

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ  
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

М. Л. ДОЛЖЕНКОВА,

О. В. КАРАВАЕВА

# **ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ**

Практикум

Часть 2

Киров  
2014

УДК 004.42(47)  
Д643

Допущено к изданию методическим советом  
факультета автоматики и вычислительной техники  
ФГБОУ ВПО «ВятГУ» в качестве практикума  
для студентов направления 230101.62 «Информатика  
и вычислительная техника» всех профилей

Рецензент

заведующий кафедрой АТ ФГБОУ ВПО «ВятГУ»,  
кандидат технических наук В. И. Семеновых

**Долженкова, М. Л.**

Д643 Технологии программирования : практикум / М. Л. Долженкова,  
О. В. Караваева. – Киров: ФГБОУ ВПО «ВятГУ». 2013. – 127 с.

УДК 004.42(47)

Учебно-методическое пособие предназначено для студентов  
направления 230101.62 «Информатика и вычислительная техника» всех  
профилей, изучающих дисциплину «Технологии программирования»

Тех. редактор А. В. Куликова

© ФГБОУ ВПО «ВятГУ», 2013

## Содержание

ВВЕДЕНИЕ	4
ЛАБОРАТОРНАЯ РАБОТА № 1. Разработка технического задания на создание программного обеспечения	5
Теоретические сведения	5
Порядок выполнения работы	20
ЛАБОРАТОРНАЯ РАБОТА № 2. Структурный подход к проектированию программного обеспечения	21
Теоретические сведения	21
Порядок выполнения работы:	42
ЛАБОРАТОРНАЯ РАБОТА № 3.	
Реализация программного обеспечения	43
Теоретическая часть	43
Порядок выполнения работы	45
ЛАБОРАТОРНАЯ РАБОТА № 4. Тестирование программ	46
Теоретическая часть	46
Порядок выполнения работы	72
ЛАБОРАТОРНАЯ РАБОТА № 5. Методология объектно-ориентированного моделирования	73
Теоретические сведения	73
Порядок выполнения работы	100
ЛАБОРАТОРНАЯ РАБОТА № 6. Методология управление проектами	101
Цель работы	101
Теоретический материал	101
Порядок выполнения работы	125
СПИСОК ЛИТЕРАТУРЫ	127

## **ВВЕДЕНИЕ**

Технология программирования – это система методов, способов и приемов разработки и отладки программ.

Под технологией программирования (programming technology) будем понимать совокупность производственных процессов, приводящую к созданию требуемого программного средства (ПС), а также описание этой совокупности процессов.

Технология программирования в широком смысле – это технология разработки программных средств, включающая все процессы, начиная с момента зарождения идеи этого средства, и, связанные с созданием необходимой программной документации.

Современная технология проектирования программ включает в себя комплекс мероприятий, руководящих документов и автоматизированных средств, предназначенных для системного анализа, разработки, отладки, документирования, управления работой специалистов.

Хорошая технология дает возможность получить высокий экономический эффект при ее использовании и повышает качество программного продукта.

Настоящий лабораторный практикум направлен на изучение основных этапов создания надежного и качественного программного обеспечения и закрепление знаний полученных в рамках теоретического курса.

## **ЛАБОРАТОРНАЯ РАБОТА № 1. Разработка технического задания на создание программного обеспечения**

### **Цель работы:**

Проанализировать требования к программному обеспечению, составить и оформить техническое задание на разработку.

Основная задача лабораторной работы познакомиться с процессом разработки требований к программному обеспечению и составления технического задания на разработку программы, получение навыков использования основных методов формирования и анализа требований.

### **Теоретические сведения**

В процессе создания программного обеспечения специалистам приходится решать ряд сложных задач. В ряде случаев трудно чётко описать те действия, которые должна выполнять система.

Описание функциональных возможностей и ограничений, накладываемых на систему, называется *требованиями* к этой системе, а сам процесс формирования, анализа, документирования и проверки функциональных возможностей и ограничений – *разработкой требований*.

Требования делят на пользовательские и системные.

Пользовательские требования – это описание функций выполняемых системой и ограничений, накладываемых на неё на естественном языке.

Системные требования – это описание архитектуры системы, требований к параметрам оборудования, программного окружения и т. д., необходимых для эффективной реализации поставленной пользователем задачи.

Разработка требований – это процесс, включающий мероприятия, необходимые для создания и утверждения документа, содержащего спецификацию системных требований. Выделяют четыре этапа разработки требований:

1. анализ технической возможности создания системы,
2. формирование и анализ требований,
3. специфицирование требований и создание соответствующей документации,
4. аттестация требований.

В ходе анализа технической возможности создания системы выполняются общее описание системы и ее назначения. В результате анализа формируется отчет, в котором дается четкая рекомендация о возможности продолжения процесса разработки требований проектируемой системы.

Анализ должен ответить на следующие вопросы.

1. Решает ли система задачи организации-заказчика и организации-разработчика?
2. Возможно ли реализовать систему, на основе существующих на данный момент технологии, не выходя за пределы заданной стоимости?
3. Возможно ли объединить разрабатываемую систему с другими системами, которые уже эксплуатируются?

Для ответов на поставленные вопросы нужно определить следующее:

1. Что произойдет с организацией, если система не будет введена в эксплуатацию?
2. Какие текущие проблемы существуют в организации и как новая система поможет их решить?
3. Каким образом система будет способствовать развитию заказчика?
4. Требуется ли разработка системы технологии, которая до этого не использовалась в организации?

Источниками для получения необходимой информации могут быть менеджеры отделов, где система будет использоваться, разработчики

программного обеспечения, знакомые с типом будущей системы, конечные пользователи и т. д.

После обработки собранной информации готовится отчет по анализу возможности создания системы. В нем даются рекомендации о продолжении разработки системы, предлагаются изменения бюджета и графика работ по созданию системы, пересматриваются требования к системе.

На втором этапе выполняется формирование и анализ требований (рис. 1). Каждая организация использует собственный вариант этой модели, зависящий от опыта работы коллектива разработчиков, типа разрабатываемой системы, используемых стандартов и т. д.

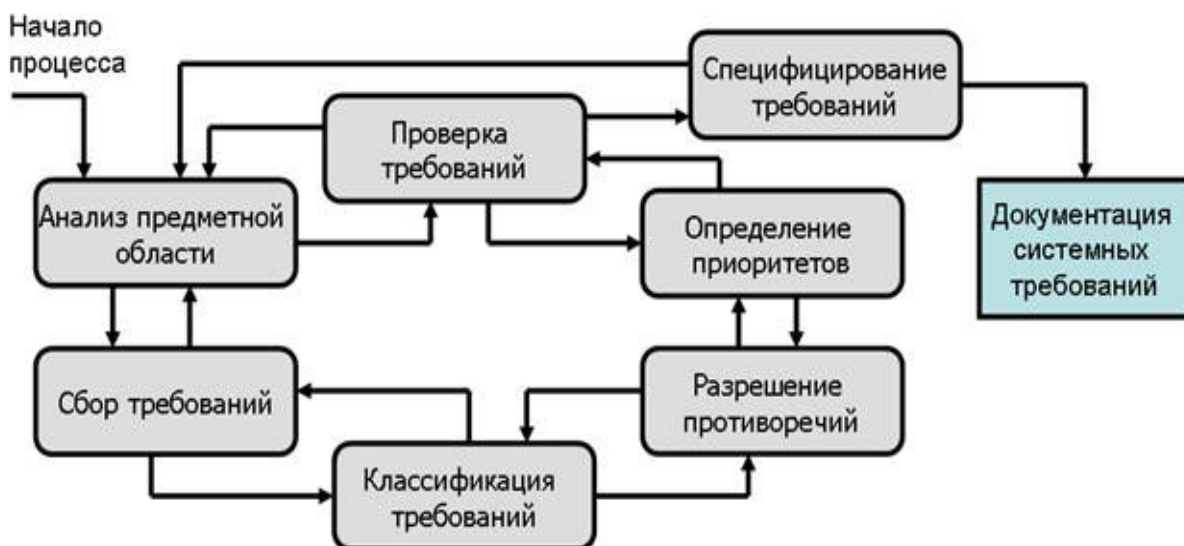


Рис. 1. Процесс формирования и анализа требований

В процессе формирования и анализа требований выполняют следующие работы:

1. *Анализ предметной области.* Аналитики изучают предметную область, в которой будет эксплуатироваться система.
2. *Сбор требований.* То есть взаимодействия с лицами, формирующими требования и продолжение анализа предметной области.
3. *Классификация требований.* Преобразование набора требований в логически связанные группы.

4. *Разрешение противоречий.* Так как требования многочисленных лиц, занятых в процессе формирования требований, могут быть противоречивыми, необходимо определить и разрешить эти противоречия.

5. *Назначение приоритетов.* В любом наборе требований некоторые из них будут более важны. Поэтому совместно с лицами, формирующими требования, определяются наиболее важные требования.

6. *Проверка требований.* Определяется полнота, последовательность и непротиворечивость выделенных требований.

Процесс формирования и анализа требований циклический, с обратной связью. Цикл начинается с анализа предметной области и заканчивается проверкой требований. Понимание требований предметной области улучшается в каждом цикле процесса формирования требований.

Существуют два основных подхода к формированию требований:

- метод, основанный на множестве опорных точек зрения;
- сценарии.

Метод с использованием различных *опорных* точек зрения к разработке требований признает различные точки зрения на проблему и использует для построения и организации как процесса формирования требований, так и непосредственно самих требований.

Точки зрения можно трактовать следующим образом.

– *Источник информации о системных данных.* На основе опорных точек зрения строится модель создания и использования данных в системе. В процессе формирования требований отбираются все такие точки зрения, которые будут использованы при работе системы, а также способы обработки этих точек зрения (данных).

– *Структура представлений.* Точки зрения рассматриваются как особая часть модели системы. На основе различных точек зрения могут разрабатываться модели «сущность-связь», модели конечного автомата и т. д.



– *Получатели системных сервисов* Точки зрения являются внешними относительно системы получателями сервисов, они помогают определить данные, необходимые для выполнения системных сервисов.

Наиболее эффективным подходом к анализу систем является использование внешних опорных точек зрения, на основе которого разработан метод VORD (Viewpoint-Oriented Requirements Definition – определение требований на основе точек зрения). Основные этапы метода VORD представлены на рис. 2.

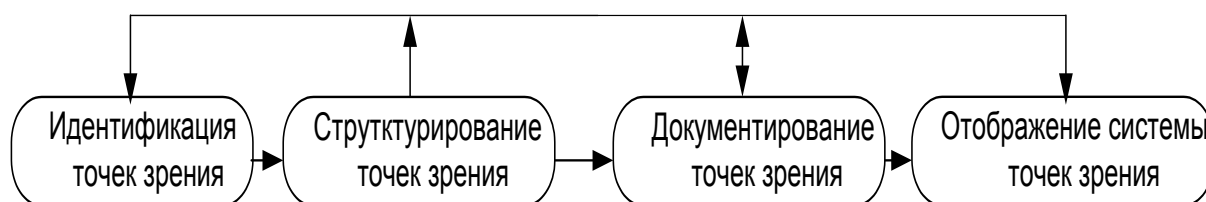


Рис. 2. Метод VORD

Идентификация точек зрения, получающих системные сервисы, и идентификация сервисов, соответствующих каждой точке зрения.

Структурирование точек зрения – создание иерархии сгруппированных точек зрения. Общесистемные сервисы предоставляются более высоким уровням иерархии и наследуются точками зрения низшего уровня.

Документирование опорных точек зрения, которое заключается в точном описании идентифицированных точек зрения и сервисов.

Отображение системы точек зрения, которая показывает системные объекты, определенные на основе информации, заключенной в опорных точках зрения.

Рассмотрим использование метода VORD для анализа требований к системе поддержки заказа и учета товаров в магазине. Пусть для каждого товара фиксируется место хранения, количество товара и его поставщик. Система учета заказов и товаров должна обеспечивать добавление информации о новом товаре, изменение или удаление информации об имеющемся товаре, хранение (добавление, изменение и удаление)

информации о поставщиках (название фирмы, ее адрес и телефон). Система позволяет составлять заказы поставщикам. Каждый заказ может содержать несколько позиций, в каждой из которых указывается наименование товара и его количество в заказе. Система по требованию пользователя формирует и выдает на печать следующую информацию:

- список всех товаров;
- список товаров, имеющихся в наличии;
- список товаров, количество которых необходимо пополнить;
- список товаров, поставляемых данным поставщиком.

При формировании требований необходимо идентифицировать опорные точки зрения. Один из подходов к идентификации точек зрения – метод «мозгового штурма», при котором определяют потенциальные системные сервисы и организации, взаимодействующие с системой. В ходе «мозгового штурма» идентифицируют потенциальные опорные точки зрения, системные сервисы, входные данные, нефункциональные требования, управляющие события и исключительные ситуации.

На рис. 3 приведена диаграмма идентификации точек зрения и сервисов. Сервисы должны соответствовать опорным точкам зрения. Но могут быть сервисы, которые не поставлены им в соответствие. Это означает, что на начальном этапе «мозгового штурма» некоторые опорные точки зрения не были идентифицированы.

Распределение сервисов для некоторых идентифицированных на рис. 3 точек зрения приведено в табл. 1. Один и тот же сервис может быть соотнесен с несколькими точками зрения.

Информация, извлеченная из точек зрения, используется для заполнения форм шаблонов точек зрения. После чего точки зрения организуются в иерархию наследования, что позволяет увидеть общие точки зрения и повторно использовать информацию. Сервисы, данные и управляющая информация наследуются подмножеством точек зрения.

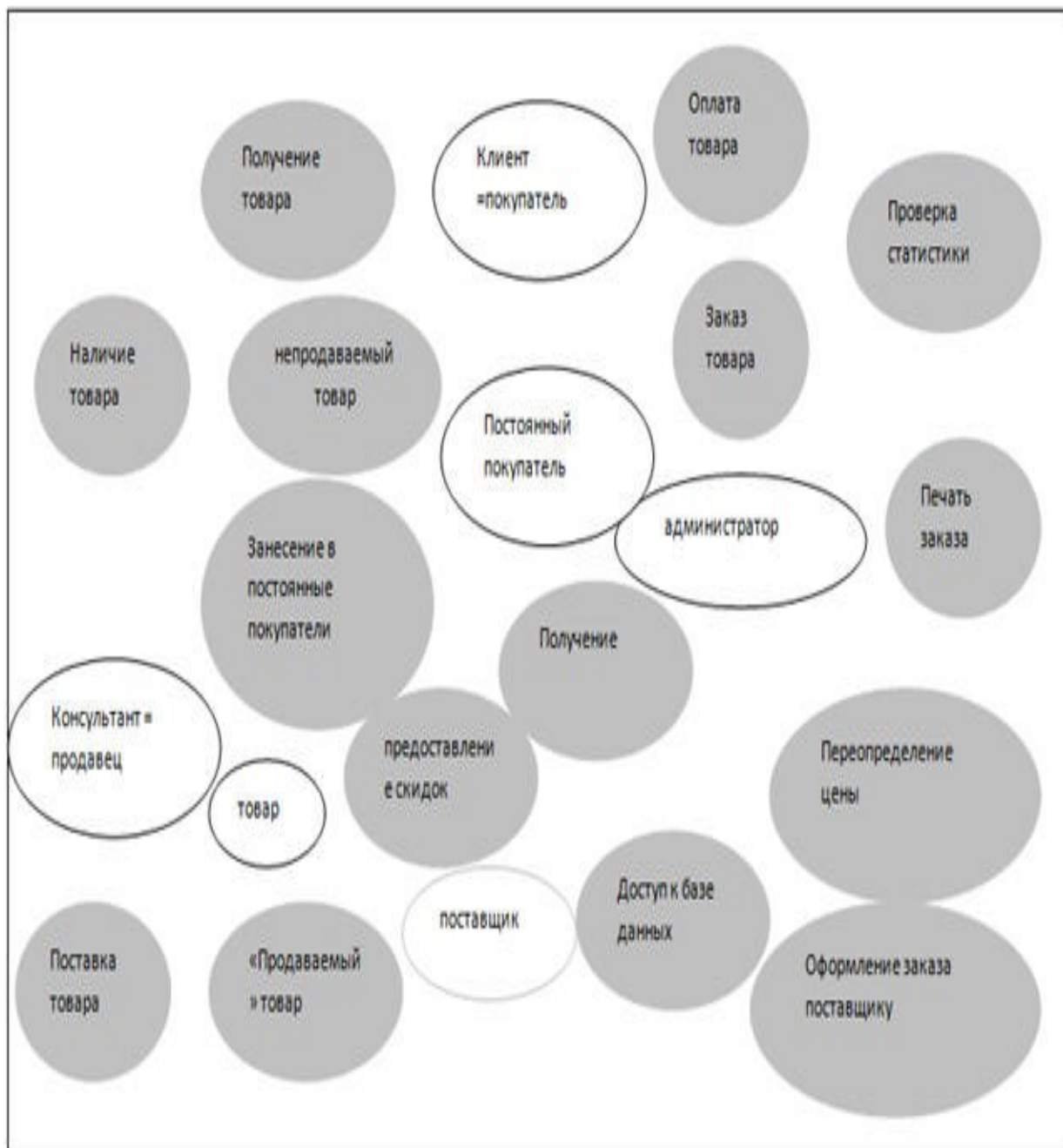


Рис. 3. Диаграмма идентификации точек зрения

На рис. 4 показана часть иерархии точек зрения для системы учета заказов и товаров.

Таблица 1

## Сервисы, соотнесенные с точками зрения

Клиент	Постоянный покупатель	Товар	Поставщик	Продавец	Администратор
Проверка наличия товара	Получение скидки	Прием товара	Занесение в базу данных (название, адрес, телефон и т. д.)	Продажа товара	Доступ к базе данных
Покупка товара	Получение информацию о новых поступлениях	Занесение в базу данных (данные о поставщике, кол-ве, месте хранения и. д.)		Печать чека	Проверка статистики
Получение чека		Назначение цены		Доступ к каталогу	Переопределение цены
Заказ товара		Переопределение цены		Проверка наличия товара	Оформление заказа поставщику
Занесение покупателя и суммы покупки в базу данных		Покупаемый или Непокупаемый товар		Оформление заказа покупателю	Печать заказа

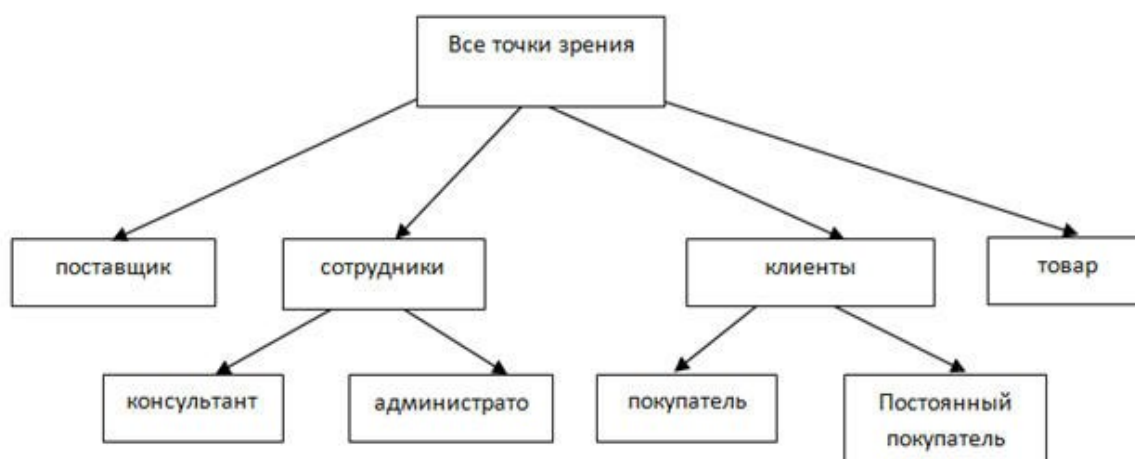


Рис. 4. Иерархия точек зрения

Сценарии особенно полезны для детализации уже сформулированных требований, так как описывают варианты взаимодействия пользователя с системой. Каждый сценарий описывает одно или несколько возможных интерактивных действий.

Сценарий начинается с общего описания, затем постепенно детализируется для создания полного описания взаимодействия пользователя с системой.

В большинстве случаев сценарий включает следующее:

- Описание состояния системы в начале сценария.
- Описание состояния системы после завершения сценария.
- Описание нормального протекания событий.
- Описание исключительных ситуаций и способов их обработки
- Информацию относительно других действий, которые можно осуществлять во время выполнения сценария.

Таким образом, сценарии содержат описание потоков данных, системных операций и исключительных ситуаций, которые могут возникнуть

Для понимания и формирования социальных и организационных аспектов эксплуатации системы используется этнографический подход к формированию системных требований, представленный на рис. 5.

Разработчик требований погружается в рабочую среду, в которой будет использоваться система, наблюдает и протоколирует реальные действия, выполняемые пользователями системы. Это помогает обнаружить неявные требования к системе, которые отражают реальные аспекты ее эксплуатации, а не формальные умозрительные процессы.

Этнографический подход позволяет детализировать требования для критических систем, чего не всегда можно добиться другими методами разработки требований. Однако, поскольку этот метод не может охватить все требования предметной области и требования организационного характера так как он ориентирован на конечного пользователя.

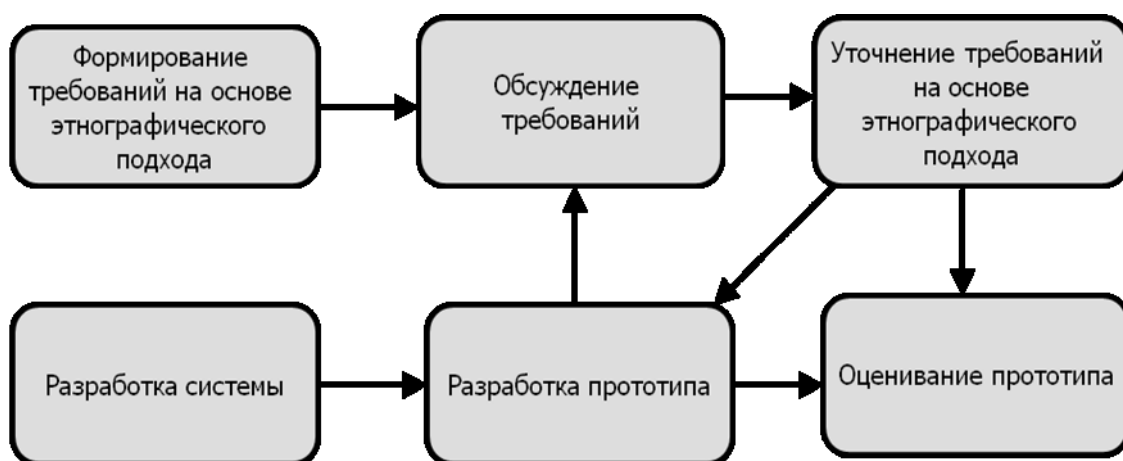


Рис. 5. Процесс разработки требований  
согласно этнографическому подходу

То что требования действительно определяют систему, которую хочет иметь заказчик демонстрирует аттестация требований, Проверка требований очень важна, так как ошибки в спецификации требований могут привести к переделке всей системы и большим затратам, если будут обнаружены в процессе разработки или после введения в эксплуатацию. Стоимость внесения в систему изменений, необходимых для устранения ошибок в требованиях, намного выше, чем исправление ошибок проектирования или кодирования. Причина в том, что изменение требований обычно влечет за собой значительные изменения в системе.

Во время процесса аттестации выполняются различные типы проверок требований.

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

2. *Проверка на непротиворечивость.* Спецификация требований не должна содержать противоречий.

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

4. *Проверка выполнимости.* На основе знания существующих технологий требования проверяются на возможность их реального выполнения, а так же возможность финансирования и соблюдения графика разработки системы.

Существует следующие методы аттестации требований, которые можно использовать как совместно, так и каждый в отдельности.

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

2. *Прототипирование.* Прототип системы демонстрируется конечному пользователю или заказчику с целью подтверждения того, что он отвечает потребностям.

3. *Генерация тестовых сценариев.* Требования должны быть такими, чтобы их реализацию можно было протестировать. Если тесты для требований разрабатываются как часть процесса аттестации, то возможно обнаружить проблемы в спецификации. Если такие тесты сложно или

невозможно разработать, это означает, что требования трудно выполнить и необходимо их пересмотреть.

#### 4. *Автоматизированный анализ непротиворечивости.*

Если требования представлены в виде структурных или формальных системных моделей, можно использовать инструментальные CASE-средства для проверки непротиворечивости моделей. Для автоматизированной проверки непротиворечивости строят базу данных требований и затем проверяют все требования в этой базе данных. Анализатор требований готовит отчет обо всех обнаруженных противоречиях.

На основании полученных моделей строятся пользовательские требования, описывающие функции, выполняемых системой, и ограничений, накладываемых на неё на естественном языке.

Пользовательские требования должны описывать внешнее поведение системы, основные функции и сервисы предоставляемые системой, её нефункциональные свойства. Пользовательские требования можно оформить простым перечислением.

Затем составляются системные требования:

1. *Требования к архитектуре системы.* Например, число и размещение хранилищ и серверов приложений.

2. *Требования к техническому обеспечению.* Например, частота процессоров серверов и клиентов, требуемый объём памяти для хранения данных, размер оперативной и видео памяти, пропускная способность канала и т. д.

3. *Требования к параметрам системы.* Например, время отклика на действие пользователя, максимальный размер передаваемого файла, максимальная скорость передачи данных, максимальное число одновременно работающих пользователей и т. д.

4. *Требования к программному интерфейсу.*



5. *Требования к структуре системы.* Например, масштабируемость (возможность распространения системы на большое количество машин, не приводящая к потере работоспособности и эффективности), распределённость (поддержка распределённого хранения данных), модульность, открытость (наличие открытых интерфейсов для возможной доработки и интеграции с другими системами).

6. *Требования по взаимодействию и интеграции с другими системами.* Например, использование общих баз данных, получение данных из баз данных других систем и т. д.

Техническое задание оформляют в соответствии с ГОСТ 34.602–89.

Техническое задание должно содержать следующие разделы:

- введение;
- наименование и область применения;
- основание для разработки;
- назначение разработки;
- технические требования к программе или программному изделию;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки;
- приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них. При необходимости допускается в техническое задание включать приложения.

В соответствии с ГОСТ содержание разделов определяется следующим образом.

Введение должно включать краткую характеристику области применения программы или программного продукта, а также объекта

(например, системы), в котором предполагается их использовать. Основное назначение введения – продемонстрировать актуальность данной разработки и показать, какое место эта разработка занимает в ряду подобных.

В разделе «Наименование и область применения» указывают наименование, краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие.

В разделе «Основание для разработки» должны быть указаны:

- документ (документы), на основании которых ведется разработка.

Таким документом может служить план, приказ, договор и т. п.;

- организация, утвердившая этот документ, и дата его утверждения;

- наименование и (или) условное обозначение темы разработки.

В разделе «Назначение разработки» должно быть указано функциональное и эксплуатационное назначение программы или программного изделия.

Раздел «Технические требования к программе или программному изделию» должен содержать следующие подразделы:

- требования к функциональным характеристикам – должны быть указаны требования к составу выполняемых функций, организации входных и выходных данных, временным характеристикам и т. п.;

- требования к надежности – должны быть указаны требования к обеспечению надежного функционирования (обеспечение устойчивого функционирования, контроль входной и выходной информации, время восстановления после отказа и т. п.);

- условия эксплуатации – должны быть указаны условия эксплуатации (температура окружающего воздуха, относительная влажность и т. п. для выбранных типов носителей данных), при которых должны обеспечиваться заданные характеристики, а также вид обслуживания, необходимое количество и квалификация персонала;

- требования к составу и параметрам технических средств – указывают необходимый состав технических средств с указанием их технических характеристик;

- требования к информационной и программной совместимости – должны быть указаны требования к информационным структурам на входе и выходе и методам решения, исходным кодам, языкам программирования. При необходимости должна обеспечиваться защита информации и программ;

- требования к маркировке и упаковке в общем случае указывают требования к маркировке программного изделия, варианты и способы упаковки;

- требования к транспортированию и хранению должны быть указаны для программного изделия условия транспортирования, места хранения, условия хранения, условия складирования, сроки хранения в различных условиях;

- специальные требования.

В разделе «Технико-экономические показатели» должны быть указаны: ориентировочная экономическая эффективность, предполагаемая годовая потребность, экономические преимущества разработки по сравнению с лучшими отечественными и зарубежными образцами или аналогами.

В разделе «Стадии и этапы разработки» устанавливают необходимые стадии разработки, этапы и содержание работ (перечень программных документов, которые должны быть разработаны, согласованы и утверждены), а также, как правило, сроки разработки и определяют исполнителей.

В разделе «Порядок контроля и приемки» должны быть указаны виды испытаний и общие требования к приемке работы.

В приложениях к техническому заданию при необходимости приводят:

- перечень научно-исследовательских и других работ, обосновывающих разработку;

- схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие документы, которые могут быть использованы при разработке;

- другие источники разработки.

В случаях, если какие-либо требования, предусмотренные техническим заданием, заказчик не предъявляет, следует в соответствующем месте указать «Требования не предъявляются».

### **Порядок выполнения работы**

1. Построить опорные точки зрения на основании метода VORD для формирования и анализа требований. Результатом должны явиться две диаграммы: диаграмма идентификации точек зрения и диаграмма иерархии точек.

2. Составить информационную модель будущего программного обеспечения, включающую в себя описание основных объектов системы и взаимодействия между ними.

3. Определить пользовательские требования, четко описывающие будущий функционал системы.

4. Определить системные требования, включающие требования к структуре, программному интерфейсу, технологиям разработки, общие требования к системе.

5. Разработать техническое задание на программный продукт.

## **ЛАБОРАТОРНАЯ РАБОТА № 2. Структурный подход к проектированию программного обеспечения**

### **Цель работы:**

Создать формальные модели и на их основе определить спецификации разрабатываемого программного обеспечения.

### **Теоретические сведения**

При построении моделей проектируемого программного обеспечения необходимо рассматривать его с разных сторон. Поэтому рекомендуется использовать сразу несколько моделей и сопровождать их описаниями. Структурный подход к проектированию программных продуктов предполагает разработку следующих моделей:



- спецификаций процессов;
- словаря терминов;
- диаграмм переходов состояний (STD – State Transition Diagrams), характеризующих поведение системы во времени;
- функциональных диаграмм;
- диаграмм потоков данных (DFD – Data Flow Diagrams), описывающих взаимодействие источников и потребителей информации через процессы, которые должны быть реализованы в системе;
- диаграмм «сущность-связь» (ERD – Entity-Relationship Diagrams), описывающих базы данных разрабатываемой системы.

Спецификации процессов могут быть представлены в виде блок-схем алгоритмов, псевдокодов, Flow-форм, диаграмм Насси-Шнейдермана или просто краткого текстового описания.

Для изображения схем алгоритмов используют ГОСТ 19.701-90

Таблица 2

## Обозначения блоков в схемах алгоритмов программ

Название	Обозначение	Назначение
Терминальная вершина	 Действие	Начало, завершение программы или подпрограммы
Операционная вершина	 Действие	Обработка данных (вычисления, пересылки и т. п.)
Ввод/вывод данных	 Данные	Операции ввода-вывода
Условная вершина	 Условие	Ветвление, выбор, поисковые и итерационные циклы
Граница цикла	 Начало	Любые циклы
	 Конеч	
Предопределенный процесс	 Имя	Вызов процедур
Соединитель	 Имя	Маркировка разрывов линий
Комментарий	 Комментарий	Пояснения к операциям

*Псевдокод* представляет собой формализованное текстовое описание алгоритма. Существует множество вариантов псевдокодов, один из них приведен в табл. 3.

## Псевдокоды

Структура	Псевдокод	Структура	Псевдокод
Следование	<Действие1> <Действие2>	Выбор	<b>Выбор</b> <код> <код1>:<Действие1> <код2>: <Действие2>  ... <b>Все-выбор</b>
Ветвление	<b>Если</b> <Условие> <b>то</b> <Действие1> <b>иначе</b> <Действие2> <b>Все-если</b>	Цикл с заданным количеством повторений	<b>Для</b> <индекс> = <n>, <k>, <h> <Действие> <b>Все-цикл</b>
Цикл с предусловием	<b>Цикл-пока</b> <Условие> <Действие> <b>Все-цикл</b>	Цикл с постусловием	<b>Выполнять</b> <Действие> <b>До</b> <Условие>

Пример алгоритма поиска некоторого значения, записанного псевдокодом:

*Программа*

*Цикл-пока* не конец файла

Прочитать запись

Сравнить поля с критерием поиска

*Если* совпали

Сохранить в выходной список

*Конец-если*

*Конец-цикл*

Вывод результирующего списка

*Конец-программа*

*Flow-формы* – это графическая нотация описания структурных алгоритмов, иллюстрирующая вложенность структур. Каждый символ Flow-формы имеет вид прямоугольника и может быть вписан в любой другой символ. Нотация Flow-форм приведена на рисунке 6.

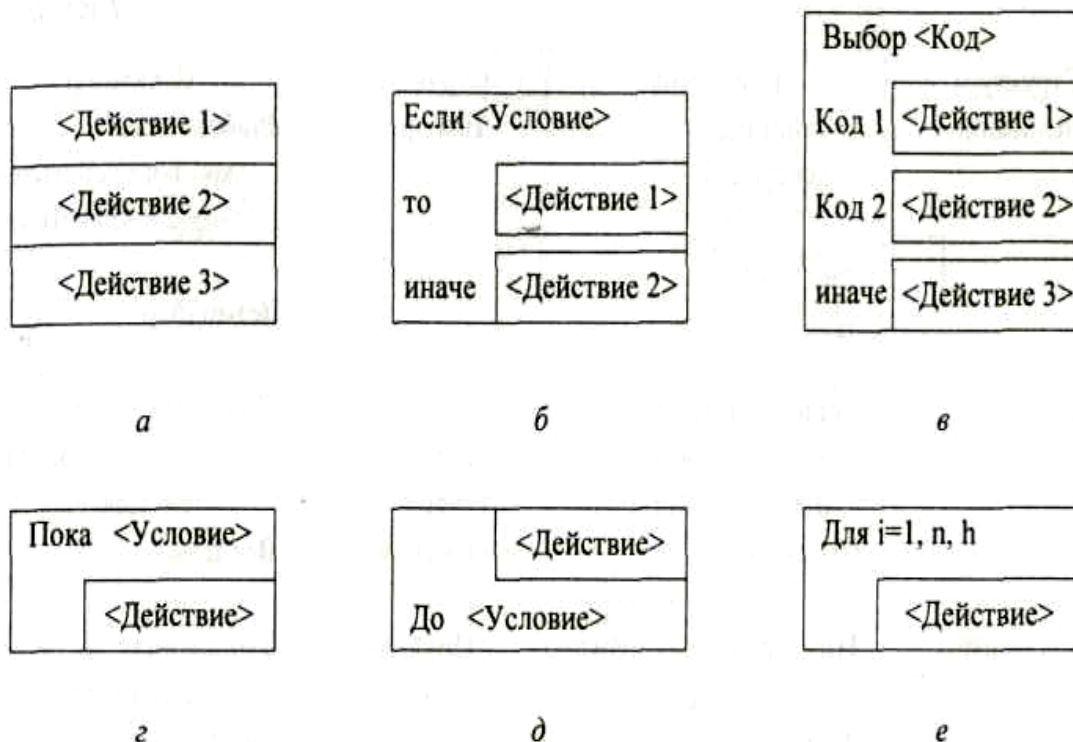


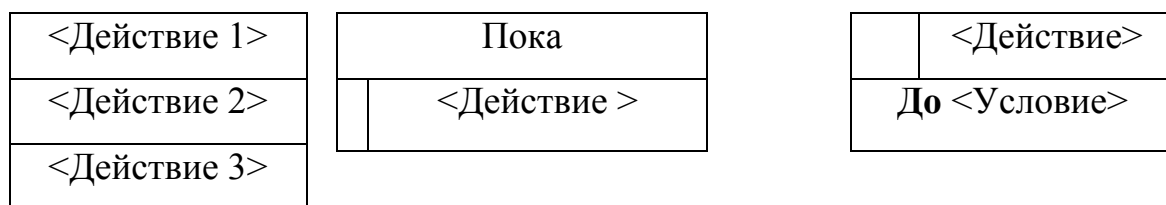
Рис. 6. Условные обозначения Flow-форм для основных конструкций:

а – следование; б – ветвление; в – выбор; г – цикл-до; е – счетный цикл

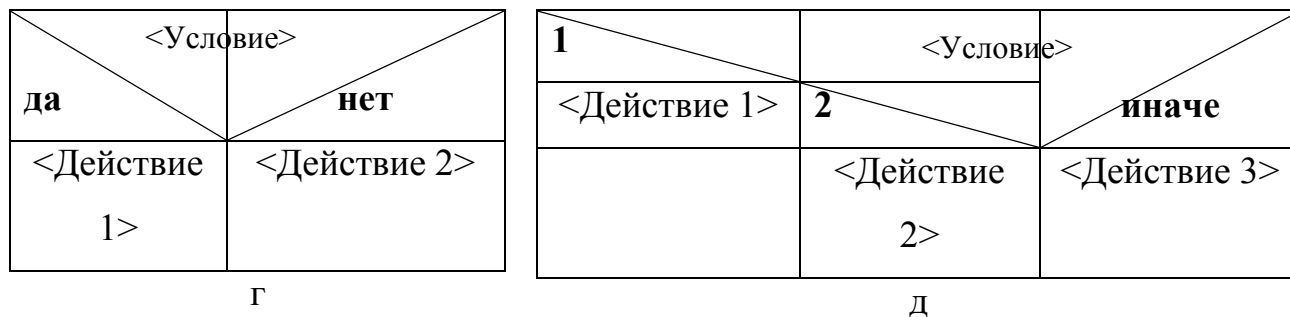
*Диаграммы Насси-Шнейдермана* отличаются от Flow-форм тем, что область обозначения условий изображают в виде треугольников (рис. 7). Это обозначение обеспечивает большую наглядность представления алгоритма.

Однако для описания неструктурных алгоритмов псевдокодов, Flow-форм и диаграмм Насси-Шнейдермана не используются, т. к. для неструктурных передач управления в этих нотациях просто отсутствуют условные обозначения.





в



г

д

Рис. 7. Условные обозначения диаграмм Насси-Шнейдермана для основных конструкций:

а – следование; б – цикл–пока; в – цикл–до; г – ветвление; д – выбор

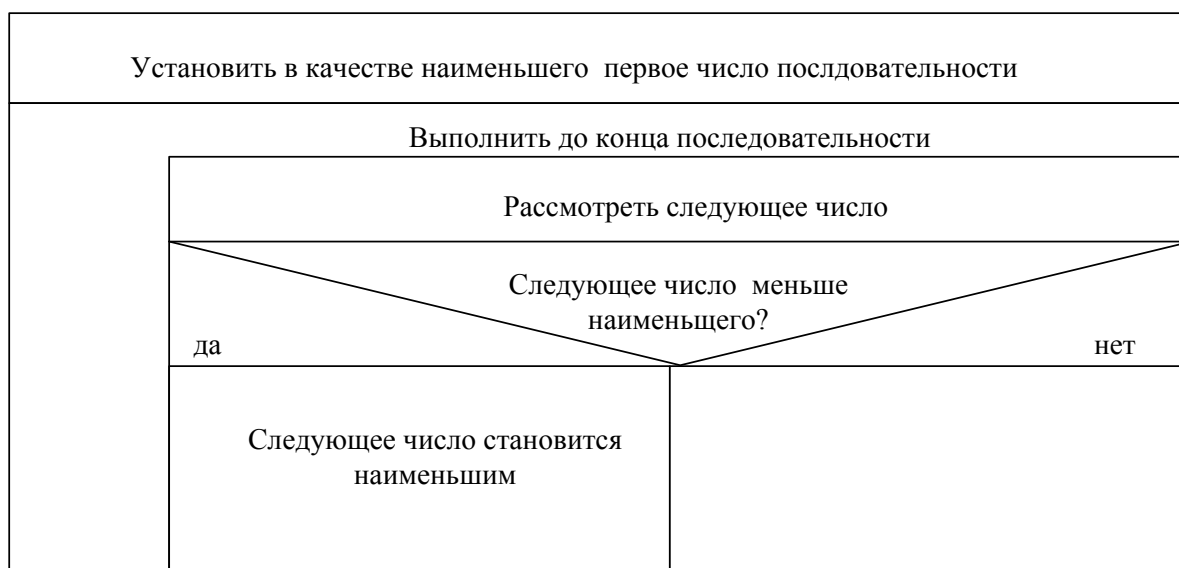


Рис. 8. Диаграмма Насси-Шнейдермана поиска минимального числа в последовательности

Flow-формы и диаграммы Насси-Шнейдермана в отличие псевдокода позволяют более наглядно отражать вложенность конструкций.

Однако при описании больших алгоритмов Flow-форм и диаграмм Насси-Шнейдермана сложны, что затрудняет их практическое применение.

*Словарь терминов* представляет собой краткое описание основных понятий, используемых при составлении спецификаций. Он предназначен

для повышения степени понимания предметной области позволяет исключить разногласия при определении моделей между заказчиками и разработчиками.

Обычно описание термина в словаре выполняют по следующей схеме:

Термин;

Категория (понятие предметной области, элемент данных, условное обозначение);

Краткое описание.

Пример:

<i>Термин</i>	Web-сайт
<i>Категория</i>	Интернет-программирование
<i>Описание</i>	Совокупность Web-страниц с повторяющимся дизайном, объединенных по смыслу, навигационно и физически находящихся на одном сервере.

*Диаграммы переходов состояний (SDT)* демонстрирует поведение разрабатываемого программного обеспечения при получении внешних управляющих воздействий.

В диаграммах такого вида узлы соответствуют состояниям динамической системы, а дуги – переходу системы из одного состояния в другое. Узел, из которого выходит дуга, является начальным состоянием, узел, в который дуга входит – следующим. Дуга помечается именем входного сигнала или события, вызывающего переход, а так же сигналом или действием, сопровождающим переход.

Условные обозначения, используемые при построении диаграмм переходов состояний, показаны на рис. 9.

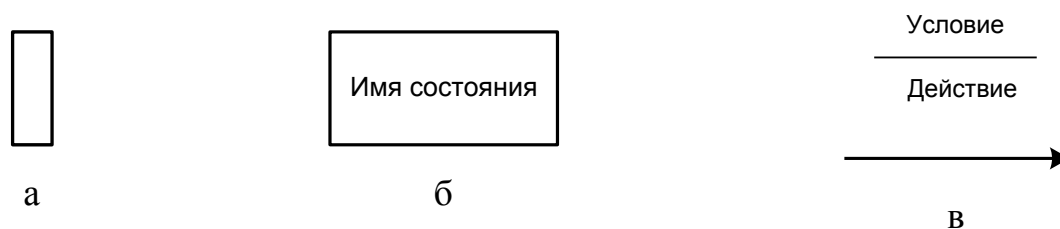


Рис. 9. Условные обозначения диаграмм переходов состояний:

а – терминальное состояние; б – промежуточное состояние; в – переход

На рис. 10 представлена диаграмма переходов торгового автомата активно взаимодействующего с покупателем.

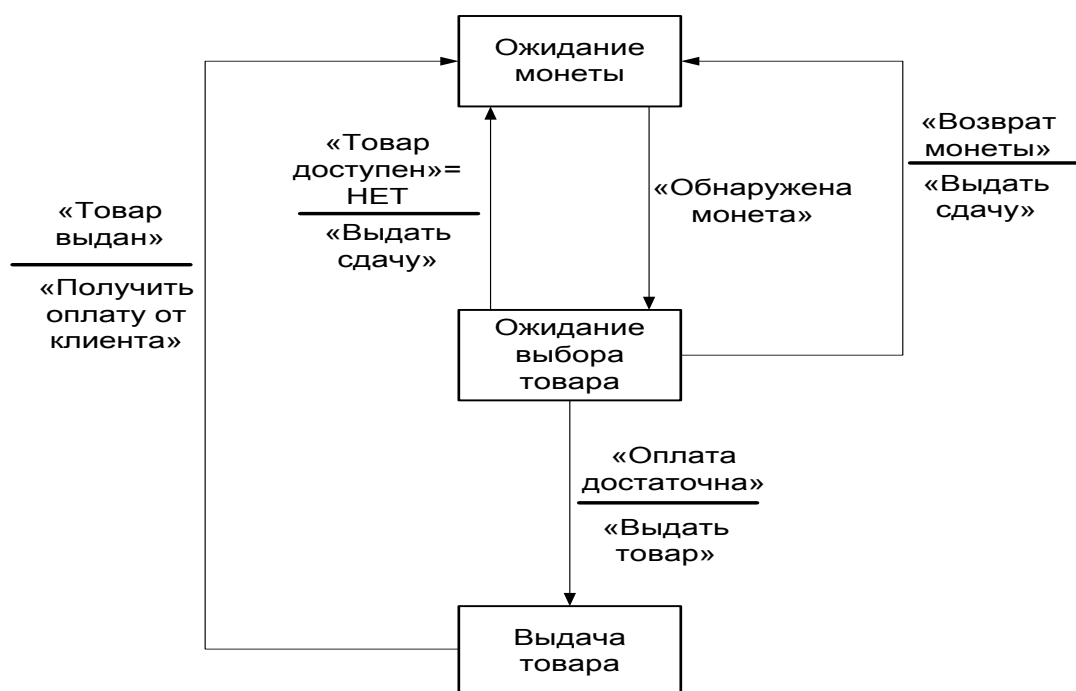


Рис. 10. Диаграмма переходов состояний торгового автомата

Характерной особенностью интерактивного программного обеспечения является наличие состояния ожидания, когда программное обеспечение приостанавливает работу до получения очередного управляющего воздействия (рис. 11).



Рис. 11. Диаграмма переходов состояний программы построения графиков/таблиц функций

*Функциональными* называют диаграммы, в первую очередь отражающие взаимосвязи функций разрабатываемого программного обеспечения.

Они создаются на ранних этапах проектирования систем, для того чтобы помочь проектировщику выявить основные функции и составные части проектируемой системы и, по возможности, обнаружить и устранить существенные ошибки. Современные методы структурного анализа и проектирования предоставляют разработчику определённые синтаксические и графические средства проектирования функциональных диаграмм информационных систем.

IDEF0 (Integrated Definition Function Modeling) -методология функционального моделирования. В основе IDEF0 методологии лежит понятие блока, который отображает некоторую бизнес-функцию. Четыре стороны блока имеют разную роль: левая сторона имеет значение «входа», правая – «выхода», верхняя – «управления», нижняя – «механизма» (рис. 12).

Взаимодействие между функциями в IDEF0 представляется в виде дуги, которая отображает поток данных или материалов, поступающий с выхода одной функции на вход другой. В зависимости от того, с какой стороной блока связан поток, его называют соответственно «входным», «выходным», «управляющим».



Рис. 12. Функциональный блок

В IDEF0 реализованы три базовых принципа моделирования процессов:

- принцип функциональной декомпозиции;
- принцип ограничения сложности;
- принцип контекста.

*Принцип функциональной декомпозиции* состоит в моделировании типовой ситуации, когда любое действие, операция, функция могут быть разбиты (декомпозированы) на более простые действия, операции, функции. Сложная функция представляется в виде совокупности элементарных функций (рис. 13).

*Принцип ограничения сложности.* При работе с IDEF0 диаграммами существенным является условие их разборчивости и удобочитаемости. Суть принципа ограничения сложности состоит в том, что количество блоков на диаграмме должно быть не менее двух и не более шести. Практика показывает, что соблюдение этого принципа приводит к тому,

что функциональные процессы, представленные в виде IDEF0 модели, хорошо структурированы, понятны и легко поддаются анализу.

*Принцип контекстной диаграммы.* Моделирование делового процесса начинается с построения контекстной диаграммы. На этой диаграмме отображается только один блок – главная бизнес-функция моделируемой системы. Если речь идет о моделировании целого предприятия или даже крупного подразделения, главная бизнес-функция не может быть сформулирована как, например, «продавать продукцию». Главная бизнес-функция системы – это «миссия» системы, ее значение в окружающем мире. Нельзя правильно сформулировать главную функцию предприятия, не имея представления о его стратегии.

При определении главной бизнес-функции необходимо всегда иметь в виду цель моделирования и точку зрения на модель. Одно и то же предприятие может быть описано по-разному, в зависимости от того, с какой точки зрения его рассматривают: директор предприятия и налоговой инспектор видят организацию совершенно по-разному.

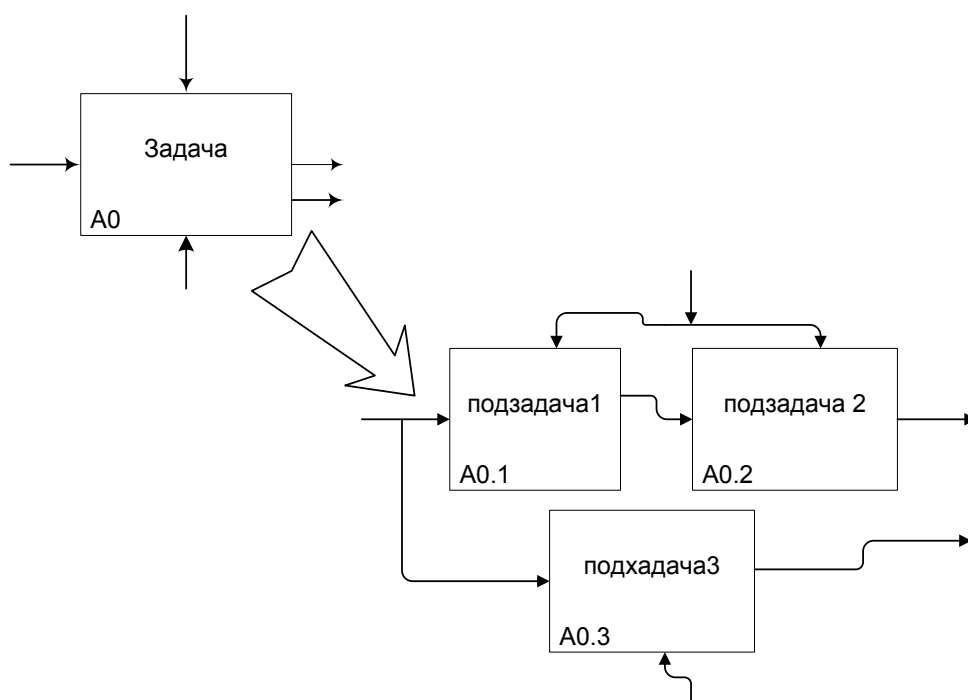


Рис. 13. Декомпозиция функционального блока

Контекстная диаграмма играет еще одну роль в функциональной модели. Она «фиксирует» границы моделируемой бизнес-системы, определяя то, как моделируемая система взаимодействует со своим окружением. Это достигается за счет описания дуг, соединенных с блоком, представляющим главную бизнес-функцию.

*Пример.*

На рис. 14 и рис. 15 представлен пример построения функциональной диаграммы, описывающей изготовление изделия.



Рис. 14. Контекстная диаграмма

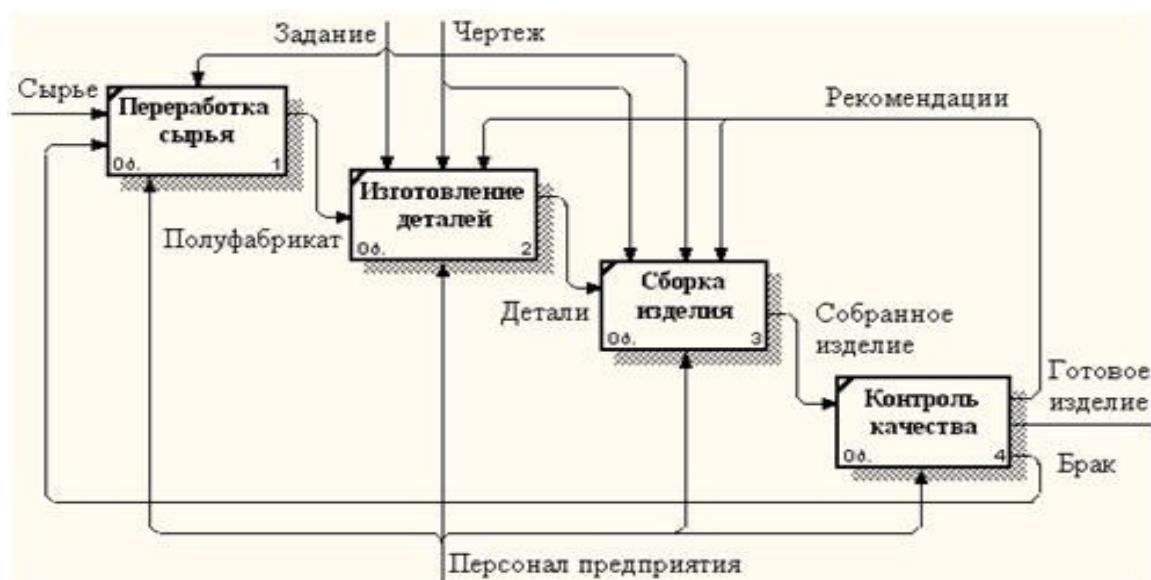


Рис. 15. Диаграмма первого уровня декомпозиции

Существует два ключевых подхода к построению функциональной модели: построение «как есть» и построение «как будет».

*Построение модели «как есть».* Обследование предприятия является обязательной частью любого проекта создания или развития корпоративной информационной системы.

Построение функциональной модели «как есть» позволяет четко зафиксировать, какие деловые процессы осуществляются на предприятии, какие информационные объекты используются при выполнении деловых процессов и отдельных операций. Функциональная модель «как есть» является отправной точкой для анализа потребностей предприятия, выявления проблем и «узких» мест и разработки проекта совершенствования деловых процессов.

*Построение модели «как будет».* Создание и внедрение корпоративной информационной системы приводит к изменению условий выполнения отдельных операций, структуры деловых процессов и предприятия в целом. Это приводит к необходимости изменения системы бизнес-правил, используемых на предприятии, модификации должностных инструкций сотрудников. Функциональная модель «как будет» позволяет уже на стадии проектирования будущей информационной системы определить эти изменения. Применение функциональной модели «как будет» позволяет не только сократить сроки внедрения информационной системы, но также снизить риски, связанные с невосприимчивостью персонала к информационным технологиям.

Для описания логики взаимодействия информационных потоков наиболее подходит IDEF3 (workflow diagramming) – методология моделирования, графически описывающая информационные потоки и взаимоотношения между процессами обработки информации и объектами, являющимися частью процессов. Диаграммы Workflow используются для анализа завершенности процедур обработки информации. Описывают сценарии действий сотрудников организации, например последовательность обработки заказа или события, которые необходимо обработать за конечное время. Каждый сценарий сопровождается



описанием процесса и может быть использован для документирования каждой функции.

IDEF3 – это метод, имеющий основной целью дать возможность аналитикам описать ситуацию, когда процессы выполняются в определенной последовательности, а также описать объекты, участвующие совместно в одном процессе.

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

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

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

*Единицы работы – Unit of Work (UOW).* UOW, также называемые работами (activity), являются центральными компонентами модели. В IDEF3 работы изображаются прямоугольниками с прямыми углами и имеют имя, выраженное отглагольным существительным, обозначающим процесс действия, одиночным или в составе фразы, и номер (идентификатор); другое имя существительное в составе той же фразы обычно отображает основной выход (результат) работы, например, «Изготовление изделия».

*Связи.* Связи показывают взаимоотношения работ. Все связи в IDEF3 однонаправлены и могут быть направлены куда угодно, но обычно

диаграммы IDEF3 стараются построить так, чтобы связи были направлены слева направо. В IDEF3 различают три типа стрелок, изображающих связи, стиль которых устанавливается через меню Edit/Arrow Style:

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

- Отношения (Relational Link) – пунктирная линия, используемая для изображения связей между единицами работ (UOW) а также между единицами работ и объектами ссылок.

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

- Старшая связь и поток объектов. Старшая связь показывает, что работа-источник заканчивается ранее, чем начинается работа-цель. Часто результатом работы-источника становится объект, необходимый для запуска работы-цели. В этом случае стрелку, обозначающую объект, изображают с двойным наконечником. Имя стрелки должно ясно идентифицировать отображаемый объект. Поток объектов имеет ту же семантику, что и старшая стрелка.

*Перекрестки (Junction).* Окончание одной работы может служить сигналом к началу нескольких работ, или же одна работа для своего запуска может ожидать окончания нескольких работ. Перекрестки используются для отображения логики взаимодействия стрелок при слиянии и разветвлении или для отображения множества событий, которые могут или должны быть завершены перед началом следующей работы. Различают перекрестки для слияния (Fan-in Junction) и разветвления (Fan-out Junction) стрелок. Перекресток не может использоваться одновременно для слияния и для разветвления.

Для внесения перекрестка служит кнопка в палитре инструментов – добавить в диаграмму перекресток Junction. В диалоге Junction Type Editor необходимо указать тип перекрестка. Смысл каждого типа приведен в табл. 4.

Таблица 4

#### Типы перекрестков

Обозначение	Наименование	Смысл в случае слияния стрелок	Смысл в случае разветвления стрелок
	Asynchronous AND	Все предшествующие процессы должны быть завершены	Все следующие процессы должны быть запущены
	Synchronous AND	Все предшествующие процессы завершены одновременно	Все следующие процессы запускаются одновременно
	Asynchronous OR	Один или несколько предшествующих процессов должны быть завершены	Один или несколько следующих процессов должны быть запущены
	Synchronous OR	Один или несколько предшествующих процессов завершены одновременно	Один или несколько следующих процессов запускаются одновременно
	XOR (Exclusive OR)	Только один предшествующий процесс завершен	Только один следующий процесс запускается

В отличие от IDEF0 в IDEF3 стрелки могут сливаться и разветвляться только через перекрестки.

*Декомпозиция работ.* В IDEF3 декомпозиция используется для детализации работ. Методология IDEF3 позволяет декомпозировать работу многократно, т. е. работа может иметь множество дочерних работ. Это позволяет в одной модели описать альтернативные потоки. Возможность множественной декомпозиции предъявляет дополнительные требования к нумерации работ. Так, номер работы состоит из номера родительской работы, версии декомпозиции и собственного номера работы на текущей диаграмме (рис. 16).

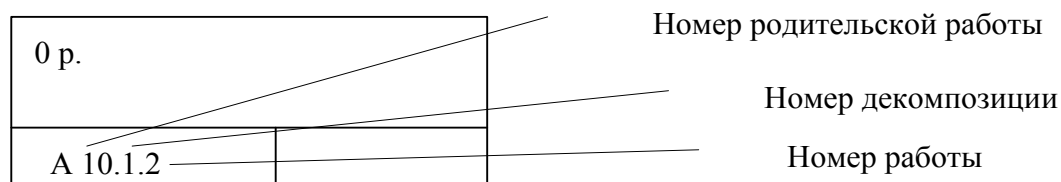


Рис. 16. Номер единицы работы (UOW)

Такая диаграмма состоит из трех типов узлов: узлов обработки данных, узлов хранения данных и внешних узлов, представляющих внешние, по отношению к используемой диаграмме, источники или потребители данных. Дуги в диаграмме соответствуют потокам данных, передаваемых от узла к узлу. Они помечены именами соответствующих данных. Описание процесса, функции или системы обработки данных, соответствующих узлу диаграммы, может быть представлено диаграммой следующего уровня детализации, если процесс достаточно сложен.

Для изображения диаграмм потоков данных традиционно используют два вида нотаций: нотации Иордана и Гейна-Сарсона (табл. 5).

Первым шагом при построении иерархии диаграмм потоков данных является построение контекстных диаграмм, показывающих как система будет взаимодействовать с пользователями и другими внешними системами. При проектировании простых систем достаточно одной контекстной диаграммы, имеющей звездную топологию, в центре которой располагается основной процесс, соединенный с источниками и приемниками информации (рис. 17)

После построения контекстных диаграмм полученную модель следует проверить на полноту исходных данных и отсутствие информационных связей с другими объектами.

Таблица 5

## Нотации диаграмм потоков данных

Понятие	Описание	Нотация Йордана	Нотация Гейна-Сарсона
Внешняя сущность	Внешний по отношению к системе объект, обменивающийся с ней потоками данных	Имя внешнего объекта	Номер Имя
Функция	Действие, выполняемое моделируемой системой	Имя функции	Номер Имя Механизм
Поток данных	Объект, над которым выполняется действие. Может быть информационным (логическим) или управляющим. (Управляющие потоки обозначаются пунктирной линией со стрелкой).	Имя объекта	Имя объекта
Хранилище данных	Структура для хранения информационных объектов	№ Имя объекта	№ Имя объекта

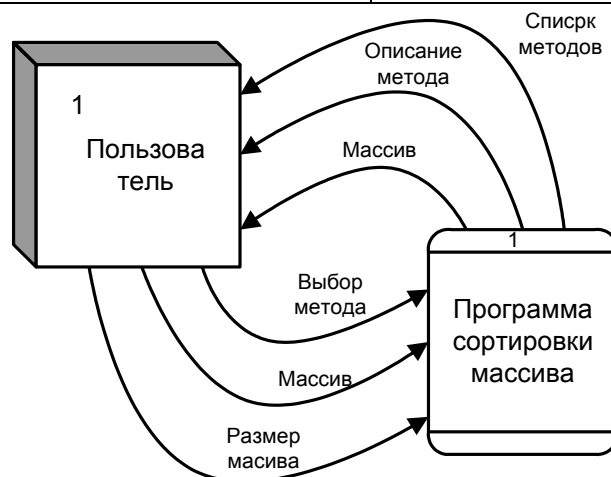


Рис. 17. Контекстная диаграмма программы построения графиков/таблиц функций

Для каждой подсистемы, присутствующей на контекстных диаграммах, выполняется ее детализация при помощи диаграмм потоков данных, при этом необходимо соблюдать следующие правила:

- правило балансировки – означает, что при детализации подсистемы можно использовать только те компоненты (подсистемы, процессы, внешние сущности, накопители данных), с которыми она имеет информационную связь на родительской диаграмме;

- правило нумерации – означает, что при детализации подсистем должна поддерживаться их иерархическая нумерация. Например, подсистемы, детализирующие подсистему с номером 1, получают номера 1.1, 1.2, 1.3 и т. д.

При построении иерархии диаграмм потоков данных переходить к детализации процессов следует только после определения структур данных, которые описывают содержание всех потоков и накопителей данных. Структуры данных могут содержать альтернативы, условные вхождения и итерации. Условное вхождение означает, что соответствующие компоненты могут отсутствовать в структуре. Альтернатива означает, что в структуру может входить один из перечисленных элементов. Итерация означает, что компонент может повторяться в структуре некоторое указанное число раз. Для каждого элемента данных может указываться его тип (непрерывный или дискретный). Для непрерывных данных может указываться единица измерения (кг, см и т. п.), диапазон значений, точность представления и форма физического кодирования. Для дискретных данных может указываться таблица допустимых значений.

После каждого уровня детализации должна проводиться проверка полноты и наглядности модели.

Базовыми понятиями ER-модели данных (ER – Entity-Relationship) являются: сущность, атрибут и связь.

Поскольку нотация Баркера является наиболее распространенной, в дальнейшем будем придерживаться именно ее.

*Сущность* – это класс однотипных объектов, информация о которых должна быть учтена в модели. Сущность имеет наименование, выраженное

существительным в единственном числе и обозначается в виде прямоугольника с наименованием (рис. 19 а). Примерами сущностей могут быть такие классы объектов как «Студент», «Сотрудник», «Товар».

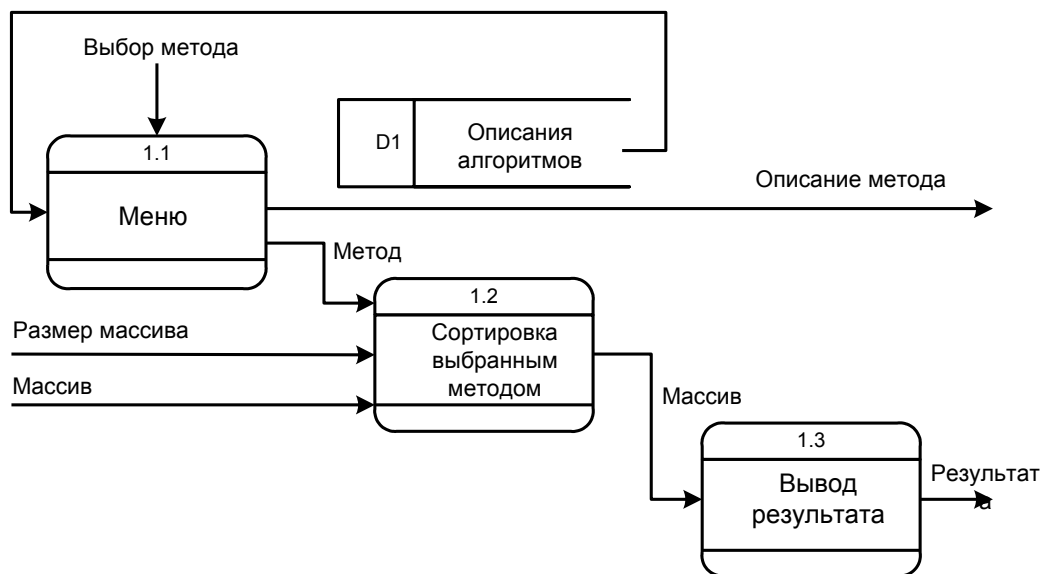


Рис. 18. Детализирующая диаграмма потоков данных программы сортировки одномерного массива (нотация Гейна-Сарсона)

*Экземпляр сущности* – это конкретный представитель данной сущности. Например, конкретный представитель сущности «Студент» – «Максимов». Причем сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности, для того чтобы различать экземпляры.

*Атрибут сущности* – это именованная характеристика, являющаяся некоторым свойством сущности. Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с описательными оборотами или прилагательными). Примерами атрибутов сущности «Студент» могут быть такие атрибуты как «Номер зачетной книжки», «Фамилия», «Имя», «Пол», «Возраст», «Средний балл» и т. п. Атрибуты изображаются в прямоугольнике, обозначающем сущность (рис. 19 б).

*Ключ сущности* – это избыточный набор атрибутов, значения которых в совокупности являются уникальными для каждого экземпляра

сущности. При удалении любого атрибута из ключа нарушается его уникальность. Ключей у сущности может быть несколько. На диаграмме ключевые атрибуты отображаются подчеркиванием (рис. 19 в).

*Связь* – это отношение одной сущности к другой или к самой себе. Возможно по одной сущности находить другие, связанные с ней. Например, связи между сущностями могут выражаться следующими фразами – «СОТРУДНИК может иметь несколько ДЕТЕЙ», «СОТРУДНИК обязан числиться точно в одном ОТДЕЛЕ». Графически связь изображается линией, соединяющей две сущности (рис. 20):

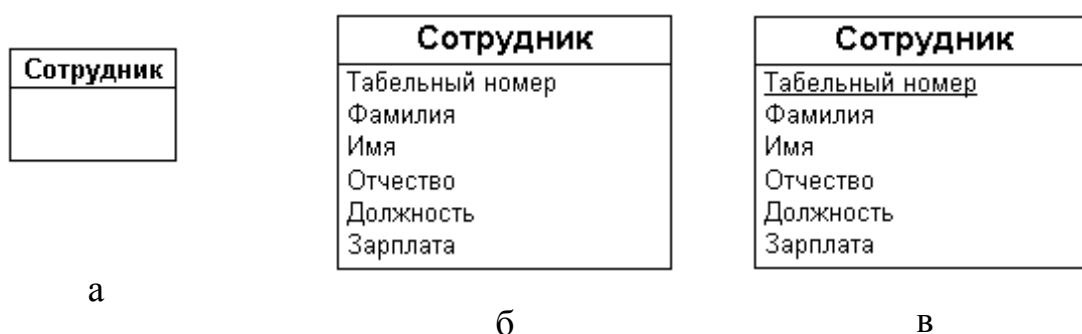


Рис. 19. Обозначения сущности в нотации Баркера:

а – без атрибутов; б – с указанием атрибутов; в – с ключевым атрибутом

Каждая связь имеет одно или два наименования. Наименование обычно выражается неопределенной формой глагола: «Продавать», «Быть проданным» и т. п. Каждое из наименований относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности.

Связь типа *один-к-одному* означает, что один экземпляр первой сущности связан точно с одним экземпляром второй сущности. Такая связь чаще всего.

Связь типа *один-ко-многим* означает, что один экземпляр первой сущности связан свидетельствует о том, что мы неправильно разделили одну сущность, на две с несколькими экземплярами второй сущности. Это наиболее часто используемый тип связи. Пример такой связи приведен на рис. 20.



Связь типа *много-ко-многим* означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и наоборот. Этот тип связи является временным, допустимым на ранних этапах разработки модели. В дальнейшем такую связь необходимо заменить двумя связями типа *один-ко-многим* путем создания промежуточной сущности.

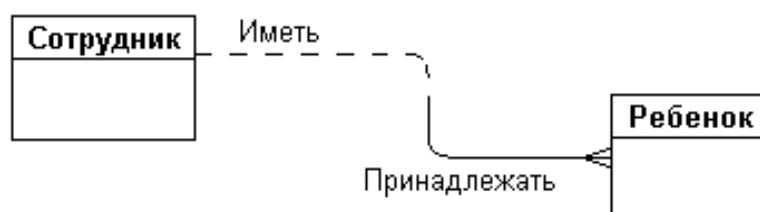


Рис. 20. Пример связи между сущностями

Связь может иметь один из следующих **типов**:

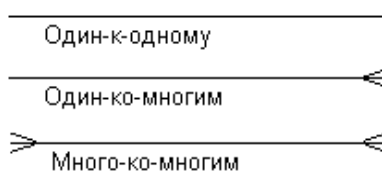


Рис.21. Типы связей

Каждая связь может иметь одну из двух модальностей связи: Связь может иметь разную модальность с разных концов (рис. 22). Каждая связь может быть прочитана как слева направо, так и справа налево. Связь на рис. 20 читается так:

Слева направо: «Сотрудник может иметь несколько детей».

Справа налево: «Ребенок должен принадлежать точно одному сотруднику».

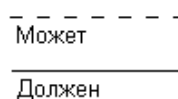


Рис. 22. Модальности связей

Пример разработки простой ER-диаграммы приведен на рисунке 23.

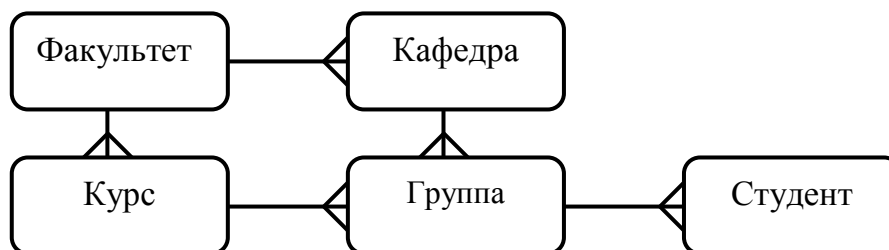


Рис. 23. Пример ER-диаграммы базы данных сведений о студентах

### Порядок выполнения работы:

1. На основе технического задания из лабораторной работы № 1 выполнить анализ функциональных и эксплуатационных требований к программному продукту.
2. Определить функциональные диаграммы (верхнего уровня и детализирующие).
3. Определить диаграммы переходов состояний.
4. Определить диаграммы потоков данных для решаемой задачи (контекстную и детализирующую).
5. Определить диаграммы «сущность-связь», если в задании присутствует база данных. Разработать укрупненную схему алгоритма, псевдокод, Flow-диаграмму (по выбору) программного модуля.
6. Добавить словарь терминов (2–3 шт.).

## **ЛАБОРАТОРНАЯ РАБОТА № 3.**

### **Реализация программного обеспечения**

#### **Цель работы**

Разработать программный продукт в соответствии с заданным вариантом.

#### **Теоретическая часть**

Составление программной документации Важным этапом разработки программного продукта является составление программной документации. Жизненный цикл программного обеспечения содержит специальный процесс, посвященный этому вопросу. На каждый программный продукт должны составляться два типа документации – для разработчиков и для различных групп пользователей. Программная документация пользователей должна содержать все необходимые сведения по эксплуатации ПО. Аналогично, документация разработчика должна содержать сведения, необходимые для разработки и сопровождения программного обеспечения.

Документирование программного обеспечения осуществляется в соответствии с Единой системой программной документации (ГОСТ 19.XXX). ГОСТ 19.101–77 содержит виды программных документов для программного обеспечения различных типов. В данном ГОСТ перечислены документы следующих типов:

- спецификация должна содержать перечень и краткое описание назначения всех файлов программного обеспечения, в том числе и файлов документации на него, и является обязательной для программных систем, а также их компонентов, имеющих самостоятельное применение;
- ведомость держателей подлинников (код вида документа – 05) должна содержать список предприятий, на которых хранятся подлинники программных документов. Необходимость этого документа определяется

на этапе разработки и утверждения технического задания только для программного обеспечения со сложной архитектурой;

- текст программы должен содержать текст программы с необходимыми комментариями. Необходимость этого документа определяется на этапе разработки и утверждения технического задания;

- описание программы должно содержать сведения о логической структуре и функционировании программы. Необходимость данного документа также определяется на этапе разработки и утверждения технического задания;

- ведомость эксплуатационных документов должна содержать перечень эксплуатационных документов на программу. Необходимость этого документа также определяется на этапе разработки и утверждения технического задания;

- формуляр должен содержать основные характеристики программного обеспечения, комплектность и сведения об эксплуатации программы;

- описание применения должно содержать сведения о назначении программного обеспечения, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств;

- руководство системного программиста должно содержать сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения;

- руководство программиста должно содержать сведения для эксплуатации программного обеспечения;

- руководство оператора содержит сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программы;

- описание языка – описание синтаксиса и семантики языка программы;

– руководство по техническому обслуживанию содержит сведения для применения программы при обслуживании технических средств.

### **Порядок выполнения работы**

1. По результатам лабораторных работ № 1, 2 написать код программ для решения поставленной задачи на выбранном языке программирования.

2. Отладить программный модуль.

3. Получить результаты работы.

4. Оформить документацию к разработанному программному обеспечению.

5. Сдать и защитить отчет по лабораторной работе.

Отчет по лабораторной работе должен состоять из:

1. Листингов программ.

2. Интерфейса пользователя.

3. Документации к программному обеспечению (руководство пользователя, руководство системного программиста, руководство программиста, руководство оператора).

4. Результатов работы программ.

## ЛАБОРАТОРНАЯ РАБОТА № 4. Тестирование программ

**Цель работы:** изучить методы тестирования логики программы, формализованные описания результатов тестирования и стандарты по составлению схем программ.

### Теоретическая часть

Тестирование программного обеспечения включает в себя целый комплекс действий, аналогичных последовательности процессов разработки программного обеспечения. В него входят :

- постановка задачи для теста;
- проектирование теста;
- написание тестов;
- тестирование тестов;
- выполнение тестов;
- изучение результатов тестирования.

Наиболее важным является проектирование тестов. Существуют разные подходы к проектированию тестов.

Первый состоит в том, что тесты проектируются на основе внешних спецификаций программ и модулей либо спецификаций сопряжения модуля с другими модулями, программа при этом рассматривается как «черный ящик». Смысл теста заключается в том, чтобы проверить, соответствует ли программа внешним спецификациям. При этом содержание модуля не имеет значения. Такой подход получил название – стратегия «черного ящика».

Второй подход – стратегия «белого ящика», основан на анализе логики программы. При таком подходе тестирование заключается в проверке каждого пути, каждой ветви алгоритма. При этом внешняя спецификация во внимание не принимается.

Ни один из этих подходов не является оптимальным. Реализация тестирования методом «черного ящика» сводится к проверке всех

возможных комбинаций входных данных. Невозможно протестировать программу, подавая на вход бесконечное множество значений, поэтому ограничиваются определенным набором данных. При этом исходят из максимальной отдачи теста по сравнению с затратами на его создание. Она измеряется вероятностью того, что тест выявит ошибки, если они имеются в программе. Затраты измеряются временем и стоимостью подготовки, выполнения и проверки результатов теста.

Тестирование методом «белого ящика» также не дает 100 %-ной гарантии того, что модуль не содержит ошибок. Даже если предположить, что выполнены тесты для всех ветвей алгоритма, нельзя с полной уверенностью утверждать, что программа соответствует ее спецификациям. Например, если требовалось написать программу для вычисления кубического корня, а программа фактически вычисляет корень квадратный, то реализация будет совершенно неправильной, даже если проверить все пути. Вторая проблема – отсутствующие пути. Если программа реализует спецификации не полностью (например, отсутствует такая специализированная функция, как проверка на отрицательное значение входных данных программы вычисления квадратного корня), никакое тестирование существующих путей не выявит такой ошибки. И наконец, проблема зависимости результатов тестирования от входных данных. Одни данные будут давать правильные результаты, а другие нет. Например, если для определения равенства трех чисел программируется выражение вида:

$$\text{IF } (A + B + C)/3 = D$$

то оно будет верным не для всех значений  $A$ ,  $B$  и  $C$  (ошибка возникает в том случае, когда из двух значений  $B$  или  $C$  одно больше, а другое на столько же меньше  $A$ ). Если концентрировать внимание только на тестировании путей, нет гарантии, что эта ошибка будет выявлена.

Таким образом, полное тестирование программы невозможно, т. е. никакое тестирование не гарантирует полное отсутствие ошибок в

программе. Поэтому необходимо проектировать тесты таким образом, чтобы увеличить вероятность обнаружения ошибки в программе.

Существуют следующие методы тестирования по принципу «белого ящика»:

- покрытие операторов;
- покрытие решений;
- покрытие условий;
- покрытие решений/условий;
- комбинаторное покрытие условий.

Метод покрытия операторов Целью этого метода тестирования является выполнение каждого оператора программы хотя бы один раз.

Если для тестирования задать значения переменных  $A = 2$ ,  $B = 0$ ,  $X=3$ , будет реализован путь *ace*, т. е. каждый оператор программы выполнится один раз (рис. 24 а). Но если внести в алгоритм ошибки – заменить в первом условии *and* на *or*, а во втором  $X > 1$  на  $X < 1$  (рис. 24 б), ни одна ошибка не будет обнаружена (табл. 4). Кроме того, путь *abd* вообще не будет охвачен тестом, и если в нем есть ошибка, она также не будет обнаружена. В табл. 4 ожидаемый результат определяется по блок-схеме на рис. 24 а, а фактический – по рис. 24 б.

Как видно из этой таблицы, ни одна из внесенных в алгоритм ошибок не будет обнаружена.

Таблица 4.

Результат тестирования методом покрытия операторов

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A = 2, B = 0, X=3$	$X=2,5$	$X=2,5$	Неуспешно



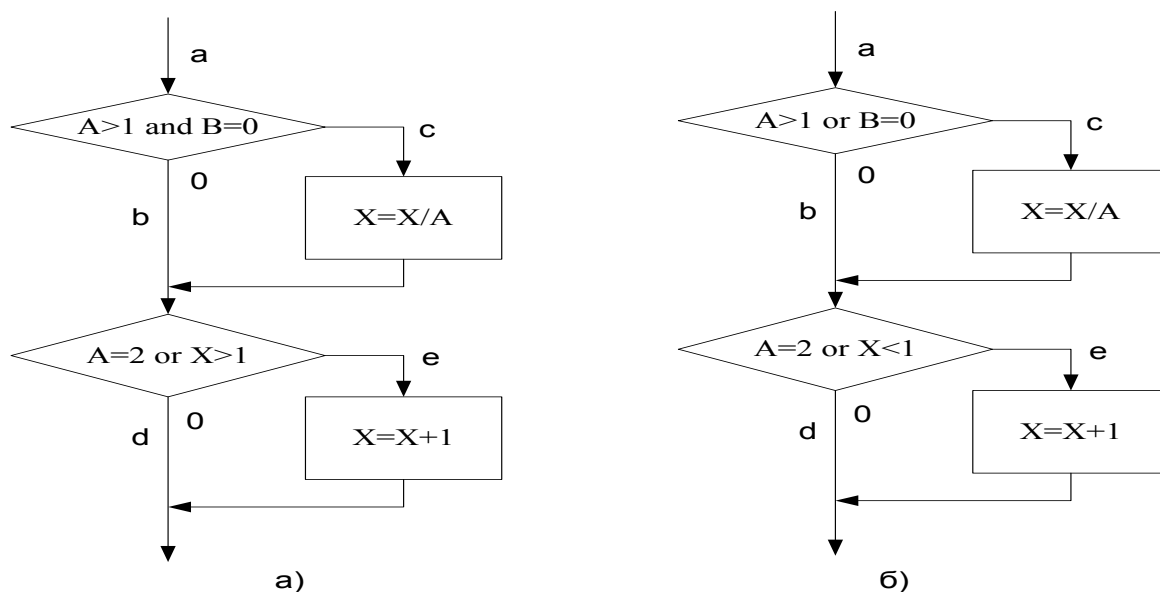


Рис. 24. Пример программы: а – правильный, б – с ошибкой

Метод покрытия решений (покрытия переходов). Согласно методу покрытия решений каждое направление перехода должно быть реализовано, по крайней мере, один раз. Этот метод включает в себя критерий покрытия операторов, так как при выполнении всех направлений переходов выполнятся все операторы, находящиеся на этих направлениях.

Для программы, приведенной на рис. 24, покрытие решений может быть выполнено двумя тестами, покрывающими пути  $\{ace, abd\}$ , либо  $\{acd, abe\}$ . Для этого выберем следующие исходные данные:  $\{A = 3, B=0, X=3\}$  – в первом случае и  $\{A = 2, B=1, X= 1\}$  – во втором. Однако путь, где  $A$  не меняется, будет проверен с вероятностью 50 %: если во втором условии вместо условия  $X > 1$  записано  $X < 1$ , то ошибка не будет обнаружена двумя тестами.

Результаты тестирования приведены в табл. 5.

Таблица 5 .

Результат тестирования методом покрытия решений

Тест	Ожидаемый	Фактический	Результат
$A = 3, B=0, X=3$	$X=1$	$X=1$	Неуспешно
$A = 2, B=1, X= 1$	$X=1$	$X=1,5$	Успешно

Метод покрытия условий. Этот метод может дать лучшие результаты по сравнению с предыдущими. В соответствии с методом покрытия условий

записывается число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз.

В рассматриваемом примере имеем условия:  $\{A > 1, B = 0\}$ ,  $\{A = 2, X > 1\}$ . Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где  $A > 1$ ,  $A < 1$ ,  $B = 0$  и  $B <> 0$  в точке  $a$  и  $A = 2$ ,  $A <> 2$ ,  $X > 1$  и  $X < 1$  в точке  $b$ . Тесты, удовлетворяющие критерию покрытия условий (табл. 6), и соответствующие им пути:

а)  $A = 2, B = 0, X = 4$  *ace*;

б)  $A = 1, B = 1, X = 0$  *abd*.

Таблица 6.

Результаты тестирования методом покрытия условий

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A = 2, B = 0, X = 4$	$X = 3$	$X = 3$	Неуспешно
$A = 1, B = 1, X = 0$	$X = 0$	$X = 1$	Успешно

Метод покрытия решений/условий. Критерий покрытия решений/условий требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия выполнялись по крайней мере один раз, все результаты каждого решения выполнялись по крайней мере один раз и, кроме того, каждой точке входа передавалось управление по крайней мере один раз.

Недостатки метода:

- не всегда можно проверить все условия;
- невозможно проверить условия, которые скрыты другими условиями;
- недостаточная чувствительность к ошибкам в логических выражениях.

Так, в рассматриваемом примере два теста метода покрытия условий

а)  $A = 2, B = 0, X = 4$  *ace*;

б)  $A=1, B=1, X=0$  *abd*

отвечают и критерию покрытия решений/условий. Это является следствием того, что одни условия приведенных решений скрывают другие условия в этих решениях. Так, если условие  $A > 1$  будет ложным, транслятор может не проверять условия  $B=0$ , поскольку при любом результате условия,  $B=0$  результат решения  $((A > 1) \& (B=0))$  примет значение *ложь*. То есть в варианте на рис. 24 не все результаты всех условий выполняются в процессе тестирования.

Рассмотрим реализацию того же примера на рис. 25. Наиболее полное покрытие тестами в этом случае осуществляется так, чтобы выполнялись все возможные результаты каждого простого решения. Для этого нужно покрыть пути *aseg* (тест  $A = 2, B=0, X=4$ ), *acdfli* (тест  $A = 3, B=1, X=0$ ), *abfli* (тест  $A = 0, B=0, X=0$ ), *abfl* (тест  $A = 0, B=0, X=2$ ).

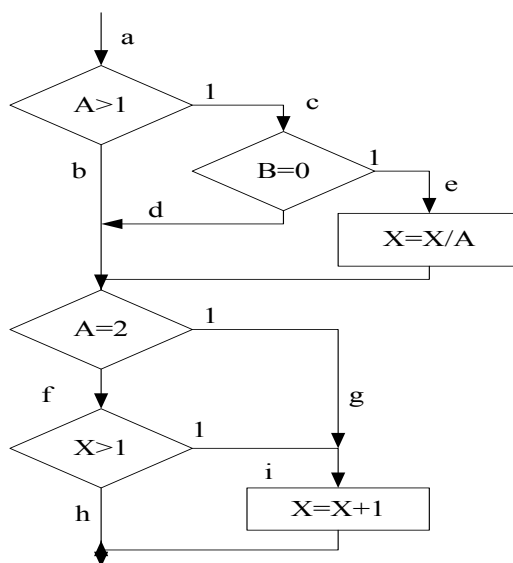


Рис. 25. Пример алгоритма программы

Протестировав алгоритм на рис. 25, нетрудно убедиться в том, что критерии покрытия условий и критерии покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

Метод комбинаторного покрытия условий. Критерий комбинаторного покрытия условий удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.

Этот метод требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении выполнялись по крайней мере один раз. По этому критерию в рассматриваемом примере должны быть покрыты тестами следующие восемь комбинаций:

1.  $A > 1, B = 0$ . 5.  $A = 2, X > 1$ .

2.  $A > 1, B < 0$ . 6.  $A = 2, X < 1$ .

3.  $A < 1, B = 0$ . 7.  $A < 2, X > 1$ .

4.  $A < 1, B < 0$ . 8.  $A < 2, X < 1$ .

Для того чтобы протестировать эти комбинации, необязательно использовать все 8 тестов. Фактически они могут быть покрыты четырьмя тестами (табл. 7):

–  $A = 2, B = 0, X = 4$  {покрывает 1, 5};

–  $A = 2, B = 1, X = 1$  {покрывает 2, 6};

–  $A = 0,5, B = 0, X = 2$  {покрывает 3, 7};

–  $A = 1, B = 0, X = 1$  {покрывает 4, 8}.

Таблица 7

Результаты тестирования методом комбинаторного покрытия условий

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A = 2, B = 0, X = 4$	$X = 3$	$X = 3$	Неуспешно
$A = 2, B = 1, X = 1$	$X = 2$	$X = 1,5$	Успешно
$A = 0,5, B = 0, X = 2$	$X = 3$	$X = 4$	Успешно
$A = 1, B = 0, X = 1$	$X = 1$	$X = 1$	Неуспешно

Функциональное тестирование или тестирование методом черного ящика базируется на том, что все тесты основаны на спецификации

системы или ее компонентов. Поведение системы определяется изучением ее входных и соответствующих выходных данных.

При тестировании «черного ящика» тестировщик имеет доступ к ПО только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов с уверенностью в том, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши.

Методология составления тестов «чёрного ящика»:

- эквивалентное разбиение;
- анализ граничных значений;
- применение функциональных диаграмм;
- предположение об ошибке.

*Эквивалентное разбиение.* Тестирование программы ограничивается использованием небольшого подмножества всех возможных входных данных. Тогда, конечно, хотелось бы выбрать для тестирования самое подходящее подмножество (т. е. подмножество с наивысшей вероятностью обнаружения большинства ошибок). Правильно выбранный тест этого подмножества должен обладать двумя свойствами:

- уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;
- покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Указанные свойства, несмотря на их кажущееся подобие, описывают два различных положения. Во-первых, каждый тест должен включать столько различных входных условий, сколько это возможно, с тем, чтобы минимизировать общее число необходимых тестов. Во-вторых, необходимо пытаться разбить входную область программы на конечное число классов эквивалентности так, чтобы можно было предположить (конечно, не абсолютно уверенно), что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если один тест класса эквивалентности обнаруживает ошибку, то следует ожидать, что и все другие тесты этого класса эквивалентности будут обнаруживать ту же самую ошибку. Наоборот, если тест не обнаруживает ошибки, то следует ожидать, что ни один тест этого класса эквивалентности не будет обнаруживать ошибки (в том случае, когда некоторое подмножество класса эквивалентности не попадает в пределы любого другого класса эквивалентности, так как классы эквивалентности могут пересекаться). Эти два положения составляют основу методологии тестирования по принципу черного ящика, известной как эквивалентное разбиение. Второе положение используется для разработки набора «интересных» условий, которые должны быть протестированы, а первое – для разработки минимального набора тестов, покрывающих эти условия. Примером класса эквивалентности для программы о треугольнике является набор «трех равных чисел, имеющих целые значения, большие нуля». Определяя этот набор как класс эквивалентности, устанавливают, что если ошибка не обнаружена некоторым тестом данного набора, то маловероятно, что она будет обнаружена другим тестом набора. Иными словами, в этом случае время тестирования лучше затратить на что-нибудь другое (на тестирование других классов эквивалентности).

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

1. выделение классов эквивалентности
2. построение тестов.

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп. Заметим, что различают два типа классов эквивалентности: *правильные классы эквивалентности*, представляющие правильные входные данные программы, и *неправильные классы эквивалентности*, представляющие все другие возможные состояния условий (т. е. ошибочные входные значения). При этом существует ряд правил:

1. Если входное условие описывает область значений (например, «целое данное может принимать значения от 1 до 999»), то определяются один правильный класс эквивалентности ( $1 \leq \text{значение целого данного} \leq 999$ ) и два неправильных (значение целого данного  $< 1$  и значение целого данного  $> 999$ ).

2. Если входное условие описывает множество входных значений и есть основание полагать, что каждое значение программа трактует особо (например, «известны способы передвижения на АВТОБУСЕ, ГРУЗОВИКЕ, ТАКСИ, ПЕШКОМ или МОТОЦИКЛЕ»), то определяется правильный класс эквивалентности для каждого назначения и один неправильный класс эквивалентности (например, «НА ПРИЦЕПЕ»).

3. Если входное условие описывает ситуацию «должно быть» (например, «первым символом идентификатора должна быть буква»), то определяется один правильный класс эквивалентности (первый символ – буква) и один неправильный (первый символ – не буква).

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

Второй шаг заключается в использовании классов эквивалентности для построения тестов. Этот процесс включает в себя:

1. Назначение каждому классу эквивалентности уникального номера.

2. Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых правильных классов эквивалентности, до тех пор, пока все правильные классы эквивалентности не будут покрыты (только не общими) тестами.

3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы эквивалентности не будут покрыты тестами. Причина покрытия неправильных классов эквивалентности индивидуальными тестами состоит в том, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами. Например, спецификация устанавливает «тип книги при поиске (ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА, ПРОГРАММИРОВАНИЕ или ОБЩИЙ) и количество (1-9999)». Тогда тест XYZ О отображает два ошибочных условия (неправильный тип книги и количество) и, вероятно, не будет осуществлять проверку количества, так как программа может ответить: «XYZ-НЕСУЩЕСТВУЮЩИЙ ТИП КНИГИ» и не проверять остальную часть входных данных.

Предположим, что при разработке компилятора для подмножества языка программирования требуется протестировать синтаксическую проверку оператора DIM[1].

Пусть определена следующая спецификация:

Оператор DIM используется для определения массивов.

$\text{DIM } ad [, ad] \dots$ , где  $ad$  есть описатель массива в форме  $n(d[,d] \dots)$ ,  $n$  – символическое имя массива, а  $d$  – индекс массива.

Символические имена могут содержать от одного до шести символов – букв или цифр, причем первой должна быть буква.

Допускается от одного до семи индексов. Форма индекса  $[lb:] ub$ , где  $lb$  и  $ub$  задают нижнюю и верхнюю границы индекса массива. Граница



может быть либо константой, принимающей значения от – 65534 до 65535, либо целой переменной (без индексов). Если lb не определена, то предполагается, что она равна единице. Значение ub должно быть больше или равно lb. Если lb определена, то она может иметь отрицательное, нулевое или положительное значение. Оператор может располагаться на нескольких строках

Первый шаг заключается в том, чтобы идентифицировать входные условия и по ним определить классы эквивалентности. Классы эквивалентности в таблице обозначены числами.

Таблица 8

### Выделение классов эквивалентности

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности
Число описателей массивов	ОДИН (1), > ОДНОГО (2)	НИ ОДНОГО (3)
Длина имени массива	1-6(4)	0(5), > 6(6)
Имя массива	Имеет в своем составе буквы (7) и цифры (8)	содержит что-то еще (9)
Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности
Имя массива начинается с буквы	да (10)	нет (11)
Число индексов	1-7(12)	0(13), > 7(14)
Верхняя граница	Константа (15), целая переменная (16)	имя элемента массива (17), что-то иное (18)

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности
Имя переменной	Имеет в своем составе буквы (19), и цифры (20)	состоит из чего-то еще (21)
Переменная начинается с буквы	да (22)	нет (23)
Константа	От -65534 до 65535 (24))	Меньше –65534 (25), больше 65535 (26)
Нижняя граница определена	да (27), нет (28)	
Верхняя к нижней границе	Больше (29), равна (30)	меньше (31)
Значение нижней границы	Отрицательное (32) Нуль (33), > 0 (34)	
Нижняя граница	Константа (35), целая переменная (36)	имя элемента массива (37), что-то иное (38)
Оператор расположен на нескольких строках	да (39), нет (40)	

Следующий шаг – построение теста, покрывающего один или более правильных классов эквивалентности. (DIM A(2) покрывает классы 1, 4, 7, 10, 12, 15, 24, 28, 29 и 40.)

Далее определяются один или более тестов, покрывающих оставшиеся правильные классы эквивалентности. (DIM A12345(I,9,J4XXXX.65535,1,KLM, X 100), BBB (–65534 : 100,0 : 1000,10 : 10,1 : 65535) покрывает оставшиеся классы.)

Неправильные классы эквивалентности и соответствующие им тесты:

(3)	DIMENSION	(21)	DIMENSION C(1.,10)
(5)	DIMENSION(10)	(23)	DIMENSION C(10,1J)
((6)	DIMENSION A234567(2)	(25)	DIMENSION D (-65535:1)
(9)	DIMENSION A.I(2)	(26)	DIMENSION D(65536)
(11)	DIMENSION1A(10)	(31)	DIMENSION D(4:3)
(13)	DIMENSION B	(37)	DIMENSION D(A(2):4)
(14)	DIMENSION B (4,4,4,4,4,4,4,4)	(38)	DIMENSION D(:4)
(17)	DIMENSION B(4,A(2))		
(18)	DIMENSION B (47)		

Хотя эквивалентное разбиение значительно лучше случайного выбора тестов, оно все же имеет недостатки (т. е. пропускает определенные типы высокоэффективных тестов).

Следующие два метода – анализ граничных значений и использование функциональных диаграмм (диаграмм причинно–следственных связей cause–effect graphing) – свободны от многих недостатков, присущих эквивалентному разбиению.

*Анализ граничных значений.* Как показывает опыт, тесты, исследующие граничные условия, приносят большую пользу, чем тесты, которые их не исследуют. Граничные условия – это ситуации, возникающие непосредственно «на», «выше» или «ниже» границ входных и выходных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения в двух отношениях:

1. Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.

2. При разработке тестов рассматривают не только входные условия (пространство входов), но и пространство результатов (т. е. выходные классы эквивалентности) .

Приведем несколько общих правил этого метода.

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений. Например, если правильная область входных значений есть  $-1,0 - +1,0$ , то написать тесты для ситуаций  $-1,0$ ,  $1,0$ ,  $-1,001$  и  $1,001$ .

2. Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то получить тесты для 0,1, 255 и 256 записей.

3. Использовать правило 1 для каждого выходного условия. Например, если программа вычисляет ежемесячный расход и если минимум расхода составляет 0,00 дол., а максимум – 1165,25 дол., то построить тесты, которые вызывают расходы с 0,00 дол. и 1165,25 дол. Кроме того, построить, если это возможно, тесты, которые вызывают отрицательный расход и расход больше 1165,25 дол. Заметим, что важно проверить границы пространства результатов, поскольку не всегда

границы входных областей представляют такой же набор условий, как и границы выходных областей (например, при рассмотрении подпрограммы вычисления синуса). Не всегда также можно получить результат вне выходной области, но, тем не менее, стоит рассмотреть эту возможность.

4. Использовать правило 2 для каждого выходного условия. Например, если система информационного поиска отображает на экране терминала наиболее релевантные рефераты в зависимости от входного запроса, но никак не более четырех рефератов, то построить тесты, такие, чтобы программа отображала нуль, один и четыре реферата, и тест, который мог бы вызвать выполнение программы с ошибочным отображением пяти рефератов.

5. Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

Существенное различие между анализом граничных значений и эквивалентным разбиением заключается в том, что анализ граничных значений исследует ситуации, возникающие на и вблизи границ эквивалентных разбиений.

Предположим, что нужно протестировать программу бинарного поиска. Нам известна *спецификация* этой программы. Поиск выполняется в массиве элементов  $M$ , возвращается индекс  $I$  элемента массива, значение которого соответствует ключу поиска  $Key$ .

Входные условия:

- 1) массив должен быть упорядочен;
- 2) массив должен иметь не менее одного элемента;
- 3) нижняя граница массива (индекс) должна быть меньше или равна его верхней границе.

Результаты:

1) если элемент найден, то флаг Result=True, значение I – номер элемента;

2) если элемент не найден, то флаг Result=False, значение I не определено.

Для формирования классов эквивалентности (и их ребер) надо построить дерево разбиений. Листья дерева разбиений дадут нам искомые классы эквивалентности. Определим стратегию разбиения. На первом уровне будем анализировать выполнимость входных условий, на втором уровне – выполнимость результатов. На третьем уровне можно анализировать специальные требования, полученные из практики разработчика. В нашем примере мы знаем, что входной массив должен быть упорядочен. Обработка упорядоченных наборов из четного и нечетного количества элементов может выполняться по-разному. Кроме того, принято выделять специальный случай одноэлементного массива. Следовательно, на уровне специальных требований возможны следующие эквивалентные разбиения:

1) массив из одного элемента;

2) массив из четного количества элементов;

3) массив из нечетного количества элементов, большего единицы.

Наконец на последнем, 4-м уровне критерием разбиения может быть анализ ребер классов эквивалентности. Очевидно, возможны следующие варианты:

1) работа с первым элементом массива;

2) работа с последним элементом массива;

3) работа с промежуточным (ни с первым, ни с последним) элементом массива.

Структура дерева разбиений приведена на рис. 26.

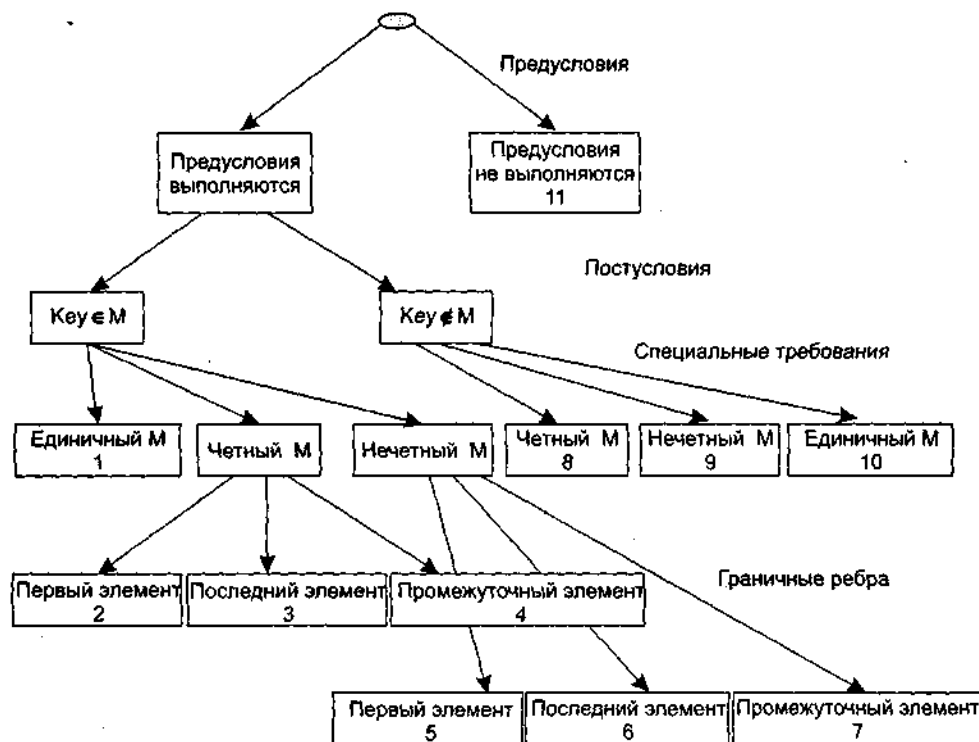


Рис. 26. Дерево разбиений области исходных данных  
бинарного поиска

Это дерево имеет 11 листьев. Каждый лист задает отдельный тестовый вариант. Покажем тестовые варианты, основанные на проведенных разбиениях.

*Тестовый вариант 1 (единичный массив, элемент найден):*

*ИД:* M=15; Key=15.

*ОЖ.РЕЗ.:* Result=True; I=1.

*Тестовый вариант 2 (четный массив, найден 1-й элемент):*

*ИД:* M=15, 20, 25, 30, 35, 40; Key=15.

*ОЖ.РЕЗ.:* Result=True; I=1.

*Тестовый вариант 3 (четный массив, найден последний элемент) :*

*ИД:* M=15, 20, 25, 30, 35, 40; Key=40.

*ОЖ.РЕЗ.:* Result=True; I=6.

*Тестовый вариант 4 (четный массив, найден промежуточный элемент):*

*ИД:* M=15, 20, 25, 30, 35, 40; Key=25.

*ОЖ.РЕЗ.:* Result=True; I=3.

*Тестовый вариант 5 (нечетный массив, найден 1-й элемент):*

*ИД:* M=15, 20, 25, 30, 35,40, 45; Key=15.

*ОЖ.РЕЗ.:* Result=True; I=1.

*Тестовый вариант 6 (нечетный массив, найден последний элемент):*

*ИД:* M=15, 20, 25, 30,35, 40,45; Key=45.

*ОЖ.РЕЗ.:* Result=True; I=7.

*Тестовый вариант 7 (нечетный массив, найден промежуточный элемент):*

*ИД:* M=15, 20, 25, 30,35, 40, 45; Key=30.

*ОЖ.РЕЗ.:* Result=True; I=4.

*Тестовый вариант 8 (четный массив, не найден элемент):*

*ИД:* M=15, 20, 25, 30, 35,40; Key=23.

*ОЖ.РЕЗ.:* Result=False; I=?

*Тестовый вариант 9 (нечетный массив, не найден элемент);*

*ИД:* M=15, 20, 25, 30, 35, 40, 45; Key=24.

*ОЖ.РЕЗ.:* Result=False; I=?

*Тестовый вариант 10 (единичный массив, не найден элемент):*

*ИД:* M=15; Key=0.

*ОЖ.РЕЗ.:* Result=False; I=?

*Тестовый вариант 11 (нарушены предусловия):*

*ИД:* M=15, 10, 5, 25, 20, 40, 35; Key=35.

*ОЖ.РЕЗ.:* Аварийное завершение: Массив не упорядочен.

Анализ граничных значений, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако он часто оказывается неэффективным из-за того, что внешне выглядит простым. Граничные условия могут быть едва уловимы и, следовательно, определение их связано с большими трудностями.

Одним из недостатков анализа граничных значений и эквивалентного разбиения является то, что они не исследуют комбинаций входных условий. Например, пусть программа предыдущего раздела не выполняется, число элементов массива превышает некоторый предел



(например, объем памяти). Такая ошибка не обязательно будет обнаружена тестированием граничных значений. Тестирование комбинаций входных условий - непростая задача, поскольку даже при построенном эквивалентном разбиении входных условий число комбинаций обычно астрономически велико. Если нет систематического способа выбора подмножества входных условий, то, как правило, выбирается произвольное подмножество, приводящее к неэффективному тесту.

Метод функциональных диаграмм или диаграмм причинно-следственных связей помогает систематически выбирать высокорезультативные тесты. Он дает полезный побочный эффект, так как позволяет обнаруживать неполноту и неоднозначность исходных спецификаций. Функциональная диаграмма представляет собой формальный язык, на который транслируется спецификация, написанная на естественном языке. Диаграмме можно сопоставить цифровую логическую цепь (комбинаторную логическую сеть), но для ее описания используется более простая нотация (форма записи), чем обычная форма записи, принятая в электронике. Построение тестов этим методом осуществляется в несколько этапов.

1. Спецификация разбивается на «рабочие» участки. Это связано с тем, что функциональные диаграммы становятся слишком громоздкими при применении данного метода к большим спецификациям. Например, когда тестируется система разделения времени, рабочим участком может быть спецификация отдельной команды. При тестировании компилятора в качестве рабочего участка можно рассматривать каждый отдельный оператор языка программирования.

2. В спецификации определяются причины и следствия. Причина есть отдельное входное условие или класс эквивалентности входных условий. Следствие есть выходное условие или преобразование системы (остаточное действие, которое входное условие оказывает на состояние программы или системы). Например, если сообщение программе приводит

к обновлению основного файла, то изменение в нем и является преобразованием системы; подтверждающее сообщение было бы выходным условием. Причины и следствия определяются путем последовательного (слово за словом) чтения спецификации. При этом выделяются слова или фразы, которые описывают причины и следствия. Каждым причине и следствию приписывается отдельный номер.

3. Анализируется семантическое содержание спецификации, которая преобразуется в булевский граф, связывающий причины и следствия. Это и есть функциональная диаграмма.

4. Диаграмма снабжается примечаниями, задающими ограничения и описывающими комбинации причин и (или) следствий, которые являются невозможными из-за синтаксических или внешних ограничений.

5. Путем методического прослеживания состояний условий диаграммы она преобразуется в таблицу решений с ограниченными входами. Каждый столбец таблицы решений соответствует тесту.

6. Столбцы таблицы решений преобразуются в тесты.

Используется автоматный подход к решению задачи.

*Шаги способа:*

1) для каждого модуля перечисляются причины (условия ввода или классы эквивалентности условий ввода) и следствия (действия или условия вывода). Каждой причине и следствию присваивается свой идентификатор;

2) разрабатывается граф причинно-следственных связей;

3) граф преобразуется в таблицу решений;

4) столбцы таблицы решений преобразуются в тестовые варианты

Базовые символы для записи функциональных диаграмм показаны  
рис. 27

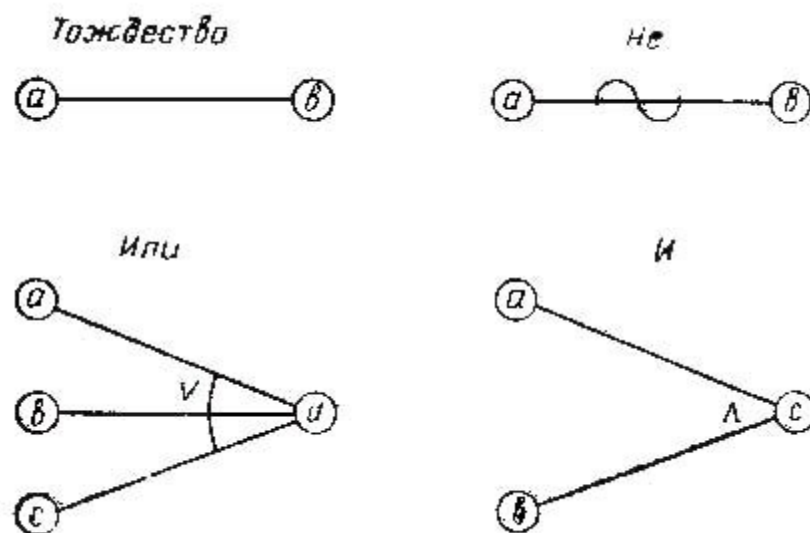


Рис. 27. Базовые логические отношения функциональных диаграмм.

Рассмотрим построение ФД и тестового набора для следующей спецификации:

Функция  $f$  принимает два параметра. Первый может принимать значение в диапазоне  $0 \div 10$  или  $20 \div 100$ . Второй параметр должен быть типа `char`. В этом случае возвращается код успешного завершения функции  $f$  - `O'k`. Если первый параметр неправильный, то выдается сообщений `ErrMes1`. Если второй параметр неправильный, то выдается сообщений `ErrMes2`.

Причинами будут:

1. Первый параметр лежит в диапазоне  $0 \div 10$ ;
2. Первый параметр лежит в диапазоне  $20 \div 100$ ;
3. Второй параметр типа `char`.

Следствия:

1. Возвращается код успешного завершения (31);
2. Выдается сообщение `ErrMes1(32)`;
3. Выдается сообщение `ErrMes2(33)`.

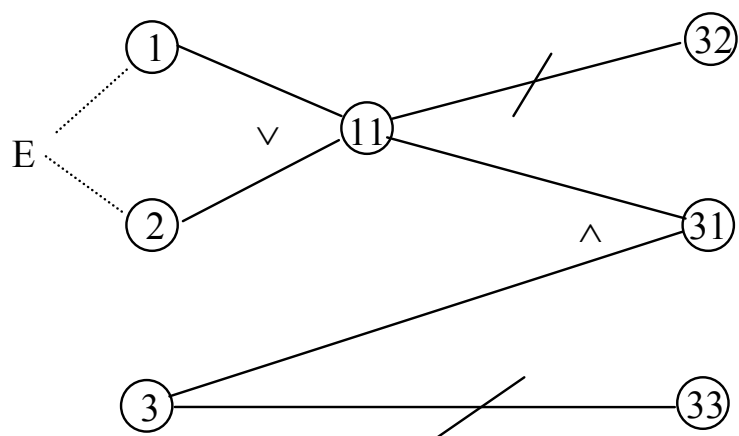


Рис. 28. Функциональная диаграмма

Косая черта означает отрицание, а объединение пунктирной линией через знак  $E$  – одновременное существование только одной из причин. В диаграмме введен узел 11, для объединения причин 1 и 2 по логическому ИЛИ.

Далее для ФД составляется таблица истинности, в которой число столбцов определяется возможными наборами значений причин.

Таблица 9

Таблица решений

Причины \ Тесты	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	0	1	1	0	0
3	0	1	0	1	0	1
11	0	0	1	1	1	1
...						
31	1	1	0	0	0	0
32	1	1	0	1	0	1
33	1	0	1	0	1	0

Каждый из 6 столбцов является тестовым набором

Отметим, что здесь создан промежуточный узел 11.

Хотя диаграмма отображает спецификацию, она содержит невозможную комбинацию причин – причины 1 и 2 не могут быть установлены в 1 одновременно. В большинстве программ определенные комбинации причин невозможны из-за синтаксических или внешних ограничений (например, символ не может принимать значения «А» и «В» одновременно). В этом случае используются дополнительные логические ограничения:

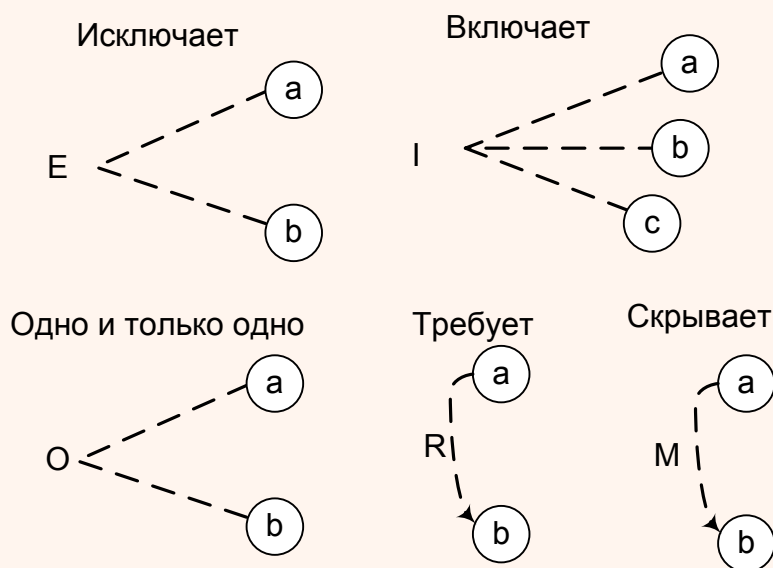


Рис. 29. Спецификация функциональных диаграмм

Ограничение E устанавливает, что E должно быть истинным, если хотя бы одна из причин – a или b – принимает значение 1 (a и b не могут принимать значение 1 одновременно). Ограничение I устанавливает, что по крайней мере, одна из величин a, b или c всегда должна быть равной 1 (a, b и c не могут принимать значение 0 одновременно). Ограничение O устанавливает, что одна и только одна из величин a или b должна быть равна 1. Ограничение R устанавливает, что если a принимает значение 1, то b должна принимать значение 1 (т. е. невозможно, чтобы a было равно 1, а b – 0). Часто возникает необходимость в ограничениях для следствий. Ограничение M устанавливает, что если следствие a имеет значение 1, то следствие b должно принять значение 0.

Как видно из рассмотренного выше примера, физически невозможно, чтобы причины 1 и 2 присутствовали одновременно, но возможно, чтобы присутствовала одна из них. Следовательно, они связаны ограничением Е.

Рассмотрим программу расчета за электричество по среднему или переменному тарифу:

При расчете по среднему тарифу

- при месячном потреблении менее 100кВт/ч, выставляется фиксированная сумма;
- при месячном потреблении  $\geq 100$  кВт/ч, применяется процедура пересчета А.

При расчете по переменному тарифу

- при месячном потреблении менее 100 кВт/ч, т/ч, применяется процедура пересчета А;
- при месячном потреблении более или  $= 100$  кВт/ч, т/ч, применяется процедура пересчета В.

I. Определим причины:

1. расчет по среднему тарифу
2. расчет по переменному тарифу
3. месячное потребление менее 100 кВт/ч
4. месячное потребление более 100 кВт/ч

Следствия

101 минимальная стоимость

102 процедура А

103 процедура В

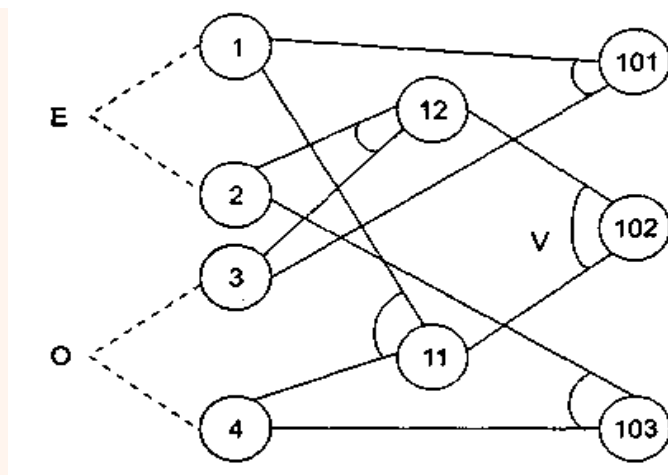


Рис. 30. Функциональная диаграмма программы расчета за электричество

Таблица решений для расчета оплаты за электричество

Номера столбцов –>			1	2	3	4
Условия	Причины	1	1	0	1	0
		2	0	1	0	1
		3	1	1	0	0
		4	0	0	1	1
	Вторичные причины	11	0	0	1	0
		12	0	1	0	0
Действия	Следствия	101	1	0	0	0
		102	0	1	1	0
		103	0	0	0	1

Преобразование каждого столбца таблицы в тестовый вариант.

В нашем примере таких вариантов четыре.

*Тестовый вариант 1 (столбец 1):*

*ИД:* расчет по среднему тарифу; месячное потребление электроэнергии 75 кВт/ч.

*ОЖ.РЕЗ.:* минимальная месячная стоимость.

*Тестовый вариант 2 (столбец 2):*

*ИД:* расчет по переменному тарифу; месячное потребление электроэнергии 90 кВт/ч.

*ОЖ.РЕЗ.:* процедура *А* планирования расчета.

*Тестовый вариант 3 (столбец 3):*

*ИД:* расчет по среднему тарифу; месячное потребление электроэнергии 100 кВт/ч.

*ОЖ.РЕЗ.:* процедура *А* планирования расчета.

*Тестовый вариант 4 (столбец 4):*

*ИД:* расчет по переменному тарифу; месячное потребление электроэнергии 100 кВт/ч.

*ОЖ.РЕЗ.:* процедура *В* планирования расчета.

### **Порядок выполнения работы**

1. Спроектировать тесты по принципу «белого ящика» для программы, разработанной в лабораторной работе № 4. Использовать схемы алгоритмов, разработанные и уточненные в лабораторных работах №№ 2, 3.

2. Выбрать несколько алгоритмов для тестирования и обозначить буквами или цифрами ветви этих алгоритмов.

3. Выписать пути алгоритма, которые должны быть проверены тестами для выбранного метода тестирования.

4. Записать тесты, которые позволят пройти по путям алгоритма.

5. Протестировать разработанную вами программу. Результаты оформить в виде таблиц (см. табл. Л5.1– Л5.4).

6. Проверить все виды тестов и сделать выводы об их эффективности.

7. Оформить отчет по лабораторной работе.

8. Сдать и защитить работу.



## **ЛАБОРАТОРНАЯ РАБОТА № 5. Методология объектно-ориентированного моделирования**

### **Цель работы:**

Ознакомление с основными элементами определения, представления, проектирования и моделирования программных систем с помощью языка UML.

### **Теоретические сведения**

В данной лабораторной работе необходимо ознакомиться с основными элементами определения, представления, проектирования и моделирования программных систем с помощью языка UML, получить навыки применения данных элементов для построения объектно-ориентированных моделей ИС на основании требований.

В ходе выполнения лабораторной работы необходимо сформировать:

- модель системы должна содержать: диаграмму вариантов использования; диаграммы взаимодействия для каждого варианта использования; диаграмму классов, позволяющая реализовать весь описанный функционал ИС; объединенную диаграмму компонентов и размещения
- для классов указать стереотипы;
- в зависимости от варианта задания диаграмма размещения должна показывать расположение компонентов в распределенном приложении или связи между встроенным процессором и устройствами.

Существует множество технологий и инструментальных средств, с помощью которых можно реализовать в некотором смысле оптимальный проект ИС, начиная с этапа анализа и заканчивая созданием программного кода системы. В большинстве случаев эти технологии предъявляют весьма жесткие требования к процессу разработки и используемым ресурсам, а попытки трансформировать их под конкретные проекты оказываются безуспешными. Эти технологии представлены CASE-средствами верхнего

уровня или CASE-средствами полного жизненного цикла (upper CASE tools или full life-cycle CASE tools). Они не позволяют оптимизировать деятельность на уровне отдельных элементов проекта, и, как следствие, многие разработчики перешли на так называемые CASE-средства нижнего уровня (lower CASE tools). Однако они столкнулись с новой проблемой – проблемой организации взаимодействия между различными командами, реализующими проект.

Унифицированный язык объектно-ориентированного моделирования Unified Modeling Language (UML) явился средством достижения компромисса между этими подходами. Существует достаточное количество инструментальных средств, поддерживающих с помощью *UML* жизненный цикл информационных систем, и, одновременно, *UML* является достаточно гибким для настройки и поддержки специфики деятельности различных команд разработчиков.

Создание UML началось в октябре 1994 г., когда Джим Рамбо и Гради Буч из Rational Software Corporation стали работать над объединением своих методов OMT и Booch. В настоящее время консорциум пользователей UML Partners включает в себя представителей таких грандов информационных технологий, как Rational Software, Microsoft, IBM, Hewlett-Packard, Oracle, DEC, Unisys, IntelliCorp, Platinum Technology.

*UML* представляет собой *объектно-ориентированный* язык моделирования, обладающий следующими основными характеристиками:

- является языком визуального моделирования, который обеспечивает разработку репрезентативных моделей для организации взаимодействия заказчика и разработчика ИС, различных групп разработчиков ИС;
- содержит механизмы расширения и специализации базовых концепций языка.

UML – это стандартная нотация визуального моделирования программных систем, принятая консорциумом Object Managing Group (OMG) осенью 1997 г., и на сегодняшний день она поддерживается многими объектно-ориентированными CASE-продуктами.

UML включает внутренний набор средств моделирования, которые сейчас приняты во многих методах и средствах моделирования. Эти концепции необходимы в большинстве прикладных задач, хотя не каждая концепция необходима в каждой части каждого приложения. Пользователям языка предоставлены возможности:

- строить модели на основе средств ядра, без использования механизмов расширения для большинства типовых приложений;
- добавлять при необходимости новые элементы и условные обозначения, если они не входят в ядро, или специализировать компоненты, систему условных обозначений (нотацию) и ограничения для конкретных предметных областей.



Рис. 31. Интегрированная модель сложной системы в нотации языка UML

Стандарт UML предлагает следующие диаграммы для моделирования:

- диаграммы вариантов использования (use case diagrams) – для моделирования бизнес-процессов организации и требований к создаваемой системе);

- диаграммы классов (class diagrams) – для моделирования статической структуры классов системы и связей между ними;
- диаграммы поведения системы (behavior diagrams):
  - диаграммы взаимодействия (interaction diagrams):
  - диаграммы последовательности (sequence diagrams)
  - кооперативные диаграммы (collaboration diagrams) – для моделирования процесса обмена сообщениями между объектами;
  - диаграммы состояний (statechart diagrams) – для моделирования поведения объектов системы при переходе из одного состояния в другое;
- диаграммы деятельности (activity diagrams) – для моделирования поведения системы в рамках различных вариантов использования, или моделирования деятельности;
- диаграммы реализации (implementation diagrams):
  - диаграммы компонентов (component diagrams) – для моделирования иерархии компонентов (подсистем) системы;
  - диаграммы развертывания (deployment diagrams) – для моделирования физической архитектуры системы.

### **Диаграммы вариантов использования**

Понятие варианта использования (use case) впервые ввел Ивар Якобсон и придал ему такую значимость, что в настоящее время вариант использования превратился в основной элемент разработки и планирования проекта.

Вариант использования представляет собой последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Вариант использования описывает типичное взаимодействие между пользователем и системой. В простейшем случае вариант использования

определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать. На языке UML вариант использования изображают следующим образом:

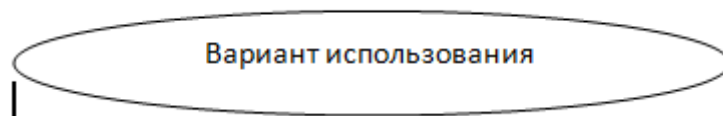


Рис. 32. Вариант использования

Действующее лицо (actor) – это роль, которую пользователь играет по отношению к системе. Действующие лица представляют собой роли, а не конкретных людей или наименования работ. Несмотря на то, что на диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок, действующее лицо может также быть внешней системой, которой необходима некоторая информация от данной системы. Показывать на диаграмме действующих лиц следует только в том случае, когда им действительно необходимы некоторые варианты использования. На языке UML действующие лица представляют в виде фигур:

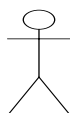


Рис. 33. Действующее лицо (актор)

Действующие лица делятся на три основных типа:

- пользователи;
- системы;
- другие системы, взаимодействующие с данной;
- время.

Время становится действующим лицом, если от него зависит запуск каких-либо событий в системе.

В языке UML на диаграммах вариантов использования поддерживается несколько типов связей между элементами диаграммы.

Это связи коммуникации (communication), включения (include), расширения (extend) и обобщения (generalization).

Связь коммуникации – это связь между вариантом использования и действующим лицом. На языке UML связи коммуникации показывают с помощью однонаправленной ассоциации (сплошной линии).

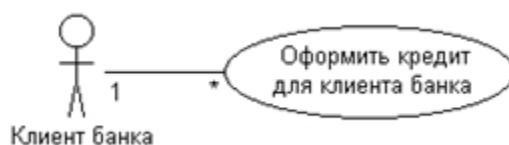


Рис. 34. Пример связи коммуникации

Связь включения применяется в тех ситуациях, когда имеется какой-либо фрагмент поведения системы, который повторяется более чем в одном варианте использования. С помощью таких связей обычно моделируют многократно используемую функциональность.

Связь расширения применяется при описании изменений в нормальном поведении системы. Она позволяет варианту использования только при необходимости использовать функциональные возможности другого.

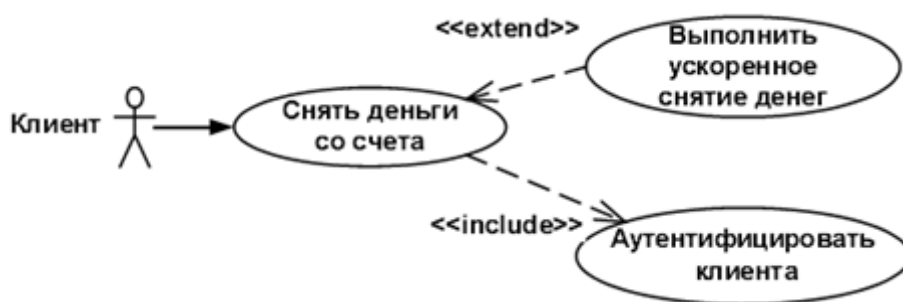


Рис. 35. Пример связи включения и расширения

С помощью связи обобщения показывают, что у нескольких действующих лиц имеются общие черты.

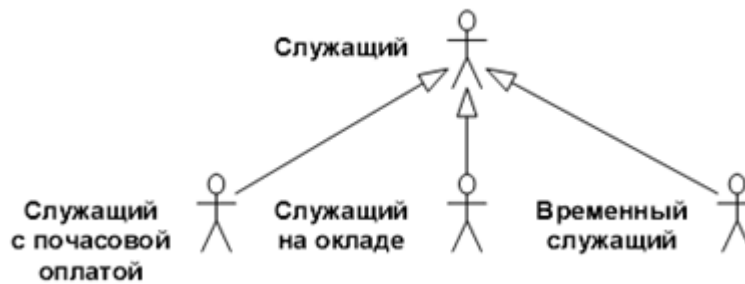


Рис. 36. Пример связи обобщения

### Диаграммы взаимодействия (interaction diagrams)

Диаграммы взаимодействия (interaction diagrams) описывают поведение взаимодействующих групп объектов. Как правило, диаграмма взаимодействия охватывает поведение объектов в рамках только одного варианта использования. На такой диаграмме отображается ряд объектов и те сообщения, которыми они обмениваются между собой.

*Сообщение (message)* – это средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение одной из его операций.

*Информационное (informative) сообщение* – это сообщение, снабжающее объект-получатель некоторой информацией для обновления его состояния.

*Сообщение-запрос (interrogative)* – это сообщение, запрашивающее выдачу некоторой информации об объекте-получателе.

*Императивное (imperative) сообщение* – это сообщение, запрашивающее у объекта-получателя выполнение некоторых действий.

Существует два вида диаграмм взаимодействия: диаграммы последовательности (sequence diagrams) и кооперативные диаграммы (collaboration diagrams).

### **Диаграмма последовательности (sequence diagrams)**

Диаграмма последовательности отражает поток событий, происходящих в рамках варианта использования.

Все действующие лица показаны в верхней части диаграммы. Стрелки соответствуют сообщениям, передаваемым между действующим лицом и объектом или между объектами для выполнения требуемых функций.

На диаграмме последовательности объект изображается в виде прямоугольника, от которого вниз проведена пунктирная вертикальная линия. Эта линия называется линией жизни (lifeline) объекта. Она представляет собой фрагмент жизненного цикла объекта в процессе взаимодействия.

Каждое сообщение представляется в виде стрелки между линиями жизни двух объектов. Сообщения появляются в том порядке, как они показаны на странице сверху вниз. Каждое сообщение помечается как минимум именем сообщения. При желании можно добавить также аргументы и некоторую управляющую информацию. Можно показать самоделегирование (self-delegation) – сообщение, которое объект посылает самому себе, при этом стрелка сообщения указывает на ту же самую линию жизни.

### **Диаграмма кооперации (collaboration diagram)**

Диаграммы кооперации отображают поток событий через конкретный сценарий варианта использования, упорядочены по времени, а кооперативные диаграммы больше внимания заостряют на связях между объектами.

На диаграмме кооперации представлена вся та информация, которая есть и на диаграмме последовательности, но кооперативная диаграмма по-другому описывает поток событий. Из нее легче понять связи между объектами, однако, труднее уяснить последовательность событий.

На кооперативной диаграмме так же, как и на диаграмме последовательности, стрелки обозначают сообщения, обмен которыми



осуществляется в рамках данного варианта использования. Их временная последовательность указывается путем нумерации сообщений.

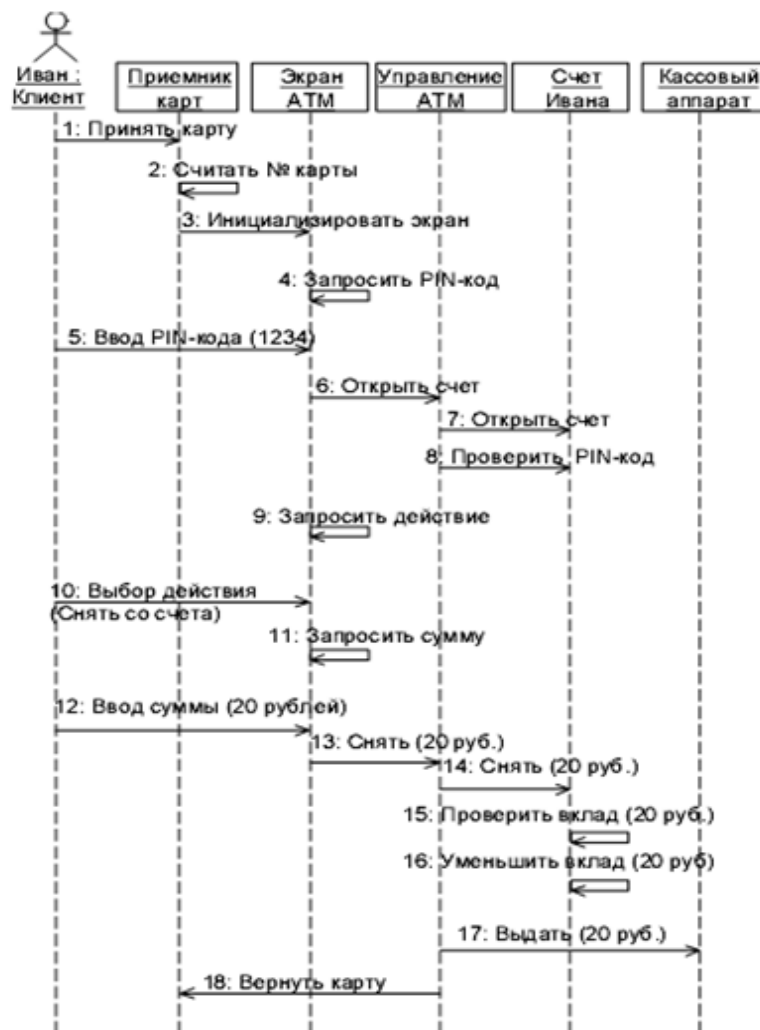


Рис.37. Пример диаграммы последовательности

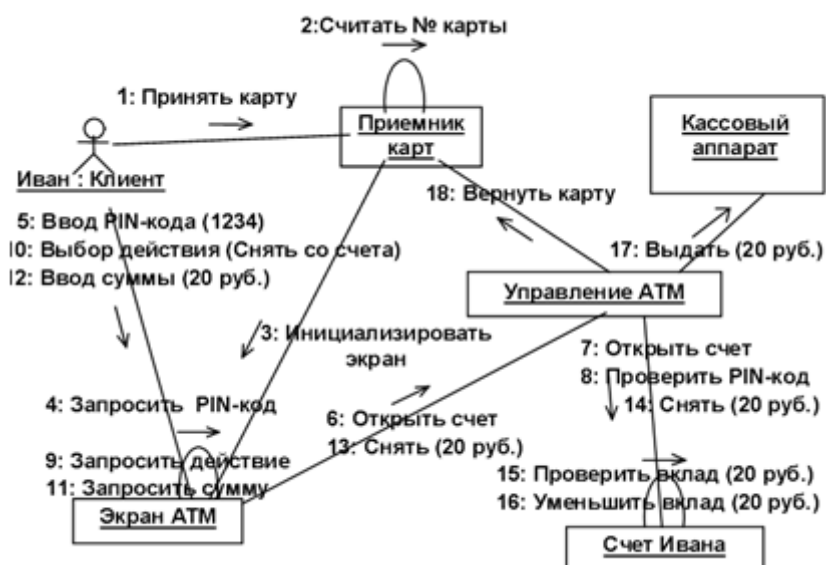


Рис. 38. Пример диаграммы кооперации

## Диаграммы классов

Диаграмма классов определяет типы классов системы и различного рода статические связи, которые существуют между ними. На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами.

Диаграмма классов UML – это граф, узлами которого являются элементы статической структуры проекта (классы, интерфейсы), а дугами – отношения между узлами (ассоциации, наследование, зависимости).

На диаграмме классов изображаются следующие элементы:

- Пакет (package) – набор элементов модели, логически связанных между собой.
- Класс (class) – описание общих свойств группы сходных объектов.
- Интерфейс (interface) – абстрактный класс, задающий набор операций, которые объект произвольного класса, связанного с данным интерфейсом, предоставляет другим объектам.

Класс – это группа сущностей (объектов), обладающих сходными свойствами, а именно, данными и поведением. Отдельный представитель некоторого класса называется объектом класса или просто объектом.

Под поведением объекта в UML понимаются любые правила взаимодействия объекта с внешним миром и с данными самого объекта.

На диаграммах класс изображается в виде прямоугольника со сплошной границей, разделенного горизонтальными линиями на 3 секции:

- Верхняя секция (секция имени) содержит имя класса и другие общие свойства (в частности, стереотип).
- В средней секции содержится список атрибутов
- В нижней – список операций класса, отражающих его поведение (действия, выполняемые классом).

Любая из секций атрибутов и операций может не изображаться (а также обе сразу). Для отсутствующей секции не нужно рисовать разделительную линию и как-либо указывать на наличие или отсутствие элементов в ней.

На усмотрение конкретной реализации могут быть введены дополнительные секции, например, исключения (Exceptions).

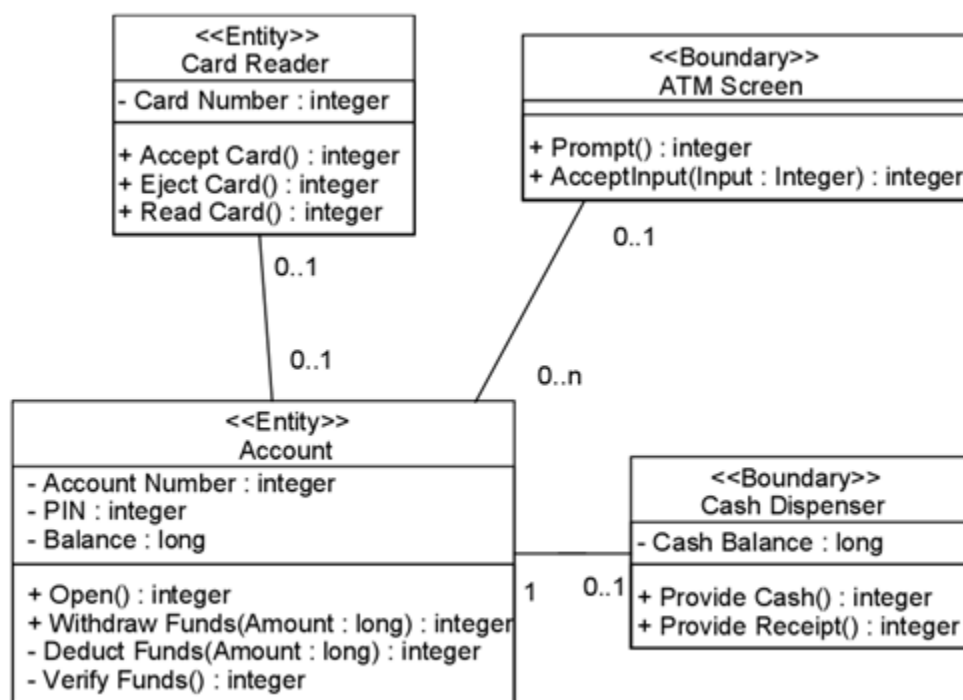


Рис. 39. Пример диаграммы классов

Стереотипы классов – это механизм, позволяющий разделять классы на категории.

В языке UML определены три основных стереотипа классов:

- Boundary (граница);
- Entity (сущность);
- Control (управление).

Граничными классами (boundary classes) называются такие классы, которые расположены на границе системы и всей окружающей среды. Это экранные формы, отчеты, интерфейсы с аппаратурой (такой как принтеры или сканеры) и интерфейсы с другими системами.

Чтобы найти граничные классы, надо исследовать диаграммы вариантов использования. Каждому взаимодействию между действующим лицом и вариантом использования должен соответствовать, по крайней мере, один граничный класс. Именно такой класс позволяет действующему лицу взаимодействовать с системой.

Классы-сущности (entity classes) содержат хранимую информацию. Они имеют наибольшее значение для пользователя, и потому в их названиях часто используют термины из предметной области. Обычно для каждого класса-сущности создают таблицу в базе данных.

Управляющие классы (control classes) отвечают за координацию действий других классов. Обычно у каждого варианта использования имеется один управляющий класс, контролирующий последовательность событий этого варианта использования. Управляющий класс отвечает за координацию, но сам не несет в себе никакой функциональности, так как остальные классы не посылают ему большого количества сообщений. Вместо этого он сам посылает множество сообщений. Управляющий класс просто делегирует ответственность другим классам, по этой причине его часто называют классом-менеджером.

В системе могут быть и другие управляющие классы, общие для нескольких вариантов использования. Например, может быть класс SecurityManager (менеджер безопасности), отвечающий за контроль событий, связанных с безопасностью. Класс TransactionManager (менеджер транзакций) занимается координацией сообщений, относящихся к транзакциям с базой данных. Могут быть и другие менеджеры для работы с другими элементами функционирования системы, такими как разделение ресурсов, распределенная обработка данных или обработка ошибок.

Помимо упомянутых выше стереотипов можно создавать и свои собственные.

Атрибут – это элемент информации, связанный с классом. Атрибуты хранят инкапсулированные данные класса.

Так как атрибуты содержатся внутри класса, они скрыты от других классов. В связи с этим может понадобиться указать, какие классы имеют право читать и изменять атрибуты. Это свойство называется видимостью атрибута (attribute visibility).

У атрибута можно определить четыре возможных значения этого параметра:

- **Public** (общий, открытый). Это значение видимости предполагает, что атрибут будет виден всеми остальными классами. Любой класс может просмотреть или изменить значение атрибута. В соответствии с нотацией UML общему атрибуту предшествует знак «+».

- **Private** (закрытый, секретный). Соответствующий атрибут не виден никаким другим классом. Закрытый атрибут обозначается знаком «-» в соответствии с нотацией UML.

- **Protected** (защищенный). Такой атрибут доступен только самому классу и его потомкам. Нотация UML для защищенного атрибута – это знак «#».

- **Package or Implementation** (пакетный). Предполагает, что данный атрибут является общим, но только в пределах его пакета. Этот тип видимости не обозначается никаким специальным значком.

В общем случае, атрибуты рекомендуется делать закрытыми или защищенными. Это позволяет лучше контролировать сам атрибут и код.

С помощью закрытости или защищенности удастся избежать ситуации, когда значение атрибута изменяется всеми классами системы. Вместо этого логика изменения атрибута будет заключена в том же классе, что и сам этот атрибут. Задаваемые параметры видимости повлияют на генерируемый код.

Операции реализуют связанное с классом поведение. Операция включает три части – имя, параметры и тип возвращаемого значения.

Параметры – это аргументы, получаемые операцией «на входе». Тип возвращаемого значения относится к результату действия операции.

На диаграмме классов можно показывать как имена операций, так и имена операций вместе с их параметрами и типом возвращаемого значения. Чтобы уменьшить загруженность диаграммы, полезно бывает на некоторых из них показывать только имена операций, а на других их полную сигнатуру.

В языке UML операции имеют следующую нотацию:

*Имя Операции (аргумент: тип данных аргумента, аргумент2:тип данных аргумента2,...): тип возвращаемого значения*

Следует рассмотреть четыре различных типа операций:

- Операции реализации.
- Операции управления.
- Операции доступа.
- Вспомогательные операции.

Операции реализации (implementor operations) реализуют некоторые бизнес-функции. Такие операции можно найти, исследуя диаграммы взаимодействия. Диаграммы этого типа фокусируются на бизнес-функциях, и каждое сообщение диаграммы, скорее всего, можно соотнести с операцией реализации.

Каждая операция реализации должна быть легко прослеживаема до соответствующего требования. Это достигается на различных этапах моделирования. Операция выводится из сообщения на диаграмме взаимодействия, сообщения исходят из подробного описания потока событий, который создается на основе варианта использования, а последний – на основе требований. Возможность проследить всю эту цепочку позволяет гарантировать, что каждое требование будет реализовано в коде, а каждый фрагмент кода реализует какое-то требование.

Операции управления (manager operations) управляют созданием и уничтожением объектов. В эту категорию попадают конструкторы и деструкторы классов.

Атрибуты обычно бывают закрытыми или защищенными. Тем не менее, другие классы иногда должны просматривать или изменять их значения. Для этого существуют операции доступа (access operations). Такой подход дает возможность безопасно инкапсулировать атрибуты внутри класса, защитив их от других классов, но позволяет осуществить к ним контролируемый доступ. Создание операций Get и Set (получения и изменения значения) для каждого атрибута класса является стандартом.

Вспомогательными (helper operations) называются такие операции класса, которые необходимы ему для выполнения его ответственностей, но о которых другие классы не должны ничего знать. Это закрытые и защищенные операции класса.

Чтобы идентифицировать операции, выполните следующие действия:

1. Изучите диаграммы последовательности и кооперативные диаграммы. Большая часть сообщений на этих диаграммах является операциями реализации. Рефлексивные сообщения будут вспомогательными операциями.

2. Рассмотрите управляющие операции. Может потребоваться добавить конструкторы и деструкторы.

3. Рассмотрите операции доступа. Для каждого атрибута класса, с которым должны будут работать другие классы, надо создать операции Get и Set.

Связь представляет собой семантическую взаимосвязь между классами. Она дает классу возможность узнавать об атрибутах, операциях и связях другого класса. Иными словами, чтобы один класс мог послать сообщение другому на диаграмме последовательности или кооперативной диаграмме, между ними должна существовать связь.

Существуют четыре типа связей, которые могут быть установлены между классами: ассоциации, зависимости, агрегации и обобщения.

Ассоциация (association) – это семантическая связь между классами. Их рисуют на диаграмме классов в виде обыкновенной линии.



Рис. 40. Связь ассоциация

Ассоциации могут быть двунаправленными, как в примере, или однонаправленными. На языке UML двунаправленные ассоциации рисуют в виде простой линии без стрелок или со стрелками с обеих ее сторон. На однонаправленной ассоциации изображают только одну стрелку, показывающую ее направление.

Направление ассоциации можно определить, изучая диаграммы последовательности и кооперативные диаграммы. Если все сообщения на них отправляются только одним классом и принимаются только другим классом, но не наоборот, между этими классами имеет место однонаправленная связь. Если хотя бы одно сообщение отправляется в обратную сторону, ассоциация должна быть двунаправленной.

Ассоциации могут быть рефлексивными. Рефлексивная ассоциация предполагает, что один экземпляр класса взаимодействует с другими экземплярами этого же класса.

Связи зависимости (dependency) также отражают связь между классами, но они всегда однонаправлены и показывают, что один класс зависит от определений, сделанных в другом. Например, класс А использует методы класса В. Тогда при изменении класса В необходимо произвести соответствующие изменения в классе А.

Зависимость изображается пунктирной линией, проведенной между двумя элементами диаграммы, и считается, что элемент, привязанный к концу стрелки, зависит от элемента, привязанного к началу этой стрелки.



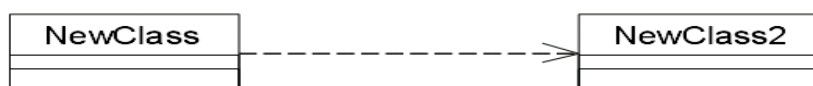


Рис. 41. Связь зависимость

При генерации кода для этих классов к ним не будут добавляться новые атрибуты. Однако, будут созданы специфические для языка операторы, необходимые для поддержки связи.

Агрегации (aggregations) представляют собой более тесную форму ассоциации. Агрегация – это связь между целым и его частью. Например, у вас может быть класс Автомобиль, а также классы Двигатель, Покрышки и классы для других частей автомобиля. В результате объект класса Автомобиль будет состоять из объекта класса Двигатель, четырех объектов Покрышек и т. д. Агрегации визуализируют в виде линии с ромбиком у класса, являющегося целым:



Рис. 42. Связь агрегация

В дополнение к простой агрегации UML вводит более сильную разновидность агрегации, называемую композицией. Согласно композиции, объект-часть может принадлежать только единственному целому, и, кроме того, как правило, жизненный цикл частей совпадает с циклом целого: они живут и умирают вместе с ним. Любое удаление целого распространяется на его части.

Такое каскадное удаление нередко рассматривается как часть определения агрегации, однако оно всегда подразумевается в том случае, когда множественность роли составляет 1..1; например, если необходимо удалить Клиента, то это удаление должно распространиться и на Заказы (и, в свою очередь, на Строки заказа).

Обобщение (наследование) – это отношение типа общее–частное между элементами модели. С помощью обобщений (generalization)

показывают связи наследования между двумя классами. Большинство объектно-ориентированных языков непосредственно поддерживают концепцию наследования. Она позволяет одному классу наследовать все атрибуты, операции и связи другого. Наследование пакетов означает, что в пакете-наследнике все сущности пакета-предка будут видны под своими собственными именами (т. е. пространства имен объединяются). Наследование показывается сплошной линией, идущей от класса-потомка к классу-предку (в терминологии ООП – от потомка к предку, от сына к отцу, или от подкласса к суперклассу). Со стороны более общего элемента рисуется большой полый треугольник.

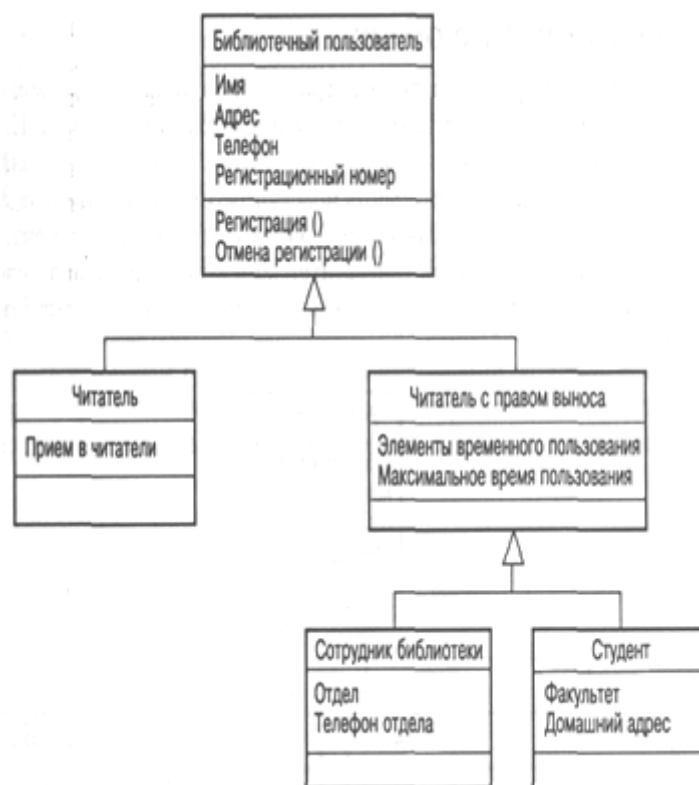


Рис. 43. Пример связи наследование

Помимо наследуемых, каждый подкласс имеет свои собственные уникальные атрибуты, операции и связи.

Множественность (multiplicity) показывает, сколько экземпляров одного класса взаимодействуют с помощью этой связи с одним экземпляром другого класса в данный момент времени.

Например, при разработке системы регистрации курсов в университете можно определить классы Course (курс) и Student (студент). Между ними установлена связь: у курсов могут быть студенты, а у студентов – курсы. Вопросы, на который должен ответить параметр множественности: «Сколько курсов студент может посещать в данный момент? Сколько студентов может за раз посещать один курс?»

Так как множественность дает ответ на оба эти вопроса, её индикаторы устанавливаются на обоих концах линии связи. В примере регистрации курсов мы решили, что один студент может посещать от нуля до четырех курсов, а один курс могут слушать от 0 до 20 студентов.

В языке UML приняты определенные нотации для обозначения множественности.

Таблица 10

Обозначения множественности связей в UML

Множественность	Значение
0..*	Ноль или больше
1..*	Один или больше
0..1	Ноль или один
1..1 (сокращенная запись :1)	Ровно один

Связи можно уточнить с помощью имен связей или ролевых имен. Имя связи – это обычно глагол или глагольная фраза, описывающая, зачем она нужна. Например, между классом Person (человек) и классом Company (компания) может существовать ассоциация. Можно задать в связи с этим вопрос, является ли объект класса Person клиентом компании, её сотрудником или владельцем? Чтобы определить это, ассоциацию можно назвать «employs» (нанимает):

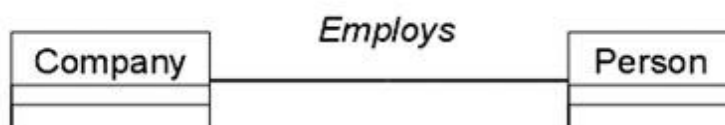


Рис. 44. Пример имен связей

Ролевые имена применяют в связях ассоциации или агрегации вместо имен для описания того, зачем эти связи нужны. Возвращаясь к примеру с классами *Person* и *Company*, можно сказать, что класс *Person* играет роль сотрудника класса *Company*. Ролевые имена – это обычно имена существительные или основанные на них фразы, их показывают на диаграмме рядом с классом, играющим соответствующую роль. Как правило, пользуются или ролевым именем, или именем связи, но не обоими сразу. Как и имена связей, ролевые имена не обязательны, их дают, только если цель связи не очевидна. Пример ролей приводится ниже:

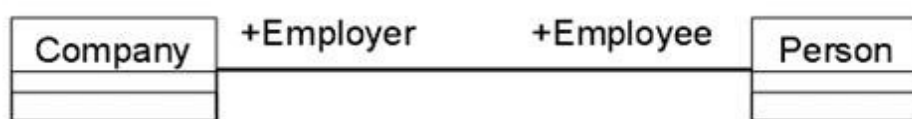


Рис. 45. Пример ролей связей

В контексте диаграмм классов, пакет – этоместилище для некоторого набора классов и других пакетов. Пакет является самостоятельным пространством имен.

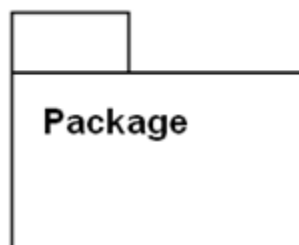


Рис. 46. Обозначение пакета в UML

В UML нет каких-либо ограничений на правила, по которым разработчики могут или должны группировать классы в пакеты. Но есть некоторые стандартные случаи, когда такая группировка уместна, например, тесно взаимодействующие классы, или более общий случай – разбиение системы на подсистемы.

Пакет физически содержит сущности, определенные в нем (говорят, что «сущности принадлежат пакету»). Это означает, что если будет уничтожен пакет, то будут уничтожены и все его содержимое.

Существует несколько наиболее распространенных подходов к группировке.

Группировать их по стереотипу. В таком случае получается один пакет с классами-сущностями, один с граничными классами, один с управляющими классами и т. д. Этот подход может быть полезен с точки зрения размещения готовой системы, поскольку все находящиеся на клиентских машинах пограничные классы уже оказываются в одном пакете.

Объединение классов по их функциональности. Например, в пакете Security (безопасность) содержатся все классы, отвечающие за безопасность приложения. В таком случае другие пакеты могут называться Employee Maintenance (Работа с сотрудниками), Reporting (Подготовка отчетов) и Error Handling (Обработка ошибок). Преимущество этого подхода заключается в возможности повторного использования.

Механизм пакетов применим к любым элементам модели, а не только к классам. Если для группировки классов не использовать некоторые эвристики, то она становится произвольной. Одна из них, которая в основном используется в UML, – это зависимость. Зависимость между двумя пакетами существует в том случае, если между любыми двумя классами в пакетах существует любая зависимость.

Таким образом, диаграмма пакетов представляет собой диаграмму, содержащую пакеты классов и зависимости между ними. Строго говоря, пакеты и зависимости являются элементами диаграммы классов, то есть диаграмма пакетов – это форма диаграммы классов.

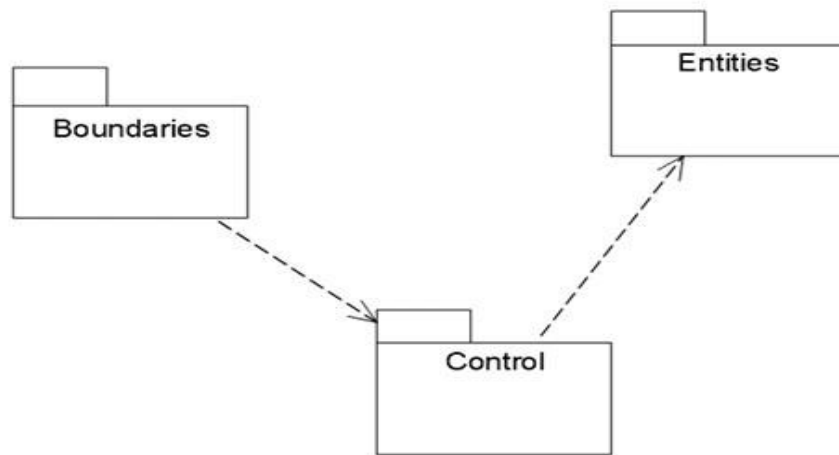


Рис. 47. Пример диаграммы пакетов

Зависимость между двумя элементами имеет место в том случае, если изменения в определении одного элемента могут повлечь за собой изменения в другом. Что касается классов, то причины для зависимостей могут быть самыми разными:

- один класс посылает сообщение другому;
- один класс включает часть данных другого класса; один класс использует другой в качестве параметра операции.

Если класс меняет свой интерфейс, то любое сообщение, которое он посылает, может утратить свою силу.

Пакеты не дают ответа на вопрос, каким образом можно уменьшить количество зависимостей в вашей системе, однако они помогают выделить эти зависимости, а после того, как они все окажутся на виду, остается только поработать над снижением их количества. Диаграммы пакетов можно считать основным средством управления общей структурой системы.

Пакеты являются жизненно необходимым средством для больших проектов. Их следует использовать в тех случаях, когда диаграмма классов, охватывающая всю систему в целом и , становится нечитаемой.

## Диаграммы состояний

Диаграммы состояний определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий.

Существует много форм диаграмм состояний, незначительно отличающихся друг от друга семантикой.

На диаграмме имеются два специальных состояния – начальное (start) и конечное (stop). Начальное состояние выделено черной точкой, оно соответствует состоянию объекта, когда он только что был создан. Конечное состояние обозначается черной точкой в белом кружке, оно соответствует состоянию объекта непосредственно перед его уничтожением. На диаграмме состояний может быть одно и только одно начальное состояние. В то же время, может быть столько конечных состояний, сколько вам нужно, или их может не быть вообще. Когда объект находится в каком-то конкретном состоянии, могут выполняться различные процессы. Процессы, происходящие, когда объект находится в определенном состоянии, называются действиями (actions).

С состоянием можно связывать данные пяти типов: деятельность, входное действие, выходное действие, событие и история состояния.

Деятельностью (activity) называется поведение, реализуемое объектом, пока он находится в данном состоянии. Деятельность – это прерываемое поведение. Оно может выполняться до своего завершения, пока объект находится в данном состоянии, или может быть прервано переходом объекта в другое состояние. Деятельность изображают внутри самого состояния, ей должно предшествовать слово do (делать) и двоеточие.

Входным действием (entry action) называется поведение, которое выполняется, когда объект переходит в данное состояние. Данное действие осуществляется не после того, как объект перешел в это состояние, а, скорее, как часть этого перехода. В отличие от деятельности, входное

действие рассматривается как непрерываемое. Входное действие также показывают внутри состояния, ему предшествует слово entry (вход) и двоеточие.

Выходное действие (exit action) подобно входному. Однако, оно осуществляется как составная часть процесса выхода из данного состояния. Оно является частью процесса такого перехода. Как и входное, выходное действие является непрерываемым.

Выходное действие изображают внутри состояния, ему предшествует слово exit (выход) и двоеточие.

Поведение объекта во время деятельности, при входных и выходных действиях может включать отправку события другому объекту. В этом случае описанию деятельности, входного действия или выходного действия предшествует знак « ^ ».

Соответствующая строка на диаграмме выглядит как

*Do: Цель.Событие (Аргументы)*

Цель – это объект, получающий событие, Событие – это посылаемое сообщение, а Аргументы являются параметрами посылаемого сообщения.

Деятельность может также выполняться в результате получения объектом некоторого события. При получении некоторого события выполняется определенная деятельность.

Переходом (Transition) называется перемещение из одного состояния в другое. Совокупность переходов диаграммы показывает, как объект может перемещаться между своими состояниями. На диаграмме все переходы изображают в виде стрелки, начинающейся на первоначальном состоянии и заканчивающейся последующим.

Переходы могут быть рефлексивными. Объект может перейти в то же состояние, в котором он в настоящий момент находится. Рефлексивные переходы изображают в виде стрелки, начинающейся и завершающейся на одном и том же состоянии.



У перехода существует несколько спецификаций. Они включают события, аргументы, ограждающие условия, действия и посылаемые события.

Событие (event) – это то, что вызывает переход из одного состояния в другое. Событие размещают на диаграмме вдоль линии перехода.

На диаграмме для отображения события можно использовать как имя операции, так и обычную фразу.

Большинство переходов должны иметь события, так как именно они, прежде всего, заставляют переход осуществиться. Тем не менее, бывают и автоматические переходы, не имеющие событий. При этом объект сам перемещается из одного состояния в другое со скоростью, позволяющей осуществиться входным действиям, деятельности и выходным действиям.

Ограждающие условия (guard conditions) определяют, когда переход может, а когда не может осуществиться. В противном случае переход не осуществится.

Ограждающие условия изображают на диаграмме вдоль линии перехода после имени события, заключая их в квадратные скобки.

Ограждающие условия задавать необязательно. Однако если существует несколько автоматических переходов из состояния, необходимо определить для них взаимно исключающие ограждающие условия. Это поможет читателю диаграммы понять, какой путь перехода будет автоматически выбран.

Действием (action), как уже говорилось, является непрерываемое поведение, осуществляющееся как часть перехода. Входные и выходные действия показывают внутри состояний, поскольку они определяют, что происходит, когда объект входит или выходит из него. Большую часть действий, однако, изображают вдоль линии перехода, так как они не должны осуществляться при входе или выходе из состояния.

Действие рисуют вдоль линии перехода после имени события, ему предшествует косая черта.

Событие или действие могут быть поведением внутри объекта, а могут представлять собой сообщение, посылаемое другому объекту. Если событие или действие посылается другому объекту, перед ним на диаграмме помещают знак «^».



Рис. 48. Пример диаграммы состояний

Диаграммы состояний не надо создавать для каждого класса, они применяются только в сложных случаях. Если объект класса может существовать в нескольких состояниях и в каждом из них ведет себя по-разному, для него может потребоваться такая диаграмма.

### Диаграммы размещения

Диаграмма размещения (deployment diagram) отражает физические взаимосвязи между программными и аппаратными компонентами системы. Она является хорошим средством для того, чтобы показать маршруты перемещения объектов и компонентов в распределенной системе.

Каждый узел на диаграмме размещения представляет собой некоторый тип вычислительного устройства – в большинстве случаев, часть аппаратуры. Эта аппаратура может быть простым устройством или датчиком, а может быть и мейнфреймом.

Диаграмма размещения показывает физическое расположение сети и местонахождение в ней различных компонентов.

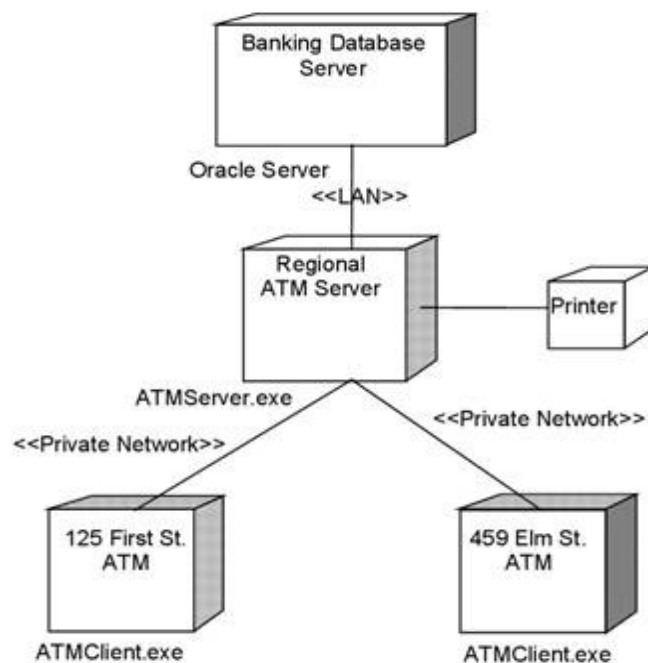


Рис. 49. Пример диаграммы размещения

Диаграмма размещения используется менеджером проекта, пользователями, архитектором системы и эксплуатационным персоналом, чтобы понять физическое размещение системы и расположение её отдельных подсистем.

### Диаграммы компонентов

Диаграммы компонентов показывают, как выглядит модель на физическом уровне. На них изображены компоненты программного обеспечения и связи между ними. При этом на такой диаграмме выделяют два типа компонентов: исполняемые компоненты и библиотеки кода.

Каждый класс модели (или подсистема) преобразуется в компонент исходного кода. После создания они сразу добавляются к диаграмме компонентов. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы.

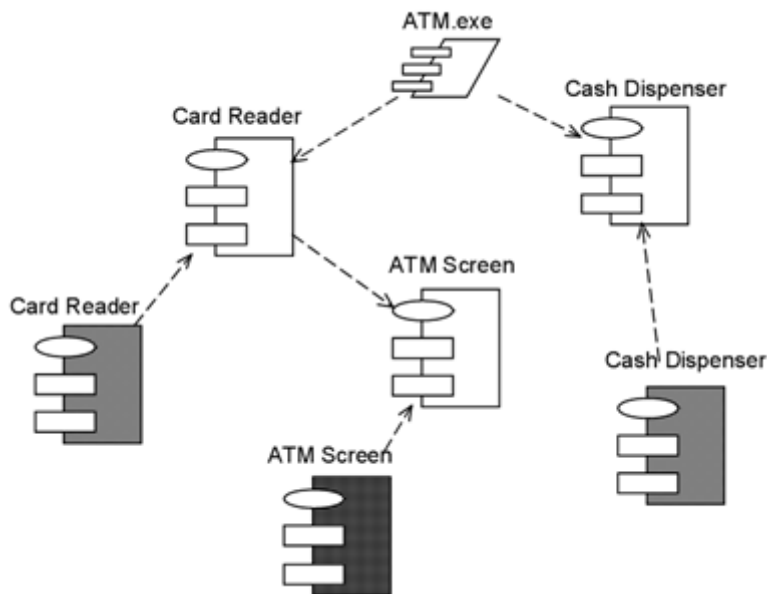


Рис. 50. Пример диаграммы компонентов

Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию системы. Из нее видно, в каком порядке надо компилировать компоненты, а также какие исполняемые компоненты будут созданы системой. На такой диаграмме показано соответствие классов реализованным компонентам. Она нужна там, где начинается генерация кода.

В некоторых случаях допускается размещать диаграмму компонентов на диаграмме развертывания. Это позволяет показать какие компоненты выполняются и на каких узлах.

### Порядок выполнения работы

1. Изучить предлагаемый теоретический материал.
2. Построить диаграмму вариантов использования для выбранной информационной системы.
3. Выполните реализацию вариантов использования в терминах взаимодействующих объектов и представляющую собой набор диаграмм:
  - диаграмм классов, реализующих вариант использования;

– диаграмм взаимодействия (диаграмм последовательности и кооперативных диаграмм), отражающих взаимодействие объектов в процессе реализации варианта использования.

4. Разделить классы по пакетам используя один из механизмов разбиения.

5. Построить диаграмму состояний для конкретных объектов информационной системы.

6. Оформить отчет, включающий все полученные уровни модели, описание функциональных блоков, потоков данных, хранилищ и внешних объектов.

## **ЛАБОРАТОРНАЯ РАБОТА № 6 Методология управление проектами**

### **Цель работы:**

Изучение методологии управления проектами. Получение навыков по применению данных методологий для планирования проекта.

### **Теоретический материал**

Процесс разработки ПО существенно отличается от процессов реализации технических проектов, что порождает определенные сложности в управлении программными проектами. .

1. *Программный продукт нематериален.* Менеджер технического проекта видит результаты выполнения своего проекта. Если реализация проекта отстает от графика, это также видно воочию. В противоположность этому программное обеспечение нематериально. Его нельзя увидеть или потрогать. Менеджер программного проекта не видит процесс «роста» разрабатываемого ПО. Он может полагаться только на документацию, которая фиксирует процесс разработки программного продукта.

2. *Не существует стандартных процессов разработки ПО.* На сегодняшний день не существует четкой зависимости между процессом

создания ПО и типом создаваемого программного продукта. Другие технические дисциплины имеют длительную историю, процессы разработки технических изделий многократно опробованы и проверены. Процессы создания большинства технических систем хорошо изучены. Изучением же процессов создания ПО специалисты занимаются только несколько последних лет. Поэтому пока нельзя точно предсказать, на каком этапе процесса разработки ПО могут возникнуть проблемы, угрожающие всему программному проекту.

3. *Большие программные проекты – это часто «одноразовые» проекты.* Большие программные проекты, как правило, значительно отличаются от проектов, реализованных ранее. Поэтому, чтобы уменьшить неопределенность в планировании проекта, руководители проектов должны обладать очень большим практическим опытом. Но постоянные технологические изменения в компьютерной технике и коммуникационном оборудовании обесценивают предыдущий опыт. Знания и навыки, накопленные опытом, могут не востребоваться в новом проекте.

Перечисленное выше может привести к тому, что реализация проекта выйдет из временного графика или превысит бюджетные ассигнования. Программные системы зачастую оказываются новинками как в «идеологическом», так и в техническом плане. Технические проекты, которые являются инновационными (например, новая транспортная система), также часто нарушают временные графики работ. Поэтому, предвидя возможные проблемы в реализации программного проекта, следует всегда помнить, что многим из них свойственно выходить за рамки временных и бюджетных ограничений.

Эффективное управление программным проектом напрямую зависит от правильного планирования работ, необходимых для его выполнения. План помогает менеджеру предвидеть проблемы, которые могут возникнуть на каких-либо этапах создания ПО, и разработать

превентивные меры для их предупреждения или решения. План, разработанный на начальном этапе проекта, рассматривается всеми его участниками как руководящий документ, выполнение которого должно привести к успешному завершению проекта. Этот первоначальный план должен максимально подробно описывать все этапы реализации проекта.

Кроме разработки плана проекта, на менеджера ложится обязанность разработки других видов планов. Эти виды планов кратко описаны в табл. 11.

Таблица 11

#### Виды планов

План	Описание
План качества	Описывает стандарты и мероприятия по поддержке качества разрабатываемого ПО
План аттестации	Описывает способы, ресурсы и перечень работ, необходимых для аттестации программной системы
План управления конфигурацией	Описывает структуру и процессы управления конфигурацией
План сопровождения ПО	Предлагает план мероприятий, требующихся для сопровождения ПО в процессе его эксплуатации, а также расчет стоимости сопровождения и необходимые для этого ресурсы
План по управлению персоналом	Описывает мероприятия, направленные на повышение квалификации членов команды разработчиков

В листинге 1 показан процесс планирования создания ПО в виде псевдокода. Здесь сделан акцент на том, что планирование – это итерационный процесс. Поскольку в процессе выполнения проекта постоянно поступает новая информация, план должен регулярно пересматриваться. Важными факторами, которые должны учитываться при разработке плана, являются финансовые и деловые обязательства организации. Если они изменяются, эти изменения также должны найти отражение в плане работ.

## Листинг 1. Процесс планирования проекта

Определение проектных ограничений

Первоначальная оценка параметров проекта

Определение этапов выполнения проекта и контрольных отметок

**while** пока проект не завершится или не будет остановлен **loop**

Составление графика работ

Начало выполнения работ

Ожидание окончания очередного этапа работ

Отслеживание хода выполнения работ

Пересмотр оценок параметров проекта

Изменение графика работ

Пересмотр проектных ограничений

**if** (возникла проблема) **then**

Пересмотр технических или организационных параметров проекта

**end if**

**end loop**

Процесс планирования начинается с определения проектных ограничений (временные ограничения, возможности наличного персонала, бюджетные ограничения и т. д.). Эти ограничения должны определяться параллельно с оцениванием проектных параметров, таких как структура и размер проекта, а также распределением функций среди исполнителей. Затем определяются этапы разработки и то, какие результаты документация, прототипы, подсистемы или версии программного продукта) должны быть получены по окончании этих этапов. Далее начинается циклическая часть планирования. Сначала разрабатывается график работ по выполнению проекта или дается разрешение на продолжение использования ранее созданного графика. После этого (обычно через 2–3 недели) проводится контроль выполнения работ и отмечаются расхождения между реальным и плановым ходом работ.



Далее, по мере поступления новой информации о ходе выполнения проекта, возможен пересмотр первоначальных оценок параметров проекта. Это, в свою очередь, может привести к изменению графика работ. Если в результате этих изменений нарушаются сроки завершения проекта, должны быть пересмотрены (и согласованы с заказчиком ПО) проектные ограничения.

Конечно, большинство менеджеров проектов не думают, что реализация их проектов пройдет гладко, без всяких проблем. Желательно описать возможные проблемы еще до того, как они проявят себя в ходе выполнения проекта. Поэтому лучше составлять «пессимистические» графики работ, чем «оптимистические». Но, конечно, невозможно построить план, учитывающий все, в том числе случайные, проблемы и задержки выполнения проекта, поэтому и возникает необходимость периодического пересмотра проектных ограничений и этапов создания программного продукта.

План проекта должен четко показать ресурсы, необходимые для реализации проекта, разделение работ на этапы и временной график выполнения этих этапов. В некоторых организациях план проекта составляется как единый документ, содержащий все виды планов, описанных выше. В других случаях план проекта описывает только технологический процесс создания ПО. В таком плане обязательно присутствуют ссылки на планы других видов, но они разрабатываются отдельно от плана проекта.

План, представленный ниже, принадлежит именно к последнему типу планов. Детализация планов проектов очень разнится в зависимости от типа разрабатываемого программного продукта и организации-разработчика. Но в любом случае большинство планов содержат следующие разделы.

1. *Введение.* Краткое описание целей проекта и проектных ограничений (бюджетных, временных и т. д.), которые важны для управления проектом.

2. *Организация выполнения проекта.* Описание способа подбора команды разработчиков и распределение обязанностей между членами команды.

3. *Анализ рисков.* Описание возможных проектных рисков, вероятности их проявления и стратегий, направленных на их уменьшение. Тема управления рисками рассмотрена ниже.

4. *Аппаратные и программные ресурсы, необходимые для реализации проекта.* Перечень аппаратных средств и программного обеспечения, необходимого для разработки программного продукта. Если аппаратные средства требуется закупать, приводится их стоимость совместно с графиком закупки и поставки.

5. *Разбиение работ на этапы.* Процесс реализации проекта разбивается на отдельные процессы, определяются этапы выполнения проекта, приводится описание результатов («выходов») каждого этапа и контрольные отметки. Эта тема представлена ниже.

6. *График работ.* В этом графике отображаются зависимости между отдельными процессами (этапами) разработки ПО, оценки времени их выполнения и распределение членов команды разработчиков по отдельным этапам.

7. *Механизмы мониторинга и контроля за ходом выполнения проекта.* Описываются предоставляемые менеджером отчеты о ходе выполнения работ, сроки их предоставления, а также механизмы мониторинга всего проекта.

План должен регулярно пересматриваться в процессе реализации проекта. Одни части плана, например график работ, изменяются часто, другие более стабильны. Для внесения изменений в план требуется

специальная организация документопотока, позволяющая отслеживать эти изменения.

Менеджеру для организации процесса создания ПО и управления им необходима информация. Поскольку само программное обеспечение неосвязаемо, эта управленческая информация может быть получена только в виде документов, отображающих выполнение очередного этапа разработки программного продукта. Без этой информации нельзя судить о степени готовности создаваемого продукта, невозможно оценить произведенные затраты или изменить график работ.

При планировании процесса определяются контрольные отметки–вехи, отмечающие окончание определенного этапа работ. Для каждой контрольной отметки создается отчет, который предоставляется руководству проекта. Эти отчеты не должны быть большими объемными документами; они должны подводить краткие итоги окончания отдельного логически завершенного этапа проекта. Этапом не может быть, например, «Написание 80% кода программ», поскольку невозможно проверить завершение такого «этапа»; кроме того, подобная информация практически бесполезна для управления, поскольку здесь не отображается связь этого «этапа» с другими этапами создания ПО.

Обычно по завершении основных больших этапов, таких как разработка спецификации, проектирование и т. п., заказчику ПО предоставляются результаты их выполнения, так называемые контрольные проектные элементы. Это может быть документация, прототип программного продукта, законченные подсистемы ПО и т. д. Контрольные проектные элементы, предоставляемые заказчику ПО, могут совпадать с контрольными отметками (точнее, с результатами выполнения какого-либо этапа). Но обратное утверждение неверно. Контрольные отметки – это внутренние проектные результаты, которые используются для контроля за ходом выполнения проекта, и они, как правило, не предоставляются заказчику ПО.

Для определения контрольных отметок весь процесс создания ПО должен быть разбит на отдельные этапы с указанным «выходом» (результатом) каждого этапа. Например, на рисунке 51 показаны этапы разработки спецификации требований в случае, когда для ее проверки используется прототип системы, а также представлены выходные результаты (контрольные отметки) каждого этапа. Здесь контрольными проектными элементами являются требования и спецификация требований.

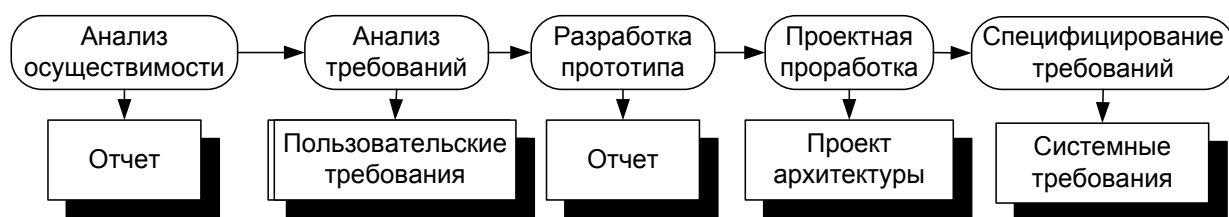


Рис. 51. Этапы процесса разработки спецификации

Составление графика – одна из самых ответственных работ, выполняемых менеджером проекта. Здесь менеджер оценивает длительность проекта, определяет ресурсы, необходимые для реализации отдельных этапов работ, и представляет их (этапы) в виде согласованной последовательности. Если данный проект подобен ранее реализованному, то график работ последнего проекта можно взять за основу для данного проекта. Но затем следует учесть, что на отдельных этапах нового проекта могут использоваться методы и подходы, отличные от использованных ранее.

Если проект является инновационным, первоначальные оценки длительности и требуемых ресурсов наверняка будут слишком оптимистичными, даже если менеджер попытается предусмотреть все возможные неожиданности. С этой точки зрения проекты создания ПО не отличаются от больших инновационных технических проектов. Новые аэропорты, мосты и даже новые автомобили, как правило, появляются позже первоначально объявленных сроков их сдачи или поступления на рынок, чему причиной являются неожиданно возникшие проблемы и

трудности. Именно поэтому графики работ необходимо постоянно обновлять по мере поступления новой информации о ходе выполнения проекта.

В процессе составления графика (рис.52) весь массив работ, необходимых для реализации проекта, разбивается на отдельные этапы и оценивается время, требующееся для выполнения каждого этапа. Обычно многие этапы выполняются параллельно. График работ должен предусматривать это и распределять производственные ресурсы между ними наиболее оптимальным образом. Нехватка ресурсов для выполнения какого-либо критического этапа – частая причина задержки выполнения всего проекта.

Длительность этапов обычно должна быть не меньше недели. Если она будет меньше, то окажется ниже точности временных оценок этапов, что может привести к частому пересмотру графика работ. Также целесообразно (в аспекте управления проектом) установить максимальную длительность этапов, не превышающую 8 или 10 недель. Если есть этапы, имеющие большую длительность, их следует разбить на этапы меньшей длительности.

При расчете длительности этапов менеджер должен учитывать, что выполнение любого этапа не обойдется без больших или маленьких проблем и задержек. Разработчики могут допускать ошибки или задерживать свою работу, техника может выйти из строя либо аппаратные или программные средства поддержки процесса разработки могут поступить с опозданием. Если проект инновационный и технически сложный, это становится дополнительным фактором появления непредвиденных проблем и увеличения длительности реализации проекта по сравнению с первоначальными оценками.

Кроме временных затрат, менеджер должен рассчитать другие ресурсы, необходимые для успешного выполнения каждого этапа. Особый вид ресурсов – это команда разработчиков, привлеченная к выполнению проекта. Другими видами ресурсов могут быть необходимое свободное

дисковое пространство на сервере, время использования какого-либо специального оборудования и бюджетные средства на командировочные расходы персонала, работающего над проектом.

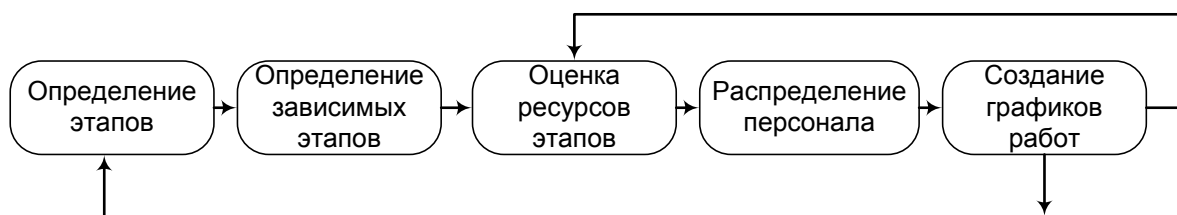


Рис. 52. Процесс составления графика работ

Существует хорошее эмпирическое правило: оценивать временные затраты так, как будто ничего непредвиденного и «плохого» не может случиться, затем увеличить эти оценки для учета возможных проблем. Возможные, но трудно прогнозируемые проблемы существенно зависят от типа и параметров проекта, а также от квалификации и опыта членов команды разработчиков. К исходным расчетным оценкам рекомендуется добавлять 30% на возможные проблемы и затем еще 20 %, чтобы быть готовым к тому, что невозможно предвидеть.

График работ по проекту обычно представляется в виде набора диаграмм и графиков, показывающих разбиение проектных работ на этапы, зависимости между этапами и распределение разработчиков по этапам.

Временные и сетевые диаграммы полезны для представления графика работ. Временная диаграмма показывает время начала и окончания каждого этапа и его длительность. Сетевая диаграмма отображает зависимости между различными этапами проекта. Эти диаграммы можно создать автоматически с помощью программных средств поддержки управления на основе информации, заложенной в базе данных проекта.

Рассмотрим этапы некоего проекта, представленные в табл. 2, из которой, в частности, видно, что этап Т3 зависит от этапа Т1. Это значит, что этап Т1 должен завершиться прежде, чем начнется этап Т3. Например,

на этапе Т1 проводится компонентный анализ создаваемого программного продукта, а на этапе Т3 – проектирование системы.

На основе приведенных значений длительности этапов и зависимости между ними строится сетевой график последовательности этапов (рис. 53). На этом графике видно, какие работы могут выполняться параллельно, а какие должны выполняться последовательно друг за другом. Этапы обозначены прямоугольниками. Контрольные отметки и контрольные проектные элементы показаны в виде овалов и обозначены буквой М с соответствующим номером. Даты на данной диаграмме соответствуют началу выполнения этапов. Сетевую диаграмму следует читать слева направо и сверху вниз.

Таблица 12

Этапы проекта

Этап	Длительность (дни)	Зависимость
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

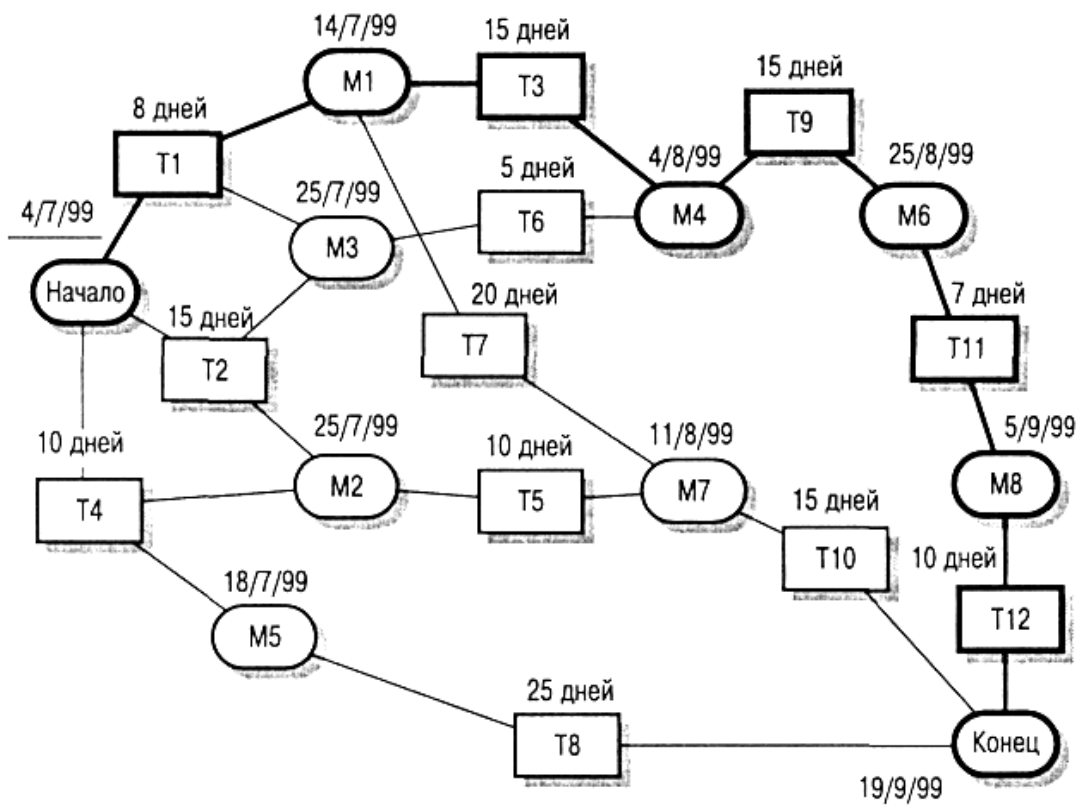


Рис. 53. Сетевая диаграмма этапов

Если для создания сетевой диаграммы используются программные средства поддержки управления проектом, каждый этап должен заканчиваться контрольной отметкой. Очередной этап может начаться только тогда, когда будет получена контрольная отметка (которая может зависеть от нескольких предшествующих этапов). Поэтому в третьем столбце таблицы 12 приведены контрольные отметки; они будут достигнуты только тогда, когда будет завершен этап, в строке которого помещена соответствующая контрольная отметка.

Любой этап не может начаться, пока не выполнены все этапы на всех путях, ведущих от начала проекта к данному этапу. Например, этап T9 не может начаться, пока не будут завершены этапы T3 и T6. Отметим, что в данном случае достижение контрольной отметки M4 говорит о том, что эти этапы завершены.

Минимальное время выполнения всего проекта можно рассчитать, просуммировав в сетевой диаграмме длительности этапов на самом длинном пути (длина пути здесь измеряется не количеством этапов на



пути, а суммарной длительностью этих этапов) от начала проекта до его окончания (это так называемый критический путь). В нашем случае продолжительность проекта составляет 11 недель или 55 рабочих дней. На рис. 53 критический путь показан более толстыми линиями, чем остальные пути. Таким образом, общая продолжительность реализации проекта зависит от этапов работ, находящихся на критическом пути. Любая задержка в завершении любого этапа на критическом пути приведет к задержке всего проекта.

Задержка в завершении этапов, не входящих в критический путь, не влияет на продолжительность всего проекта до тех пор, пока суммарная длительность этих этапов (с учетом задержек) на каком-нибудь пути не превысит продолжительности работ на критическом пути. Например, задержка этапа T8 на срок, меньший 20 дней, никак не влияет на общую продолжительность проекта. На рис. 54 представлена временная диаграмма, на которой показаны возможные задержки на каждом этапе.

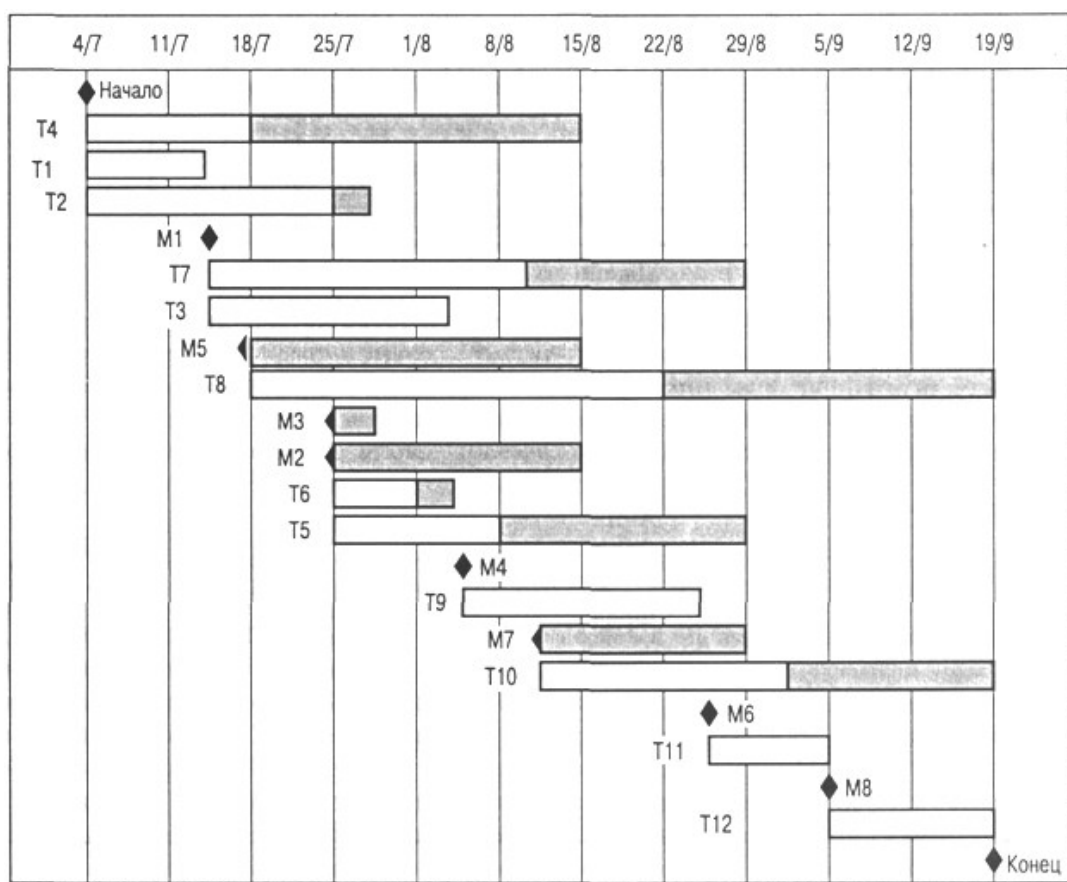


Рис. 54. Временная диаграмма длительности этапов

Сетевая диаграмма позволяет увидеть в зависимости этапов значимость того или иного этапа для реализации всего проекта. Внимание к этапам критического пути часто позволяет найти способы их изменения с тем, чтобы сократить длительность всего проекта. Менеджеры используют сетевую диаграмму для распределения работ.

Временная диаграмма (иногда называемая по имени ее изобретателя диаграммой Гантта) может быть построена программными средствами поддержки процесса управления. Она показывает длительность выполнения каждого этапа и возможные их задержки (показаны затененными прямоугольниками), а также даты начала и окончания каждого этапа. Этапы критического пути не имеют затененных прямоугольников; это означает, что задержка с завершением данных этапов приведет к увеличению длительности всего проекта.

Подобно распределению времени выполнения этапов, менеджер должен рассчитать распределение ресурсов по этапам, в частности назначить исполнителей на каждый этап. В таблице 13 приведено распределение разработчиков на каждый этап, представленный на рис. 55.

Приведенная таблица может быть использована программными средствами поддержки процесса управления для построения временной диаграммы занятости сотрудников на определенных этапах работ. Персонал не занят в работе над проектом все время его реализации. В течение периода незанятости сотрудники могут быть в отпуске, работать над другими проектами, проходить обучение и т. д.

В больших организациях обычно работает много специалистов, которые задействуются в проекте по мере необходимости. Конечно, такой подход может создать определенные проблемы для менеджеров проектов. Например, если специалист занят в проекте, который задерживается, это может создать прямые сложности для других проектов, где он также должен участвовать.

Распределение исполнителей по этапам

Этап	Исполнитель
T1	Джейн
T2	Анна
T3	Джейн
T4	Фред
T5	Мэри
T6	Анна
T7	Джим
T8	Фред
T9	Джейн
T10	Анна
T11	Фред
T12	Фред

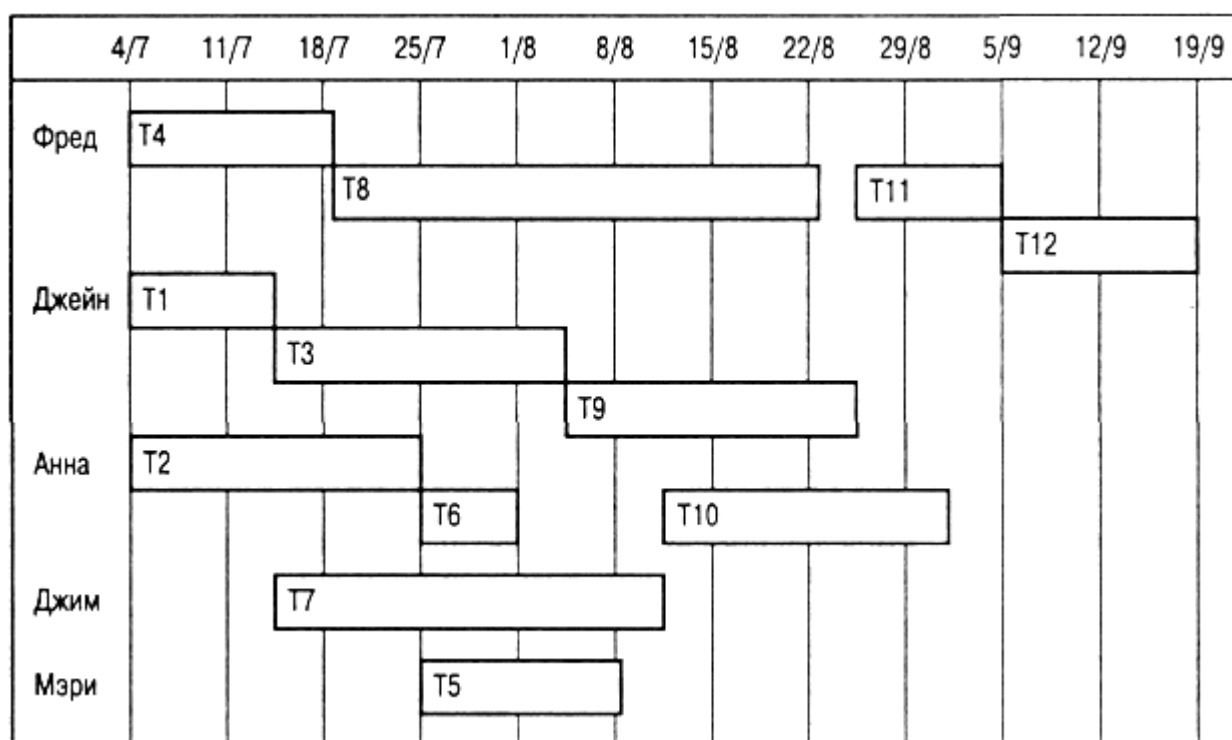


Рис. 55. Временная диаграмма распределения работников по этапам

Первоначальный график работ неизбежно содержит какие-нибудь ошибки или недоработки. По мере реализации проекта рассчитанные

оценки длительности выполнения этапов работ должны сравниваться с реальными сроками выполнения этих этапов. Результаты сравнения должны использоваться в качестве основы для пересмотра графика работ еще не реализованных этапов проекта, в частности для того, чтобы попытаться уменьшить длительность этапов критического пути.

Важной частью работы менеджера проекта является оценка рисков, которые могут повлиять на график работ или на качество создаваемого программного продукта, и разработка мероприятий по предотвращению рисков. Результаты анализа рисков должны быть отражены в плане проекта. Определение рисков и разработка мероприятий по уменьшению их влияния на ход выполнения проекта называется управлением рисками.

Упрощенно риск можно понимать как вероятность проявления каких-либо неблагоприятных обстоятельств, негативно влияющих на реализацию проекта. Риски могут угрожать проекту в целом, создаваемому программному продукту или организации-разработчику. Можно выделить три типа рисков.

1. *Риски для проекта*, которые влияют на график работ или ресурсы, необходимые для выполнения проекта.
2. *Риски для разрабатываемого продукта*, влияющие на качество или производительность разрабатываемого программного продукта.
3. *Бизнес-риски*, относящиеся к организации-разработчику или поставщикам.

Конечно, эти типы рисков могут пересекаться. Например, если опытный программист покидает проект, это будет риском для проекта (поскольку задерживается срок сдачи готового продукта), риском для продукта (так как новый программист, заменивший ушедшего, может оказаться не слишком опытным и сделать ошибки в программе) и бизнес-риском (поскольку задержка данного проекта может негативно повлиять на будущие деловые контакты между заказчиком и организацией-разработчиком). Конкретные типы рисков, которые могут оказать влияние

на данный проект, зависят от вида создаваемого программного продукта и от организационного окружения, где реализуется программный проект. Вместе с тем многие типы рисков способны повлиять на любые программные проекты, эти риски приведены в табл. 14.

Таблица 14

Возможные риски программных проектов

Риск	Типы риска	Описание риска
Текучесть разработчиков	Риск для проекта	Опытные разработчики покидают проект до его завершения
Изменение в управлении организацией	Риск для проекта	Организация меняет свои приоритеты в управлении проектом
Неготовность аппаратных средств	Риск для проекта	Аппаратные средства, которые необходимы для проекта, не поступили вовремя или не готовы к эксплуатации
Изменение требований	Риск для проекта и для разрабатываемого продукта	Появление большого количества непредвиденных изменений в требованиях, предъявляемых к разрабатываемому ПО
Задержка в разработке спецификации	Риск для проекта и для разрабатываемого продукта	Спецификации основных интерфейсов подсистем не поступили к разработчикам в соответствии с графиком работ
Недостаточная эффективность CASE-средств	Риск для разрабатываемого продукта	CASE-средства, предназначенные для поддержки проекта, оказались менее эффективными, чем ожидалось
Изменения в технологии разработки ПО	Бизнес-риск	Основные технологии построения программной системы заменяются новыми
Появление конкурирующего программного продукта	Бизнес-риск	На рынке программных продуктов до окончания проекта появилась конкурирующая программная система

Схема процесса управления рисками показана на рис. 56. Этот процесс состоит из четырех стадий.

1. *Определение рисков.* Определяются возможные риски для проекта, для разрабатываемого продукта и бизнес-риски.
2. *Анализ рисков.* Оценивается вероятность и последовательность появления рисковых ситуаций.
3. *Планирование рисков.* Планируются мероприятия по предотвращению рисков или минимизации их воздействия на проект.
4. *Мониторинг рисков.* Постоянное оценивание вероятностей рисков и выполнение мероприятий по смягчению последствий проявления рисковых ситуаций.

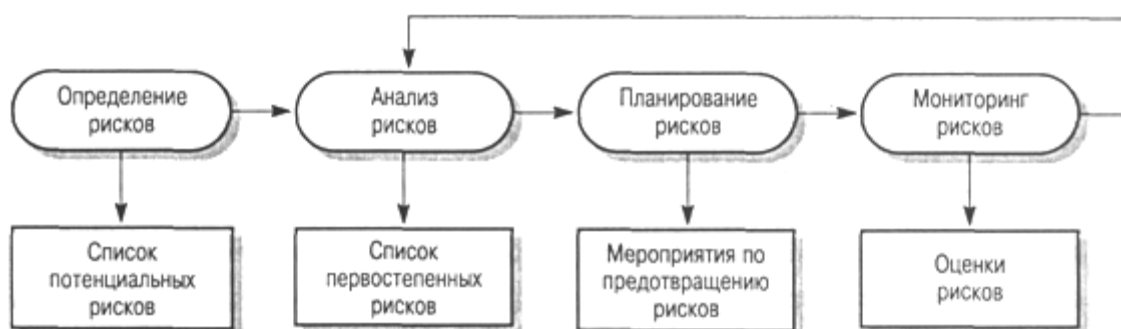


Рис. 56. Процесс управления рисками

Процесс управления рисками, как и другие процессы планирования, является итерационным, выполняемым в течение всего срока реализации проекта. Сначала разрабатываются планы управления рисками, затем постоянно отслеживается ситуация вокруг реализации проекта. При поступлении новой информации о возможных рисках заново проводится анализ рисков и первостепенное внимание уделяется новым рискам. По мере поступления новой информации также изменяются планы мероприятий по предотвращению и смягчению рисков.

Результаты процесса управления рисками документируются в виде планов управления рисками. Они должны включать описание возможных проектных рисков, их анализ и перечень мероприятий, необходимых для управления рисками.

Определение рисков – первая стадия процесса управления рисками. На этой стадии описываются риски, которые могут проявиться при

реализации проекта. В принципе на этой стадии не должна оцениваться вероятность и значимость рисков, но на практике маловероятные риски с незначительными последствиями обычно отбрасываются сразу.

Определение рисков может выполняться в режиме командной работы с использованием подхода «мозговой штурм» либо основываться на опыте менеджера. При определении рисков может помочь приведенный ниже список возможных категорий рисков.

1. *Технологические риски.* Проистекают из программных и аппаратных технологий, на основе которых разрабатывается система.

2. *Риски, связанные с персоналом.* Связаны с членами команды разработчиков.

3. *Организационные риски.* Проистекают из организационного окружения, в котором выполняется проект.

4. *Инструментальные риски.* Связаны с используемыми CASE-средствами и другими средствами поддержки процесса создания ПО.

5. *Риски, связанные с системными требованиями.* Проявляются при изменении требований, предъявляемых к разрабатываемой системе.

6. *Риски оценивания.* Связаны с оцениванием характеристик программной системы и ресурсов, необходимых для реализации проекта.

В таблице 15 представлены некоторые примеры, относящиеся к каждой из описанных категорий рисков. Результатом этапа определения рисков будет длинный список возможных рисков, которые могут повлиять на разрабатываемый программный продукт, проект или организацию-разработчика.

При анализе для каждого определенного риска подсчитывается вероятность его проявления и ущерб, который он может нанести. Не существует простых методов выполнения анализа рисков – в значительной мере он основан на мнении и опыте менеджера. Можно привести следующую шкалу вероятностей рисков и их последствий.

1. Вероятность риска считается очень низкой, если она имеет значение менее 10 %; низкой, если ее значение от 10 до 25 %; средней при значениях от 25 до 50 %; высокой, если значение колеблется от 50 до 75 %; очень высокой при значениях более 75 %.

2. Возможный ущерб от рискованных ситуаций можно подразделить на катастрофический, серьезный, терпимый и незначительный.

Результаты анализа рисков должны быть представлены в виде таблицы рисков, упорядоченных по степени возможного ущерба. В табл. 16 приведен упорядоченный список рисков, описанных в табл. 15; там же указаны вероятности этих рисков. Здесь вероятности рисков и степень ущерба от них указаны произвольно. На практике для их определения необходима подробная информация о проекте, технологии создания ПО, команде разработчиков и о самой организации.

Конечно, как вероятность рисков, так и возможный ущерб от них должны пересматриваться при поступлении дополнительной информации об этих рисках и по мере реализации мероприятий по управлению ими. Поэтому подобные таблицы рисков должны переделываться на каждой итерации процесса управления рисками.

После проведения анализа рисков определяются наиболее значимые риски, которые затем отслеживаются на протяжении всего срока выполнения проекта. Определение этих значимых рисков зависит от их вероятностей и возможного ущерба. В общем случае всегда отслеживаются риски с катастрофическими последствиями, а также риски с серьезным ущербом, значение вероятности которых выше среднего.



## Категории рисков

Категория рисков	Примеры рисков
Технологические риски	База данных, которая используется в программной системе, не обеспечивает обработку ожидаемого объема транзакций. Программные компоненты, которые используются повторно, имеют дефекты, ограничивающие их функциональные возможности
Риски, связанные с персоналом	Невозможно подобрать работников с требуемым профессиональным уровнем. Ведущий разработчик заболел в самое критическое время. Невозможно организовать необходимое обучение персонала
Организационные риски	В организации, выполняющей разработку ПО, произошла ре-организация, в результате чего изменились приоритеты в управлении проектом. Финансовые затруднения в организации привели к уменьшению бюджета проекта
Инструментальные риски	Программный код, генерируемый CASE-средствами, не эффективен. CASE-средства невозможно интегрировать с другими средствами поддержки проекта
Риски, связанные с системными требованиями	Изменения требований приводят к значительным повторным темными требованиями работам по проектированию системы. Первоначальная нечеткая формулировка пользовательских требований привела к значительным изменениям системных требований, проявившихся на поздних стадиях разработки проекта
Риски оценивания	Недооценки времени выполнения проекта. Скорость выявления дефектов в системе ниже ранее запланированной. Размер системы значительно превышает первоначально рассчитанный

## Список рисков после проведения их анализа

Риск	Вероятность	Степень ущерба
Финансовые затруднения в организации привели к уменьшению бюджета проекта	Низкая	Катастрофическая
Невозможно подобрать работников с требуемым профессиональным уровнем	Высокая	Катастрофическая
Ведущий разработчик заболел в самое критическое время	Средняя	Серьезная
Программные компоненты, используемые повторно, имеют дефекты, ограничивающие их функциональные возможности	Средняя	Серьезная
Изменения требований приводят к значительным повторным работам по проектированию системы	Средняя	Серьезная
Риск	Вероятность	Степень ущерба
В организации, выполняющей разработку ПО, произошла реорганизация, в результате чего изменились приоритеты в управлении проектом	Высокая	Серьезная
База данных, которая используется в программной системе, не обеспечивает обработку ожидаемого объема транзакций	Средняя	Серьезная
Недооценки времени выполнения проекта	Высокая	Серьезная
CASE-средства невозможно интегрировать с другими средствами поддержки проекта	Высокая	Терпимая
Невозможно организовать необходимое обучение персонала	Средняя	Терпимая
Скорость выявления дефектов в системе ниже ранее спланированной	Средняя	Терпимая
Размер системы значительно превышает первоначально рассчитанный	Высокая	Терпимая
Программный код, генерируемый CASE-средствами, неэффективен	Средняя	Незначительная

В некоторых статьях рекомендуется определить и отслеживать «10 верхних» рисков, но это не всегда обоснованная рекомендация. Количество рисков, которые необходимо отслеживать, зависит от

конкретного проекта. Это может быть пять рисков, а может – пятнадцать. Но, конечно, количество рисков, по которым проводится мониторинг, должно быть обозримым. Большое количество отслеживаемых рисков потребует огромного количества собираемой информации. Из списка рисков, представленных в таблице 16, для мониторинга следует отобрать те риски, которые могут привести к катастрофическим и серьезным последствиям для вашего проекта.

Планирование заключается в определении стратегии управления каждым значимым риском, отобранным для мониторинга после анализа рисков. Здесь также не существует общепринятых подходов для разработки таких стратегий – многое основывается на «чутье» и опыте менеджера проекта. В табл. 16 показаны возможные стратегии управления основными рисками, приведенными в табл. 17.

Существует три категории стратегий управления рисками.

1. *Стратегии предотвращения рисков.* Согласно этим стратегиям следует проводить мероприятия, снижающие вероятность проявления рисков. Примером может служить стратегия исключения потенциально дефектных компонентов.

*Минимизационные стратегии.* Направлены на уменьшение возможного ущерба от рисков. Примером служит стратегия уменьшения ущерба от болезни членов команды разработчиков .

2. *Планирование «аварийных» ситуаций.* Согласно этим стратегиям необходимо иметь план мероприятий, которые следует выполнить в случае проявления рисков ситуации. Мониторинг рисков заключается в регулярном пересчете вероятностей рисков и ущерба, который они могут нанести. Для этого необходимо постоянно отслеживать факторы, которые влияют на вероятность рисков и возможный ущерб. Эти факторы зависят от типов риска. В табл. 18 приведены признаки, которые помогают определить тип риска.

## Стратегии управления рисками

Риск	Стратегия
Финансовые проблемы организации	Подготовить краткий документ для руководства организации, показывающий важность данного проекта для достижения финансовых целей организации
Проблемы неквалифицированного персонала	Предупредить заказчика о потенциальных трудностях и возможной задержке проекта, рассмотреть вопрос о покупке компонентов системы
Болезни персонала	Реорганизовать работу команды разработчиков таким образом, чтобы обязанности и работа членов команды перекрывали друг друга, вследствие этого разработчики будут знать и понимать задачи, выполняемые другими сотрудниками
Дефектные системные компоненты	Заменить потенциально дефектные системные компоненты покупными компонентами, гарантирующими качество работы
Изменения требований	Попытаться определить требования, наиболее вероятно подверженные изменениям; в структуре системы не отображать детальную информацию
Реорганизация компании-разработчика	Подготовить краткий документ для руководства компании, показывающий важность данного проекта для достижения финансовых целей компании
Недостаточная производительность базы данных	Рассмотреть возможность покупки более производительной базы данных
Недооценки времени выполнения проекта	Рассмотреть вопрос о покупке системных компонентов, исследовать возможность использования генератора программного кода

## Признаки рисков

Тип риска	Признаки
Технологические риски	Задержки в поставке оборудования или программных средств поддержки процесса создания ПО, многочисленные документированные технологические проблемы
Риски, связанные с персоналом	Низкое моральное состояние персонала, натянутые отношения между членами команды разработчиков, низкое качество выполненной работы
Организационные риски	Разговоры среди персонала о пассивности и недостаточной компетентности высшего руководства организации
Инструментальные риски	Нежелание разработчиков использовать программные средства поддержки, неодобрительные отзывы о CASE-средствах, запросы на более мощные инструментальные средства
Риски, связанные с системными требованиями	Необходимость пересмотра многих системных требований, недовольство заказчика ПО
Риски оценивания	Изменения графика работ, многочисленные отчеты о нарушении графика работ

Мониторинг рисков должен быть непрерывным процессом, отслеживающим ход выполнения мероприятий по управлению рисками, при этом каждый основной риск должен рассматриваться отдельно.

**Порядок выполнения работы**

В ходе выполнения лабораторной работы необходимо:

1. Построить модель управления проектом, включающую:
  - определение всех этапов проекта, зависимых этапов, определение длительности этапов;
  - построение на основе полученных данных сетевой и временной диаграмм;
  - построение диаграммы распределения работников по этапам;

а) при определении этапа указывается его название – отражающее суть этапа (например, определение пользовательских требований, проектирование интерфейса и т. д.);

б) этапов должно быть не менее 7, срок реализации проекта – с 1.09 по 31.12;

в) в проекте задействовано 3 человека (группа разработчиков).

2. Изучить предлагаемый теоретический материал.
3. Построить временную и сетевую диаграммы для выбранного проекта.
4. Построить диаграмму распределения участников группы по этапам.
5. Построить список возможных рисков с указанием названия риска, его описание и типа.
6. Провести анализ рисков.
7. Описать стратегию планирования рисков.
8. Оформить отчёт, включающий все полученные диаграммы и описание стратегии планирования

## СПИСОК ЛИТЕРАТУРЫ

1. Брауде, Э. Технология разработки программного обеспечения [Текст] / Э. Брауде. – Санкт-Петербург : Питер, 2004. – 655 с.
2. Вендров, А. М. CASE-технологии. Современные методы и средства проектирования информационных систем [Текст] / А. М. Вендров. – Москва : Финансы и статистика, 1998. – 176 с.
3. Грекул, В. И. Проектирование информационных систем [Электронный ресурс] : [курс лекций] / В. И. Грекул. – Режим доступа : <http://www.intuit.ru/department/se/devis/>. – 15.10.2014.
4. Кантор, М. Управление программными проектами [Текст] : практ. рук. по разработке успеш. программ. обеспечения / М. Кантор. – Москва : Вильямс, 2002. – 173 с.
5. Константайн, Л. Разработка программного обеспечения [Текст] / Л. Константайн, Л. Локвуд. – Санкт-Петербург : Питер, 2004. – 592 с.
6. Ларман, К. Применение UML 2.0 и шаблонов проектирования [Текст] : пер. с англ. / К. Ларман. – Москва : Вильямс, 2007. – 736 с.
7. Орлов, С. Технологии разработки программного обеспечения [Текст] : учеб. для вузов / С. Орлов. – 3-е издание. – Санкт-Петербург : Питер, 2004. – 528 с.
8. Соммервилл, И. Инженерия программного обеспечения [Текст] : пер. с англ. / И. Соммервилл. – 6-е изд. – Москва : Вильямс, 2002. – 624 с.
9. Якобсон, А. Унифицированный процесс разработки программного обеспечения [Текст] / А. Якобсон, Г. Буч, Д. Рамбо. – Санкт-Петербург : Питер, 2002. – 496 с.

Учебное издание

Долженкова Мария Львовна  
Караваева Ольга Владимировна

# **ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ**

Учебно-методическое пособие

Подписано к использованию 30. 07.2014. Заказ № 1838.

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Вятский государственный университет»

610000, г. Киров, ул. Московская, 36, тел.: (8332) 64-23-56, <http://vyatsu.ru>