключении отметим, что разработка ПО носит *творческий характер* (на каждом шаге приходится делать какой-либо выбор, принимать какое-либо решение), а не сводится к выполнению какой-либо последовательности регламентированных действий. Тем самым эта разработка ближе к процессу проектирования каких-либо сложных устройств, но никак не к их массовому производству. Этот творческий характер разработки ПО сохраняется до самого ее конца. Как же преодолеет разногласии между пользователем и разработчиком, которые несомненно возникнут в процессе работе? Как обеспечить, чтобы ПО

выполняло то, что пользователю разумно ожидать от него? Для этого разработчикам необходимо правильно понять, во-первых, чего хочет пользователь, и, во-вторых, его уровень подготовки и окружающую его обстановку. Ясное описание соответствующей сферы деятельности пользователя или интересующей его проблемной области во многом облегчает достижение разработчиками этой цели. При разработке ПО следует привлекать пользователя для участия в процессах принятия решений, а также тщательно освоить особенности его работы. С учетом этого при разработки ПО необходимо применять везде, где это возможно смежный контроль и сочетание как статических, так и динамических методов контроля.

Смежный контроль означает, проверку полученного документа лицами, не участвующими в его разработке, с двух сторон: во-первых, со стороны автора исходного для контролируемого процесса документа, и, во-вторых, лицами, которые будут использовать полученный документ в качестве исходного в последующих технологических процессах. Такой контроль позволяет обеспечивать однозначность интерпретации полученного документа.

Сочетание статических и динамических методов контроля означает, что нужно не только контролировать документ как таковой, но и проверять, какой процесс обработки данных он описывает.

Реальное применение любой технологии проектирования, разработки и сопровождения ПО в конкретной организации и конкретном проекте невозможно без выработки ряда стандартов (правил, соглашений), которые должны соблюдаться всеми участниками проекта. К таким стандартам относятся следующие:

стандарт проектирования;

стандарт оформления проектной документации;

стандарт пользовательского интерфейса.

Стандарт проектирования должен устанавливать:

набор необходимых моделей (диаграмм) на каждой стадии проектирования и степень их детализации;

правила фиксации проектных решений на диаграммах, в том числе: правила именования объектов (включая соглашения по терминологии), набор атрибутов для всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм, включая требования к форме и размерам объектов, и т.д.;

требования к конфигурации рабочих мест разработчиков, включая настройки операционной системы, настройки CASE-средств, общие настройки проекта и т.д.;

механизм обеспечения совместной работы над проектом, в том числе: правила интеграции подсистем проекта, правила поддержания проекта в одинаковом для всех разработчиков состоянии (регламент обмена проектной информацией, механизм фиксации общих объектов и т.д.), правила проверки проектных решений на непротиворечивость и т.д.

Стандарт оформления проектной документации должен устанавливать:

комплектность, состав и структуру документации на каждой стадии проектирования;

требования к ее оформлению (включая требования к содержанию разделов, подразделов, пунктов, таблиц и т.д.),

правила подготовки, рассмотрения, согласования и утверждения документации с указанием предельных сроков для каждой стадии;

требования к настройке издательской системы, используемой в качестве встроенного средства подготовки документации;

требования к настройке CASE-средств для обеспечения подготовки документации в соответствии с установленными требованиями.

Стандарт интерфейса пользователя должен устанавливать:

правила оформления экранов (шрифты и цветовая палитра), состав и расположение окон и элементов управления;

правила использования клавиатуры и мыши;

правила оформления текстов помощи;

перечень стандартных сообщений;

правила обработки реакции пользователя.

3 Декомпозиция задачи. Структурный и модульный подход к проектированию

3.1 Архитектуры программного средства

Архитектура ПС – это его строение как оно видно (или должно быть видно) извне его, т.е. представление ПС как системы, состоящей из некоторой совокупности взаимодействующих подсистем. В качестве таких подсистем выступают обычно отдельные программы. Разработка архитектуры является первым этапом борьбы со сложностью ПС, на котором реализуется принцип выделения относительно независимых компонент.

Основные задачи разработки архитектуры ПС:

выделение программных подсистем и отображение на них внешних функций (заданных во внешнем описании) ПС;

определение способов взаимодействия между выделенными программными подсистемами.

С учетом принимаемых на этом этапе решений производится дальнейшая конкретизация и функциональных спецификаций.

Различают следующие основные классы архитектур программных средств:

цельная программа;

комплекс автономно выполняемых программ;

слоистая программная система;

коллектив параллельно выполняемых программ.

3.2 Методы проектирования

Приступая к разработке каждой программы ПС, следует иметь в виду, что она, как правило, является большой системой, поэтому мы должны принять меры для ее упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями. А сам такой метод разработки программ называют *модульным* программированием.

Программный модуль – это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен с другими модулями программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю. Таким образом, программный модуль может рассматриваться и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании (т.е. как средство накопления и многократного использования программистских знаний).

Модульное программирование является воплощением, в процессе разработки программ, борьбы со сложностью, обеспечивает независимость компонент системы, и использование иерархических структур.

Для избегания сложностей формулируются определенные требования, которым должен удовлетворять программный модуль, т.е. выявляются основные характеристики «хорошего» программного модуля и используют древовидные модульные структуры программ (включая деревья со сросшимися ветвями). В узлах такого дерева размещаются программные модули, а направленные дуги (стрелки) показывают статическую подчиненность модулей, т.е. каждая дуга показывает, что в тексте модуля, из которого она исходит, имеется ссылка на модуль, в который она входит. Другими словами, каждый модуль может обращаться к подчиненным ему модулям, т.е. выражается через эти модули. При этом модульная структура программы, в конечном счете, должна включать и совокупность спецификаций модулей, образующих эту программу.

Спецификация программного модуля содержит:

синтаксическую спецификацию его входов, позволяющую построить на используемом языке программирования синтаксически правильное обращение к нему (к любому его входу),

функциональную спецификацию модуля (описание семантики функций, выполняемых этим модулем по каждому из его входов).

Функциональная спецификация модуля строится так же, как и функциональная спецификация ПС.

В процессе разработки программы ее модульная структура может по-разному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Поэтому можно говорить о разных методах разработки структуры программы. Обычно в литературе обсуждаются два метода: метод восходящей разработки и метод нисходящей разработки.

Метод *восходящей разработки* заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модулей самого нижнего уровня (листья дерева модульной структуры программы), в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в принципе в таком же (восходящем) порядке, в каком велось их программирование.

Метод *нисходящей разработки* заключается в следующем. Как и в предыдущем методе сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модуля самого верхнего уровня (головного), переходя к программированию какого-либо другого модуля только в том случае, если уже запрограммирован модуль, который к нему обращается. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (нисходящем) порядке. При этом первым тестируется головной модуль программы, который представляет всю тестируемую программу и поэтому тести-

руется при «естественном» состоянии информационной среды, при котором начинает выполняться эта программа. При этом те модули, к которым может обращаться головной, заменяются их имитаторами (так называемыми заглушками). Каждый *имитатор модуля* представляется весьма простым программным фрагментом, который, в основном, сигнализирует о самом факте обращения к имитируемому модулю, производит необходимую для правильной работы программы обработку значений его входных параметров (иногда с их распечаткой) и выдает, если это необходимо, заранее запасенный подходящий результат. После завершения тестирования и отладки головного и любого последующего модуля производится переход к тестированию одного из модулей, которые в данный момент представлены имитаторами, если таковые имеются. Для этого имитатор выбранного для тестирования модуля заменяется самим этим модулем и, кроме того, добавляются имитаторы тех модулей, к которым может обращаться выбранный для тестирования модуль. При этом каждый такой модуль будет тестироваться при «естественных» состояниях информационной среды, возникающих к моменту обращения к этому модулю при выполнении тестируемой программы. Таким образом, большой объем «отладочного» программирования при восходящем тестировании заменяется программированием достаточно простых имитаторов используемых в программе модулей. Кроме того, имитаторы удобно использовать для того, чтобы подыгрывать процессу подбора тестов путем задания нужных результатов, выдаваемых имитаторами. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т.е. ликвидируется весьма неприятный источник просчетов при программировании модулей.

Некоторым недостатком нисходящей разработки, приводящим к определенным затруднениям при ее применении, является необходимость абстрагироваться от базовых возможностей используемого языка программирования, выдумывая абстрактные операции, которые позже нужно будет реализовать с помощью выделенных в программе модулей. Однако способность к таким абстракциям представляется необходимым условием разработки больших программных средств, поэтому ее нужно развивать.

Особенностью рассмотренных методов восходящей и нисходящей разработок (которые мы будем называть *классическими*) является требование, чтобы модульная структура программы была разработана до начала программирования (кодирования) модулей.

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модулей. Разработка программы при конструктивном подходе начинается с программирования головного модуля, исходя из спецификации программы в целом. При этом спецификация программы принимается в качестве спецификации ее головного модуля, который полностью берет на себя ответственность за выполнение функций программы. В процессе программирования головного модуля, в случае, если эта программа достаточно большая, выделяются подзадачи (внутренние функции), в терминах которых программируется головной модуль. Это

означает, что для каждой выделяемой подзадачи (функции) создается спецификация реализующего ее фрагмента программы, который в дальнейшем может быть представлен некоторым поддеревом модулей. Важно заметить, что здесь также ответственность за выполнение выделенной функции несет головной (может быть, и единственный) модуль этого поддерева, так что спецификация выделенной функции является одновременно и спецификацией головного модуля этого поддерева. В головном модуле программы для обращения к выделенной функции строится обращение к головному модулю указанного поддерева в соответствии с созданной его спецификацией.

Таким образом, на первом шаге разработки программы (при программировании ее головного модуля) формируется верхняя начальная часть дерева, например, такая, которая показана на рисунок 6.

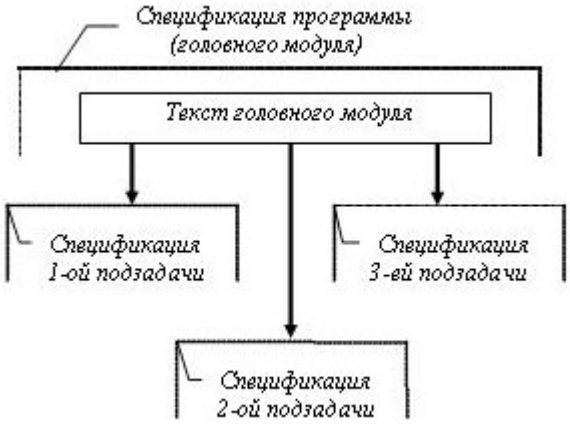

Рисунок 6 – Первый шаг формирования модульной структуры программы при конструктивном подходе

Аналогичные действия производятся при программировании любого другого модуля, который выбирается из текущего состояния дерева программы из числа специфицированных, но пока еще не запрограммированных модулей. В результате этого производится очередное доформирование дерева программы, например, такое, которое показано на рисунке 7.

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Но при этом ставится существенно другая цель разработки: повышение уровня используемого языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяются типичные функции, каждая из которых может использоваться при решении разных задач в этой области, и специфицируются, а затем и программируются отдельные программные модули, выполняющие эти функции. Так как процесс выделения таких функций связан с накоплением и обобщением опыта решения задач в заданной предметной области, то обычно сначала выделяются и реализуются отдельными модулями более простые функции, а затем постепенно появляются модули, использующие ранее выделенные функции.

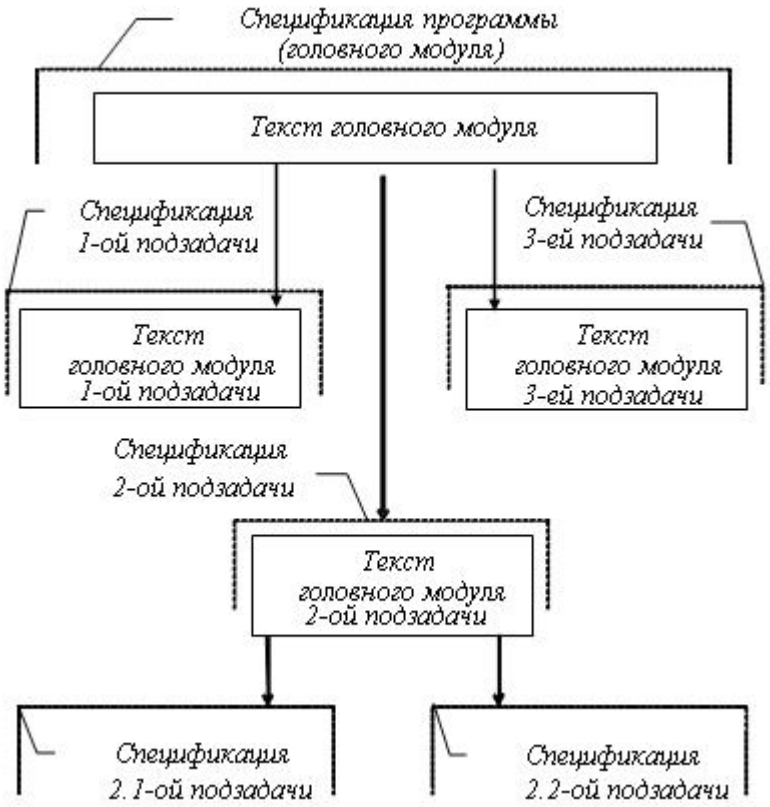

Рисунок 7 – Второй шаг формирования модульной структуры программы при конструктивном подходе

Такой набор модулей создается в расчете на то, что при разработке той или иной программы заданной предметной области в рамках конструктивного подхода могут оказаться приемлемыми некоторые из этих модулей. Это позволяет существенно сократить трудозатраты на разработку конкретной программы путем подключения к ней заранее заготовленных и проверенных на практике модульных структур нижнего уровня. Так как такие структуры могут многократно использоваться в разных конкретных программах, то архитектурный подход может рассматриваться как путь борьбы с дублированием в программировании. В связи с этим программные модули, создаваемые в рамках архитектурного подхода, обычно параметризуются для того, чтобы усилить применимость таких модулей путем настройки их на параметры.

Все эти методы имеют еще различные разновидности в зависимости от того, в какой последовательности обходятся узлы (модули) древовидной структуры программы в процессе ее разработки. Это можно делать, например, по слоям (разрабатывая все модули одного уровня, прежде чем переходить к следующему уровню). При нисходящей разработке дерево можно обходить также в лексикографическом порядке (сверху вниз, слева направо). Возможны и другие варианты обхода дерева. Так, при конструктивной реализации для обхода дерева программы целесообразно следовать идеям Фуксмана, которые он использовал в предложенном им методе вертикального слоения. Сущность такого обхода заключается в следующем. В рамках конструктивного подхода сначала реализуются только те модули, которые необходимы для самого простейшего

варианта программы, которая может нормально выполняться только для весьма ограниченного множества наборов входных данных, но для таких данных эта задача будет решаться до конца. Вместо других модулей, на которые в такой программе имеются ссылки, в эту программу вставляются лишь их имитаторы, обеспечивающие, в основном, сигнализацию о выходе за пределы этого частного случая. Затем к этой программе добавляются реализации некоторых других модулей (в частности, вместо некоторых из имеющихся имитаторов), обеспечивающих нормальное выполнение для некоторых других наборов входных данных. И этот процесс продолжается поэтапно до полной реализации требуемой программы. Таким образом, обход дерева программы производится с целью кратчайшим путем реализовать тот или иной вариант (сначала самый простейший) нормально действующей программы. В связи с этим такая разновидность конструктивной реализации получила название метода *целенаправленной конструктивной реализации*. Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика. Поэтому этот метод является весьма привлекательным.

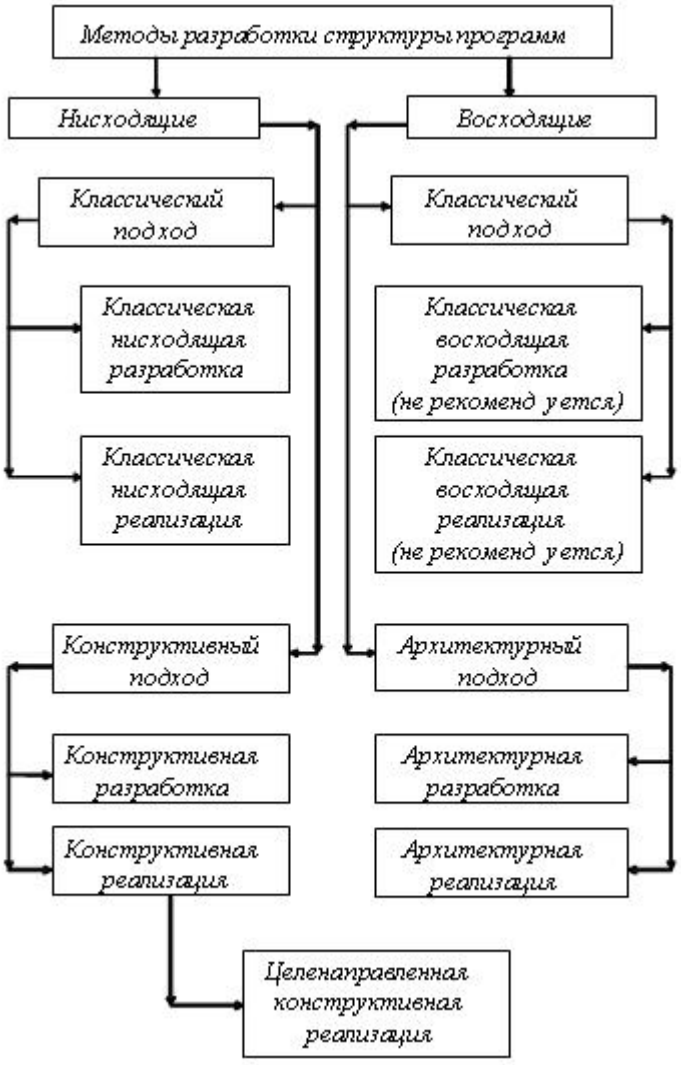

Рисунок 8 – Классификация методов разработки структуры программ

Подводя итог сказанному, на рисунке 8 представлена общая классификация рассмотренных методов разработки структуры программы.

3.3 Декомпозиция программы по SADT технологии

Сущность структурного подхода к разработке ПС заключается в ее декомпозиции (разбиении) на автоматизируемые функции: система разбивается на функциональные подсистемы, которые в свою очередь делятся на подфункции, подразделяемые на задачи и так далее. Процесс разбиения продолжается вплоть до конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны. При разработке ПС используется методы проектирования, описанные выше.

Все наиболее распространенные методологии структурного подхода базируются на ряде общих принципов. В качестве двух базовых принципов используются следующие:

принцип "разделяй и властвуй" – принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;

принцип иерархического упорядочивания – принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

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

–принцип абстрагирования – заключается в выделении существенных аспектов системы и отвлечения от несущественных;

–принцип формализации – заключается в необходимости строгого методического подхода к решению проблемы;

–принцип непротиворечивости – заключается в обоснованности и согласованности элементов;

–принцип структурирования данных – заключается в том, что данные должны быть структурированы и иерархически организованы. В структурном анализе используются в основном две группы средств, иллюстрирующих функции, выполняемые системой и отношения между данными. Каждой группе средств соответствуют определенные виды моделей (диаграмм), наиболее распространенными среди которых являются следующие:

SADT (Structured Analysis and Design Technique) модели и соответствующие функциональные диаграммы;

DFD (Data Flow Diagrams) диаграммы потоков данных;

ERD (Entity-Relationship Diagrams) диаграммы "сущность-связь".

Перечисленные модели в совокупности дают полное описание ПС независимо от того, является ли она существующей или вновь разрабатываемой. Со-

став диаграмм в каждом конкретном случае зависит от необходимой полноты описания системы.

На стадии проектирования ПС модели расширяются, уточняются и дополняются диаграммами, отражающими структуру программного обеспечения: архитектуру ПО, структурные схемы программ и диаграммы экранных форм.

Методология SADT разработана Дугласом Россом и получила дальнейшее развитие в работе. На ее основе разработана, в частности, известная методология IDEF0 (Icam DEFinition), которая является основной частью программы ICAM (Интеграция компьютерных и промышленных технологий), проводимой по инициативе ВВС США.

Методология SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Основные элементы этой методологии основываются на следующих концепциях:

графическое представление блочного моделирования. Графика блоков и дуг SADT-диаграммы отображает функцию в виде блока, а интерфейсы входа/выхода представляются дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описываются посредством интерфейсных дуг, выражающих "ограничения", которые в свою очередь определяют, когда и каким образом функции выполняются и управляются;

строгость и точность. Выполнение правил SADT требует достаточной строгости и точности, не накладывая в то же время чрезмерных ограничений на действия аналитика. Правила SADT включают:

ограничение количества блоков на каждом уровне декомпозиции (правило 3-6 блоков);

связность диаграмм (номера блоков);

уникальность меток и наименований (отсутствие повторяющихся имен);

синтаксические правила для графики (блоков и дуг);

разделение входов и управлений (правило определения роли данных).

отделение организации от функции, т.е. исключение влияния организационной структуры на функциональную модель.

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

Результатом применения методологии SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария, имеющих ссылки друг на друга. Диаграммы - главные компоненты модели, все функции ПС и интерфейсы на них представлены как блоки и дуги. Место соединения дуги с блоком

определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как информация, которая подвергается обработке, показана с левой стороны блока, а результаты выхода показаны с правой стороны. Механизм (человек или автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу (рисунок 9).

Одной из наиболее важных особенностей методологии SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель.

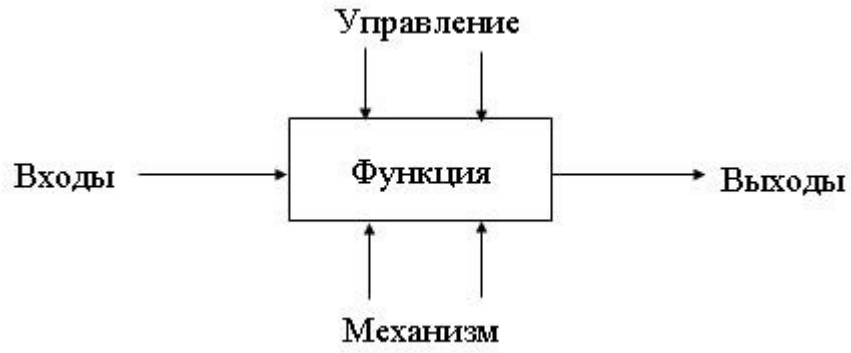

Рисунок 9 – Функциональный блок и интерфейсные дуги

На рисунке 10, где приведены четыре диаграммы и их взаимосвязи, показана структура SADT-модели. Каждый компонент модели может быть декомпозирован на другой диаграмме. Каждая диаграмма иллюстрирует "внутреннее строение" блока на родительской диаграмме.

3.3.1 Иерархия диаграмм

Построение SADT-модели начинается с представления всей системы в виде простейшей компоненты - одного блока и дуг, изображающих интерфейсы с функциями вне системы. Поскольку единственный блок представляет всю систему как единое целое, имя, указанное в блоке, является общим. Это верно и для интерфейсных дуг - они также представляют полный набор внешних интерфейсов системы в целом.

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

Во всех случаях каждая подфункция может содержать только те элементы, которые входят в исходную функцию. Кроме того, модель не может опустить какие-либо элементы, т.е., как уже отмечалось, родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить, и из него не может быть ничего удалено.

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

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

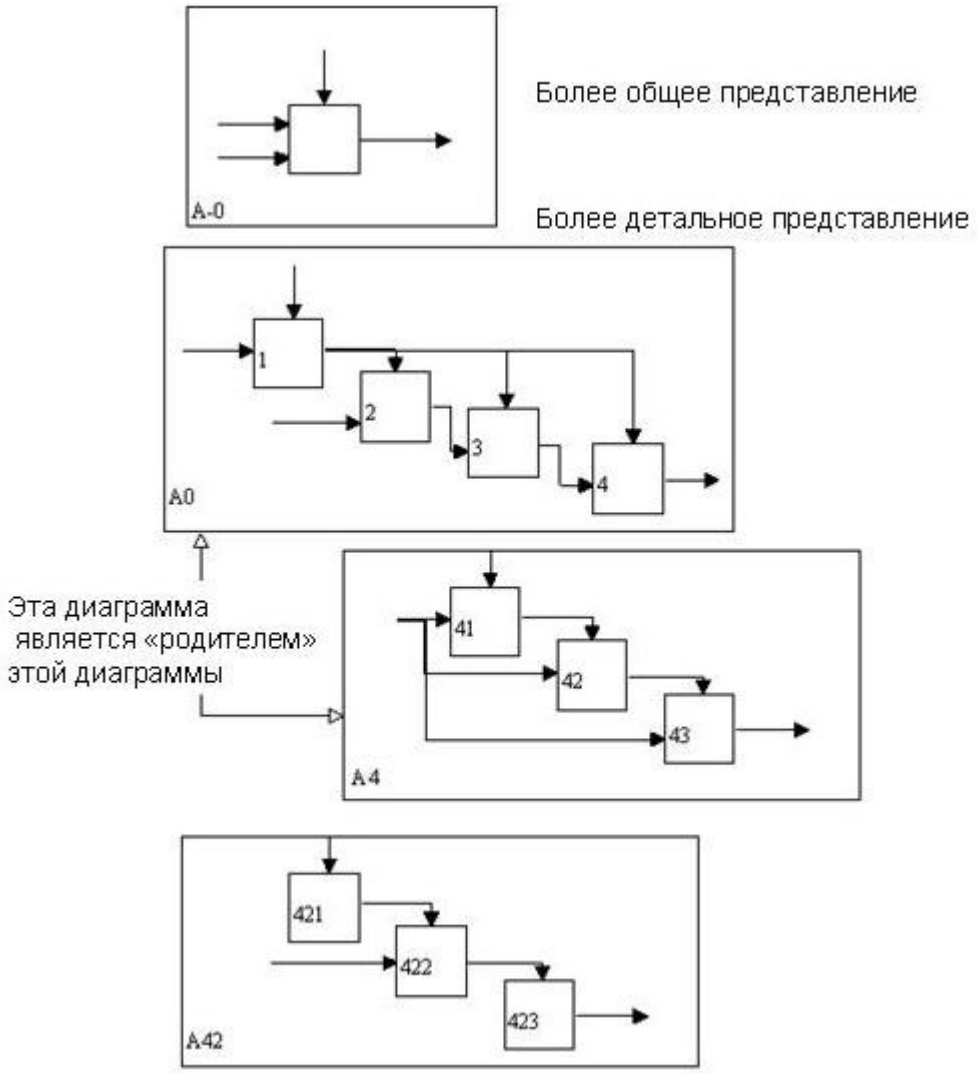

Рисунок 10 – Структура SADT-модели. Декомпозиция диаграмм

На рисунках 11 – 12 представлены различные варианты выполнения функций и соединения дуг с блоками.

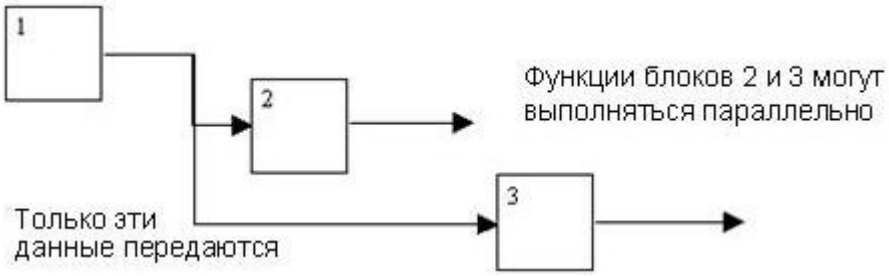

Рисунок 11 – Одновременное выполнение

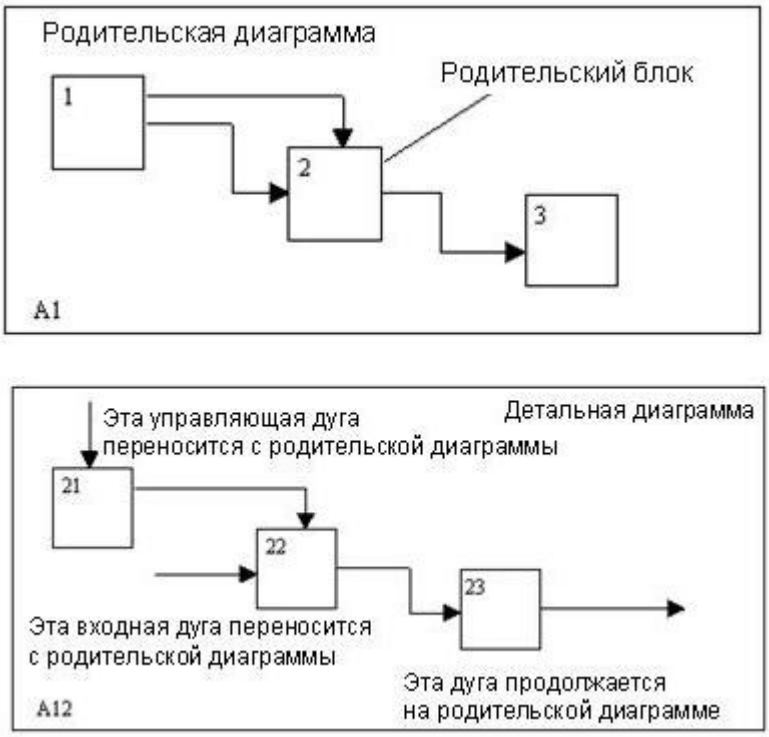

Рисунок 12 – Соответствие должно быть полным и непротиворечивым

Некоторые дуги присоединены к блокам диаграммы обоими концами, у других же один конец остается не присоединенным. Не присоединенные дуги соответствуют входам, управлениям и выходам родительского блока. Источник или получатель этих пограничных дуг может быть обнаружен только на родительской диаграмме. Не присоединенные концы должны соответствовать дугам на исходной диаграмме. Все граничные дуги должны продолжаться на родительской диаграмме, чтобы она была полной и непротиворечивой.

На SADT-диаграммах не указаны явно ни последовательность, ни время. Обратные связи, итерации, продолжающиеся процессы и перекрывающиеся (по времени) функции могут быть изображены с помощью дуг. Обратные связи могут выступать в виде комментариев, замечаний, исправлений и т.д. (рисунок 13).

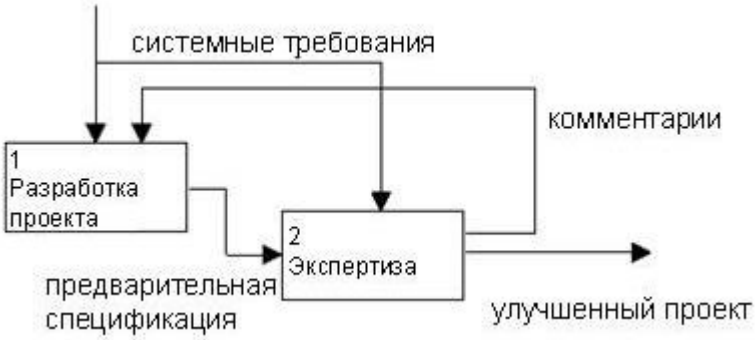

Рисунок 13 – Пример обратной связи

Как было отмечено, механизмы (дуги с нижней стороны) показывают средства, с помощью которых осуществляется выполнение функций. Механизм может быть человеком, компьютером или любым другим устройством, которое помогает выполнять данную функцию (рисунок 14).

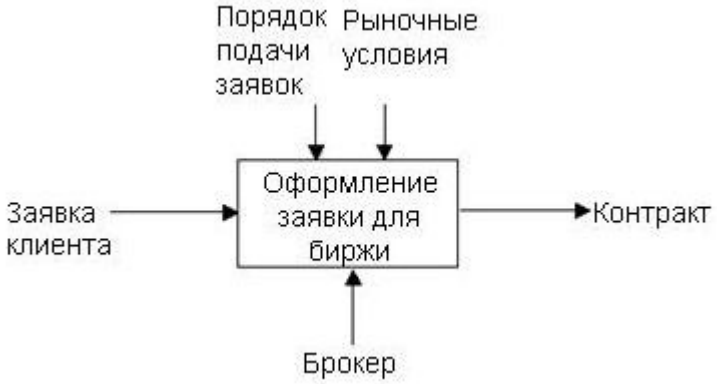

Рисунок 14 – Пример механизма

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

Для того, чтобы указать положение любой диаграммы или блока в иерархии, используются номера диаграмм. Например, A21 является диаграммой, которая детализирует блок 1 на диаграмме A2. Аналогично, A2 детализирует блок 2 на диаграмме A0, которая является самой верхней диаграммой модели. На рисунке 15 показано типичное дерево диаграмм.

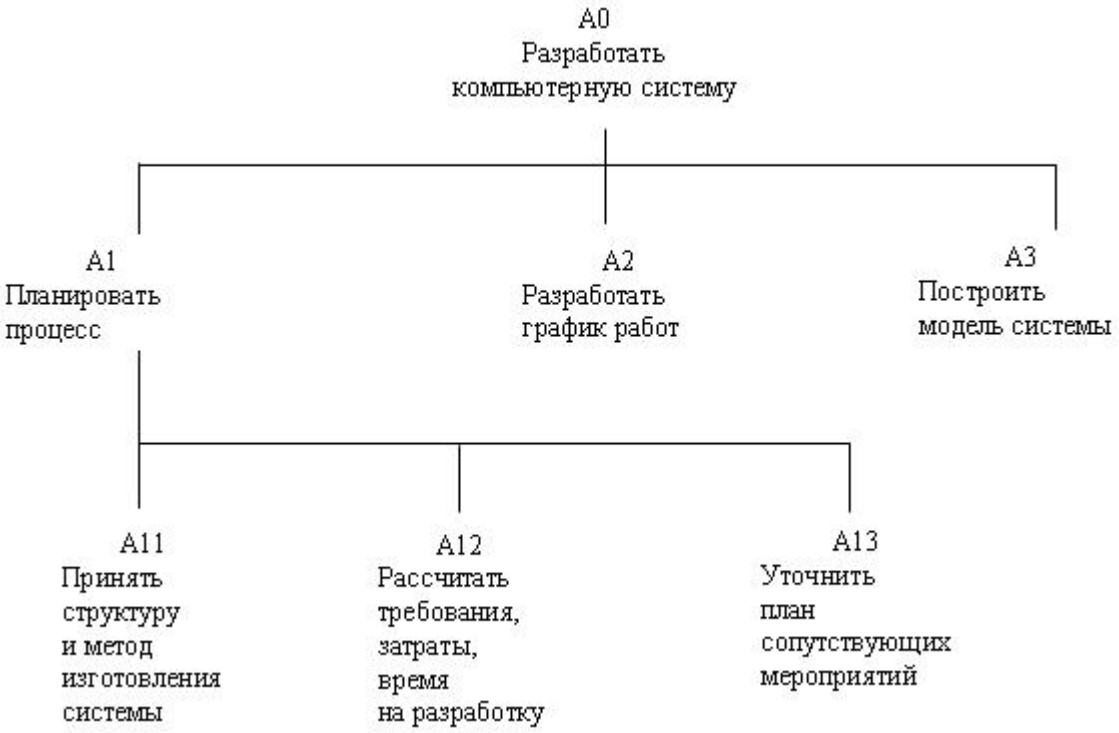

Рисунок 15 – Иерархия диаграмм

3.4 Порядок разработки и описание программного модуля

При разработке программного модуля целесообразно придерживаться следующего порядка:

изучение и проверка спецификации модуля, выбор языка программирования;

выбор алгоритма и структуры данных;

программирование (кодирование) модуля;

шлифовка текста модуля;

проверка модуля;

компиляция модуля.

Первый шаг разработки программного модуля в значительной степени представляет собой смежный контроль структуры программы снизу: изучая спецификацию модуля, разработчик должен убедиться, что она ему понятна и достаточна для разработки этого модуля. В завершении этого шага выбирается язык программирования: хотя язык программирования может быть уже предопределен для всего ПС, все же в ряде случаев (если система программирования это допускает) может быть выбран другой язык, более подходящий для реализации данного модуля (например, язык ассемблера).

На втором шаге разработки программного модуля необходимо выяснить, не известны ли уже какие-либо алгоритмы для решения поставленной и или близкой к ней задачи. И если найдется подходящий алгоритм, то целесообразно им воспользоваться. Выбор подходящих структур данных, которые будут использоваться при выполнении модулем своих функций, в значительной степени

предопределяет логику и качественные показатели разрабатываемого модуля, поэтому его следует рассматривать как весьма ответственное решение.

На третьем шаге осуществляется построение текста модуля на выбранном языке программирования. Обилие всевозможных деталей, которые должны быть учтены при реализации функций, указанных в спецификации модуля, легко могут привести к созданию весьма запутанного текста, содержащего массу ошибок и неточностей. Искать ошибки в таком модуле и вносить в него требуемые изменения может оказаться весьма трудоемкой задачей. Поэтому весьма важно для построения текста модуля пользоваться технологически обоснованной и практически проверенной дисциплиной программирования. Впервые на это обратил внимание Дейкстра, сформулировав и обосновав основные принципы структурного программирования. На этих принципах базируются многие дисциплины программирования, широко применяемые на практике.

Следующий шаг разработки модуля связан с приведением текста модуля к завершенному виду в соответствии со спецификацией качества ПС. При программировании модуля разработчик основное внимание уделяет правильности реализации функций модуля, оставляя недоработанными комментарии и допуская некоторые нарушения требований к стилю программы. При шлифовке текста модуля он должен отредактировать имеющиеся в тексте комментарии и, возможно, включить в него дополнительные комментарии с целью обеспечить требуемое качества. С этой же целью производится редактирование текста программы для выполнения стилистических требований.

Шаг проверки модуля представляет собой ручную проверку внутренней логики модуля до начала его отладки (использующей выполнение его на компьютере), реализует общий принцип, сформулированный для обсуждаемой технологии программирования, о необходимости контроля принимаемых решений на каждом этапе разработки ПС.

И, наконец, последний шаг разработки модуля означает завершение проверки модуля (с помощью компилятора) и переход к процессу отладки модуля.

При программировании модуля следует иметь в виду, что программа должна быть понятной не только компьютеру, но и человеку: и разработчик модуля, и лица, проверяющие модуль, и тестовики, готовящие тесты для отладки модуля, и сопроводители ПС, осуществляющие требуемые изменения модуля, вынуждены будут многократно разбирать логику работы модуля. В связи с этим Дейкстра и предложил строить программу как композицию из нескольких типов управляющих конструкций (структур), которые позволяют сильно повысить понимаемость логики работы программы. Программирование с использованием только таких конструкций назвали *структурным*.

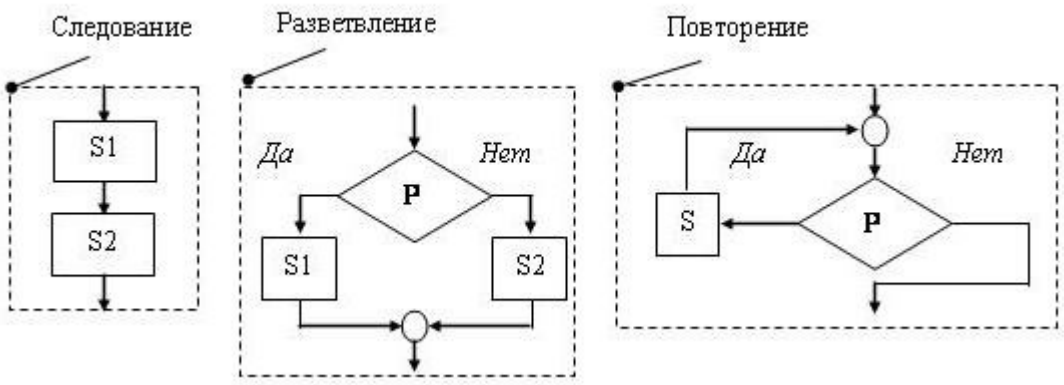

Рисунок 16 – Основные управляющие конструкции структурного программирования

Основными конструкциями структурного программирования являются: следование, разветвление и повторение (рисунок 16). Компонентами этих конструкций являются обобщенные операторы (узлы обработки) S, S1, S2 и условие (предикат) P. В качестве обобщенного оператора может быть либо простой оператор используемого языка программирования (операторы присваивания, ввода, вывода, обращения к процедуре), либо фрагмент программы, являющийся композицией основных управляющих конструкций структурного программирования. Существенно, что каждая из этих конструкций имеет по управлению только один вход и один выход. Тем самым, и обобщенный оператор имеет только один вход и один выход.

4 Характеристика программного модуля. Потоки данных и процессы

4.1 Характеристики программного модуля

Приступая к разработке каждой программы ПС, следует иметь в виду, что она, как правило, является большой системой, поэтому необходимо принять меры для ее упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями. А сам такой метод разработки программ называют *модульным* программированием.

Программный модуль – это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен с другими модулями программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю. Таким образом, программный модуль может рассматриваться и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании (т.е. как средство накопления и многократного использования программистских знаний).

Модульное программирование призвано, в процессе разработки программ, борьбы со сложностью, обеспечивает независимость компонент системы, и использование иерархических структур.

Для избежание сложностей формулируются определенные требования, которым должен удовлетворять программный модуль, т.е. выявляются основные характеристики «хорошего» программного модуля. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Так, Хольт предложил следующие два общих таких критерия:

- хороший модуль снаружи проще, чем внутри;
- хороший модуль проще использовать, чем построить.

Майерс предлагает для оценки приемлемости программного модуля использовать более конструктивные его характеристики:

- размер модуля,
- прочность (связность) модуля,
- сцепление с другими модулями,
- рутинность модуля (независимость от предыстории обращений к нему).

4.2 Характеристики программного модуля по Майерсу

Размер модуля измеряется числом содержащихся в нем операторов или строк.

Сцепление модуля – это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления Майерс предлагает упорядоченный набор из шести видов сцепления модулей.

Рутинность модуля – это его независимость от предыстории обращений к нему. Модуль будем называть *рутинным*, если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль будем называть *зависящим от предыстории*, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, изменяемого в результате предыдущих обращений к нему. Существуют некоторые рекомендации по использованию зависящих от предыстории модулей:

- всегда следует использовать рутинный модуль, если это не приводит к плохим (не рекомендуемым) сцеплениям модулей;
- зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;
- в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение (эффект выполнения) данного модуля при разных последующих обращениях к нему.

В связи с последней рекомендацией может быть полезным определение внешнего представления (ориентированного на информирование человека) состояний зависящего от предыстории модуля. В этом случае эффект выполнения каждой функции (операции), реализуемой этим модулем, следует описывать в терминах этого внешнего представления, что существенно упростит прогнозирование поведения данного модуля.

Прочность(связность) модуля – это мера его внутренних связей. Чем выше прочность модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести. Для оценки степени прочности модуля Майерс предлагает упорядоченный по степени прочности набор из семи классов модулей.

Функционально прочный модуль – это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется для использования.

Информационно прочный модуль – это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Такой класс следует рассматривать как класс программных модулей с высшей степенью прочности. Информационно прочный модуль может реализовывать, например, абстрактный тип данных

4.3 Характеристики программного модуля по методологии SADT

Методология SADT предлагает несколько другую классификацию связности. Различают по крайней мере семь типов связывания (таблица 1):

Таблица 1 – Типы связанности

Значимость	Тип связности	Для функций	Для данных
0	Случайная	Случайная	Случайная
1	Логическая	Функции одного и того же множества или типа (например, "редактировать все входы")	Данные одного и того же множества или типа
2	Временная	Функции одного и того же периода времени (например, "операции инициализации")	Данные, используемые в каком-либо временном интервале
3	Процедурная	Функции, работающие в одной и той же фазе или итерации (например, "первый проход компилятора")	Данные, используемые во время одной и той же фазы или итерации
4	Коммуникационная	Функции, использующие одни и те же данные	Данные, на которые воздействует одна и та же деятельность
5	Последовательная	Функции, выполняющие последовательные преобразования одних и тех же данных	Данные, преобразуемые последовательными функциями
6	Функциональная	Функции, объединяемые для выполнения одной функции	Данные, связанные с одной функцией

Ниже каждый тип связи кратко определен и проиллюстрирован с помощью типичного примера из SADT.

(0) Тип случайной связности: наименее желательный.

Случайная связность возникает, когда конкретная связь между функциями мала или полностью отсутствует. Это относится к ситуации, когда имена данных на SADT-дугах в одной диаграмме имеют малую связь друг с другом. Крайний вариант этого случая показан на рисунке 17.

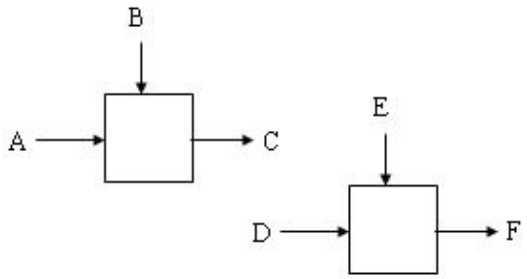

Рисунок. 17 – Случайная связность

(1) Тип логической связности. Логическое связывание происходит тогда, когда данные и функции собираются вместе вследствие того, что они попадают в общий класс или набор элементов, но необходимых функциональных отношений между ними не обнаруживается.

(2) Тип временной связности. Связанные по времени элементы возникают вследствие того, что они представляют функции, связанные во времени, когда данные используются одновременно или функции включаются параллельно, а не последовательно.

(3) Тип процедурной связности. Процедурно-связанные элементы появляются сгруппированными вместе вследствие того, что они выполняются в течение одной и той же части цикла или процесса. Пример процедурно-связанной диаграммы приведен на рисунке 18.

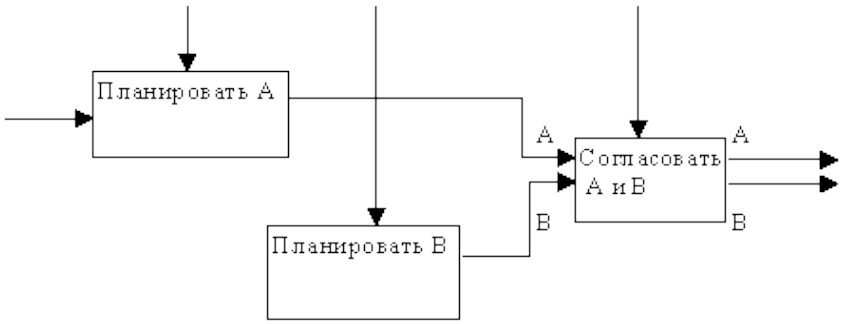

Рисунок 18 – Процедурная связность

(4) Тип коммуникационной связности. Диаграммы демонстрируют коммуникационные связи, когда блоки группируются вследствие того, что они используют одни и те же входные данные и/или производят одни и те же выходные данные (рисунок 19).

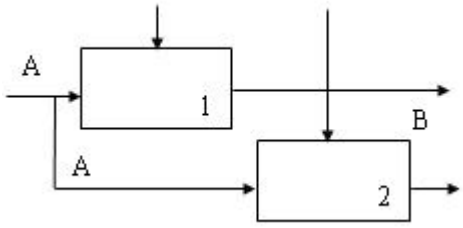

Рисунок 19 – Коммуникационная связность

*(5) Тип последовательной связности***.** На диаграммах, имеющих последовательные связи, выход одной функции служит входными данными для следующей функции. Связь между элементами на диаграмме является более тесной, чем на рассмотренных выше уровнях связок, поскольку моделируются причинно-следственные зависимости (рисунок 20).

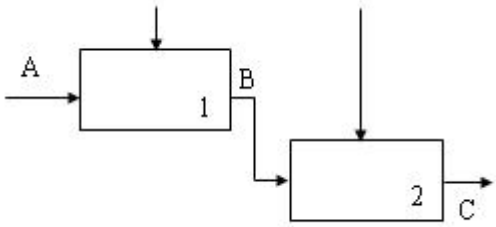

Рисунок 20 – Последовательная связность

*(6) Тип функциональной связности***.** Диаграмма отражает полную функциональную связность, при наличии полной зависимости одной функции от другой. Диаграмма, которая является чисто функциональной, не содержит чужеродных элементов, относящихся к последовательному или более слабому типу связности. Одним из способов определения функционально-связанных диаграмм является рассмотрение двух блоков, связанных через управляющие дуги, как показано на рисунке 21.

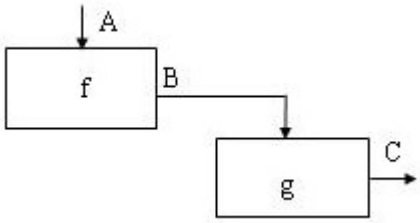

Рисунок 21 – Функциональная связность

В математических терминах необходимое условие для простейшего типа функциональной связности, показанной на рисунке 21, имеет следующий вид:

$$C = g(B) = g(f(A))$$

Ниже в таблице представлены все типы связей, рассмотренные выше. Важно отметить, что уровни 4-6 устанавливают типы связностей, которые разработчики считают важнейшими для получения диаграмм хорошего качества.

4.4 Потоки данных и процессы

В третьей части были описаны понятия декомпозиция задачи и методы проектирования. Рассмотрим некоторые аспекты проектирования более подробно. При декомпозиции задачи необходимо помнить и четко представлять каким образом будет происходить взаимодействие между частями задачи, т.е. необходимо определить потоки данных.

Потоки данных определяются согласно следующим рекомендациям:

- каждому источнику данных соответствует один входной поток;
- если имеется совокупный набор данных, полученный из нескольких источников, эти наборы распределяются по группам, обрабатываемых совместно потоков данных;
- если не все потоки данных подвергаются обработки одновременно, процесс обработки делиться на этапы, в каждом из которых участвуют группы совместно обрабатываемых потоков, кроме того, должны существовать внутренние потоки данных, связывающие последовательные этапы;
- для каждого этапа в системе выделяются основной входной поток, для выдачи результатов обработки, и дополнительный, для выдачи оперативных отчетов, сообщений об ошибках и т.д.

После того как определены потоки данных, необходимо определить процессы, оперирующие этими потоками. Здесь так же существует ряд правил:

- если потоки обрабатываются раздельно, то для каждого из них требуется отдельный процесс;
- если некоторые системные функции должны выполняться в разное время или чаще чем другие, они реализуются в виде отдельных процессов;
- если некоторые из промежуточных потоков данных необходимо сохранять с целью их последующего использования, должны быть предусмотрены процессы, осуществляющие их запоминание.

Поток данных определяет информацию, передаваемую через некоторое соединение от источника к приемнику. Реальный поток данных может быть информацией, передаваемой по кабелю между двумя устройствами, пересылаемыми по почте письмами, магнитными лентами или дискетами, переносимыми с одного компьютера на другой и т.д.

Рассмотрим пример построения диаграмма потоков данных.

В качестве предметной области используется описание работы видеобиблиотеки, которая получает запросы на фильмы от клиентов и ленты, возвращаемые клиентами. Запросы рассматриваются администрацией видеобиблиотеки с использованием информации о клиентах, фильмах и лентах. При этом проверяется и обновляется список арендованных лент, а также проверяются записи о членстве в библиотеке. Администрация контролирует также возвраты лент, используя информацию о фильмах, лентах и список арендованных лент, который обновляется. Обработка запросов на фильмы и возвратов лент включает следующие действия: если клиент не является членом библиотеки, он не имеет права

на аренду. Если требуемый фильм имеется в наличии, администрация информирует клиента об арендной плате. Однако, если клиент просрочил срок возврата имеющихся у него лент, ему не разрешается брать новые фильмы. Когда лента возвращается, администрация рассчитывает арендную плату плюс пени за несвоевременный возврат.

Видео-библиотека получает новые ленты от своих поставщиков. Когда новые ленты поступают в библиотеку, необходимая информация о них фиксируется. Информация о членстве в библиотеке содержится отдельно от записей об аренде лент.

Администрация библиотеки регулярно готовит отчеты за определенный период времени о членах библиотеки, поставщиках лент, выдаче определенных лент и лентах, приобретенных библиотекой.

При решении данной задачи необходимо провести: анализ, глобальное проектирование (проектирование архитектуры системы), детальное проектирование и реализация (программирование).

На фазе анализа строится модель среды (Environmental Model). Построение модели среды включает:

–анализ поведения системы (определение назначения ИС, построение начальной контекстной диаграммы потоков данных (DFD) и формирование матрицы списка событий (ELM), построение контекстных диаграмм);

–анализ данных (определение состава потоков данных и построение диаграмм структур данных (DSD), конструирование глобальной модели данных в виде ER-диаграммы).

Назначение ИС определяет соглашение между проектировщиками и заказчиками относительно назначения будущей ИС, общее описание ИС для самих проектировщиков и границы ИС. Назначение фиксируется как текстовый комментарий в "нулевом" процессе контекстной диаграммы.

Например, в данном случае назначение ИС формулируется следующим образом: ведение базы данных о членах библиотеки, фильмах, аренде и поставщиках. При этом руководство библиотеки должно иметь возможность получать различные виды отчетов для выполнения своих задач.

Перед построением контекстной DFD необходимо проанализировать внешние события (внешние объекты), оказывающие влияние на функционирование библиотеки. Эти объекты взаимодействуют с ИС путем информационного обмена с ней.

Из описания предметной области следует, что в процессе работы библиотеки участвуют следующие группы людей: клиенты, поставщики и руководство. Эти группы являются внешними объектами. Они не только взаимодействуют с системой, но также определяют ее границы и изображаются на начальной контекстной DFD как терминаторы (внешние сущности).

Начальная контекстная диаграмма изображена на рисунке 22. В отличие от нотации Gane/Sarson внешние сущности обозначаются обычными прямоугольниками, а процессы - окружностями.

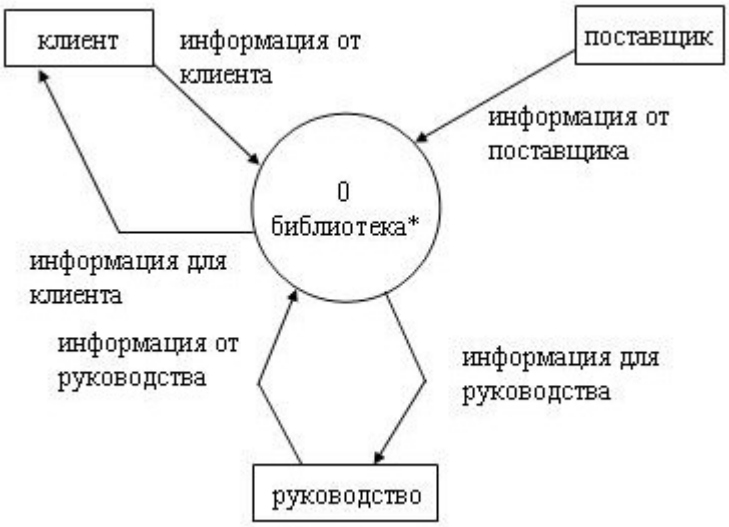

Рисунок 22- Начальная контекстная диаграмма

Список событий строится в виде матрицы (ELM) и описывает различные действия внешних сущностей и реакцию ИС на них. Эти действия представляют собой внешние события, воздействующие на библиотеку. Различают следующие типы событий:

Аббревиатура	**Тип**
NC	Нормальное управление
ND	Нормальные данные
NCD	Нормальное управление/данные
TC	Временное управление
TD	Временные данные
TCD	Временное управление/данные

Все действия помечаются как нормальные данные. Эти данные являются событиями, которые ИС воспринимает непосредственно, например, изменение адреса клиента, которое должно быть сразу зарегистрировано. Они появляются в DFD в качестве содержимого потоков данных.

Матрица списка событий имеет следующий вид:

№	**Описание**	**Тип**	**Реакция**
1	Клиент желает стать членом библиотеки	ND	Регистрация клиента в качестве члена библиотеки
2	Клиент сообщает об изменении адреса	ND	Регистрация измененного адреса клиента
3	Клиент запрашивает аренду фильма	ND	Рассмотрение запроса
4	Клиент возвращает фильм	ND	Регистрация возврата
5	Руководство предоставляет полномочия новому поставщику	ND	Регистрация поставщика
6	Поставщик сообщает об изменении адреса	ND	Регистрация измененного адреса поставщика

7	Поставщик направляет фильм в библиотеку	ND	Получение нового фильма
8	Руководство запрашивает новый отчет	ND	Формирование требуемого отчета для руководства

Для завершения анализа функционального аспекта поведения системы строится полная контекстная диаграмма, включающая диаграмму нулевого уровня. При этом процесс "библиотека" декомпозируется на 4 процесса, отражающие основные виды административной деятельности библиотеки. Существующие "абстрактные" потоки данных между терминаторами и процессами трансформируются в потоки, представляющие обмен данными на более конкретном уровне. Список событий показывает, какие потоки существуют на этом уровне: каждое событие из списка должно формировать некоторый поток (событие формирует входной поток, реакция - выходной поток). Один "абстрактный" поток может быть разделен на более чем один "конкретный" поток.

Потоки на диаграмме верхнего уровня	**Потоки на диаграмме нулевого уровня**
Информация от клиента	Данные о клиенте, Запрос об аренде
Информация для клиента	Членская карточка, Ответ на запрос об аренде
Информация от руководства	Запрос отчета о новых членах, Новый поставщик, Запрос отчета о поставщиках, Запрос отчета об аренде, Запрос отчета о фильмах
Информация для руководства	Отчет о новых членах, Отчет о поставщиках, Отчет об аренде, Отчет о фильмах
Информация от поставщика	Данные о поставщике, Новые фильмы

На приведенной DFD (рисунок 23) накопитель данных "библиотека" является глобальным или абстрактным представлением хранилища данных.

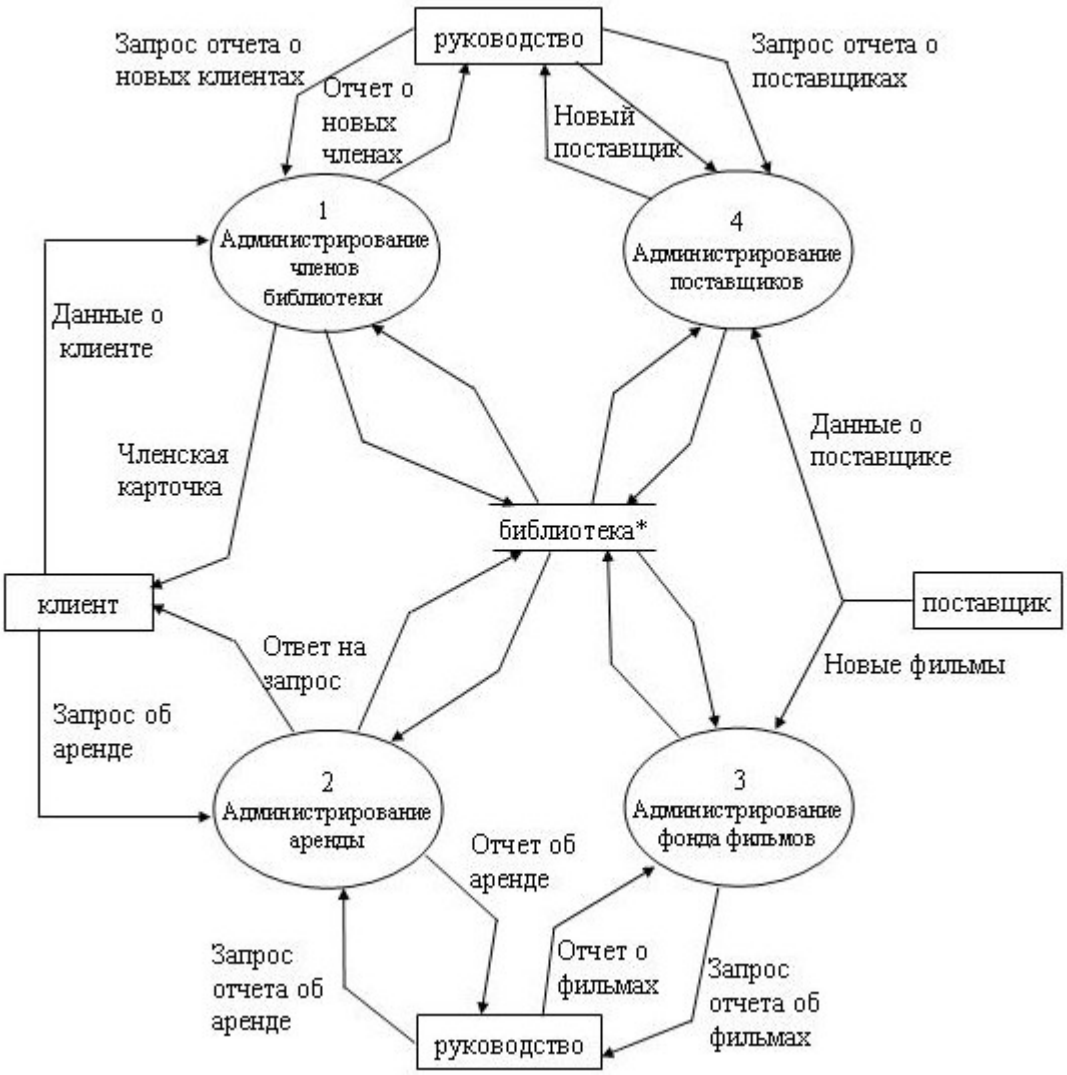

Рисунок 23 - Контекстная диаграмма

Анализ функционального аспекта поведения системы дает представление об обмене и преобразовании данных в системе. Взаимосвязь между "абстрактными" потоками данных и "конкретными" потоками данных на диаграмме нулевого уровня выражается в диаграммах структур данных (рисунок 24).

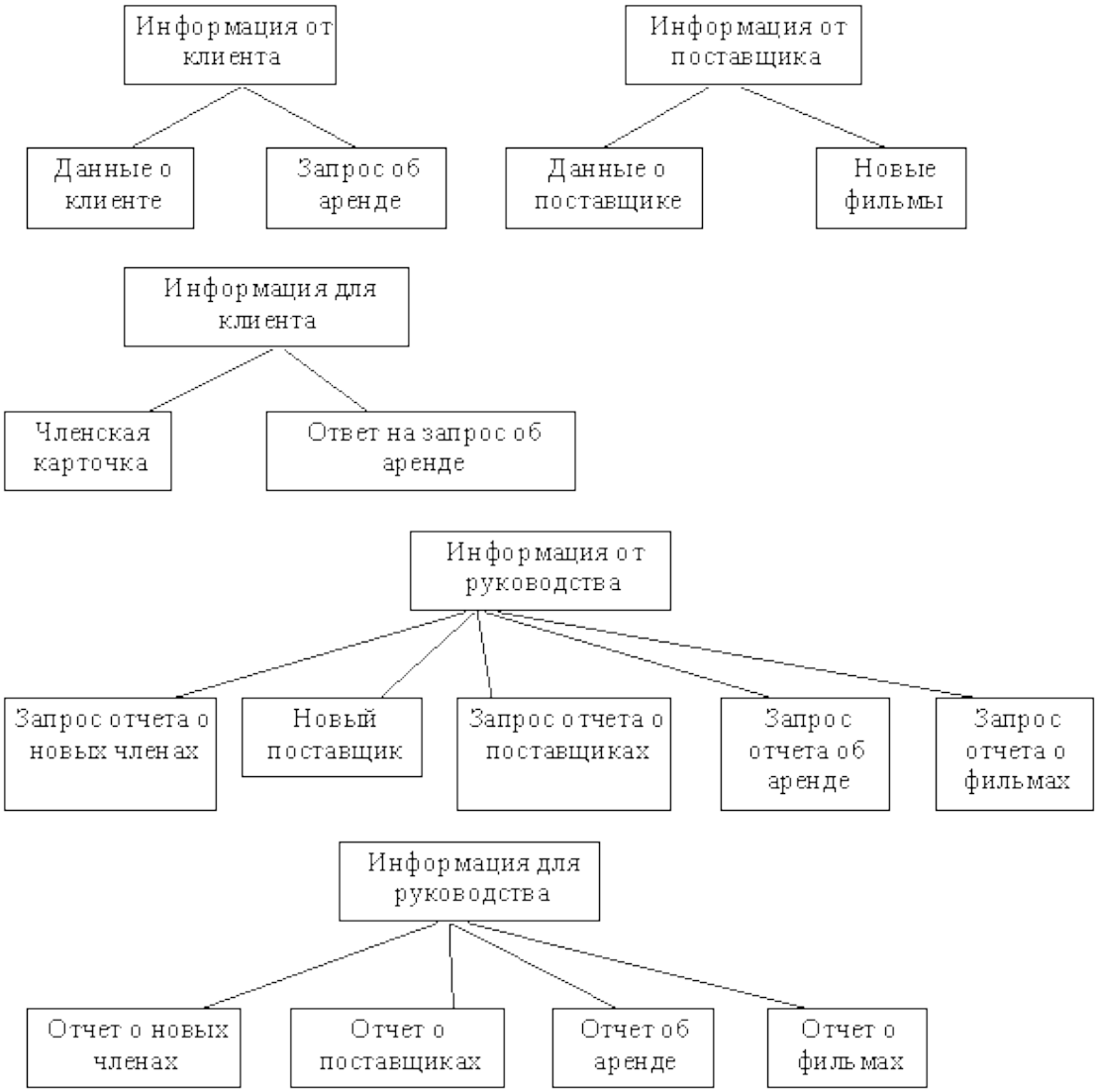

Рисунок 24 - Диаграмма структур данных

На фазе анализа строится глобальная модель данных, представляемая в виде диаграммы "сущность-связь" (рисунок 25).

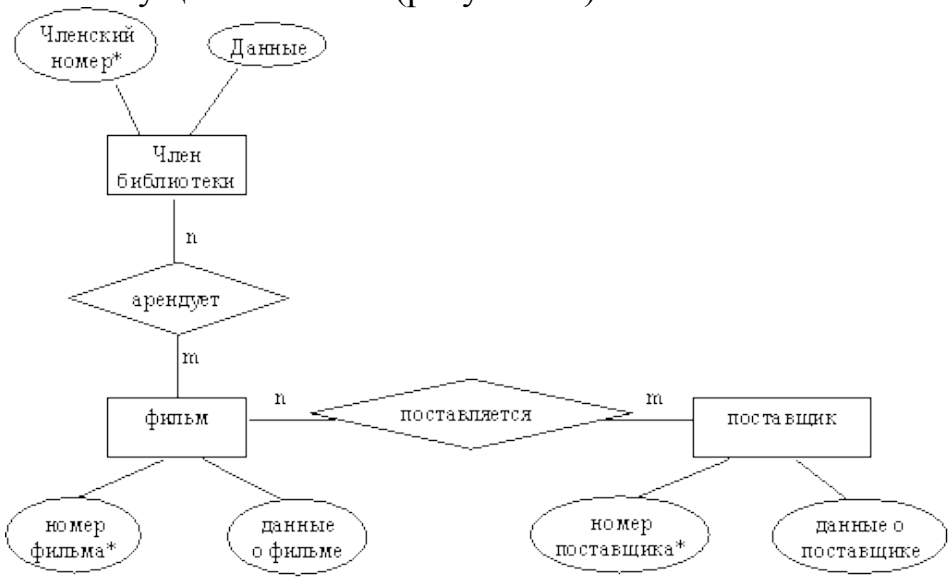

Рисунок 25 - Диаграмма "сущность-связь"

Между различными типами диаграмм существуют следующие взаимосвязи:

- ELM-DFD: события - входные потоки, реакции - выходные потоки
- DFD-DSD: потоки данных - структуры данных верхнего уровня
- DFD-ERD: накопители данных - ER-диаграммы
- DSD-ERD: структуры данных нижнего уровня - атрибуты сущностей

На фазе проектирования архитектуры строится предметная модель. Процесс построения предметной модели включает в себя:

- детальное описание функционирования системы;
- дальнейший анализ используемых данных и построение логической модели данных для последующего проектирования базы данных;
- определение структуры пользовательского интерфейса, спецификации форм и порядка их появления;
- уточнение диаграмм потоков данных и списка событий, выделение среди процессов нижнего уровня интерактивных и не интерактивных, определение для них мини-спецификаций.

Результатами проектирования архитектуры являются:

- модель процессов (диаграммы архитектуры системы (SAD) и мини-спецификации на структурированном языке);
- модель данных (ERD и подсхемы ERD);
- модель пользовательского интерфейса (классификация процессов на интерактивные и не интерактивные функции, диаграмма последовательности форм (FSD – Form Sequence Diagram), показывающая, какие формы появляются в приложении и в каком порядке. На FSD фиксируется набор и структура вызовов экранных форм. Диаграммы FSD образуют иерархию, на вершине которой находится главная форма приложения, реализующего подсистему. На втором уровне находятся формы, реализующие процессы нижнего уровня функциональной структуры, зафиксированной на диаграммах SAD.

На фазе детального проектирования строится модульная модель. Под модульной моделью понимается реальная модель проектируемой прикладной системы. Процесс ее построения включает в себя:

- уточнение модели базы данных для последующей генерации SQL-предложений;
- уточнение структуры пользовательского интерфейса;
- построение структурных схем, отражающих логику работы пользовательского интерфейса и модель бизнес-логики (Structure Charts Diagram – SCD) и привязка их к формам.

Результатами детального проектирования являются:

- модель процессов (структурные схемы интерактивных и не интерактивных функций);
- модель данных (определение в ERD всех необходимых параметров для приложений);
- модель пользовательского интерфейса (диаграмма последовательности форм (FSD), показывающая, какие формы появляются в приложении, и в каком

порядке, взаимосвязь между каждой формой и определенной структурной схемой, взаимосвязь между каждой формой и одной или более сущностями в ERD).

На фазе реализации строится реализационная модель. Процесс ее построения включает в себя:

–генерацию SQL-предложений, определяющих структуру целевой БД (таблицы, индексы, ограничения целостности);

–уточнение структурных схем (SCD) и диаграмм последовательности форм (FSD) с последующей генерацией кода приложений.

На основе анализа потоков данных и взаимодействия процессов с хранилищами данных осуществляется окончательное выделение подсистем (предварительное должно было быть сделано и зафиксировано на этапе формулировки требований в техническом задании). При выделении подсистем необходимо руководствоваться принципом функциональной связанности и принципом минимизации информационной зависимости. Необходимо учитывать, что на основании таких элементов подсистемы как процессы и данные на этапе разработки должно быть создано приложение, способное функционировать самостоятельно. С другой стороны при группировке процессов и данных в подсистемы необходимо учитывать требования к конфигурированию продукта, если они были сформулированы на этапе анализа.

5 Тестирование и отладка

5.1 Виды ошибок

Ошибки анализа. Связаны либо с неполным учетом ситуации, которые могут возникнуть, либо с неверным решением задачи. К первому случаю относятся, например, пренебрежение возможностью появления отрицательных значений переменных, малых и больших величин. Во втором случае обычно имеют место крупные и мелкие логические ошибки, из которых можно назвать:

1. Отсутствие заданий начальных значений переменных.
2. Неверные условия окончания цикла.
3. Неверную индексацию цикла.
4. Отсутствие задания условий инициирования цикла.
5. Неправильное указание ветви алгоритма для продолжения процесса решения задачи.

Ошибки общего характера. После того, как найден подходящий алгоритм решения задачи, на этапе программирования также могут появиться ошибки, независимо от выбранного языка. Такими ошибками могут быть:

–ошибки из-за недостаточного знания или понимания программистом языка программирования или самой машины

–ошибки, допущенные при программировании алгоритма, когда команды, используемые в программе, не обеспечивают последовательности событий, установленной алгоритмом.

Ошибки физического характера. Можно назвать несколько типов ошибок, вызываемых неверными действиями программиста:

–пропуск некоторых операторов.

–отсутствие необходимых данных.

–непредусмотренные данные.

–неверный формат данных.

Аккуратное и логичное написание программы, позволяет избежать ошибок или выявить их в случае возникновения. Следует избегать возможных программистских трюков, т.к. чем их больше, тем труднее отладка программы для самого автора, а кто-то другой этого сделать просто не сможет.

5.2 Отладка и тестирование

Отладка программного средства (ПС) – это деятельность, направленная на обнаружение и исправление ошибок в программе с использованием процессов выполнения его программ.

Тестирование ПС – это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ. Указанный набор данных называется *тестовым* или просто *тестом*.

Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПС ошибки, поиска места ошибки в программах и документации ПС и редактирования программ и документации с целью устранения обнаруженной ошибки. Другими словами:

Отладка = Тестирование + Поиск ошибок + Редактирование.

5.2.1 Тестирование

Тестирование не может доказать правильность ПС, в лучшем случае оно может продемонстрировать наличие в нем ошибки. Другими словами, нельзя гарантировать, что тестированием ПС практически выполнимым набором тестов можно установить наличие каждой имеющейся в ПС ошибки. Поэтому возникает две задачи. Первая задача: подготовить такой набор тестов и применить к ним ПС, чтобы обнаружить в нем по возможности большее число ошибок. Однако чем дольше продолжается процесс тестирования (и отладки в целом), тем большей становится стоимость ПС. Отсюда вторая задача: определить момент окончания отладки ПС (или отдельной его компоненты). Признаком возможности окончания отладки является полнота охвата пропущенными через ПС тестами (т.е. тестами, к которым применено ПС) множества различных ситуаций, возникающих при выполнении программ ПС, и относительно редкое проявление ошибок в ПС на последнем отрезке процесса тестирования. Последнее определяется в соответствии с требуемой степенью надежности ПС, указанной в спецификации его качества.

Для оптимизации набора тестов, т.е. для подготовки такого набора тестов, который позволял бы при заданном их числе (или при заданном интервале времени, отведенном на тестирование) обнаруживать большее число ошибок в ПС, необходимо, во-первых, заранее планировать этот набор и, во-вторых, использовать рациональную стратегию планирования (проектирования) тестов. Проектирование тестов можно начинать сразу же после завершения этапа внешнего описания ПС.

Возможны разные подходы к выработке стратегии проектирования тестов, которые можно условно графически разместить (рисунок 26) между следующими двумя крайними подходами. Левый крайний подход заключается в том, что тесты проектируются только на основании изучения спецификаций ПС (внешнего описания, описания архитектуры и спецификации модулей). Строение модулей при этом никак не учитывается, т.е. они рассматриваются как черные ящики. Фактически такой подход требует полного перебора всех наборов входных данных, так как в противном случае некоторые участки программ ПС могут не работать при пропуске любого теста, а это значит, что содержащиеся в них ошибки не будут проявляться. Однако тестирование ПС полным множеством наборов входных данных практически неосуществимо. Правый крайний подход заключается в том, что тесты проектируются на основании изучения текстов программ с целью протестировать все пути выполнения каж-

дой программ ПС. Если принять во внимание наличие в программах циклов с переменным числом повторений, то различных путей выполнения программ ПС может оказаться также чрезвычайно много, так что их тестирование также будет практически неосуществимо.

Рисунок 26 - Спектр подходов к проектированию тестов

Оптимальная стратегия проектирования тестов расположена внутри интервала между этими крайними подходами, но ближе к левому краю. Она включает проектирование значительной части тестов по спецификациям, но она требует также проектирования некоторых тестов и по текстам программ. При этом в первом случае эта стратегия базируется на принципах:

–на каждую используемую функцию или возможность – хотя бы один тест,

–на каждую область и на каждую границу изменения какой-либо входной величины – хотя бы один тест,

–на каждую особую (исключительную) ситуацию, указанную в спецификациях, – хотя бы один тест.

Во втором случае эта стратегия базируется на принципе: каждая команда каждой программы ПС должна проработать хотя бы на одном тесте.

Оптимальную стратегию проектирования тестов можно конкретизировать на основании следующего принципа: для каждого программного документа (включая тексты программ), входящего в состав ПС, должны проектироваться свои тесты с целью выявления в нем ошибок. Во всяком случае, этот принцип необходимо соблюдать в соответствии с определением ПС и содержанием понятия технологии программирования как технологии разработки надежного ПС. В связи с этим Майерс даже определяет разные виды тестирования в зависимости от вида программного документа, на основании которого строятся тесты. Описание таких тестов приведено ниже.

Тестирование архитектуры ПС. Целью тестирования является поиск несоответствия между описанием архитектуры и совокупностью программ ПС. К моменту начала тестирования архитектуры ПС должна быть уже закончена автономная отладка каждой подсистемы. Ошибки реализации архитектуры могут быть связаны, прежде всего, с взаимодействием этих подсистем, в частности, с реализацией архитектурных функций (если они есть). Поэтому хотелось бы проверить все пути взаимодействия между подсистемами ПС. При этом желательно хотя бы протестировать все цепочки выполнения подсистем без повторного вхождения последних. Если заданная архитектура представляет ПС в

качестве малой системы из выделенных подсистем, то число таких цепочек будет вполне обозримо.

Тестирование внешних функций. Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС. Несмотря на то, что все эти программы автономно уже отлажены, указанные расхождения могут быть, например, из-за несоответствия внутренних спецификаций программ и их модулей (на основании которых производилось автономное тестирование) функциональной спецификации ПС. Как правило, тестирование внешних функций производится так же, как и тестирование модулей на первом шаге, т.е. как черного ящика.

Тестирование качества ПС. Целью тестирования является поиск нарушений требований качества, сформулированных в спецификации качества ПС. Это наиболее трудный и наименее изученный вид тестирования. Ясно лишь, что далеко не каждый примитив качества ПС может быть испытан тестированием. Завершенность ПС проверяется уже при тестировании внешних функций. На данном этапе тестирование этого примитива качества может быть продолжено, если требуется получить какую-либо вероятностную оценку степени надежности ПС. Однако методика такого тестирования еще требует своей разработки. Могут тестироваться такие примитивы качества, как точность, устойчивость, защищенность, временная эффективность, в какой-то мере – эффективность по памяти, эффективность по устройствам, расширяемость и, частично, независимость от устройств. Каждый из этих видов тестирования имеет свою специфику и заслуживает отдельного рассмотрения. Мы здесь ограничимся лишь их перечислением. Легкость применения ПС (критерий качества, включающий несколько примитивов качества) оценивается при тестировании документации по применению ПС.

Тестирование документации по применению ПС. Целью тестирования является поиск несогласованности документации по применению и совокупностью программ ПС, а также выявление неудобств, возникающих при применении ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки ПС (тестированию определения требований к ПС и аттестации ПС), поэтому весьма важно разработчикам сначала самим воспользоваться ПС так, как это будет делать пользователь. Все тесты на этом этапе готовятся исключительно на основании только документации по применению ПС. Прежде всего, должны тестироваться возможности ПС как это делалось при тестировании внешних функций, но только на основании документации по применению. Должны быть протестированы все неясные места в документации, а также все примеры, использованные в документации. Далее тестируются наиболее трудные случаи применения ПС с целью обнаружить нарушение требований относительности легкости применения ПС.

Тестирование определения требований к ПС. Целью тестирования является выяснение, в какой мере ПС не соответствует предъявленному определению требований к нему. Особенность этого вида тестирования заключается в том, что его осуществляет организация-покупатель или организация-пользователь ПС как один из путей преодоления барьера между разработчиком и поль-

зователем. Обычно это тестирование производится с помощью контрольных задач – типовых задач, для которых известен результат решения. В тех случаях, когда разрабатываемое ПС должно придти на смену другой версии ПС, которая решает хотя бы часть задач разрабатываемого ПС, тестирование производится путем решения общих задач с помощью, как старого, так и нового ПС (с последующим сопоставлением полученных результатов). Иногда в качестве формы такого тестирования используют *опытную* эксплуатацию ПС – ограниченное применение нового ПС с анализом использования результатов в практической деятельности. По существу, этот вид тестирования во многом перекликается с испытанием ПС при его аттестации, но выполняется до аттестации, а иногда и вместо аттестации.

5.2.2 Методы тестирования

Метод "черного ящика". Основан на принципе "вход-выход". Программе подаются некоторые данные на вход и проверяются результаты, в надежде найти несоответствия. При этом как именно работает программа считается несущественным. При таком подходе необходимо иметь спецификацию программы для того, чтобы было с чем сравнивать результаты.

Метод "белого ящика". В этом методе тестовые данные получают путем анализа логики программы.

Категории тестовых данных. Существует методология относящаяся к методу ящика, которая называется эквивалентным разбиением. Согласно ей сначала выделяют классы эквивалентности, а затем строят тесты.

Различают два типа классов эквивалентности:

–правильные (представляющие правильные входные данные);

–неправильные (ошибочные входные данные).

Необходимо сосредоточить внимание на неправильных и неожиданных условиях.

–если входное условие описывает область значений, то определяется один правильный класс и два неправильных;

–если входное условие описывает множество входных значений, каждый из которых программа трактует особо, то определяется правильный класс эквивалентности для каждого значения и один неправильный класс;

–если входное условие описывает ситуацию "должна быть", то определяется один правильный и один неправильный классы;

–если есть основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс разбивается на меньшие классы.

Построение тестов включает в себя:

1. Назначение каждому классу эквивалентности уникального номера
2. Проектирование новых тестов, каждый из которых покрывает как можно большее количество неоткрытых правильных классов.
3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов.

Данные для тестирования.

1. В качестве некоторых тестовых данных используют экстремальные значения. Например, если целая величина должна находиться в диапазоне от *a* до *b*, то следует проверить: *a-1, a, a+1, b-1, b, b+1.*
2. Используют специальные значения. К ним относятся: константы, 0, 1, пустая строка, пустой файл, строка из одного символа и т.д.
3. Для циклов, организованных с помощью оператора Do, выбрать значения, при которых цикл выполняется 0, 1 и максимальное число раз.

Также методы тестирования можно разделить на:

1. *Статическое тестирование* – ручная проверка программы за столом.
2. *Детерминированное тестирование* – при различных комбинациях исходных данных.
3. *Стохастическое* – исходные данные выбираются произвольно, на выходе определяется качественное совпадение результатов или примерная оценка.

5.2.3 Отладка

Успех отладки ПС в значительной степени предопределяет рациональная организация тестирования. При отладке ПС отыскиваются и устраняются, в основном, те ошибки, наличие которых в ПС устанавливается при тестировании.

Рассмотрим основные заповеди отладки программного средства

Заповедь 1. Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам; нежелательно тестировать свою собственную программу.

Заповедь 2. Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.

Заповедь 3. Готовьте тесты как для правильных, так и для неправильных данных.

Заповедь 4. Документируйте пропуск тестов через компьютер; детально изучайте результаты каждого теста; избегайте тестов, пропуск которых нельзя повторить.

Заповедь 5. Каждый модуль подключайте к программе только один раз; никогда не изменяйте программу, чтобы облегчить ее тестирование.

Заповедь 6. Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в нее были внесены изменения (например, в результате устранения ошибки).

В нашей стране различаются два основных вида отладки (включая тестирование): автономную и комплексную отладку ПС.

Автономная отладка ПС означает последовательное раздельное тестирование различных частей программ, входящих в ПС, с поиском и исправлением в них фиксируемых при тестировании ошибок. Она фактически включает отладку каждого программного модуля и отладку сопряжения модулей.

Комплексная отладка означает тестирование ПС в целом с поиском и исправлением фиксируемых при тестировании ошибок во всех документах (включая тексты программ ПС), относящихся к ПС в целом. К таким документам относятся определение требований к ПС, спецификация качества ПС, функциональная спецификация ПС, описание архитектуры ПС и тексты программ ПС.

Автономная отладка программного средства. При автономной отладке ПС каждый модуль на самом деле тестируется в некотором программном окружении, кроме случая, когда отлаживаемая программа состоит только из одного модуля. Это окружение состоит из других модулей, часть которых является модулями отлаживаемой программы, которые уже отлажены, а часть – модулями, управляющими отладкой (*отладочными* модулями). Таким образом, при автономной отладке тестируется всегда некоторая программа (*тестируемая программа*), построенная специально для тестирования отлаживаемого модуля. Эта программа лишь частично совпадает с отлаживаемой программой, кроме случая, когда отлаживается последний модуль отлаживаемой программы. В процессе автономной отладки ПС производится наращивание тестируемой программы отлаженными модулями: при переходе к отладке следующего модуля в его программное окружение добавляется последний отлаженный модуль. Такой процесс наращивания программного окружения отлаженными модулями называется *интеграцией* программы. Отладочные модули, входящие в окружение отлаживаемого модуля, зависят от порядка, в каком отлаживаются модули этой программы, от того, какой модуль отлаживается и, возможно, от того, какой тест будет пропускаться.

При восходящем тестировании это окружение будет содержать только один отладочный модуль (кроме случая, когда отлаживается последний модуль отлаживаемой программы), который будет головным в тестируемой программе. Такой отладочный модуль называют *ведущим* (или драйвером). Ведущий отладочный модуль подготавливает информационную среду для тестирования отлаживаемого модуля (т. е. формирует ее состояние, требуемое для тестирования этого модуля, в частности, путем ввода некоторых тестовых данных), осуществляет обращение к отлаживаемому модулю и после окончания его работы выдает необходимые сообщения. При отладке одного модуля для разных тестов могут составляться разные ведущие отладочные модули.

При нисходящем тестировании окружение отлаживаемого модуля в качестве отладочных модулей содержит *отладочные имитаторы* (заглушки) некоторых еще не отлаженных модулей. К таким модулям относятся, прежде всего, все модули, к которым может обращаться отлаживаемый модуль, а также еще не отлаженные модули, к которым могут обращаться уже отлаженные модули (включенные в это окружение). Некоторые из этих имитаторов при отладке одного модуля могут изменяться для разных тестов.

На практике в окружении отлаживаемого модуля могут содержаться отладочные модули обоих типов, если используется смешанная стратегия тестирования. Это связано с тем, что как восходящее, так и нисходящее тестирование имеет свои достоинства и свои недостатки.

Часто применяют также комбинацию восходящего и нисходящего тестирования, которую называют методом *сандвича*. Сущность этого метода заключается в одновременном осуществлении как восходящего, так и нисходящего тестирования, пока эти два процесса тестирования не встретятся на каком-либо модуле где-то в середине структуры отлаживаемой программы. Этот метод при разумном порядке тестирования позволяет воспользоваться достоинствами как восходящего, так и нисходящего тестирования, а также в значительной степени нейтрализовать их недостатки.

Весьма важным при автономной отладке является тестирование сопряжения модулей. Дело в том, что спецификация каждого модуля программы, кроме головного, используется в этой программы в двух ситуациях: во-первых, при разработке текста (иногда говорят: тела) этого модуля и, во-вторых, при написании обращения к этому модулю в других модулях программы. И в том, и в другом случае в результате ошибки может быть нарушено требуемое соответствие заданной спецификации модуля. Такие ошибки требуется обнаруживать и устранять. Для этого и предназначено тестирование сопряжения модулей. При нисходящем тестировании тестирование сопряжения осуществляется попутно каждым пропускаемым тестом, что считают достоинством нисходящего тестирования. При восходящем тестировании обращение к отлаживаемому модулю производится не из модулей отлаживаемой программы, а из ведущего отладочного модуля. В связи с этим существует опасность, что последний модуль может приспособиться к некоторым "заблуждениям" отлаживаемого модуля. Поэтому, приступая (в процессе интеграции программы) к отладке нового модуля, приходится тестировать каждое обращение к ранее отлаженному модулю с целью обнаружения несогласованности этого обращения с телом соответствующего модуля (и не исключено, что виноват в этом ранее отлаженный модуль). Таким образом, приходится частично повторять в новых условиях тестирование ранее отлаженного модуля, при этом возникают те же трудности, что и при нисходящем тестировании.

Автономное тестирование модуля целесообразно осуществлять в четыре последовательно выполняемых шага.

Шаг 1. На основании спецификации отлаживаемого модуля подготовьте тесты для каждой возможности и каждой ситуации, для каждой границы областей допустимых значений всех входных данных, для каждой области изменения данных, для каждой области недопустимых значений всех входных данных и каждого недопустимого условия.

Шаг 2. Проверьте текст модуля, чтобы убедиться, что каждое направление любого разветвления будет пройдено хотя бы на одном тесте. Добавьте недостающие тесты.

Шаг 3. Проверьте текст модуля, чтобы убедиться, что для каждого цикла существуют тесты, обеспечивающие, по крайней мере, три следующие ситуации: тело цикла не выполняется ни разу, тело цикла выполняется один раз и тело цикла выполняется максимальное число раз. Добавьте недостающие тесты.

Шаг 4. Проверьте текст модуля, чтобы убедиться, что существуют тесты, проверяющие чувствительность к отдельным особым значениям входных данных. Добавьте недостающие тесты.

Комплексная отладка программного средства. Как уже было сказано выше, при комплексной отладке тестируется ПС в целом, причем тесты готовятся по каждому из документов ПС. Тестирование этих документов производится, как правило, в порядке, обратном их разработке. Исключение составляет лишь тестирование документации по применению, которая разрабатывается по внешнему описанию параллельно с разработкой текстов программ – это тестирование лучше производить после завершения тестирования внешнего описания. Тестирование при комплексной отладке представляет собой применение ПС к конкретным данным, которые в принципе могут возникнуть у пользователя (в частности, все тесты готовятся в форме, рассчитанной на пользователя), но, возможно, в моделируемой (а не в реальной) среде. Например, некоторые недоступные при комплексной отладке устройства ввода и вывода могут быть заменены их программными имитаторами.

Лабораторная работа №1

Тема: *Основные характеристики качества и надежности программного продукта*

Цель: *Изучение и применение принципов оценки программного обеспечения с точки зрения качества и надежности.*

1 Постановка задачи

Разработать программное средство, в котором присутствовали бы некоторые критерии и примитивы качественного программного обеспечения. Сделать выводы о проделанной работе.

2 Содержание отчета

Отчет должен содержать:
–постановку задачи;
–теоретические сведения;
–обоснование качества программного средства;
–текст программы;
–контрольный пример;
–выводы о проделанной работе.

Лабораторная работа №2

Тема: *Рассмотрение этапов жизненного цикла программного обеспечения*

Цель: *Изучение жизненного цикла программного обеспечения. Выбор модели жизненного цикла при разработке программного обеспечения. Преодоление барьера между пользователем и разработчиком.*

Студенты делятся на группы по три человека. Каждый участник группы выбирает одну из ролей: заказчик, программист, тестировщик (он же проводит аттестацию). В соответствии с выбранными ролями студенты должны выполнить следующее задание

1 Постановка задачи

Разработать программный продукт (область применения любая). Рассмотреть этапы жизненного цикла и выбрать наиболее подходящую модель.

В результате работы группа должна на каждом этапе сформировать требования к программному продукту:

1. постановка задачи,

2. программное средство, написанная на каком-либо языке или приложении,

3. набор тестовых данных.

2 Содержание отчета

Отчет по лабораторной работе состоит из следующих пунктов:

Введение

1 Постановка задачи, которая формулируется совместными усилиями заказчика и программиста. В постановке задачи оговариваются:

обоснование постановки задачи,

выбор структуры программного средства,

выбор языка программирования, набор тестовых данных,

требования к документации,

сопровождение программного продукта,

требования к аппаратному обеспечению и т.д.

В результате составляется спецификация в которой присутствуют следующие данные:

наименование задачи.

назначение.

достигаемая цель.

для кого предназначена.

системные и технические средства.

периодичность использования.

входная информация и выходная информация (раскрывается состав и форма представления входной, промежуточной и выходной информации)

источник возникновения данных

характеризуются формы и методы контроля достоверности информации

описываются формы взаимодействия пользователя с ЭВМ

завершается постановка задачи описанием контрольного примера, демонстрирующего порядок решения задачи традиционным способом.

Основное требование к контрольному примеру - это отражение всего многообразия возможных форм существования исходных данных.

2. Теоретическая часть. Здесь описывается теория по разработке ПО

3. Практическая часть. Программист объясняет выбор модели ЖЦ, а также предоставляет анализ предметной области, т.е. определяет входные и выходные данные, способ их получение, хранения, обработки и вывода. Если необходимо, программист, строит математическую или другие модели, позволяющие более наглядно представить нужную информацию. Так же он предоставляет код программного продукта

4.Руководсто пользователю. Здесь указывается:

название ПС,

условия его работы (технические и системные требования),

действия оператора для работы с ПС,

дополнительные сообщения оператору.

5. Тестирование. В данном разделе предоставляются наборы тестов, разработанные тестировщиком, максимально полно отражающие ситуации, которые могут возникнуть в процессе работы и реакции ПО на них.

Приложение А. Код программы

Лабораторная работа №3

Тема: *Декомпозиция задачи. Структурный и модульный подход к проектированию*

Цель: *Произвести декомпозицию задачи. Построить иерархию модулей и определить связи между ними. Построить функциональную схему и схему информационных связей. Определить архитектуру программного средства.*

Лабораторная работа выполняется на основе второй лабораторной.

1 Постановка задачи

На основе программного средства, разработанного на второй лабораторной работе провести анализ архитектуры программного средства и метода программирования, а так же провести декомпозицию, построить иерархическую, функциональную и схему информационных связей ПС.

2 Содержание отчета

Отчет по лабораторной работе состоит из следующих пунктов:

1. Постановка задача

2. Теоретическая часть. Раскрываются основы построения ПС, определяются понятия архитектуры, кратко рассматриваются методы проектирования и средства, используемые при этом.

3. Практическая часть. Представляется архитектура программного средства. Производиться декомпозиция задачи, строятся схемы, необходимые при проектировании ПС (иерархическая, функциональная, схема информационных связей).

Лабораторная работа №4

Тема: *Характеристика программного модуля. Потоки данных и процессы.*

Цель: *Изучение характеристик программного модуля. Построение моделей потоков данных и процессов в различных методологиях. 0ценка модулей.*

1 Постановка задачи

Написать программу анализа функции, разделив задачу на три части:
- интерфейс;
- анализ функции;
- построение графика функции.

2 Содержание отчета

Отчет состоит из следующих частей:

1. Постановка задачи.

2. Теоретическая часть. Раскрываются основы характеристик программного модуля, определяются основные понятия, кратко рассматриваются методы моделирования потоков данных и процессов.

3. Практическая часть. Производиться декомпозиция задачи, строятся схемы, отображающие работу программы, проводится анализ модулей, присутствующих в программе.

Лабораторная работа №5

Тема: *Тестирование и отладка*

Цель: *Изучить принципы и методы тестирования и отладки программного средства. Научиться создавать тестовые данные*

1 Постановка задачи

Разработать программу, выполняющую не менее трех операций над матрицами.

В результате работы студент должен получить программу и отчет о проделанной работе.

2 Содержание отчета

Отчет по лабораторной работе состоит из следующих пунктов:

Введение

1 Постановка задачи, которая описывает какие операции производятся над матрицами, и если необходимо указываются какие-либо ограничения.

2. Теоретическая часть. Здесь описывается теория тестирования и отладки программного средства.

3. Практическая часть. В данном разделе предоставляются наборы тестов, разработанные тестировщиком, которые максимально полно отражают ситуации, которые могут возникнуть в процессе работы и реакции ПО на них.

Приложение А. Код программы

Вопросы к защите лабораторных работ

Лабораторная работа № 1

1. Понятие качества программного средства?
2. Что такое надежность программного средства?
3. Что понимают под отказом в программном средстве?

Лабораторная работа № 2

1. Что такое жизненный цикл программного средства (ПО)?
2. Что такое анализ ПО?
3. Что такое сопровождение ПО?
4. Что такое RAD?
5. Что такое смежный контроль?

Лабораторная работа № 3

1. Что такое архитектура программного средства?
2. Что такое программный модуль?
3. Что такое структурное программирование?
4. Что такое пошаговая детализация программного модуля?

Лабораторная работа № 4

1. Что такое программный модуль?
2. Характеристики программного модуля по Майерсу?
3. Характеристики программного модуля по методологии SADT?
4. Что представляет собой модель SADT?

5. Что из себя представляет функциональная схема?

Лабораторная работа № 5

1. Что такое отладка программного средства?
2. Что такое тестирование программного средства?
3. Что такое автономная отладка программного средства?
4. Что такое комплексная отладка программного средства?
5. Что такое ведущий отладочный модуль?
6. Что такое отладочный имитатор программного модуля?