Московский государственный технический университет им. Н.Э. Баумана
Факультет Информатики и систем управления
Кафедра Компьютерные системы и сети

Г.С. Иванова, Т.Н. Ничушкина

Проектирование программного обеспечения

Учебное пособие

по выполнению и оформлению курсовых, дипломных и квалификационных работ

МОСКВА 2002

АННОТАЦИЯ

Настоящее учебное пособие содержит указания и рекомендации по выполнению и оформлению курсовых и квалификационных работ, связанных с разработкой программных продуктов. В пособии описываются порядок выполнения, оформления и требования к представляемым документам. Особое внимание обращено на оформление текстовых и графических документов: технического задания, расчетно-пояснительной записки и плакатов. В приложении приводятся примеры технического задания и оглавления расчетно-пояснительной записки.

Пособие предназначено для студентов всех курсов специальности «Компьютерные системы и сети».

Оглавление

ВВЕДЕНИЕ

Создание современной программной системы – весьма трудоемкая задача: обычный размер ПО превышает сотни тысяч операторов. Для эффективного создания подобных программных продуктов специалист должен иметь представление о методах анализа, проектирования, реализации и тестирования программных систем; ориентироваться в существующих подходах и технологиях.

Проектирование программных продуктов, как и любых других сложных систем, выполняется поэтапно с использованием блочно-иерархического подхода, который подразумевает разработку продукта по частям с последующей сборкой. На каждом этапе выполняются определенные проектные операции, которые соответствующим образом документируются. Последовательность выполнения этапов и их результаты непосредственно следуют из используемой модели жизненного цикла программного обеспечения (ПО).

Кроме того, реализованная система также должна сопровождаться разного рода программной документацией, например, спецификацией, руководством программиста, руководством пользователя, руководством оператора. Таким образом, владение навыками создания программной документации, безусловно, необходимо будущему разработчику ПО.

1. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Жизненным циклом ПО называют период от момента появления идеи создания некоторого ПО до момента завершения его поддержки фирмой-разработчиком или фирмой, выполнявшей сопровождение.

Состав процессов жизненного цикла регламентируется международным стандартом ISO/IEC 12207: 1995 «Information Technology – Software Life Cycle Process» («Информационные технологии – Процессы жизненного цикла ПО»). ISO – International Organization of Standardization – Международная организация по стандартизации, IEC – International Electrotechnical Commission – Международная комиссия по электротехнике.

Этот стандарт только называет и определяет процессы жизненного цикла, не конкретизируя в деталях, как реализовывать или выполнять действия и задачи, включенные в эти процессы. При этом процесс жизненного цикла определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Каждый процесс характеризуется определенными задачами и методами их ре-

шения, а также исходными данными и результатами. Так в соответствии со стандартом все процессы делятся на три группы:

* основные процессы (приобретение, поставка, разработка, эксплуатация, сопровождение);

* вспомогательные процессы; обеспечивают выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит);

* организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение жизненного цикла ПО, обучение).

Указанные действия можно условно сгруппировать, выделив следующие основные этапы (в скобках указаны соответствующие стадии разработки по ГОСТ 19.102–77 «Стадии разработки»):

- постановка задачи (стадия «Техническое задание»);
- анализ требований и разработка спецификаций (стадия «Эскизный проект»);
- проектирование (стадия «Технический проект»);
- реализация (стадия «Технический проект»).

Постановка задачи. В процессе постановки задачи четко формулируют назначение ПО и определяют основные *функциональные*, *эксплуатационные* и *технологические* требования к нему**.** Функциональные требования определяют функции разрабатываемого ПО, эксплуатационные – особенности его эксплуатации, а технологические – особенности процесса разработки: подход, архитектуру, технологию, среду или язык программирования.

Требования к ПО, имеющему прототипы, обычно выполняют по аналогии, учитывая структуру и характеристики уже существующих программных продуктов. Для формулирования требований к ПО, не имеющему аналогов, иногда необходимо провести специальные исследования, называемые *предпроектными*. В процессе таких исследований определяют разрешимость задачи, возможно, разрабатывают методы ее решения (если они новые) и устанавливают наиболее существенные характеристики разрабатываемого ПО. Обычно этап постановки задачи заканчивается разработкой *технического задания*.

Анализ требований и определение спецификаций. *Спецификациями* называют точное формализованное описание функций и ограничений разрабатываемого ПО. Соответственно различают *функциональные* и *эксплуатационные* спецификации. Сово-

купность спецификаций представляет собой *общую* логическую модель проектируемого ПО.

Для получения спецификаций выполняют анализ требований технического задания, формулируют содержательную постановку задачи, выбирают математический аппарат формализации, строят модель предметной области, определяют подзадачи и выбирают или разрабатывают методы их решения. Часть спецификаций может быть определена в процессе предпроектных исследований и, соответственно, зафиксирована в техническом задании.

На этапе также целесообразно сформировать тесты для поиска ошибок в проектируемом ПО, обязательно указав ожидаемые результаты.

Проектирование. Задачей этого этапа является определение *подробных* спецификаций разрабатываемого ПО.

Процесс проектирование сложного ПО обычно включает:

* проектирование общей структуры – определение основных частей (*компонентов*) и их взаимосвязей по управлению и данным;

* декомпозицию компонентов и построение структурных иерархий в соответствии с рекомендациями блочно-иерархического подхода;

* проектирование компонентов.

Результатом проектирования является детальная модель разрабатываемого ПО вместе со спецификациями его компонентов всех уровней. Тип модели зависит от выбранного или заданного подхода (структурный, объектно-ориентированный или компонентный) и конкретной технологии проектирования. Однако в любом случае процесс проектирования охватывает как проектирование обрабатывающих программ (подпрограмм) и определение взаимосвязей между ними, так и проектирование данных, с которыми взаимодействуют эти программы или подпрограммы.

Принято различать также два аспекта проектирования:

* логическое проектирование, включающее те проектные операции, которые непосредственно не зависят от имеющихся технических и программных средств, составляющих среду функционирования будущего программного продукта;

* физическое проектирование, которое заключается в привязке к конкретным техническим и программным средствам среды функционирования.

Реализация. Реализация представляет собой процесс поэтапного написания кодов программы на выбранном языке программирования (кодирование), их тестирование и отладку.

Сопровождение. Сопровождение – это процесс выпуска и внедрения новых версий программного продукта.

Причинами выпуска новых версий могут служить:

* необходимость исправления ошибок, выявленных в процессе эксплуатации предыдущих версий;

* необходимость совершенствования предыдущих версий, например, улучшения интерфейса или расширения состава выполняемых функций;

* изменение среды функционирования, например, появление новых технических средств и/или программных продуктов.

На этом этапе в программный продукт вносят необходимые изменения, которые могут потребовать пересмотра проектных решений, принятых на любом этапе.

В настоящее время при разработке ПО в основном используется *спиральная схема* (рис. 1.1), согласно которой программный продукт создается не сразу, а итерационно с использованием прототипов. *Прототипом* называют действующий программный продукт, реализующий отдельные функции и внешние интерфейсы разрабатываемого ПО.

При использовании спиральной схемы на первой итерации, как правило, специфицируют, проектируют, реализуют и тестируют интерфейс пользователя. На второй – добавляют некоторый ограниченный набор функций. На последующих этапах этот набор расширяют, наращивая возможности данного продукта.

Основным достоинством данной схемы является то, что, начиная с некоторой итерации, обеспечившей определенную функциональную полноту, продукт можно предоставлять пользователю. Это, в свою очередь, позволяет:

* сократить время до появления первых версий программного продукта;

* заинтересовать большое количество пользователей, обеспечивая быстрое продвижение следующих версий продукта на рынке;

* ускорить формирование и уточнение спецификаций за счет появления практики использования продукта;

* уменьшить вероятность морального устаревания системы за время разработки.

Появление CASE-технологий (Computer-Aided Software/System Engineering – Автоматизированная технология разработки ПО/систем) снизило трудоемкость отдельных

этапов жизненного цикла ПО за счет автоматизации трудоемких операций. Современные CASE-средства существенно увеличивают производительность труда программистов, улучшают качество создаваемого ПО и автоматизируют формирование проектной документации для всех этапов жизненного цикла в соответствии с современными стандартами.

2. ПОСТАНОВКА ЗАДАЧИ. РАЗРАБОТКА ТЕХНИЧЕСКОГО ЗАДАНИЯ

Процесс создания нового ПО начинают с постановки задачи, в процессе которой определяют требования к программному продукту. Это один из наиболее ответственных этапов в процессе создания ПО. От того насколько полно определены функции будущего программного продукта и другие требования к нему, во многом зависит стоимость разработки и ее качество.

Техническое задание представляет собой документ, в котором формулируют основные цели разработки, требования к программному продукту, определяют сроки и этапы разработки и регламентируют процесс приемно-сдаточных испытаний. В формулировании технического задания участвуют как представители заказчика, так и представители исполнителя. В основе этого документа лежат исходные требования заказчика, анализ передовых достижений техники, результаты выполнения научно-исследовательских работ, предпроектных исследований, научного прогнозирования и т.п.

Основными факторами, определяющими характеристики разрабатываемого ПО, являются:

* исходные данные и требуемые результаты, которые определяют *функции* программы или системы;

* среда (программная и аппаратная), в которой разрабатываемое ПО будет функционировать, может быть задана, а может выбираться для обеспечения параметров, указанных в техническом задании;

* возможное взаимодействие с другим ПО и/или конкретными техническими средствами – также может быть определено, а может выбираться исходя из набора выполняемых функций.

Разработка технического задания выполняется в следующей последовательности. Прежде всего, устанавливают набор выполняемых функций, а также перечень и характеристики исходных данных. Затем определяют набор результатов, их характеристики

и способы представления. Далее уточняют среду функционирования ПО: конкретную комплектацию и параметры технических средств, версию используемой операционной системы и, возможно, версии и параметры другого установленного ПО, с которым предстоит взаимодействовать будущему программному продукту.

В тех случаях, когда разрабатываемое ПО собирает и хранит некоторую информацию или включается в управление каким-либо техническим процессом, необходимо также четко регламентировать действия программы в случае сбоев оборудования и энергоснабжения.

На техническое задание существует стандарт (ГОСТ 19.201–78). В соответствие с этим стандартом техническое задание должно содержать следующие разделы:

* введение;
* основания для разработки;
* назначение разработки;
* требования к программе или программному изделию;
* требования к программной документации;
* технико-экономические показатели;
* стадии и этапы разработки;
* порядок контроля и приемки.

При необходимости допускается в техническое задание включать приложения.

Рассмотрим более подробно содержание каждого раздела.

Введение должно включать наименование и краткую характеристику области применения программы или программного продукта. Основное назначение введения – продемонстрировать актуальность данной разработки и показать, какое место эта разработка занимает в ряду подобных.

Раздел *Основания для разработки* должен содержать наименование документа, на основании которого ведется разработка, организации, утвердившей данный документ, и наименование или условное обозначение темы разработки. Таким документом может служить план, приказ, договор и т.п.

Раздел *Назначение разработки* должен содержать описание функционального и эксплуатационного назначения программного продукта с указанием категорий пользователей.

Раздел *Требования к программе или программному изделию* должен включать следующие подразделы:

* требования к функциональным характеристикам;

* требования к надежности;

* условия эксплуатации;

* требования к составу и параметрам технических средств;

* требования к информационной и программной совместимости;

* требования к маркировке и упаковке;

* требования к транспортированию и хранению;

* специальные требования.

Наиболее важным из перечисленных выше является подраздел *Требований* к *функциональным характеристикам*. В этом разделе должны быть перечислены выполняемые функции и описаны состав, характеристики и формы представления исходных данных и результатов. В этом же разделе при необходимости указывают критерии эффективности: максимально допустимое время ответа системы, максимальный объем используемой оперативной и/или внешней памяти и др.

В подразделе *Требования к надежности* указывают уровень надежности, который должен быть обеспечен разрабатываемой системой и время восстановления системы после сбоя. Для систем с обычными требованиями к надежности в этом разделе иногда регламентируют действия разрабатываемого продукта по увеличению надежности результатов (контроль входной и выходной информации, создание резервных копий промежуточных результатов и т. п.).

В подразделе *Условия эксплуатации*, указывают особые требования к условиям эксплуатации: температуре окружающей среды, относительной влажности воздуха и т.п. Как правило, подобные требования формулируют, если разрабатываемая система будет эксплуатироваться в нестандартных условиях или использует специальные внешние устройства, например, для хранения информации. Здесь же указывают вид обслуживания, необходимое количество и квалификация персонала.

В подразделе *Требования к составу и параметрам технических средств* указывают необходимый состав технических средств с указанием их основных технических характеристик: тип микропроцессора, объем памяти, наличие внешних устройств и т.п. При этом часто указывают два варианта конфигурации: минимальный и рекомендуемый.

В подразделе *Требования к информационной и программной совместимости* при необходимости можно задать методы решения, определить язык или среду программирования для разработки, а также используемую операционную систему и другие сис-

темные и пользовательские программные средства, с которыми должно взаимодействовать разрабатываемое ПО. В этом же разделе при необходимости указывают, какую степень защиты информации необходимо предусмотреть.

В разделе *Требования к программной документации* указывают необходимость наличия руководства программиста, руководства пользователя, руководства системного программиста, пояснительной записки и т.п. На все эти типы документов также существуют ГОСТы.

В разделе *Технико-экономические показатели* рекомендуется указывать ориентировочную экономическую эффективность и экономические преимущества по сравнению с существующими аналогами.

В разделе *Стадии и этапы разработки* указывают стадии разработки, этапы и содержание работ с указанием сроков разработки и исполнителей.

В разделе *Порядок контроля и приемки* указывают виды испытаний и общие требования к приемке работы.

В приложениях при необходимости приводят: перечень научно-исследовательских работ, обосновывающих разработку; схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие документы, которые могут быть использованы при разработке.

В зависимости от особенностей разрабатываемого продукта разрешается уточнять содержание разделов, т.е. использовать подразделы, вводить новые разделы или объединять их.

Пример титульного листа технического задания на учебный программный продукт представлен в приложении 1.

3. АНАЛИЗ ТРЕБОВАНИЙ И ОПРЕДЕЛЕНИЕ СПЕЦИФИКАЦИЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРИ СТРУКТУРНОМ ПОДХОДЕ

Спецификации программного обеспечения при структурном подходе

Собственно разработка любого ПО начинается с анализа требований к будущему программному продукту. В результате анализа получают спецификации разрабатываемого ПО: выполняют декомпозицию и содержательную постановку решаемых задач, уточняют их взаимодействие и эксплуатационные ограничения. В целом в процессе определения спецификаций строят общую модель предметной области, как некоторой

части реального мира, с которой будет тем или иным способом взаимодействовать ПО, и конкретизируют его основные функции.

Как уже упоминалось ранее, спецификации представляют собой *полное* и *точное* описание функций и ограничений разрабатываемого ПО. При этом часть спецификаций (*функциональные)*, описывают функции разрабатываемого ПО, а другая часть (*эксплуатационные)* определяет требования к техническим средствам, надежности, безопасности и т.д.

Определение отражает главные требования к спецификациям. Применительно к функциональным спецификациям при этом подразумевается, что:

* требование *полноты* означает, что спецификации должны содержать всю существенную информацию, чтобы ничего важного не было упущено, и не должны содержать несущественной информации, например, деталей реализации, чтобы не препятствовать разработчику в выборе наиболее эффективных решений;

* требование *точности* означает, что спецификации должны однозначно восприниматься как заказчиком, так и разработчиком.

Последнее требование выполнить достаточно сложно, так как естественный язык для описания спецификаций не подходит: подробные спецификации на естественном языке не обеспечивают необходимой точности. Точные спецификации разрабатываемого ПО можно определить, только разработав некоторую *формальную модель* ПО.

Формальные модели, разрабатываемые на этапе определения спецификаций можно разделить на две группы: модели, *зависящие от подхода к разработке* (структурного или объектно-ориентированного), и модели, *не зависящие* от него. Так диаграммы переходов состояний, которые демонстрируют особенности поведения разрабатываемого ПО при получении тех или иных сигналов извне и математические модели предметной области используют при любом подходе к разработке.

В рамках структурного подхода на этапе анализа и определения спецификаций используют три типа моделей: ориентированные на функции, ориентированные на данные и ориентированные на потоки данных, каждый из которых целесообразно использовать для своего специфического класса программных разработок.

На рис. 3.1 показана классификация моделей, используемых в качестве спецификаций разрабатываемого ПО.

Следует иметь в виду, что все функциональные спецификации описывают одни и те же характеристики разрабатываемого ПО: перечень функций и состав обрабатывае-

мых данных. Они различаются только системой приоритетов (акцентов), которая используется разработчиком в процессе анализа требований и определения спецификаций. Так диаграммы переходов состояний определяют некоторые аспекты поведения ПО во времени, диаграммы потоков данных – направление и структуру потоков данных, а концептуальные диаграммы классов – отношение между основными понятиями предметной области.

Поскольку разные модели описывают проектируемое ПО с разных сторон, рекомендуется использовать сразу несколько моделей и сопровождать их текстами, дополняющими соответствующие диаграммы.

Так методологии *структурного анализа и проектирования*, основанные на моделировании потоков данных, обычно используют комплексное представление проектируемого ПО в виде совокупности моделей:

* диаграмм потоков данных (DFD – Data Flow Diagrams), описывающих взаимодействие источников и потребителей информации через процессы, которые должны быть реализованы в системе (см. § 3.4);

* диаграмм «сущность-связь» (ERD – Entity-Relationship Diagrams), описывающих базы данных разрабатываемой системы (см. § 3.5);

* диаграмм переходов состояний (STD – State Transition Diagrams), характеризующих поведение системы во времени (см. § 3.2);

* спецификаций процессов;

* словаря данных.

Взаимосвязь компонент такой обобщенной модели показана на рис. 3.2.

Диаграмма переходов состояний

Диаграмма переходов состояний является графической формой представления конечного автомата – математической абстракции, используемой для моделирования детерминированного поведения технических объектов или объектов реального мира.

На этапе анализа требований и определения спецификации диаграмма переходов состояний демонстрирует поведение разрабатываемой программной системы при получении управляющих воздействий. Под *управляющими воздействиями* или *сигналами* в данном случае понимают управляющую информацию, получаемую системой извне, например, управляющими воздействиями считают команды пользователя и сигналы датчиков, подключенных к компьютерной системе. Получив такое управляющее воздейст-

вие, разрабатываемая система должна выполнить определенные действия, а затем, или остаться в том же состоянии, или перейти в другое состояние, зафиксировав некоторые изменения в системе.

Для построения диаграммы переходов состояний необходимо в соответствие с теорией конечных автоматов определить основные состояния, управляющие воздействия (или условия перехода), выполняемые действия и возможные переходы разрабатываемого ПО. Условные обозначения, используемые при построении диаграмм переходов состояний, показаны на рис. 3.3.

Для интерактивного ПО с развитым пользовательским интерфейсом основные управляющие воздействия – команды пользователя, для ПО реального времени – сигналы от датчиков и/или оператора производственного процесса. Общим для этих типов ПО является наличие состояния ожидания, когда система приостанавливает работу до получения очередного управляющего воздействия (рис. 3.4). Для интерактивного ПО наиболее характерно получение команд различных типов, а для ПО реального времени – однотипных сигналов, либо от многих датчиков, либо требующих продолжительной обработки.

В отличие от интерактивных систем для систем реального времени обычно установлено более жесткое ограничение на время обработки полученного сигнала. Такое ограничение часто требует выполнения дополнительных исследований поведения системы во времени, например, с использованием сетей Петри или марковских процессов.

К ПО, при разработке которого требуется уточнение особенностей поведения посредством построения диаграммы переходов состояний, относится и ПО, ориентированное на работу в сети. При этом обычно отдельно строят модели поведения сервера и клиента, представляя сообщения, передаваемые между ними в виде управляющих воздействий.

Пример 3.1. Рассмотрим диаграмму переходов состояний для программы построения таблиц значений и графиков функций одной переменной. Программа должна обеспечивать возможность изменения типа представления, шага и отрезка, а также сохранения введенных формул.

Программа относится к классу интерактивных, соответственно на этапе анализа и определения спецификаций целесообразно уточнить поведение программы на уровне интерфейса с пользователем. Один из возможных вариантов поведения представлен на рис. 3.5.

Полученную диаграмму переходов состояний следует согласовать с заказчиком ПО.

Функциональные диаграммы

Функциональными называют диаграммы, в первую очередь отражающие взаимосвязи функций разрабатываемого ПО. В качестве примера функциональной модели рассмотрим активностную модель, предложенную Д. Россом в составе методологии функционального моделирования SADT (Structured Analysis and Design Technique) в 1973 году.

Отображение взаимосвязи функций активностной модели осуществляется посредством построения *иерархии* функциональных диаграмм.

Функциональная диаграмма представляет собой схематическое представление взаимосвязей нескольких функций. Каждый блок такой диаграммы соответствует некоторой функции, для которой должны быть определены исходные данные, результаты, управляющая информация и механизмы ее осуществления – человек или технические средства.

Все перечисленные выше связи функции представляются дугами, причем тип связи и ее направление строго регламентированы: дуги, изображающие каждый тип связей, должны подходить к блоку с определенной стороны (рис. 3.6), а направление связи должно указываться стрелкой в конце дуги.

Физически дуги трех первых типов представляют собой наборы данных, передаваемые между функциями. Дуги, определяющие механизм выполнения функции, в основном используют при описании спецификаций сложных информационных систем, которые включают как автоматизированные, так и ручные операции.

Блоки и дуги маркируются текстами на естественном языке. Дуги могут разветвляться и соединяться вместе различными способами. Разветвление означает, что часть или вся информация может использоваться в каждом ответвлении дуги. Дуга всегда помечается до ветвления, чтобы идентифицировать передаваемый набор данных. Если ветвь дуги после ветвления не помечена, то непомеченная ветвь содержит весь набор данных. Каждая метка ветви уточняет, что именно содержит данная ветвь (рис. 3.7).

Построение иерархии функциональных диаграмм ведется поэтапно с увеличением уровня детализации: диаграммы каждого следующего уровня уточняют структуру родительского блока. Построение модели начинают с единственного блока, для которого

определяют исходные данные, результаты, управление и механизмы реализации. Затем он последовательно детализируется с использованием метода пошаговой детализации. При этом рекомендуется каждую функцию представлять не более чем 3–7-ю блоками. Во всех случаях *каждая подфункция может использовать или продуцировать только те элементы данных, которые использованы или продуцируются родительской функцией*, причем никакие элементы не могут быть опущены, что обеспечивает непротиворечивость построенной модели.

Стрелки, приходящие с родительской диаграммы или уходящие на нее нумеруют, используя символы и числа. Символ обозначает тип связи: I – входные, C – управляющие, M – механизмы, R – результаты. Число – номер связи по соответствующей стороне родительского блока, считая сверху вниз и слева направо.

Все диаграммы связывают друг с другом иерархической нумерацией блоков: первый уровень – A0, следующий – A1, A2 и т.п., следующий – A11, A12, A13 и т.д., где первые цифры – номер родительского блока, а последняя – номер конкретного субблока родительского блока.

Детализацию завершают при получении функций, назначение которых хорошо понятно, как заказчику, так и разработчику. Эти функции описывают, применяя естественный язык или псевдокоды.

В процессе построения иерархии диаграмм фиксируют всю уточняющую информацию и строят словарь данных, в котором определяют структуры и элементы данных, показанных на диаграммах.

Таким образом, в результате получают спецификацию, которая состоит из иерархии функциональных диаграмм, описаний функций нижнего уровня и словаря, имеющих ссылки друг на друга.

Пример 3.2. Разработку функциональных диаграмм продемонстрируем на примере уточнения спецификаций программы построения графиков и таблиц функций одной переменной (см. пример 3.1).

Диаграмма, показанная на рис. 3.8, *а,* является диаграммой верхнего уровня. На ней хорошо видно, что является исходными данными для программы, и получения каких результатов мы ожидаем.

Диаграмма, показанная на рис. 3.8, *б*, уточняет функции программы. На ней показаны четыре блока: ввод/выбор и ее разбор, добавление функции в список, построение таблицы значений и построение графика функции. Для каждого блока определены ис-

ходные данные, управляющие воздействия и результаты. Согласно правилам обозначения входов/выходов, имеющих продолжение на родительской диаграмме, на диаграмме использованы следующие обозначения: I1 – функция; I2 – отрезок; I3 – шаг; C1 – вид график/таблица; R1 – график функции на отрезке; R2 – таблица значений функции на отрезке.

Словарь в этом случае должен содержать описание всех данных, используемых в системе.

Функциональную модель целесообразно применять для определения спецификаций ПО, не предусматривающего работу со сложными структурами данных, так как она ориентирована на декомпозицию функций.

Диаграммы потоков данных

Диаграммы потоков данных позволяют специфицировать как функции разрабатываемого ПО, так и обрабатываемые им данные. При использовании этой модели систему представляют в виде иерархии диаграмм потоков данных, описывающих асинхронный процесс преобразования информации от ее ввода в систему до выдачи пользователю. На каждом следующем уровне происходит уточнение процессов, пока очередной процесс не будет признан элементарным.

В основе модели лежат понятия внешней сущности, процесса, хранилища (накопителя) данных потока данных.

Внешняя сущность – материальный объект или физическое лицо, выступающие в качестве источников или приемников информации, например, заказчики, персонал, поставщики, клиенты, банк и т.п.

Процесс представляет собой преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом. Каждый процесс в системе имеет свой номер и связан с исполнителем, который осуществляет данное преобразование. Как в случае функциональных диаграмм, физически преобразование может осуществляться компьютерами, вручную или специальными устройствами. На верхних уровнях иерархии, когда процессы еще не определены, вместо понятия «процесс» используют понятия «система» и «подсистема», которые обозначают соответственно систему в целом или ее функционально законченную часть.

Хранилище данных – абстрактное устройство для хранения информации. Тип устройства и способы помещения, извлечения и хранения для такого устройства не дета-

лизируют. Физически это может быть база данных, файл, таблица в оперативной памяти, картотека на бумаге и т.п.

Поток данных представляет собой процесс передачи некоторой информации от источника к приемнику. Физически процесс передачи информации может происходить по кабелям под управлением программы или программной системы, а также вручную при участии устройств или людей вне проектируемой системы.

Диаграмма, таким образом, иллюстрирует как потоки данных, порожденные некоторыми внешними сущностями, трансформируются соответствующими процессами (или подсистемами), сохраняются накопителями данных и передаются другим внешним сущностям – потребителям информации. В результате мы получаем сетевую модель хранения/обработки информации.

Для изображения диаграмм потоков данных традиционно используют два вида нотаций: нотации Йордана и Гейна-Сарсона (табл. 3.1).

Над линией потока, направление которого обозначают стрелкой, указывают, какая конкретно информация в данном случае передается (рис. 3.9).

Построение иерархии диаграмм потоков данных начинают с диаграммы особого вида – *контекстной диаграммы*, которая определяет наиболее общий вид системы. На такой диаграмме показывают, как разрабатываемая система будет взаимодействовать с приемниками и источниками информации без указания исполнителей, т.е. описывают интерфейс системы с внешним миром. Обычно начальная контекстная диаграмма имеет форму звезды.

Если проектируемая система содержит большое количество внешних сущностей (более 10), имеет распределенную природу или включает уже существующие подсистемы, то строят *иерархии* контекстных диаграмм.

Полученную таким образом модель системы проверяют на полноту исходных данных об объектах системы и изолированность объектов (отсутствие информационных связей с другими объектами). На следующем этапе каждую подсистему контекстной диаграммы детализируют при помощи диаграмм потоков данных.

В процессе детализации соблюдают правило б а л а н с и р о в к и – *при детализации подсистемы могут использоваться компоненты только тех подсистем, с которыми у разрабатываемой подсистемы существует информационная связь* (т.е. с которыми она связана потоками данных).

Решение о завершении детализации процесса принимают в следующих случаях:

* процесс взаимодействует с 2-3 потоками данных;

* возможно описание процесса последовательным алгоритмом;

* процесс выполняет единственную логическую функцию преобразования входной информации в выходную.

На недетализируемые процессы составляют спецификации, которые должны содержать описание логики (функций) данного процесса. Такое описание может выполняться: на естественном языке, с применением структурированного естественного языка (псевдокодов), с применением таблиц и деревьев решений и в виде схем алгоритмов.

Для облегчения восприятия процессы детализируемой подсистемы нумеруют, соблюдая иерархию номеров: так процессы, полученные при детализации процесса или подсистемы «1», должны нумероваться «1.1», «1.2» и т.д. Кроме этого желательно размещать на каждой диаграмме от 3 до 6-7 процессов и не загромождать диаграммы деталями, не существенными на данном уровне.

Декомпозицию потоков данных необходимо осуществлять параллельно с декомпозицией процессов. Полная спецификация процессов включает описание *структур данных*, используемых как при передаче информации в потоке, так и при хранении в накопителе (см. § 3.5).

Законченную модель необходимо проверить на полноту и согласованность. Под согласованностью модели в данном случае понимают выполнение для всех потоков данных правила *сохранения информации*: все поступающие куда-либо данные должны быть считаны и записаны.

Пример 3.3. Разработать иерархию диаграмм потоков данных системы учета успеваемости студентов (см. техническое задание в приложении 1).

В качестве внешних сущностей для системы выступают Декан, Заместитель декана по курсу и Сотрудник деканата. Определяем потоки данных между этими сущностями и системой.

Декан должен получать:

* сводку успеваемости по факультету (процент успеваемости групп, курсов и в целом по факультету) на текущий или указанный момент времени;

* полные сведения об учебе конкретного студента (успеваемость по всем изученным предметам всех завершенных семестров обучения с учетом пересдач).

Заместитель декана по курсу должен получать:

* сводку успеваемости по курсу (процент успеваемости по группам) на текущий или указанный момент;

* сведения о сдаче экзаменов и зачетов указанной группой;

* текущие сведения об успеваемости конкретного студента;

* полные сведения об учебе конкретного студента (успеваемость по всем изученным предметам всех завершенных семестров обучения с учетом пересдач);

* список задолжников по факультету с указанием несданных предметов и групп.

Сотрудник деканата должен обеспечивать:

* ввод списков студентов, зачисленных на первый курс;

* корректировку списков студентов в соответствии с приказами о зачислении, отчислении, переводе и т.п.;

* ввод учебных планов кафедр;

* ввод расписания сессии;

* ввод результатов сдачи зачетов и экзаменов на основании ведомостей и направлений.

Кроме того, сотрудник декана должен иметь возможность получать:

* справку о прослушанных студентом предметах с указанием часов и итоговых оценок;

* приложение к диплому выпускника также с указанием часов и итоговых оценок.

В результате получаем контекстную диаграмму, которая изображена на рис. 3.10 (нотации Гейна-Сарсона).

Далее детализируем процессы в системе. На рис. 3.11 представлена детализирующая диаграмма потоков данных, на которой выделены две подсистемы: Подсистема наполнения базы и Подсистема формирования отчетов, а также хранилище данных, которое может быть реализовано как с помощью средств СУБД, так и без них.

Дальнейшая детализация процессов может не выполняться, так как их сущность для разработчика очевидна. Однако становится ясно, что полная спецификация данной разработки должна включать описание базы данных. Такое описание в виде диаграммы «сущность-связь» будет рассмотрено в § 3.5.

Кроме этого, как уже упоминалось в § 3.1, целесообразно выполнить моделирование управляющих процессов в системе.

Диаграммы отношений компонентов данных

Диаграммы отношений компонентов данных в отличие от функциональных диаграмм предназначены для определения спецификаций структур данных программы.

Структурой данных называют совокупность правил и ограничений, которые отражают связи, существующие между отдельными частями (элементами) данных. Различают *абстрактные структуры* данных, используемые для уточнения связей между элементами, и *конкретные структуры*, применяемые для представления данных в программах.

Абстрактные структуры можно разделить на три группы: структуры, элементы которых не связаны между собой (множества, кортежи), структуры с неявными связями элементов (таблицы, структуры) и структуры, связь элементов которых указывается явно (графы). Очень существенно, что в реальности возможно вложение структур данных, в том числе и разных типов, поэтому для их описания могут потребоваться специальные модели. В зависимости от описываемых типов отношений, модели структур данных принято делить на иерархические и сетевые.

Иерархические модели позволяют описывать упорядоченные или неупорядоченные отношение *вхождения* элементов данных в компонент более высокого уровня, т.е. множества, таблицы и их комбинации. К иерархическим моделям относят модель Джексона-Орра, для графического представления которой могут использоваться:

* диаграммы Джексона, предложенные в составе методики проектирования ПО того же автора в 1975 г.;

* скобочные диаграммы Орра, предложенные в составе методики проектирования ПО Варнье-Орра (1974 г.).

Сетевые модели основаны на графах, а потому позволяют описывать связность взаимодействующих компонентов независимо от вида отношения, в том числе комбинации множеств, таблиц и графов. Примером сетевой модели является модель «сущность-связь» (ER – Entity-Relationship), обычно используемую при разработке баз данных.

Диаграммы Джексона и Орра. В основе диаграмм Джексона и диаграмм Орра лежит предположение о том, что структуры данных, так же как и программ, можно строить из элементов с использованием всего трех основных конструкций: последовательности, выбора и повторения.

В нотации Джексона каждая конструкция представляется в виде двухуровневой иерархии, на верхнем уровне которой расположен блок конструкции, а на нижнем – блоки элементов. Нотации конструкций различаются специальными символами в правом верхнем углу блоков элементов. В изображении последовательности символ отсутствует. В изображении выбора ставится символ «о» (латинское) – сокращение английского «или» (or). Конструкции последовательности и выбора должны содержать по два или более элементов второго уровня. В изображении повторения в блоке единственного (повторяющегося) элемента ставится символ «*».

Так схема, показанная на рис. 3.12, *а*, означает, что конструкция A состоит из элементов B, C и D, следующих в указанном порядке. Схема на рис. 3.12, *б* означает, что конструкция S состоит либо из элемента P, либо из элемента Q, либо из элемента R. Схема, изображенная на рис. 3.12, *в*, показывает, что конструкция I состоит из 0 или более элементов X.

В том случае, если необходимо показать, что конструкция повторения должна включать 1 или более элементов, используют комбинацию из двух структур последовательности и повторения (рис. 3.13).

Отличие диаграмм Орра – в нотации. Автор предлагает для представления основных конструкций данных использовать фигурные скобки (рис. 3.14).

Пример 3.4. Рассмотрим описание структуры данных файла «Электронная ведомость», содержащего сведения о сдаче экзаменов студентами. Файл состоит из записей о результатах сдачи сессии студентами одной группы. Он имеет следующую структуру: номер группы, затем записи об успеваемости студентов (ФИО студента, затем название предмета и оценка, полученная студентом, в завершении записи специальный символ «конец записи») и специальный символ «конец файла».

На рис. 3.15 показано, как выглядит описание данной структуры с использованием диаграммы Джексона.

Сетевая модель данных. Сетевые модели данных используют в тех случаях, если отношение между компонентами данных не исчерпываются включением. Для графического представления разновидностей этой модели используют несколько нотаций. Наиболее известными являются: нотация П. Чена, нотация Р. Баркера и нотация IDEF1 (более современный вариант этой нотации – IDEFIX используется в CASE-системах, например, ERWin).

Нотация Баркера является наиболее распространенной. Далее в настоящем разделе будем придерживаться именно этой нотации.

Базовыми понятиями сетевой модели данных являются: сущность, атрибут и связь.

Сущность – реальный или воображаемый объект, имеющий существенное значение для рассматриваемой предметной области. Каждая сущность должна:

* иметь уникальное имя,

* обладать одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через связь;

* обладать одним или несколькими атрибутами, которые однозначно идентифицируют каждый экземпляр сущности.

Сущность представляет собой множество экземпляров реальных или абстрактных объектов (людей, событий, состояний, предметов и т.п.). Имя сущности должно отражать тип или класс объекта, а не его конкретный экземпляр (Аэропорт, а не Внуково).

На диаграмме в нотации Баркера сущность изображается прямоугольником, иногда, с закругленными углами (рис. 3.16, *а*).

Каждая сущность обладает одним или несколькими атрибутами. *Атрибут* – любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности (рис. 3.16, *б*).

В сетевой модели атрибуты ассоциируются с конкретными сущностями, и, соответственно, экземпляр сущности должен обладать единственным определенным значением для ассоциированного атрибута. Атрибут, таким образом, представляет собой некоторый тип характеристик или свойств, ассоциированных с множеством реальных или абстрактных объектов. Экземпляр атрибута – определенная характеристика конкретного экземпляра сущности. Он определяется типом характеристики и ее значением, называемым значением атрибута.

Атрибуты делятся на ключевые, т.е. входящие в состав уникального идентификатора, который называют первичным ключом, и описательные – прочие.

Первичный ключ – это атрибут или совокупность атрибутов и/или связей, предназначенная для уникальной идентификации каждого экземпляра сущности (совокупность признаков, позволяющих идентифицировать объект). Ключевые атрибуты помещают в начало списка и помечают символом «#» (рис. 3.16, *в*).

Описательные атрибуты могут быть обязательными или необязательными. Обязательные атрибуты для каждой сущности всегда имеют конкретное значение, необязательные – могут быть не определены. Обязательные и необязательные описательные атрибуты помечают символами «*» и «о» соответственно.

Для сущностей определено понятие *супертип* и *подтип*. Супертип – сущность, обобщающая некую группу сущностей (подтипов). Он характеризуется общими для подтипов атрибутами и отношениями. Например, супертип «учащийся» обобщает подтипы «школьник» и «студент» (рис. 3.17).

Связь – поименованная ассоциация между двумя или более сущностями, значимая для рассматриваемой предметной области. Связь, таким образом, означает, что каждый экземпляр одной сущности ассоциирован с произвольным (в том числе и нулевым) количеством экземпляров второй сущности и наоборот. Если любой экземпляр одной сущности связан хотя бы с одним экземпляром другой сущности, то связь является обязательной (рис. 3.18, *а*). Необязательная связь представляет собой условное отношение между сущностями (рис. 3.18, *б*).

Каждая сущность может обладать любым количеством связей с другими сущностями модели. Связь предполагает некоторое отношение сущностей, которое характеризуется количеством экземпляров сущности, участвующих в связи с каждой стороны. Различают три типа отношений: 1*1 – «один-к-одному»; 1*n – «один-ко-многим»; n*m – «многие-ко-многим» (рис. 3.19).

Кроме того сущности бывают независимые, зависимые и ассоциированные сущности.

Независимая сущность представляет независимые данные, которые всегда присутствуют в системе. Они могут быть, как связаны с другими сущностями, так и нет.

Зависимая сущность представляет данные, зависящие от других сущностей системы, поэтому она всегда должна быть связана с другими сущностями.

Ассоциированная сущность представляет данные, которые ассоциируются с отношениями между двумя и более сущностями. Обычно данный вид сущностей появляется в модели для разрешения отношения «многие-ко-многим» (рис. 3.20).

Если экземпляр сущности полностью идентифицируется своими ключевыми атрибутами, то говорят о *полной идентификации сущности*. В противном случае идентификация сущности осуществляется с использованием атрибутов связанной сущности, что указывается черточкой на линии связи (рис. 3.21).

Пример 3.5. Рассмотрим структуру базы данных для системы учета успеваемости студентов. Основными сущностями для решения указанной задачи являются:

* «Студент»;

* «Предмет» (изучаемый учебный курс).

Отношение между ними относится к типу «многие-ко-многим». Для разрешения этого отношения введем ассоциированную сущность «Экзамен/Зачет», которая отражает текущее выполнение предметов учебного плана студентом.

Предметы, которые изучает и по которым отчитывается студент, запланированы кафедрой в учебном плане. Учебный план включает список предметов каждого семестра (сущность «Семестр»).

Для получения справок различного рода потребуются сущности, определяющие структуру организации:

* «Факультет»;

* «Курс» (как совокупность студентов, поступивших в институт в одном году);

* «Кафедра»;

* «Группа».

Для определения момента времени, начиная с которого отсутствие положительных результатов сдачи экзамена следует считать задолженностью, необходимо хранить даты экзаменов для каждой группы (сущность «Дата экзамена»).

На рис. 3.22 показаны основные отношения между указанными сущностями.

На следующем шаге определяем атрибуты каждой сущности и уточняем их типы (атрибуты, используемые для дополнительной идентификации сущности другой сущностью, не указаны, так как они описаны в соответствующей сущности).

Факультет:

* DepID – уникальное имя факультета (ключевое поле);
* DepName – название факультета.

Курс:

* CursID – уникальное имя кафедры (ключевое поле);
* EnterYear – год начала обучения для большинства студентов курса.

Кафедра:

* SpecID – уникальное имя кафедры (ключевое поле);
* SpecName – название кафедры.

Семестр:

* SemestrID – уникальное имя семестра обучения на конкретной кафедре (ключевое поле);
* SemName – название семестра обучения на кафедре.

Группа:

* GroupID – уникальное имя группы (ключевое поле);
* GroupName – название группы.

Предмет:

* SubjectID – уникальное имя предмета (ключевой атрибут);
* SubjectName – название предмета;
* ExamKind – вид оценки знаний (необязательный атрибут): экзамен/зачет/экзамен+зачет.

Дата экзамена:

* Date – дата экзамена;
* AudNumber – номер аудитории.

Студент:

* StudentID – уникальное имя студента (ключевое поле);
* SerName – фамилия;
* FirstName – имя;
* SecondName – отчество;
* StEnterYear – год поступления в институт.

Экзамен/Зачет:

* Date – дата сдачи экзамена или зачета;
* ExemType – тип (экзамен или зачет);
* Mark – оценка.

Полученная диаграмма «сущность-связь» приведена на рис. 3.23.

Данная диаграмма должна быть проверена с точки зрения возможности получения всех справок, указанных в техническом задании или показанных на диаграмме потоков данных разрабатываемой системы (рис. 3.10).

4. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРИ СТРУКТУРНОМ ПОДХОДЕ

Разработка структурной и функциональной схем

Процесс проектирования сложного ПО начинают с уточнения его структуры, т.е. определения структурных компонентов и связей между ними. Результат уточнения структуры может быть представлен в виде структурной и/или функциональной схем.

Структурная схема разрабатываемого ПО. *Структурной* называют схему, отражающую *состав и взаимодействие по управлению* частей разрабатываемого ПО.

Самый простой вид ПО – программа в качестве структурных компонентов может включать только подпрограммы и библиотеки ресурсов. Разработка структурной схемы программы обычно выполняется методом пошаговой детализации (см. § 4.2).

Структурными компонентами программной системы или программного комплекса могут служить программы, подсистемы, базы данных, библиотеки ресурсов и т. п. Так структурная схема программной системы, как правило, показывает наличие подсистем или других структурных компонентов (рис. 4.1).

Более полное представление о проектируемом ПО с точки зрения взаимодействия его компонентов между собой и с внешней средой дает функциональная схема.

Функциональная схема. *Функциональная схема* или *схема данных* (ГОСТ 19.701–90) – схема взаимодействия компонентов ПО с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств. Для изображения функциональных схем используют специальные обозначения, установленные стандартом.

Функциональные схемы более информативны, чем структурные. Так функциональные схемы программных комплексов и систем наглядно демонстрируют различие между ними (рис. 4.2).

Все компоненты структурных и функциональных схем должны быть описаны. При структурном подходе особенно тщательно необходимо прорабатывать спецификации межпрограммных интерфейсов, так как от качества их описания зависит количество самых дорогостоящих ошибок. К самым дорогим при структурном подходе относятся ошибки, обнаруживаемые при комплексном тестировании, так как для их устранения могут потребоваться серьезные изменения уже отлаженных текстов.

Использование метода пошаговой детализации для проектирования структуры программного обеспечения

Структурный подход предлагает осуществлять декомпозицию программ методом пошаговой детализации. Результат декомпозиции – структурная схема программы – представляет собой многоуровневую иерархическую схему взаимодействия подпрограмм по управлению. Минимально такая схема отображает два уровня иерархии, т.е. показывает общую структуру программы. Однако тот же метод позволяет получить структурные схемы с большим количеством уровней.

Метод пошаговой детализации реализует нисходящий подход и базируется на основных конструкциях структурного программирования. Он предполагает пошаговую разработку алгоритма. Каждый шаг при этом включает разложение функции на подфункции. Так на первом этапе описывают решение поставленной задачи, выделяя общие подзадачи. На следующем аналогично описывают решение подзадач, формулируя уже подзадачи следующего уровня. Таким образом, на каждом шаге происходит уточнение функций проектируемого ПО. Процесс продолжают, пока не доходят до подзадач, алгоритмы решения которых очевидны.

Декомпозируя программу методом пошаговой детализации следует придерживаться основного п р а в и л а структурной декомпозиции, следующего из принципа вертикального управления: в первую очередь детализировать *управляющие процессы* декомпозируемого компонента, оставляя уточнение операций с данными напоследок.

Кроме этого целесообразно придерживаться следующих рекомендаций:

* не отделять операции инициализации и завершения от соответствующей обработки, так как модули инициализации и завершения имеют плохую связность (временную) и сильное сцепление (по управлению);

* не проектировать слишком специализированных или слишком универсальных модулей, так как проектирование излишне специальных модулей увеличивает их количество, а проектирование излишне универсальных модулей – увеличивает их сложность;

* избегать дублирования действий в различных модулях, так как при их изменении исправления придется вносить во все места, где они выполняются – в этом случае целесообразно просто реализовать эти действия в отдельном модуле;

* группировать сообщения об ошибках в один модуль по типу библиотеки ресурсов, тогда будет легче согласовать формулировки, избежать дублирования сообщений, а также перевести сообщения на другой язык.

При этом, описывая решение каждой задачи, желательно использовать не более одной-двух структурных управляющих конструкций, таких как цикл-пока или ветвление, что позволяет четче представить себе структуру организуемого вычислительного процесса.

Пример 4.1. Разработать алгоритм программы построения графиков функций одной переменной на заданном интервале изменения аргумента $[x_1, x_2]$ при условии непрерывности функции на всем интервале определения.

В общем виде задача построения графика функции ставится как задача отображения реального графика (рис. 4.3, *а*), выполненного в некотором масштабе, в соответствующее изображение в окне на экране (рис. 4.3, *б*).

Для того чтобы построить график необходимо задать функцию, интервал изменения аргумента [x1, x2], на котором функция непрерывна, количество точек графика n, размер и положение окна экрана, в котором необходимо построить график: wx1, wy1, wx2, wy2 и количество линий сетки по горизонтали и вертикали nlx, nly. Значения wx1, wy1, wx2, wy2, nlx, nly можно задать, исходя из размера экрана, а интервал и число точек графика надо вводить.

Разработку алгоритма выполняем методом пошаговой детализации, используя для его документирования псевдокод.

Примем, что программа будет взаимодействовать с пользователем через традиционное иерархическое меню, которое содержит пункты: «Формула», «Отрезок», «Шаг», «Вид результата» и «Выход». Для каждого пункта этого меню необходимо реализовать сценарий, предусмотренный в техническом задании.

Шаг 1. Определяем структуру управляющей программы, которая для нашего случая реализует работу с меню через клавиатуру:

Программа.

Инициализировать глобальные переменные

Вывести заголовок и меню

Выполнять

Если *выбрана Команда*

то *Выполнить Команду*

иначе *Обработать нажатие клавиш управления*

Все-если

до *Команда=Выход*

Конец.

Очистка экрана, вывод заголовка и меню, а также выбор Команды – операции сравнительно простые, следовательно, их можно не детализировать.

Шаг 2. Детализируем операцию Выполнить команду:

Выполнить Команду:

Выбор *Команда*

Функция:

Ввести или выбрать формулу Fun

Выполнить разбор формулы

Отрезок:

Ввести значения x1,x2

Шаг:

Ввести значение h

Вид результата:

Ввести вид_результата

Выполнить:

Рассчитать таблицу значений функции.

Если *Вид_результата=График*

то Построить график

иначе *Вывести таблицу*

все-если

Все-выбор

Определим, какие фрагменты имеет смысл реализовать в виде подпрограмм. Во-первых, фрагмент Вывод заголовка и меню, так как это достаточно длинная линейная последовательность операторов и ее выделение в отдельную процедуру позволит сократить управляющую программу. Во-вторых, фрагменты Разбор формулы, Расчет значений функции, Построение графика и Вывод таблицы, так как это достаточно сложные операции. Это – подпрограммы первого уровня, которые в основном определяют структуру программы (рис. 4.4).

Определим для этих подпрограмм интерфейсы по данным с основной программой, т.е. в данном случае списки параметров.

Подпрограмма Вывод заголовка и меню параметров не имеет.

Подпрограмма Разбор формулы должна иметь два параметра: Fun – аналитическое задание функции, Tree – возвращаемый параметр – адрес дерева разбора.

Подпрограмма Расчет Значений функции должна получать адрес дерева разбора Tree, отрезок x1 и x2, а также шаг h. Обратно в программу она должна возвращать таблицу значений функции X(n) и Y(n), где n – количество точек функции.

Подпрограммы Вывода таблицы и Построения графика должны получать таблицу значений функции и количество точек.

После уточнения имен переменных алгоритм основной программы будет выглядеть следующим образом:

Программа.

Вывод заголовка и меню

Выполнять

Если *выбрана Команда*

то

Выбор *Команда*

Функция:

Ввести или выбрать формулу Fun

Разбор формулы (Fun; Var Tree)

Отрезок:

Ввести значения x1,x2

Шаг:

Ввести значения h

Вид результата:

Ввести Вид_результата

Выполнить:

Расчет таблицы(x1,x2,h,Tree; Var X, Y, n)

Если *Вид_результата=График*

то Построение графика(X, Y, n)

иначе Вывеод таблицы(X, Y, n)

все-если

Все-выбор

иначе *Обработать нажатие клавиш управления*

Все-если

до *Команда=Выход*

Конец.

На следующих шагах необходимо выполнить детализацию алгоритмов подпрограмм. Детализацию выполняют, пока алгоритм программы не станет полностью понятен. Один из возможных вариантов полной структурной схемы данной программы показан на рис. 4.5.

Использование метода пошаговой детализации при проектировании алгоритмов программ обеспечивает высокий уровень технологичности разрабатываемого ПО, так как метод позволяет использовать только структурные способы передачи управления.

Разбиение на модули при данном виде проектирования выполняется эвристически, исходя из рекомендуемых размеров модулей (20-60 строк) и сложности структуры (две-три вложенных управляющих конструкции). Определяющую роль при разбиении программы на модули играют принципы обеспечения технологичности модулей.

Для анализа технологичности полученной иерархии модулей целесообразно использовать структурные карты Константайна или Джексона.

Структурные карты Константайна

На структурной карте отношения между модулями представляют в виде графа, вершинам которого соответствуют модули и общие области данных, а дугам – межмодульные вызовы и обращения к общим областям данных.

Различают четыре типа вершин:

* модуль – подпрограмма;

* подсистема – программа;

* библиотека – совокупность подпрограмм, размещенных в отдельном модуле;

* область данных – специальным образом оформленная совокупность данных, к которой возможно обращение извне.

Обозначения соответствующих вершин приведены на рис. 4.6. (Обозначения даны в соответствии со стандартами IBM, ISO и ANSI.)

Если стрелка, изображающая вызов касается блока, то обращение происходит к модулю целиком, а если входит в блок, то – к элементу внутри модуля.

При необходимости на структурной карте можно уточнить особые условия вызова: циклический вызов (рис. 4.7, *а*), условный вызов (рис. 4.7, *б*) и однократный вызов – при повторном вызове основного модуля однократно вызываемый модуль не активизируется (рис. 4.7, *в*).

Связи по данным и управлению обозначают стрелками, параллельными дуге вызова, причем направление стрелки указывает направление связи (рис. 4.8).

Структурные карты Константайна позволяют наглядно представить результат декомпозиции программы на модули и оценить ее качество, т.е. соответствие рекомендациям структурного программирования.

Пример 4.2. Представим в виде структурной карты Константайна полную структурную схему, полученную в предыдущем примере (рис. 4.5).

Подпрограммы Очистка окна, Вывод прямоугольника, Вывод строки текста, Вывод отрезка прямой, Задание цвета рисования и Задания цвета фона являются частью библиотеки графических примитивов практически в любой среде программирования универсального языка, поэтому их включать в структурную карту не будем.

Для остальных подпрограмм покажем особые условия вызова и типы связей (рис. 4.9).

Модули Расчет значений функции, Вывод таблицы и Построение графика связаны с основной программой по образцу, так как параметры X и Y структурные (массивы), следовательно, программа считается сцепленной по образцу.

Примечание. Анализ показывает, что количество сцеплений по образцу в программе можно уменьшить, если подпрограмму Расчет значений функции перенести на следующий уровень и вызывать из функций Вывод таблицы и Построение графика. Однако в этом случае мы при смене вида результата лишний раз будем считать таблицу значений.

Аналогично можно перенести Разбор формулы на более низкий уровень и вызывать ее, например, из подпрограммы Расчет значений функции, но поскольку велика вероятность многократного расчета значений одной функции на разных интервалах, вряд ли это целесообразно.

После внесения соответствующих изменений в алгоритм следует определить полную спецификацию модулей. Спецификация должна включать: имя, краткое описание назначения, перечень входных и выходных параметров с указанием типа и области до-

пустимых входных и выходных значений. Затем можно приступать к реализации модулей.

В соответствии с требованиями нисходящей разработки (комбинированный подход) можно предложить следующий порядок реализации модулей: Основная программа – Вывод окна с текстом – Вывод заголовка и меню – Разбор формулы – Расчет значений функции – Вывод таблицы – Построение графика.

В завершении этого этапа необходимо убедиться, что разработанная иерархия корректно реализует спецификации проектируемой системы.

5. АНАЛИЗ ТРЕБОВАНИЙ И ОПРЕДЕЛЕНИЕ СПЕЦИФИКАЦИЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРИ ОБЪЕКТНОМ ПОДХОДЕ

UML – стандартный язык описания разработки программных продуктов с использование объектного подхода

Модели разрабатываемого ПО при объектном подходе основаны на предметах и явлениях реального мира. В основе всех моделей лежит описание требуемого *поведения* разрабатываемого ПО, т.е. его функциональности, но это поведение связывается с *состояниями* элементов (объектов) конкретной предметной области.

Таким образом, на этапе анализа ставятся две задачи:

* уточнить требуемое поведение разрабатываемого ПО;
* разработать концептуальную модель его предметной области с точки зрения поставленных задач.

В основе объектного подхода к разработке ПО лежит *объектная декомпозиция*, т.е. представлении разрабатываемого ПО в виде совокупности объектов, в процессе взаимодействия которых посредством передачи сообщений и происходит выполнение требуемых функций (рис. 5.1).

Разрабатываемое с помощью объектного подхода программное обеспечение, как правило, очень сложно, поэтому для описания разработки в настоящее время используют специальный язык – универсальный язык моделирования UML. Этот язык, авторами которого являются Г. Буч, Д. Рамбо и А. Джакобсон [1], недавно был признан стандартным средством описания объектных разработок.

Полное описание разработки с использованием UML включает несколько моделей, характеризующих определенный аспект проектируемой системы (рис. 5.2):

- *модель использования* – представляет собой описание функциональности ПО с точки зрения пользователя;
- *логическая модель* – описывает ключевые абстракции ПО (классы, интерфейсы, и т.п.), т.е. средства, обеспечивающие требуемую функциональность;
- *модель реализации* – определяет реальную организацию программных модулей;
- *модель процессов* – отображает организацию вычислений и оперирует понятиями «процессы» и «нити». Она позволяет оценить производительность, масштабируемость и надежность ПО;
- *модель развертывания* – показывает особенности размещения программных компонентов на конкретном оборудовании.

Всего UML предлагает девять дополняющих друг друга диаграмм, входящих в различные модели:

* диаграммы вариантов использования;
* диаграммы классов;
* диаграммы пакетов:
* диаграммы последовательностей действий;
* диаграммы кооперации:
* диаграммы деятельностей:
* диаграммы состояний объектов:
* диаграммы компонентов:
* диаграммы размещения.

Полная спецификация разрабатываемого ПО, помимо указанных диаграмм, как и при структурном подходе, обязательно включает словарь терминов и различного рода описания и текстовые спецификации. Конкретный набор документации определяется разработчиком.

UML и предлагаемая теми же авторами методика разработки ПО названная Rational Unified Process поддерживаются пакетом **Rational Rose** фирмы Rational Software Corporation. Ряд диаграмм можно получить также средствами программы Microsoft Visual Modeler и других CASE-средств.

Определение вариантов использования

Разработку спецификаций начинают с анализа требований к функциональности, указанных в техническом задании. В процессе этого анализа выявляют внешних поль-

зователей разрабатываемого ПО и перечень отдельных аспектов его поведения в процессе взаимодействия с конкретными пользователями. Аспекты поведения ПО были названы «вариантами использования» или «прецедентами» (use cases).

Вариант использования представляет собой характерную процедуру применения разрабатываемой системы конкретным действующим лицом, в качестве которого могут выступать не только люди, но и другие системы и устройства.

Не следует путать вариант использования с конкретными операциями будущей системы. Каждый вариант использования связан с некоторой целью, имеющей самостоятельное значение, например, для текстового редактора Формирование оглавления – это вариант использования, а Связывание заголовков со специальными стилями – операция, которую необходимо выполнить, чтобы стало возможно автоматическое построение оглавления.

В зависимости от цели выполнения процедуры различают следующие варианты использования:

* основные – обеспечивают требуемую функциональность разрабатываемого ПО;

* вспомогательные – обеспечивают выполнение необходимых настроек системы и ее обслуживание (например, архивирование информации и т.п.);

* дополнительные – обеспечивают дополнительные удобства для пользователя (как правило, реализуются в том случае, если не требуют серьезных затрат каких-либо ресурсов ни при разработке, ни при эксплуатации).

В зависимости от уровня абстракции вариант использования может описываться кратко или более подробно. Краткая форма описания содержит: название варианта использования, его цель, действующих лиц, тип варианта использования (основная, второстепенная или дополнительная) и его краткое описание. Ниже приведено краткое описание варианта использование Выполнение задания системы решения комбинаторно-оптимизационных задач. Предполагается, что эта система должна обеспечивать возможность решения нескольких комбинаторно-оптимизационных задач на графах (задачи коммивояжера, задачи определения кратчайшего пути и т.п.) разными алгоритмами и хранить в базе данных исходные данные и результаты.

Название варианта	***Выполнение задания***
Цель	*Получение результатов решения задачи*
Действующие лица	*Пользователь*

Краткое описание	*Решение задачи предполагает выбор задачи, выбор алгоритма, задание данных и получение результатов решения.*
Тип	*Основной*

Основные варианты использования обычно описывают подробно, стараясь отразить особенности предметной области разрабатываемого ПО. Подробная форма, кроме указанной выше информации, включает описание типичного хода событий и возможных альтернатив. Типичный ход событий представляют в виде диалога между пользователями и системой, последовательно нумеруя события. Если пользователь может выбирать варианты, то их описывают в отдельных таблицах. Также отдельно приводят альтернативы, связанные с нарушением типичного хода событий. Например:

Вариант использования *Выполнение задания*

Типичный ход событий

Действия исполнителя	Отклик системы
1. *Пользователь инициирует новое задание*	2. *Система регистрирует новое задание и предлагает список типов задач*
3. *Пользователь выбирает тип задачи.*	4. *Система регистрирует тип задачи и предлагает список способов задания данных*
5. *Пользователь выбирает способ задания данных.* *а) Если выбран ввод с клавиатуры, см. раздел Ввод данных.* *б) Если выбран ввод из базы данных, см. раздел Выбор данных из базы*	6. *Система регистрирует данные и предлагает список алгоритмов решения*
7. *Пользователь выбирает алгоритм*	8. *Система регистрирует алгоритм и предлагает начать решение*
9. *Пользователь инициирует процесс решения*	10. *Система запускает подпрограмму решения задачи*
11. *Пользователь ожидает*	12. *Система демонстрирует результаты и предлагает сохранить их в базе*

13. *Пользователь анализирует результаты и выбирает, сохранять их в базе или нет*	14. *Если выбрано сохранение данных, то система выполняет запись данных задания в базу*
	15. *Система переходит в состояние ожидания*

Альтернатива:

10. *Если время выполнения программы с точки зрения пользователя велико, то он прерывает процесс выполнения.*

11. *Система прерывает расчеты, предлагает список алгоритмов решения и возвращается на шаг 7.*

Дополнительная информация.

1. *Необходимо обеспечить произвольную последовательность выбора типа задачи, данных и алгоритма*.

2. *Необходимо обеспечить возможность выхода из варианта на любом этапе.*

Раздел ***Ввод данных***

Типичный ход событий

Действия исполнителя	Отклик системы
1. *Пользователь выбрал Ввод данных.*	2. *Система последовательно запрашивает ввод данных.*
3. *Пользователь вводит данные.*	4. *Система проверяет данные и запрашивает, сохранять ли данные в базе.*
5. *Пользователь отвечает на запрос.*	6. *Если выбран вариант сохранения данных, то система выполняет запись данных в базу и регистрирует их в текущем задании.*

Альтернатива:

4. *Если обнаружены некорректные данные, то система выдает сообщение об ошибке с диагностикой и предлагает их исправить, возвращаясь на предыдущий шаг.*

Раздел ***Выбор данных из базы***

Типичный ход событий

Действия исполнителя	Отклик системы
1. *Пользователь выбрал Выбор данных из базы*	2. *Система демонстрирует список данных в базе.*
3. *Пользователь выбирает данные.*	4. *Система читает данные и регистрирует их в текущем задании.*

Диаграммы вариантов использования. Диаграммы вариантов использования позволяют наглядно представить ожидаемое поведение системы.

Основными понятиями диаграмм вариантов использования являются: действующее лицо, вариант использования, связь.

Действующее лицо – внешняя по отношению к разрабатываемому ПО сущность, которая взаимодействует с ним с целью получения или предоставления какой-либо информации. Как уже упоминалось выше, действующими лицами могут быть пользователи, другое ПО или какие-либо технические средства, взаимодействующее с разрабатываемым ПО.

Вариант использования – некоторая очевидная для действующего лица процедура, решающая его конкретную задачу. Все варианты использования, так или иначе, связаны с требованиями к функциональности разрабатываемой системы и могут сильно отличаться по объему выполняемой работы.

Связь – взаимодействие действующих лиц и соответствующих вариантов использования.

Варианты использования также могут быть связаны между собой. При этом фиксируют связи использования и расширения.

Использование подразумевает, что существует некоторый фрагмент поведения разрабатываемого ПО, который повторяется в нескольких вариантах использования. Этот фрагмент оформляют, как отдельный вариант использования и указывают связь с ним типа «использование».

Расширение применяют, если имеется два подобных варианта использования, различающиеся наличием в одном из них некоторых дополнительных действий. В этом

случае дополнительные действия определяют как отдельный вариант использования, который связан с основным вариантом связью типа «расширение».

На рис. 5.3 приведены условные обозначения, которые применяют при изображении диаграмм вариантов использования.

Пример 5.1. Построить диаграмму вариантов использования для системы решения комбинаторно-оптимизационных задач.

Действующее лицо у данной системы одно – Пользователь, который по сути дела обращается к системе либо для решения новой задачи, либо для просмотра результатов ранее решенной задачи, которые должны сохраняться в базе данных. Представим эти варианты использования на диаграмме.

Вариант Выполнение задания на самом деле включает несколько вариантов, которые различаются способом определения данных (ввод с клавиатуры или чтение из базы), и сохранением введенных данных в базе. Изобразим эти варианты на схеме, указав соответствующие расширения данного варианта.

Помимо двух основных вариантов использования система должна также предусматривать вспомогательные варианты использования для удаления лишних данных и результатов из базы.

Полученная диаграмма показана на рис. 5.4.

Естественно все варианты использования определить, как правило, не удается: новые варианты фиксируют постоянно, даже в процессе эксплуатации. Но, чем больше вариантов выявлено в процессе уточнения спецификаций, тем лучше, так как при этом мы получаем более точную модель предметной области, что уменьшает вероятность ее пересмотра при добавлении функций.

Построение концептуальной модели предметной области

Диаграммы классов – центральное звено объектно-ориентированных методов разработки ПО, поэтому все существующие методы используют диаграммы классов в одной из известных нотаций. Однако в основном диаграммы классов в этих методах применяют на этапе проектирования для того, чтобы показать особенности построения конкретных классов. В отличие от ранее существующих нотаций UML предлагает использовать три уровня диаграмм классов в зависимости от степени их детализации:

* *концептуальный уровень*, на котором диаграммы классов, называемые в этом случае *контекстными*, демонстрируют связи между основными *понятиями* предметной области;

* *уровень спецификаций*, на котором диаграммы классов отображают *интерфейсы* классов предметной области, т.е. связи объектов этих классов;

* *уровень реализации*, на котором диаграммы классов непосредственно показывают поля и методы конкретных *классов*.

Практически это три разных модели, связь между которыми неоднозначна. Так, если концептуальная модель определяет некоторое понятие предметной области как класс, то это не означает, что для реализации этого понятия будет использован отдельный класс. Однако во всех трех моделях нас интересуют типы объектов (классы) и их статические отношения, что позволяет использовать единую нотацию.

Каждая из перечисленных моделей используется на конкретном этапе разработки ПО:

* концептуальную модель – на этапе анализа;

* диаграммы классов уровня спецификации – на этапе проектирования;

* диаграммы классов уровня реализации – на этапе реализации.

Концептуальные модели в соответствии с определением оперируют: понятиями предметной области, атрибутами этих понятий и отношениями между ними. Понятию в предметной области разрабатываемого ПО могут соответствовать как материальные предметы, так и абстракции, которые применяют специалисты предметной области.

Основным понятиям в модели ставится в соответствие класс. *Класс* при этом традиционно понимают как совокупность общих признаков некоторой группы объектов предметной области. В соответствии с этим определением на диаграмме классов каждому классу соответствует группа объектов, общие признаки которых и фиксирует класс. Так класс «студент» объединяет общие признаки группы людей, обучающихся в высших учебных заведениях. Экземпляр класса или объект (например, Иванов И.И.) обязательно обладает всей совокупностью признаков своего класса и может иметь собственные признаки, не фиксированные в классе. Так, например, помимо того, что Иванов И.И. является студентом, он еще может быть спортсменом, музыкантом и т.д. Строго говоря, таким собственным признаком является и идентифицирующее студента имя.

На диаграммах класс изображается в виде прямоугольника, внутри которого указано имя класса (рис. 5.5, *а*). При необходимости допускается указывать характеристики класса, например атрибуты, используя специальные секции условного обозначения (рис. 5.5, *б*).

В качестве *атрибутов* представляют некоторые, существенные с точки зрения решаемой задачи характеристики объектов, например, идентифицирующие значения (имя, номер). Для конкретного объекта атрибут всегда имеет конкретное значение для конкретного объекта. На диаграмме классов атрибуты обычно показывают в секции атрибутов.

Под *отношением* классов понимают статическую, т.е. не зависящую от времени, связь между классами. Различают два основных вида отношений: ассоциация и обобщение.

Отношение *ассоциации* означает наличие связи между экземплярами классов или объектами, например, класс «студент» ассоциирован с классом «институт». Ассоциация может иметь имя, например, «Обучается». Рядом с именем ассоциации обычно ставят стрелку, указывающую направление чтения имени («Студент обучается в институте», а не наоборот).

Связь между экземплярами классов подразумевает некоторые *роли*, которые соответствующие объекты играют по отношению друг к другу. Роль связана с направлением ассоциации. Так по отношению к студентам институт – организация, осуществляющая их обучение, т.е. роль института можно назвать «Место учебы». Студент для института – объект обучающей деятельности института, т.е. «Обучаемый». Если роль собственного имени не имеет, то можно считать, что ее имя совпадает с именем класса, по отношению к которому определяется эта роль. Для рассматриваемого примера это соответственно роли «Студент» и «Институт» (рис. 5.6, *а*), но роль можно указать и явно (рис. 5.6, *б*).

Роль также обладает характеристикой *множественности*, которая показывает, сколько объектов может участвовать в одной связи с каждой стороны. Допускается указывать множественность:

* – от 0 до бесконечности;

<целое>.. * – от заданного числа до бесконечности;

<целое> – точно определенное количество объектов;

<целое1>, <целое2> – несколько вариантов точного количества объектов;

<целое1>..<целое2> – диапазон объектов.

Обобщением называют такое отношении между классами, при котором любой объект одного класса (*подтипа*) обязательно является также и объектом другого класса, называемого в данном контексте *супертипом*. Так, если некоторый конкретный студент Иванов И.И. является объектом подтипа Студент первого курса супертипа Студент, то тот же самый Иванов И.И. является объектом указанного супертипа. Следовательно, все, что известно об объектах супертипа (ассоциации, атрибуты, операции) касается и объектов подтипа. На диаграмме классов обобщение обозначают линией с треугольной стрелкой на конце, подходящем к супертипу (рис. 5.7).

На практике определение основных понятий предметной области, которые должны представляться на контекстной диаграмме в виде классов, является не тривиальной задачей. Обычно используют следующий способ:

* формируют множество понятий-кандидатов из существительных, характеризующих предметную область в описании вариантов использования;

* исключают понятия, не существенные для данного варианта использования, например в предыдущем примере, «информация», «ввод» и т.п.

Для определения множества понятий-кандидатов полезно использовать и перечень возможных категорий понятий-кандидатов.

Пример 5.2. Построить концептуальную модель для системы решения комбинаторно-оптимизационных задач. Множество понятий-кандидатов для данной разработки включает следующие словосочетания:

задание, тип задачи, список типов задач, способ задания данных, ввод данных, выбор данных из базы, алгоритм решения задачи, список конкретных алгоритмов решения задачи, полнота описания задания, результаты, данные, база данных.

Попробуем выделить основные понятия и связать их между собой.

Цель основного варианта использования системы – выполнение задания. Полное описание задания включает: тип задачи, данные и указание на алгоритм. С ним же будут связаны и полученные результаты. Данные могут сохраняться в базе и вводиться. Описание задания и все, что с ним связано, может сохраняться в базе.

Определим возможные обобщения:

1) способ задания данных: ввод данных, выбор данных из базы;

2) алгоритм: алгоритм решения задачи: конкретный алгоритмы решения задачи.

Переходим к построению концептуальной модели.

Основной класс-понятие, исходя из описания – Задание. Связываем с ним классы-понятия Данные, Алгоритм и Результаты.

В разрабатываемой системе планируется реализовать алгоритмы решения задач трех типов: поиск цикла минимальной длины, проходящего через все вершины; поиск кратчайшего пути и поиск минимального покрывающего дерева. Следовательно, класс-понятие Алгоритм является супертипом для классов Алгоритм поиска цикла минимальной длины, Алгоритм поиска кратчайшего пути и Алгоритм поиска минимального покрывающего дерева, от которых, в свою очередь, будут наследоваться Алгоритмы, реализующие конкретные методы. Алгоритм также связан с Данными и Результатами. Для алгоритма очень существенной характеристикой является его точность, соответственно добавим атрибут Точность.

Данные и Задания должны храниться в Базе данных, что показывают ассоциациями соответствующих классов. Способ задания данных для понимания основной концепции проектируемой системы пока не очень существенен.

Вид задачи в нашем случае, скорее атрибут класса Задание, чем самостоятельный класс, так как в реальном мире это имя, которое позволяет уточнить группу возможных алгоритмов решения и структуры исходных данных и получаемых результатов.

На рис. 5.8 показана полученная контекстная диаграмма классов.

Описание поведения. Системные события и операции

Концептуальная модель характеризует статические свойства разрабатываемого ПО. Для описания особенностей его поведения, т.е. возможных действий системы, целесообразно использовать диаграммы последовательностей системы, системные события, системные операции, диаграммы деятельностей, а при необходимости и диаграммы состояний объектов (см. § 6.4).

Диаграммы последовательностей системы. *Диаграмма последовательностей системы* – графическая модель, которая для определенного сценария варианта использования показывает генерируемые действующими лицами события и их порядок. При этом система рассматривается как единое целое.

Для построения диаграммы последовательностей системы необходимо:

* представить систему как «черный ящик» и изобразить для нее *линию жизни* – вертикальную пунктирную линию, подходящую к блоку снизу;

* идентифицировать каждое действующее лицо и изобразить для него линию жизни (много действующих лиц бывает в вариантах совместного использования ПО);

* из описания варианта использования определить множество системных событий и их последовательность;

* изобразить системные события в виде линий со стрелкой на конце между линиями жизни действующих лиц и системы, а также указать имя события и список передаваемых значений.

В отличие от внутренних событий, события, которые генерируются для системы действующими лицами, называют *системными*. Системные события инициируют выполнение соответствующего множества операций, также называемых *системными***.** Каждую системную операцию называют по имени соответствующего сообщения.

Множество всех системных операций определяют, идентифицируя системные события всех вариантов использования. Для наглядности системные операции изображают в виде операций абстрактного класса (типа) System. Если желательно разделить множество операций на подмножества, инициируемые разными пользователями, то используют несколько абстрактных классов: System1, System2 и т.д.

Каждую системную операцию необходимо описать. Обычно описание системной операции содержит:

* имя операции и ее параметры;

* описание обязанности;

* указание типа;

* названия вариантов использования, в которых она используется;

* примечания для разработчиков алгоритмов и т.д.;

* описание обработки возможных исключений;

* описание вывода неинтерфейсных сообщений;

* предположение о состоянии системы до выполнения операции (предусловие);

* описание изменения состояния системы после выполнения операции (постусловие).

Пример 5.3. Разработать диаграмму последовательностей системы для варианта использования Выполнение задания решения комбинаторно-оптимизационных задач.

Анализируем описание варианта использования и определяем, что действующее лицо должно генерировать девять системных событий, включая загрузку задания из базы, которая логически следует из операции сохранения.

Покажем эти события на диаграмме последовательностей (рис. 5.9). В скобках укажем параметры, которые должны формировать эти события

Следовательно, система должна обеспечивать выполнения соответствующих операций. Полученное множество операций приписывают классу System (рис. 5.10).

Далее каждую операцию необходимо описать. Для примера опишем операцию Инициировать решение ().

Раздел	**Описание**
Имя	Инициировать решение ().
Обязанности	Выполнить задание и вывести пользователю полученные результаты
Тип	Системная
Ссылки	Вариант использования Выполнить задание
Примечания	Предусмотреть возможность прерывания процесса решения пользователем
Исключения	1. Если в задании указаны не все исходные данные, то вывести сообщение об ошибке 2. Если при указанных исходных данных решение задачи указанным методом не возможно, то вывести сообщение об ошибке
Вывод	–
Предусловия	Предполагает наличие всех исходных данных задания
Постусловие	Получен результат

Диаграммы деятельностей. В зависимости от степени детализации диаграммы деятельностей так же, как диаграммы классов, могут использоваться на разных этапах разработки. На этапе анализа требований и уточнения спецификаций диаграммы деятельностей позволяют конкретизировать основные функции разрабатываемого ПО.

Под *деятельностью* в данном случае понимают задачу (операцию), которую необходимо выполнить вручную или с помощью средств автоматизации. Каждому варианту использования соответствует своя последовательность задач. В теоретическом плане

диаграммы деятельности – обобщенное представление алгоритма, реализующего анализируемый вариант использования. На диаграмме деятельность обозначается прямоугольником с закругленными углами (рис. 5.11, *а*).

Диаграммы деятельности позволяют описывать альтернативные и параллельные процессы. Для обозначения альтернативных процессов используют ромб (рис. 5.11, *б*), условие указывают рядом, а альтернативы «да», «нет» – рядом с соответствующими выходами. С помощью этого же блока можно построить циклический процесс. Множественность активации деятельности обозначают символом «*», помещенным рядом со стрелкой активации деятельности, и при необходимости уточняют надписью вида «для каждой строки».

Для обозначения параллельных процессов используют линейки синхронизации (рис. 5.11, *в*), причем условие синхронизации можно уточнить, указав его на диаграмме. На рис. 5.12 показано, что «Деятельность 1» и «Деятельность 2» могут выполняться параллельно.

На этапе определения спецификаций имеет смысл уточнять только те варианты использования, краткое описание которых недостаточно для понимания сущности решаемых проблем. Диаграммы деятельностей таким образом можно использовать вместо описания вариантов использования или в дополнение к ним.

Пример 5.4. Построить диаграмму деятельности, уточняющую вариант использования Выполнение задания системы решения комбинаторно-оптимизационных задач.

Разбиваем процесс на операции, учитывая описание предметной области в виде контекстной диаграммы классов, и показываем их на диаграмме деятельности (рис. 5.13).

6. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРИ ОБЪЕКТНОМ ПОДХОДЕ

Разработка структуры программного обеспечения при объектном подходе

Основная задача логического проектирования при объектном подходе – разработка классов для реализации объектов, полученных при объектной декомпозиции, что предполагает полное описание полей и методов каждого класса. Физическое проектирование при объектном подходе включает проектирование объединения классов и других

программных ресурсов в программные компоненты и размещения этих компонентов на конкретных вычислительных установках.

Большинство классов можно отнести к определенному типу, который применительно к классам называют *стереотипом*, например: классы-сущности (классы предметной области), граничные (интерфейсные) классы, управляющие классы, исключения и т.д.

Классы-сущности используют представления сущностей реального мира или внутренних элементов системы, например, структур данных. Обычно они не зависят от окружения и, соответственно, могут использоваться в различных приложениях. Для выявления классов-сущностей, как правило, изучают описания вариантов использования, концептуальную модель и диаграммы деятельностей. Полученный таким образом список классов-кандидатов фильтруют, удаляя слова, не относящиеся к предметной области, языковые выражения и т.п. Среди оставшихся отбирают классы-кандидаты, объекты которых обладают как состоянием, так и поведением.

Граничные классы обеспечивают взаимодействие между действующими лицами и внутренними элементами системы. К этому типу относятся как классы, реализующие пользовательские интерфейсы, так и классы, обеспечивающие интерфейс с техническими средствами или программными системами. Для обнаружения граничных классов изучают пары «действующее лицо – вариант использования».

Управляющие классы служат для моделирования последовательного поведения, заложенного в один или несколько вариантов использования.

Если количество классов-кандидатов и других ресурсов велико, то их целесообразно разбить на группы – пакеты.

Пакетом при объектном подходе называют совокупность описаний классов и других программных ресурсов, в том числе пакетов. Объединение в пакеты используют только для удобства создания больших проектов, количество классов в которых велико. При этом в один пакет обычно собирают классы и другие ресурсы *единого назначения*.

Диаграмма пакетов показывает, из каких частей состоит проектируемая программная система, и как эти части зависят друг от друга.

Связь между пакетами фиксируют, если изменения в одном пакете могут повлечь за собой изменения в другом. Она определяется зависимостью классов и других ресурсов, объединенных в пакет. Возможны различные виды зависимости классов, например:

- объекты одного класса посылают сообщения объектам другого класса;

- объекты одного класса обращаются к компонентам объектов другого;
- объекты одного класса используют объекты другого в списке параметров методов и т.п.

На рис. 6.1 приведены обозначения UML, которые допустимо использовать на диаграммах пакетов. Кроме указанных обозначений на диаграммах пакетов допустимо показывать обобщения (рис. 6.2), что подразумевает наличие единого интерфейса нескольких пакетов. В этом случае фиксируется связь от подтипа к супертипу.

Пример 6.1. Разработать структуру системы решения комбинаторно-оптимизационных задач.

Анализ концептуальной модели и вариантов использования позволяют выделить следующие группы классов или пакеты:

* пользовательский интерфейс – классы, реализующие объекты интерфейса с пользователем;

* библиотека интерфейсных компонентов – классы, реализующие интерфейсные компоненты: окна, кнопки, метки и т.п.;

* объекты управления – классы, реализующие сценарии вариантов использования;

* объекты задачи – классы, реализующие объекты предметной области системы;

* интерфейс с базой данных – классы, реализующие интерфейс с базой данных;

* база данных;

* базовые структуры данных – классы, реализующие внутренние структуры данных, такие как деревья, n-связные списки и т.п.;

* обработка ошибок – классы исключений, реализующие обработку нештатных ситуаций.

Последние два пакета объявим глобальными, так их элементы могут использоваться классами всех пакетов.

Предположим зависимости классов и изобразим диаграмму (рис. 6.3).

Определение отношений между объектами

После определения основных пакетов разрабатываемого ПО переходят к детальному проектированию классов, входящих в каждый пакет. Классы-кандидаты, которые предположительно должны войти в конкретный пакет, показывают на диаграмме классов этапа проектирования и уточняют отношения между объектами указанных классов..

Пример 6.2. Определить классы-кандидаты пакета Объекты задачи.

Используя рекомендации, данные в § 6.1, выполним анализ концептуальной модели предметной области (рис. 5.8), описания основного варианта использования Решение задачи (см. § 5.2) и его диаграммы деятельностей (рис. 5.13).

Список классов-кандидатов, полученный на основе данного анализа, выглядит следующим образом:

- класс Задание – объекты данного класса должны создаваться каждый раз, когда пользователь инициирует новое задание;
- семейство классов с базовым классом Алгоритм – объекты данного класса должны создаваться, когда определен алгоритм решения задачи;
- класс Данные – объекты данного класса должны создаваться при определении (вводе или выборе из базы) данных;
- класс Результаты – объекты данного класса должны создаваться при решении конкретной задачи конкретным алгоритмом с использованием конкретных данных.

Исходный вариант диаграммы классов пакета Объекты задачи показан на рис. 6.4.

Основой для проектирования классов является уточнение взаимодействия объектов этих классов в процессе реализации вариантов использования. При этом применяют диаграммы последовательности действий и диаграммы кооперации. Если же необходимо описать взаимодействие объектов при обработке конкретного сообщения, удобны именно диаграммы последовательностей.

Диаграммы последовательностей этапа проектирования. Диаграммы последовательностей этапа проектирования отображают взаимодействие объектов, упорядоченное по времени. В отличие от диаграмм последовательностей этапа анализа на ней показывают и внутренние объекты, а также последовательность сообщений, которыми обмениваются объекты в процессе реализации фрагмента варианта использования, обычно называемого *сценарием*.

Объекты изображают в виде прямоугольников, внутри которых указана информация, идентифицирующая объект: имя, имя объекта и имя класса или только имя класса (рис. 6.5).

Каждое сообщение представляют в виде линии со стрелкой, соединяющей линии жизни двух объектов. Эти линии помещают на диаграмму в порядке генерации сообщений (сверху вниз и слева направо). Сообщению приписывают имя, но можно указать аргументы и управляющую информацию, например, условие формирования или

маркер итерации (*). Возврат при передаче синхронных сообщений подразумеваются по умолчанию.

Если объект создается сообщением, то его рисуют справа от стрелки сообщения так, чтобы стрелка сообщения входила в него слева.

Диаграммы последовательностей также позволяют изображать параллельные процессы. *Асинхронные* сообщения, которые не блокируют работу вызывающего объекта, показывают половинкой стрелки (рис. 6.6, *а*). Такие сообщения могут выполнять одну из трех функций:

* создавать новую ветвь процесса;

* создавать новый объект (рис. 6.6, *б*);

* устанавливать связь с уже выполняющейся ветвью процесса.

На линии жизни для параллельных процессов дополнительно показывают *активации*, которые обозначаются прямоугольником, наложенным поверх линии жизни (рис. 6.6, *в*).

Уничтожение объекта показывают большим знаком «X» (рис. 6.6, *г*).

При необходимости линию жизни можно прервать, чтобы не уточнять обработку, не связанную с анализируемыми объектами (рис. 6.6, *д*).

Пример 6.3. Разработать диаграмму последовательностей для сценария Решение задачи (фрагмент варианта использования Выполнение задания от момента инициализации пользователем процесса решения до его завершения).

Анализ описания варианта использования показывает, что необходимо рассмотреть три варианта последовательности действий: нормальный процесс, прерывание процесса пользователем и возникновение исключения при выполнении алгоритма.

Нормальный процесс предполагает, что при выдаче команды Создать создается объект Решение, управляющий данным сценарием. Следующее сообщение Начать активизирует этот объект. Объект Решение запрашивает у объекта класса Задание тип объекта Алгоритм, создает объект требуемого класса и активизирует его, сохраняя способность получать и обрабатывать сообщения (параллельный процесс).

Объект класса Алгоритм, реализующий метод, запрашивает у объекта класса Задание данные и начинает обработку, используя вспомогательные объекты. Нормально завершив обработку, объект класса Алгоритм, реализующий метод, передает объекту класса Задание результаты и возвращает объекту Решение признак нормального завершения. Объект Решение уничтожает объект класса Алгоритм, реализующий метод, и

возвращает вызвавшему его объекту признак нормального завершения решения (рис. 6.7, *а*).

В случае прерывания процесса объект Решение прерывает процесс решения, уничтожает объект Алгоритм и возвращает признак прерванного выполнения (рис. 6.7, *б*).

В последнем случае при выполнении обработки возникает аварийная ситуация, результатом которой является генерация исключения. Обрабатывая исключение, объект класса Решение, выдает соответствующее сообщение пользователю, уничтожает объект класса Алгоритм, реализующий метод, и возвращает признак завершения с ошибкой (рис. 6.8).

Диаграммы кооперации. Диаграммы кооперации – это альтернативный способ представления взаимодействия объектов в процессе реализации сценария. В отличие от диаграмм последовательностей действий диаграммы кооперации показывают потоки данных между объектами классов, что позволяет уточнить связи между объектами.

Пример 6.4. Разработать диаграммы кооперации для сценария Процесс решения. Изобразим на одной диаграмме все три случая, которые возможны при реализации сценария (рис. 6.9), нумеруя сообщения в порядке их возможной генерации.

Такое представление позволяет описать потоки данных, передаваемых между объектами классов Решение, Задание и Алгоритм, реализующий метод, для сценария Процесс решения.

Уточнение отношений классов

Процесс проектирования классов начинают с уточнения отношений между ними. На этапе проектирования помимо ассоциации и обобщения различают еще два типа отношения между классами: агрегацию и композицию.

К сожалению, до настоящего времени не существует единой устоявшейся терминологии объектно-ориентированного проектирования. В табл. 6.1 приведены соответствия между основными терминами, используемыми наиболее известными авторами в этой области.

Таблица 6.1

UML	**Класс**	**Ассоциация**	**Обобщение**	**Агрегация**
Буч	Класс	Использование	Наследование	Включение
Коад	Класс,	Связь экземпляров	Обобщение-	Часть-целое

	объект		специализа-ция	
Якобсон	Объект	Ассоциация родст-ва	Наследование	Состоит из
Оделл	Тип объекта	Связь	Подтип	Композиция
Рамбо	Класс	Ассоциация	Обобщение	Агрегация
Шле-ер/Меллор	Объект	Связь	Подтип	Не опреде-лена

Агрегацией называют ассоциацию между целым и его частью или частями. Агрегацию вместо ассоциации указывают, если отношение «целое-часть» в конкретном случае существенно. Например, если колесо нас интересует только как часть автомобиля, то между соответствующими классами целесообразно указать отношении агрегации, а если колесо – товар, так же как и автомобиль, то связь целое часть не существенна.

Композиция – более сильная разновидность агрегации, которая подразумевает, что объект-часть может принадлежать только единственному целому. Объект-часть при этом создается и уничтожается только вместе со своим целым.

Уточненные отношения между классами фиксируют на диаграмме классов. Для этого используют специальные уловные обозначения (рис. 6.10).

Поскольку отношение ассоциации и его подвиды: агрегация и композиция означают наличие обмена сообщениями между объектами классов, целесообразно уточнить направление передачи сообщений. Н*авигацию* (направление ассоциации) показывают стрелкой на конце линии ассоциации. Если стрелки указаны с обеих сторон, то это означает *двунаправленную* ассоциацию.

Специальное обозначение на диаграмме классов этапа проектирования используют для указания абстрактных классов и методов: на диаграмме классов их имена выделяют курсивом, либо перед именем класса указывают стереотип «abstract».

UML также включает специальную нотацию для обозначения параметризованных классов или шаблонов (рис. 6.11).

Получение из класса-шаблона класса с конкретными типами элементов называют *связыванием*. Связывание можно обозначить двумя способами, явно указав тип-параметр (рис. 6.12, *а*) и используя условное обозначение *уточнения* (рис. 6.12, *б*).

Диаграммы классов позволяют также отобразить *ограничения*, которые невозможно показать, используя только понятия, рассмотренные выше (ассоциации, обобщения, атрибуты, операции). Например, показать, что средний балл студентов должен определяться по соответствующей формуле. Подобную информацию на диаграмме классов можно представить в виде записи на естественном языке или в виде математической формулы, поместив их в фигурные скобки.

Особое место в процессе проектирования классов занимает проектирование интерфейсов.

Интерфейсы. *Интерфейсом* в UML называют класс, содержащий только объявление операций. Отдельное описаний интерфейсов улучшает технологические качества ПО. Интерфейсы широко применяют при разработке сетевого ПО, которое должно функционировать в гетерогенных средах, а также для организации взаимодействия с базами данных и т.п., так как механизм полиморфного наследования позволяет создавать различные реализации одного и того же интерфейса.

С точки зрения теории объектно-ориентированного программирования интерфейс представляет собой особый вид абстрактного класса, отличающийся тем, что он не содержит методов, реализующих указанные операции, и объявления полей. Другими словами абстрактные классы позволяют определить реализацию некоторых методов, а интерфейсы требуют отложить определение всех методов.

На диаграмме классов интерфейс можно показать двумя способами: с помощью специального условного обозначения (рис. 6.13, *а*) или, объявив для класса стереотип «interface» (рис. 6.13, *б*).

Реализацию интерфейса также можно показать двумя способами: сокращенно (рис. 6.14, *а*) или, используя отношение реализации (рис. 6.14, *б*).

Для остальных классов, ассоциированных с интерфейсом, следует уточнить ассоциацию, показав отношение зависимости. Это отношение в данном случае означает, что класс использует указанный интерфейс (рис. 6.15).

Одновременно с уточнением отношений классов в пакете следует продумать и отношения классов, включенных в различные пакеты между собой.

Пример 6.5. Уточнить отношения классов пакета Объекты задачи между собой с классом Решение из пакета Объекты управления, используя результаты детализации отношений между объектами рассматриваемых классов.

Анализ диаграммы кооперации, представленной на рис. 6.9 показывает, что:

- класс Задание по сути дела представляет собой таблицу, в которой фиксируется все информация о конкретной задаче: вид задачи, алгоритм решения, данные и результат, причем результат связан с заданием неразрывно, так как теряет смысл вне контекста задания (отношение композиции), а данные имеют смысл сами по себе (отношение агрегации);
- класс Алгоритм целесообразно разрабатывать как абстрактный, обеспечивающий интерфейс между объектом класса Решение и конкретным алгоритмом и между объектом класса Задание и опять же конкретным алгоритмом;
- отношение между классами Задание и Алгоритм, Решение и Алгоритм, а также Задание и Решение – ассоциации, направленные к классу Задание.

Кроме того, анализ структур исходных данных и результатов решаемых задач показывает их существенное различие, следовательно, классы Данные и Результаты необходимо реализовать, как абстрактные, и наследовать от них классы, уточняющие структуры данных и результатов для каждого случая.

Результат уточнения показан на рис. 6.16.

Проектирование классов

Собственно проектирование классов предполагает окончательное определение структуры и поведения его объектов. *Структура* объектов определяется совокупностью атрибутов класса. Каждый атрибут это поле или совокупность полей, содержащееся в объекте класса.

Поведение объектов класса определяется реализуемыми *обязанностями*. Обязанности выполняются посредством *операций* класса.

Таким образом, при проектировании класса помимо имени и максимально полного списка атрибутов, необходимо уточнить его ответственность и операции. Причем как атрибуты, так и операции в процессе проектирования целесообразно дополнительно специфицировать. В зависимости от степени детализации диаграммы классов обозначение атрибута может включать, помимо имени, тип, описание видимости и значение по умолчанию в следующем виде:

<признак видимости> <имя>:<тип>=<значение по умолчанию>

где признак видимости может принимать одно из трех значений: «+» – общий; «#» – защищенный; «-» – скрытый.

Операциями называют основные действия, реализуемые классом. В отличие от методов операции не всегда реализуются классом непосредственно. Например, операция Ввод числа может быть реализована агрегированным интерфейсным элементом «окно ввода».

Полное описание операции на диаграмме класса в UML может выглядеть следующим образом:

<признак видимости> <имя>(<список параметров>): <тип возвращаемого значения>

Ответственностью класса называют краткое неформальное перечисление основных функций объектов класса. Ответственность класса обычно определяют на начальных этапах проектирования, когда атрибуты и операции класса еще не определены. Эту информацию отображают на диаграмме классов в специальных секциях условного изображения класса (рис. 6.17).

Исходный список операций класса формируют, анализируя диаграммы деятельностей, диаграммы кооперации и диаграммы последовательностей, построенные для различных сценариев с участием объектов проектируемого класса. На начальных этапах проектирования в секции операций класса обычно указывают лишь имена основных операций, определяющих наиболее общее поведение объектов соответствующих классов. По мере уточнения добавляют новые операции, а информацию об уже имеющихся операциях детализируют.

Большинство атрибутов выявляется при анализе предметной области, требований технического задания и описаний потоков событий.

Кроме того, как уже указывалось выше, отношение ассоциации и его подвиды – агрегация и композиция – означают наличие обмена сообщениями между объектами классов. Для организации передачи сообщений необходимо, чтобы генерирующий сообщения объект содержал информацию о вызываемом объекте, что означает наличие у этого объекта соответствующего указателя. Причем при отношении композиции объекты-части могут быть организованы как объектные поля объекта-целого.

В том случае, если объекты проектируемого класса должны реализовывать сложное поведение, для них целесообразно разрабатывать диаграммы состояний.

Диаграммы состояний объекта. Под *состоянием* применительно к диаграмме состояний понимают ситуацию в жизненном цикле объекта, во время которой он: удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает

некоторого события. Изменение состояния, связанное с нарушением условия или, соответственно, завершением деятельности или наступлением события называют *переходом*.

Диаграммы состояний показывают состояния объекта, возможные переходы, а также события или сообщения, вызывающие каждый переход.

Условные обозначения состояний приведены на рис. 6.18. Действие, указанное после слова Вход, выполняется при входе в состояние, а действие, указанное после слова Выход – при выходе из него. Деятельность связывается с нахождением в состоянии.

Переход обозначается линией со стрелкой и может быть помечен меткой, состоящей из трех частей, каждую из которых можно опустить:

<Событие> [<Условие>]/<Действие>

Если событие не указано, то это означает, что переход выполняется по завершению деятельности, связанной с данным состоянием. Если же оно указано – то при наступлении события. Условие записывается в виде логического выражения. Переход происходит, если результат выражения – «истина». Объект не может одновременно перейти в два разных состояния, поэтому условия перехода для любого события должны быть взаимоисключающими.

В отличие от деятельностей, действия, указанные для перехода, связывают с последним и рассматривают как мгновенные и непрерываемые.

При необходимости можно определять *суперсостояния* (рис. 6.18, *г*), которые объединяют несколько состояний в одно. Этот механизм обычно используют, чтобы показать переход из нескольких состояний в одно и тоже состояние, например, при отмене каких либо действий.

Пример 6.6. Разработать диаграмму состояний для объекта класса Решение.

Диаграмму строим, анализируя соответствующие диаграммы последовательностей (рис. 6.7–6.8). При этом необходимо уточнить, в какой момент разрешить прерывание процесса извне. Чтобы показать, что прерывание процесса возможно еще во время его инициализации, вводим суперсостояние Процесс. При реализации следует учесть возможность прерывания процесса до активации Алгоритма (рис. 6.19).

Результаты уточнения структуры и поведения объектов классов отразим на диаграмме классов.

Пример 6.7. Уточнить атрибуты и операции классов Решение, Задание, Алгоритм, Данные и Результаты, используя полученные в данном параграфе сведения.

К л а с с З а д а н и е. Поскольку объект класса Задание должен хранить идентификатор задачи и тип алгоритма, он должен иметь соответствующие поля и включать операции Определить задачу (), Определить алгоритм(), Сообщить тип задачи(), Сообщить тип алгоритма().

Кроме того, объект класса Задание отвечает за объекты классов Данные и Результаты, связанные с ним, следовательно, он должен хранить их адреса и соответственно выполнять операции Определить данные(), Сообщить данные(), Фиксировать результаты(), Сообщить результаты().

К л а с с ы Д а н н ы е и Р е з у л ь т а т ы. Данные задач и их результаты должны храниться в базе данных, но имеют различные структуры. Эту проблему можно решить, если хранить и данные, и результаты в упакованном виде, распаковывая их по мере надобности. Поэтому, соответствующие классы должны объявлять абстрактные операции Упаковать() и Распаковать(), которые будут реализовываться классами-подтипами в зависимости от реальной структуры данных, определяемой типом задачи. Следовательно, указанные классы должны также хранить тип задачи, с которой они связаны, и содержать операции Определить тип задачи() и Сообщить тип Задачи().

К л а с с А л г о р и т м. Объекты класса Алгоритм отвечают за реализацию метода решения задачи. Поскольку они посылают сообщение объектам класса Задания, то естественно должны хранить его адрес. Кроме того, класс Алгоритм должен объявлять абстрактную операцию Выполнить(). Эта операция должна переопределяться классами Алгоритм, реализующий метод.

Примечание. Возможно более удачное решение: в классе Алгоритм определить операцию Выполнить() и внутреннюю абстрактную операцию Реализовать метод(), которая вызывается из первой. Такое решение позволит не дублировать общее поведение всех алгоритмов, например, запрос и распаковку данных, а также упаковку и запись результата. Это решение не приведено, чтобы не усложнять и так достаточно сложный пример.

К л а с с Р е ш е н и е. Объект класса Решение обращается к объектам классов Задание и Алгоритм, следовательно, необходимо хранить их адреса. Операции Начать() и

Прервать() получены из диаграмм последовательностей действий. Операция Обработать исключение() получена оттуда же, но должна реализовываться особым способом, так как будет получать управление через механизм исключений.

Результаты уточнения приведены на рис. 6.20.

Проектирование методов класса. Некоторую достаточно существенную информацию о действиях, которые должны выполняться методами класса, можно получить, анализируя диаграммы последовательности действий. Однако алгоритмы всех сколько-нибудь сложных методов должны быть проработаны детально. При этом можно использовать как уже известные нотации (схемы алгоритмов и псевдокоды), так и диаграммы деятельностей.

Ранее уже были описаны диаграммы деятельности, которые предлагалось использовать в процессе уточнения спецификаций для описания вариантов использования. Эти же диаграммы могут использоваться и при проектировании методов обработки сообщений, в том числе и затрагивающих несколько объектов. В последнем случае целесообразно указать вертикальными пунктирными линиями ответственности объектов соответствующих классов, что позволит проследить вызовы других объектов.

Следует помнить, что в соответствии с общими правилами процедурной декомпозиции любую деятельность можно декомпозировать и изобразить в виде диаграммы деятельности более низкого уровня.

Пример 6.8. Построить диаграмму деятельности для операции Начать() класса Решение.

Анализ рис. 6.4, 6.7-6.8 показывает, что данная деятельность затрагивает три объекта уже детализированных классов Решение, Алгоритм и Задание. Определим зоны ответственности объектов этих классов (рис. 6.21).

Полностью спроектированные классы описывают на конкретном языке программирования.

Компоновка программных компонентов

Диаграмма компонентов применяют при проектировании физической структуры разрабатываемого ПО. Эти диаграммы показывают, как выглядит ПО на физическом уровне, т.е. из каких частей оно состоит и как эти части связаны между собой.

Диаграмма компонентов оперирует понятиями *компонент* и зависимость. Под компонентами понимают физические заменяемые части ПО, которые соответствуют

некоторому набору интерфейсов и обеспечивают их реализацию. По сути дела, это отдельные файлы различных типов: исполняемые (.exe), текстовые, графические, таблицы баз данных и т.п., составляющие разрабатываемое ПО. Условные графические обозначения компонент различных типов приведены на рис. 6.22.

Зависимость между компонентами фиксируют, если один компонент содержит некоторый ресурс (модуль, объект, класс и т.д.), а другой его использует. На рис. 6.23 в качестве примера приведена диаграмма компонентов системы решения комбинаторно-оптимизационных задач. Показан также интерфейс, через который система взаимодействует с базой данных.

При «сборке» исполняемых файлов диаграммы компонентов применяют для отображения взаимосвязей файлов, содержащих исходный код. Так на рис. 6.24 показано, что основной файл Main.cpp зависит от заголовочного файла Model.h, реализация которого находится в файле Model.cpp.

Используя UML можно построить диаграмму компоновки практически для любого случая, например, для Интернет-приложения. На рис. 6.25 приведен пример диаграммы компонентов клиентской части Интернет-приложения, написанного с использованием Java, которое в процессе работы демонстрирует некоторый рисунок.

Проектирование размещения программных компонентов для распределенных программных систем

При физическом проектировании распределенных программных систем необходимо определить наиболее целесообразный вариант размещения программных компонентов на реальном оборудовании в локальной или глобальной сетях. Для этого используют специальную модель UML– диаграмму размещения.

Диаграмма размещения отражает физические взаимосвязи между программными и аппаратными компонентами системы. Каждой части аппаратных средств системы, например, компьютеру или датчику, соответствует *узел* на диаграмме размещения. *Соединения* узлов означают наличие в системе соответствующих коммуникационных каналов. Внутри узлов указывают размещенные на данном оборудовании программные компоненты разрабатываемой программной системы, сохраняя указанные на диаграмме компонентов отношения зависимости.

На рис. 6.26 показаны условное обозначение узлов (процессора и устройства), а диаграмме размещения.

Пример 6.9. Разработать диаграмму размещения для системы учета успеваемости студентов.

Поместим основную часть системы на сервер деканата, а на компьютерах сети – соответствующие клиентские части (рис. 6.27).

7. ПРАВИЛА ОФОРМЛЕНИЯ ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ

Оформление текстового и графического материала

В соответствии с ГОСТ 7.32 – 91 расчетно-пояснительная записка должна включать: титульный лист, реферат, содержание, введение, основную часть, заключение, список использованных источников и, возможно, приложение.

Пример титульного листа записки (ГОСТ 19.104 –78) приведен в приложении 2.

Пояснительная записка оформляется на листах формата А4. Графический материал можно оформлять на листах формата А3. Поля на листе определяются в соответствии с общими требованиями: левое – не менее 30 мм, правое – не менее 10 мм, верхнее – не менее 15, а нижнее – не менее 20. При использовании текстовых редакторов для оформления записки параметры страницы заказываются в зависимости от устройства печати. При ручном оформлении выбираются из соображений удобства.

Нумерация страниц – сквозная. Номер проставляется сверху справа арабской цифрой. Страницами являются листы с текстами, рисунками и текстами приложения.

Первая страница – титульный лист. Номер страницы на титульном листе не проставляется. Образец титульного листа представлен в приложении 2.

Вторая страница – аннотация на разрабатываемый программный продукт, в которой в сжатом виде описывается назначение и особенности разработки.

Третья страница – оглавление, отражающее содержание изложенного материала (пример оглавления приведен в приложении 3). Ни аннотация, ни само оглавление в содержании не упоминаются.

Затем следуют разделы записки в порядке, определенном логикой изложения материала.

Записка завершается списком литературы.

Далее могут следовать приложения, содержащие материал, не вошедший в записку по причине ее ограниченного размера, но интересный для более глубокого понимания

назначения и возможностей разработки. Расчетно-пояснительная записка может содержать одно и более приложений.

Наименование разделов пишутся прописными буквами в середине строки. Расстояние между заголовками и текстом, а также между заголовками раздела и подразделов должно быть равно:

- при выполнении документа машинописным способом – двум интервалам;
- при выполнении рукописным способом –10 мм;
- при использовании текстовых редакторов – определяется возможностями редактора.

Наименования подразделов и пунктов следует размещать с абзацного отступа и печатать с прописной буквы вразрядку, не подчеркивая и без точки в конце. Расстояние между последней строкой текста предыдущего раздела и последующим заголовком при расположении их на одной странице должно быть равно:

- при выполнении документа машинописным способом – трем интервалам;
- при выполнении рукописным способом – не менее 15 мм;
- при использовании текстовых редакторов – определяется возможностями редактора.

Разделы и подразделы нумеруются арабскими цифрами с точкой. Разделы должны иметь порядковые номера 1, 2, и т. д. Номер подраздела включает номер раздела и порядковый номер подраздела, входящего в данный раздел, разделенные точкой. Например: 2.1., 3.5. Ссылка на пункты, разделы и подразделы указывается порядковым номером, например, «в разд. 4», «в п. 3.3.4».

Текст разделов выполняется через 1,5 интервала, а при использовании текстовых редакторов высота букв и цифр должна быть не менее 1,8 мм (шрифт № 12).

Перечисления надо нумеровать арабскими цифрами со скобкой; Например: 2), 3) и т.д. - с абзацного отступа. Допускается выделять перечисление простановкой дефиса перед пунктом текста или символом, его заменяющим, в текстовых редакторах.

Оформление рисунков, схем алгоритмов, таблиц и формул

Иллюстрации (графики, схемы, диаграммы) могут быть приведены как в основном тексте, так и в приложении. Все иллюстрации именуются рисунками. Все рисунки, таблицы и формулы нумеруются арабскими цифрами последовательно (сквозная нумерация). В приложении - в пределах приложения.

Чертежи, графики, диаграммы и схемы должны быть выполнены с учетом ЕСКД.

Каждый рисунок должен иметь подрисуночную подпись, помещаемую, как следует из названия под рисунком, например:

Рис.12. Форма окна основного меню

На все рисунки, таблицы и формулы в записке должны быть ссылки в виде: «(рис. 12)» или « форма окна основного меню приведена на рис. 12.».

Рисунки и таблицы должны размещаться сразу после той страницы, на которой в тексте записки она упоминается в первый раз. Если позволяет место, рисунок (таблица) может размещаться в тексте на той же странице, где на него дается первая ссылка.

Если рисунок занимает более одной страницы, на всех страницах, кроме первой, проставляется номер рисунка и слово «Продолжение». Например:

Рис. 12. Продолжение

Рисунки следует размещать так, чтобы их можно было рассматривать без поворота записки. Если такое размещение невозможно, рисунки следует располагать так, чтобы для рассматривания надо было повернуть записку по часовой стрелке. В этом случае верхним краем является левый край страницы. Расположение и размеры полей сохраняются в соответствии с выбранными.

Схемы алгоритмов должны быть выполнены в соответствии со стандартом ЕСПД. Толщина сплошной линии при вычерчивании схем алгоритмов должна быть в пределах от 0,6 до 1,5 мм. Надписи на схемах должны быть выполнены чертежным шрифтом. Высота букв и цифр должна быть менее 3,5 мм.

Номер таблицы размещается в правом верхнем углу перед заголовком таблицы, если он есть. Заголовок, кроме первой буквы , выполняется строчными буквами. В аббревиатурах используются только заглавные буквы. Например: ПЭВМ (ГОСТ 2.105).

Ссылки на таблицы в тексте пояснительной записки должны быть в виде слова «табл.» и номера таблицы. Например:

Результаты тестов приведены в табл. 4.

Уравнения и формулы следует выделять из текста в отдельную строку, оставив выше и ниже формулы не менее одной свободной строки. Если формула не умещается на одной строке, ее переносят, прервав на любом математическом знаке.

Номер формулы ставится с правой стороны страницы в круглых скобках на уровне формулы. Например:

$$z:=\sin(x)+\ln(y); \qquad (12)$$

Ссылка на номер формулы дается в скобках. Например: «расчет значений производится по формуле (12)».

Примечание следует помещать при необходимости пояснения содержания теста таблицы или рисунка. Оно размещается непосредственно после пункта, подпункта, таблицы или рисунка, к которому относится. Слово «примечание» печатается с абзацного отступа с прописной буквы вразрядку и не подчеркивается. Если примечаний несколько, то они нумеруются, например:

Примечания:

1. ….
2. …..

Оформление текстов программ

Тексты программ должны оформляться в соответствии с «хорошим стилем» программирования, т.е. должны быть легко читаемы и хорошо документированы. В текстах должны быть комментарии:

1) после заголовка программы или подпрограммы – общая информация: назначение, входные данные, результаты, метод решения; данные о программисте, дата написания, версия;

2) при объявлении данных - назначение переменных;

3) в начале и в конце определенной функционально законченной части программы;

4) для пояснения логических частей программы (ветвлений, циклов).

Однако, комментарии не должны затенять структуру текста и должны быть ясными и краткими.

Наименование программ и подпрограмм должны отражать их назначение. Логическая структура программы должна быть отражена в ее тексте с помощью:

1) пустых строк между текстами подпрограмм и отдельных ее функционально законченных частей;

2) сдвигами текста в строке при написании:

- заголовков вложенных циклов;

- тела цикла после его заголовка;
- альтернатив разветвлений процесса обработки данных.

Оформление приложений

Каждое приложение должно начинаться с новой страницы с указанием в правом углу слова «ПРИЛОЖЕНИЕ» прописными буквами и иметь тематический заголовок. При наличии более одного приложения все они нумеруются арабскими цифрами: ПРИЛОЖЕНИЕ 1, ПРИЛОЖЕНИЕ 2 и т.д. Например:

ПРИЛОЖЕНИЕ 2.

Титульный лист расчетно-пояснительной записки.

Рисунки и таблицы, помещаемые в приложении, нумеруются арабскими цифрами в пределах каждого приложения с добавлением буквы «П». Например:

рис. П.12 – 12-й рисунок приложения;

рис. П1.2 – 2-й рисунок 1-го приложения.

Каждый файл в приложении оформляется как рисунок с наименованием файла и его назначением. Например:

Рис. П1.3. Файл **mod1.pas** – исходные тексты библиотеки процедур обработки.

Рис. П2.4. Фал **menuran.pas** – программа движения курсора основного меню.

Оформление списка литературы

Список литературы должен включать все использованные источники. Сведения о книгах (монографиях, учебниках, пособиях, справочниках и т.д.) должны содержать: фамилию и инициалы автора, заглавие книги, место издания, издательство, год издания. При наличии трех и более авторов допускается указывать фамилию и инициалы только первого из них со словами «и др.». Наименование места издания надо приводить полностью в именительном падеже: допускается сокращение названия только двух городов: Москва (М.) и Санкт-Питербург (СПб.).

Сведения о статье из периодического издания должны включать: фамилию и инициалы автора, наименование статьи, наименование издания (журнала), наименование серии, если она есть, год выпуска, том, если есть, номер издания (журнала) и номера страниц, на которых помещена статья.

При ссылке на источник из списка литературы (особенно при обзоре аналогов) надо указывать порядковый номер по списку литературы, заключенный в квадратные скобки; например: [5].

СПИСОК ЛИТЕРАТУРЫ

1. Буч Г., Рамбо Д., Джакобсон А. Язык UML. Руководство пользователя. М.: ДМК Пресс, 2001.

2. Иванова Г.С. Технология программирования. М.: Из-во МГТУ им. Баумана, 2002.

3. Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. Объектно-ориентированное программирование. М: Из-во МГТУ им. Баумана, 2001.

4. Кватрани Т. Rational Rose и UML. Визуальное моделирование. М.: ДМК Пресс, 2001.

5. Ларман К. Применение UML и шаблонов проектирования. М.: Изд. дом «Вильямс», 2001.

5. Леоненков А. Самоучитель UML. СПб.: БХВ-Петербург, 2001.

ПРИЛОЖЕНИЕ 1. ТИТУЛЬНЫЙ ЛИСТ И ПРИМЕР ТЕХНИЧЕСКОГО ЗАДАНИЯ

Министерство образования Российской Федерации
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ИМ. Н.Э. БАУМАНА
Факультет «Информатика и системы управления»
Кафедра «Компьютерные системы и сети»

УТВЕРЖДАЮ
Зав. кафедрой ИУ6,
д.т.н., проф. _____Сюзев В.В.
“ ___” _____2002 г.

СИСТЕМА УЧЕТА ТЕКУЩЕЙ УСПЕВАЕМОСТИ СТУДЕНТОВ

Техническое задание на курсовую работу.

Листов 3

Руководитель,
к.т.н., доцент _____ Петров П.П.

Исполнитель,
студ. гр. ИУ6-21 _____ Иванов И. И.

2002

1. ВВЕДЕНИЕ

Во время сессии необходимо получение оперативной информации о ходе ее сдачи студентами, однако выполнение такого контроля вручную требует значительного времени.

Автоматизированная система учета успеваемости позволит улучшить качество контроля сдачи сессии со стороны куратора и деканата и обеспечит получение сведений о динамике работы каждого студента, группы в целом и курса.

Кроме того, хранение информации о сдаче сессий в течение всего времени обучения позволит осуществлять автоматическую генерацию справок о прослушанных курсах и приложений к диплому выпускника.

2. ОСНОВАНИЕ ДЛЯ РАЗРАБОТКИ

Основанием разработки является план мероприятий по совершенствованию учебного процесса на 2001-2002 учебный год.

3. НАЗНАЧЕНИЕ

Система предназначена для хранения и обработки сведений о успеваемости студентов учебных групп факультета в течение всего срока обучения. Обработанные сведения об успеваемости студентов могут быть использованы для оценки успеваемости каждого студента, группы, курса и факультета в целом.

4. ТРЕБОВАНИЯ К ПРОГРАММЕ ИЛИ ПРОГРАММНОМУ ИЗДЕЛИЮ

4.1. Требования к функциональным характеристи*кам*

Система должна обеспечивать возможность выполнения следующих функций.

4.1.1. Инициализацию системы (ввод списков групп, перечней изучаемых дисциплин в соответствии с учебными планами и т.п.).

4.1.2. Ввод и коррекцию текущей информации о ходе сдачи сессии конкретными студентами.

4.1.3. Хранение информации об успеваемости в течение времени обучения студента.

4.1.4. Получение сведений о текущем состоянии сдачи сессии студентами в следующих вариантах:

– результаты сдачи сессии конкретным студентом;

– результаты сдачи сессии студентами конкретной группы;

– процент успеваемости по всем студентам группы при сдаче конкретного предмета в целом на текущий момент;

– проценты успеваемости по всем группам специальности на текущий момент;

– проценты успеваемости по всем группам курса на текущий момент;

– проценты успеваемости по всем курсам и в целом по факультету на текущий момент;

– список задолжников группы на текущий момент;

– список задолжников курса на текущий момент;

Исходные данные:

– списки студентов учебных групп;

– учебные планы кафедр – перечень предметов и контрольных мероприятий по каждому предмету;

– расписания сессий;

– текущие сведения о сдаче сессии каждым студентом.

4.2. Требования к надежности

4.2.1. Предусмотреть контроль вводимой информации.

4.2.2. Предусмотреть блокировку некорректных действий пользователя при работе с системой.

4.2.3. Обеспечить целостность хранимой информации.

4.3. Требования к составу и параметрам технических средств

Система должна работать на IBM совместимых персональных компьютерах.

Минимальная конфигурация:

тип процессора…….Pentium и выше;

объем ОЗУ ……..32 Мб и более

4.4. Требования к информационной и программной совместимости

Система должна работать под управлением семейства операционных систем Win 32 (Windows 95, Windows 98, Windows 2000, Windows NT и т.п.).

5. ТРЕБОВАНИЯ К ПРОГРАММНОЙ ДОКУМЕНТАЦИИ

5.1. Разрабатываемые программные модули должны быть самодокументированны, т.е. тексты программ должны содержать все необходимые комментарии.

5.2. Программная система должна включать справочную информацию о работе и подсказки пользователю.

5.3. В состав сопровождающей документации должны входить:

- пояснительная записка, содержащая описание разработки;
- руководство системного программиста;
- руководство пользователя.

6. ЭТАПЫ РАЗРАБОТКИ

№	Название этапа	Срок	Отчетность
1	2	3	4
1.	Анализ требований и уточнение спецификаций	1.10.2002–15.10.2002	Сетевая модель данных. Описание алгоритмов обработки информации.
2.	Проектирование: разработка структуры ПО, интерфейса пользователя и проектирование компонентов.	16.10.2002–31.10..2002	Схема структурная ПО. Прототип системы на уровне интерфейса. Спецификации компонентов системы.
3.	Реализация: кодирование, тестирование и отладка программных компонентов и системы в целом	1.11.2002–15.12.2002	Программный продукт.
4.	Составление программной документации	16.12.2002–22.12.2002	Пояснительная записка и программная документация.

ПРИЛОЖЕНИЕ 2. ТИТУЛЬНЫЙ ЛИСТ РАСЧЕТНО-ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ

Министерство образования Российской Федерации
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. Н.Э. БАУМАНА
Факультет «Информатики и систем управления»
Кафедра «Компьютерные системы и сети»

СИСТЕМА УЧЕТА ТЕКУЩЕЙ УСПЕВАЕМОСТИ СТУДЕНТОВ

Расчетно-пояснительная записка
к курсовой работе

Листов 25

Руководитель,
к.т.н., доцент _____ Петров П.П.

Исполнитель,
студ. гр. ИУ6-21 _____ Иванов И. И.

2002

ПРИЛОЖЕНИЕ 3. ПРИМЕРЫ СОДЕРЖАНИЯ РАСЧЕТНО-ПОЯСНИТЕЛЬНЫХ ЗАПИСОК

1. К курсовой работе по «Технологии программирования» (объектный подход):

2. К курсовой работе по Технологии программирования (структурный подход):

3. К квалификационной работе бакалавра

(порядок частей – произвольный):

Приложение 1. Техническое задание на программный продукт (нумерация отдельная)

Приложение 2. Руководство пользователя (нумерация отдельная)

4. К дипломному проекту

(порядок частей – произвольный):