Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

В. В. Бахтизин, Л. А. Глухова

Технология разработки программного обеспечения

*Допущено Министерством образования Республики Беларусь
в качестве учебного пособия
для студентов высших учебных заведений по специальности
«Программное обеспечение информационных технологий»*

Минск БГУИР 2010

УДК 004.413(075.8)
ББК 32.973.26 – 018.2я73
Б30

Рецензенты:

кафедра дискретной математики и алгоритмики
Белорусского государственного университета,
заведующий кафедрой, доктор физико-математических наук,
профессор В. М. Котов;
доцент Гродненского государственного университета им. Янки Купалы,
кандидат технических наук, доцент А. М. Кадан

Бахтизин, В. В.

Б30 Технология разработки программного обеспечения : учеб. пособие / В. В. Бахтизин, Л. А. Глухова. – Минск : БГУИР, 2010. – 267 с. : ил.
ISBN 978-985-488-512-4

В учебном пособии доступно и наглядно рассмотрены жизненный цикл программных средств, стратегии разработки и реализующие их модели жизненного цикла, процедура выбора модели жизненного цикла для конкретного проекта. Описаны классические и современные методологии и технологии анализа и проектирования программных средств. Приведены основы организации и классификация CASE-средств.

Учебное пособие предназначено для студентов высших учебных заведений, чья специализация связана с программным обеспечением, а также для специалистов в области разработки программного обеспечения.

УДК 004.413(075.8)
ББК 32.973.26–018.2я73

ISBN 978-985-488-512-4

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время во все сферы деятельности человека широко внедряются информационные технологии. Это приводит к разработке огромного количества программных средств (ПС) различного функционального назначения. При этом объем и сложность используемых ПС постоянно возрастают.

В этой связи многие подходы к разработке ПС, применяемые на начальных этапах развития вычислительной техники, теряют свои позиции, поскольку не позволяют в полной мере получить ПС необходимого уровня качества за заданный промежуток времени при ограниченных финансовых, людских и технических ресурсах. Связано это с рядом причин.

Во-первых, интуитивный поход к разработке ПС, основанный на знаниях, умениях и талантах отдельных программистов-одиночек, не позволяет разрабатывать сложные ПС и противоречит принципам их коллективной разработки.

Во-вторых, использование коллективных методов разработки требует структурированного подхода к понятиям жизненного цикла (ЖЦ) и модели жизненного цикла программных средств (ЖЦ ПС). В противном случае возникают существенные риски не довести проект до конца или не получить продукт с заданными свойствами.

В-третьих, используемые методологии разработки ПС с ростом сложности и критичности последних перестают удовлетворять целям и задачам, стоящим перед их разработчиками.

В-четвертых, рост сложности и объема разрабатываемых ПС автоматически приводит к появлению достаточно сложных в применении методологий анализа, проектирования и последующих этапов разработки. Использование таких методологий становится невозможным без применения инструментальных средств их поддержки.

Вышеназванные причины зачастую приводят к неудовлетворительным результатам выполнения проектов.

Для иллюстрации вышесказанного приведем некоторые данные статистики [37, 18].

Известно, что 30 – 40 % проектов по разработке ПС не доходят до завершения. Около 70 % всех проектов реализуют поставленные задачи не полностью. Средний проект завершается с опозданием на 220 %.

В 10 % проектов результат не соответствует требованиям. В 12 % заказчик недостаточно привлекался к работе, чтобы обеспечить требуемые характеристики продукта. В 22 % проектов не все вносимые изменения принимались во внимание.

Поэтому в последние десятилетия во всем мире ведущими специалистами в области теории и практики программного обеспечения (ПО) активно выполняются работы по усовершенствованию подходов к разработке ПС. Эти работы ведутся в различных направлениях. Основными из них являются следующие.

Стандартизация жизненного цикла программных средств. В настоящее время разрабатывается и постоянно обновляется большое количество международных и национальных стандартов, посвященных различным аспектам ЖЦ ПС. В 2008 г. Международной организацией по стандартизации ИСО принята вторая редакция основного в данном направлении международного стандарта *ISO/IEC 12207:2008 – Системная и программная инженерия – Процессы жизненного цикла программных средств*. В Республике Беларусь действует национальный стандарт *СТБ ИСО/МЭК 12207–2003 – Информационная технология – Процессы жизненного цикла программных средств*, являющийся аутентичным аналогом первой редакции международного стандарта *ISO/IEC 12207:1995*.

Структуризация моделей жизненного цикла программных средств. С 80-х г. XX в. ведутся работы по усовершенствованию стратегий разработки ПС и созданию моделей ЖЦ, реализующих данные стратегии. В настоящее время широко используются три базовые стратегии разработки ПС: каскадная, инкрементная, эволюционная. Разработано большое количество моделей ЖЦ, реализующих данные стратегии.

Разработка методов выбора моделей жизненного цикла. К настоящему моменту разработан ряд методик и процедур выбора моделей ЖЦ, исходя из условий и характеристик конкретного проекта.

Создание методологий анализа и проектирования программных средств. В настоящее время создано большое количество методологий, направленных в первую очередь на начальные этапы процесса разработки ПС – анализ предметной области, разработку требований к системе и ПС, проектирование системы и ПС.

Разработка инструментальных средств поддержки современных методологий разработки программных средств и систем. С 80-х г. XX в. бурно развиваются CASE-средства, предназначенные для автоматизации процессов ЖЦ ПС и систем. К настоящему времени многими компаниями разработаны линейки CASE-средств, поддерживающие практически весь ЖЦ ПС и систем.

Управление качеством разрабатываемых программных средств. Основу управления качеством составляет оценка качества ПС. В настоящее время ведутся активные работы в области стандартизации оценки качества ПС и их сертификации. Основными международными стандартами, регламентирующими вопросы качества ПС и их оценки, являются следующие серии стандартов:

- *ISO/IEC 9126–1–4:2001–2004 – Программная инженерия – Качество продукта;*
- *ISO/IEC 14598–1–6:1998–2001 – Информационная* (*программная*) *инженерия – Оценка программного продукта*.

В настоящее время разрабатывается серия стандартов *ISO/IEC 250XX – Разработка программного обеспечения – Требования к качеству и оценка программного продукта (SQuaRE*). Стандарты данной серии призваны заменить две вышеназванные серии стандартов.

Вопросы стандартизации ЖЦ ПС и оценки качества ПС подробно рас-

смотрены в предыдущем учебном пособии авторов «Стандартизация и сертификация программного обеспечения» [15].

В данном учебном пособии рассматриваются остальные вопросы процесса разработки и технологий разработки ПС.

С учетом изложенного сформирована структура учебного пособия.

Пособие состоит из семи разделов.

В **первом разделе** рассматриваются основные понятия и определения в области технологий разработки ПС и систем. Кратко описываются процессы ЖЦ ПС и систем, регламентированные стандартом *СТБ ИСО/МЭК 12207–2003*.

Второй раздел посвящен стратегиям разработки ПС и систем. Проанализированы три базовые стратегии разработки: каскадная, инкрементная, эволюционная. Рассмотрены модели ЖЦ ПС и систем, реализующие данные стратегии, оценены их достоинства, недостатки и области применения.

В **третьем разделе** рассмотрены принципы выбора модели ЖЦ ПС и систем, исходя из условий конкретного проекта. Приведена классификация проектов и процедура выбора модели ЖЦ, предложенные Институтом качества программного обеспечения SQI (Software Quality Institute, США).

Четвертый раздел посвящен рассмотрению классических методологий структурного проектирования ПС. Рассмотрены общие принципы и методы реализации структурного программирования, модульного проектирования, нисходящего проектирования, методов расширения ядра. Данные методологии являются основой ряда современных методологий и технологий разработки ПС. Описаны такие характеристики структурного разбиения программ на модули, как связность и сцепление.

В **пятом разделе** описываются CASE-технологии структурного анализа и проектирования ПС. Даны общие сведения о CASE-технологиях. Детально рассмотрены широко используемые методологии функционального моделирования IDEF0, структурного анализа потоков данных DFD, информационного моделирования IDEF1X. Описаны метод JSD Джексона и диаграммы Варнье–Орра.

В **шестом разделе** приведены основы объектно-ориентированного анализа и проектирования. Подробно рассмотрены правила построения диаграмм вариантов использования.

Седьмой раздел посвящен рассмотрению инструментальных средств разработки ПО. Описаны основы CASE-средств, их состав и функциональные возможности, дана классификация CASE-средств. Рассмотрены линейки инструментальных средств Telelogic и AllFusion, предназначенные для автоматизации ЖЦ организаций, систем и ПС.

В пособии используются следующие сокращения:

ЖЦ – жизненный цикл;

ПО – программное обеспечение;

ПС – программные средства.

РАЗДЕЛ 1. ВВЕДЕНИЕ В ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ

1.1. Основные понятия и определения

Существуют различные определения технологии разработки ПО. Ниже приведены наиболее распространенные из них.

Технология разработки программного обеспечения – это совокупность процессов и методов создания программного продукта.

Технология разработки программного обеспечения – это система инженерных принципов для создания экономичного ПО, которое надежно и эффективно работает в реальных компьютерах [30]. Данное определение имеет частный характер, поскольку учитывает только две из шести характеристик качества ПО – надежность и эффективность [6, 10, 15]. С учетом этого можно сформулировать более общее и точное определение.

Технология разработки программного обеспечения – это система инженерных принципов для создания экономичного ПО с заданными характеристиками качества.

Одно из определений современных технологий разработки ПО приведено в подразд. 5.1.

Близким по смыслу к термину ***технология разработки ПО*** является широко используемый в настоящее время термин ***программная инженерия*** (*software engineering).*

Любая технология разработки ПО базируется на некоторой методологии или совокупности методологий.

Под ***методологией*** понимается система принципов и способов организации процесса разработки программных средств. *Цель* методологии разработки ПО – внедрение методов разработки ПС, обеспечивающих достижение соответствующих характеристик качества.

В настоящее время широкую известность приобрели два базовых принципа разработки ПС: *модульный* и *объектно-ориентированный*.

Разработка модульных ПС основывается на использовании *структурных методов проектирования*, целью которых является разбиение по некоторым правилам проектируемого программного средства на структурные компоненты. К структурным методам проектирования относятся такие классические методы, как структурное программирование, нисходящее проектирование, расширение

ядра, восходящее проектирование и их комбинации, а также ряд современных методов и методологий разработки ПО.

Объектно-ориентированная разработка базируется на применении объектных методов, к которым относятся методологии объектно-ориентированного анализа, проектирования и программирования.

В учебном пособии используются следующие термины [5, 9].

Программные средства, программное обеспечение (software) – полный набор (программное обеспечение) или часть (программные средства) программ, процедур, правил и связанной с ними документации системы обработки информации. *Программное средство* – ограниченная часть программного обеспечения системы обработки информации, имеющая определенное функциональное назначение.

Программный модуль (software unit) – отдельно компилируемая часть программного кода (программы).

Программный продукт (software product) – набор компьютерных программ, процедур, а также связанных с ними документации и данных. Продукты включают промежуточные продукты и продукты, предназначенные для пользователей типа разработчиков и персонала сопровождения.

Система (system) – комплекс, состоящий из процессов, технических и программных средств, устройств и персонала, обладающий возможностью удовлетворять установленным потребностям или целям.

Нотация (notation) – система графических обозначений для записи промежуточных и конечного результатов разработки ПС (в том числе предметной области, требований, результатов проектирования и т.п.).

Базовыми понятиями технологии разработки ПС являются понятия жизненного цикла, стратегии разработки и модели жизненного цикла, реализующей данную стратегию. Эти понятия рассматриваются в подразд. 1.2 и разд. 2 пособия.

Резюме

Технология разработки программного обеспечения – это система инженерных принципов для создания экономичного ПО с заданными характеристиками качества. Методология разработки программного обеспечения – это система принципов и способов организации процесса разработки программных средств. Целью методологии является внедрение методов разработки ПО, обеспечивающих достижение соответствующих характеристик качества.

1.2. Жизненный цикл программных средств

В настоящее время базовым стандартом в области ЖЦ ПС и систем является международный стандарт ***ISO/IEC 12207:2008 – Системная и программ-***

ная инженерия – Процессы жизненного цикла программных средств [4]. В Республике Беларусь с 2004 г. действует национальный стандарт ***СТБ ИСО/МЭК 12207–2003 – Информационная технология – Процессы жизненного цикла программных средств*** [9, 15], являющийся аутентичным аналогом предыдущей редакции международного стандарта ***ISO/IEC 12207:1995*** [3].

В соответствии со стандартом *СТБ ИСО/МЭК 12207–2003* под ***жизненным циклом*** программного средства или системы подразумевается совокупность процессов, работ и задач, включающая в себя разработку, эксплуатацию и сопровождение программного средства или системы и охватывающая их жизнь от формулирования концепции до прекращения использования. ЖЦ ПС состоит из *процессов*. Каждый процесс ЖЦ разделен на набор *работ*. Каждая работа разделена на набор *задач*.

Процессы ЖЦ ПС делятся на три *группы*:

- основные;
- вспомогательные;
- организационные.

Основные процессы жизненного цикла – это процессы, которые реализуются под управлением основных сторон, участвующих в жизненном цикле программных средств. Основными сторонами являются заказчик, поставщик, разработчик, оператор и персонал сопровождения программных продуктов. К *основным процессам* относится пять процессов:

- заказ;
- поставка;
- разработка;
- эксплуатация;
- сопровождение.

Процесс заказа определяет работы и задачи заказчика и состоит из определения потребностей заказчика в системе или программном продукте, подготовки и выпуска заявки на подряд, выбора поставщика и управления процессом заказа до завершения приемки системы или программного продукта.

Процесс поставки определяет работы и задачи поставщика. Данный процесс начинается с решения о подготовке предложения в ответ на заявку на подряд, присланную заказчиком, или с подписания договора с заказчиком на поставку системы или программного продукта. Затем определяются процедуры и ресурсы, необходимые для управления и обеспечения проекта, включая разработку проектных планов и их выполнение.

Процесс разработки состоит из работ и задач, выполняемых разработчиком. Данный процесс содержит *тринадцать работ*:

1) подготовка процесса разработки;
2) анализ требований к системе;
3) проектирование системной архитектуры;
4) анализ требований к программным средствам;
5) проектирование программной архитектуры;
6) техническое проектирование программных средств;

7) программирование и тестирование программных средств;
8) сборка программных средств;
9) квалификационные испытания программных средств;
10) сборка системы;
11) квалификационные испытания системы;
12) ввод в действие программных средств;
13) обеспечение приемки программных средств.

При выполнении работы 1 «Подготовка процесса разработки» выбирается модель ЖЦ ПС и систем. В данную модель структурируются процессы, работы и задачи стандарта *СТБ ИСО/МЭК 12207–2003*. Выбираются и адаптируются стандарты, методы, инструментальные средства разработки, языки программирования. Формируется план проведения работ процесса разработки.

При выполнении работы 2 «Анализ требований к системе» анализируется область применения системы. На основании результатов анализа предметной области определяются требования к ней. Выполняется оценка разработанных требований.

При выполнении работы 3 «Проектирование системной архитектуры» определяется общая архитектура системы. Осуществляется распределение требований к системе между объектами технических и программных средств архитектуры и ручными операциями. Производится дальнейшее уточнение требований. Осуществляется оценка архитектуры системы и требований к объектам архитектуры.

При выполнении работы 4 «Анализ требований к программным средствам» анализируется назначение программного средства. Исходя из результатов анализа определяются и уточняются требования к программному средству. Выполняется оценка разработанных требований.

При выполнении работы 5 «Проектирование программной архитектуры» разрабатывается эскизный проект программного средства. Требования к программному средству преобразуются в его архитектуру, распределяются между его компонентами и уточняются далее. Производится оценка результатов эскизного проектирования.

При выполнении работы 6 «Техническое проектирование программных средств» осуществляется детальное проектирование программного средства (разрабатывается технический проект для компонентов программного объекта). Компоненты проектируются до уровня их представления в виде набора программных модулей. Сложность модулей должна обеспечить возможность их непосредственного кодирования в следующей работе (работе 7). Производится распределение технических требований к компонентам между программными модулями и дальнейшее уточнение требований. Выполняется оценка технического проекта.

При выполнении работы 7 «Программирование и тестирование программных средств» производится кодирование и тестирование программных модулей. Осуществляется оценка полученных результатов программирования и тестирования.

При выполнении работы 8 «Сборка программных средств» производится сборка программных модулей и компонентов в программное средство и тестирование промежуточных и конечных результатов сборки. Выполняется оценка результатов сборки и тестирования.

При выполнении работы 9 «Квалификационные испытания программных средств» проводятся квалификационные испытания программного средства в моделируемой среде с моделируемыми исходными данными. Оцениваются результаты испытаний. При необходимости производится доработка программного продукта.

При выполнении работы 10 «Сборка системы» осуществляется сборка объектов программной и технической конфигурации, ручных операций, других подсистем в единую систему. Проводятся испытания собранной системы. Выполняется оценка собранной системы.

При выполнении работы 11 «Квалификационные испытания системы» проводятся квалификационные испытания собранной системы в моделируемой среде с моделируемыми исходными данными. По результатам испытаний выполняется оценка системы и при необходимости доработка программного продукта.

При выполнении работы 12 «Ввод в действие программных средств» программный продукт вводится в действие в среде эксплуатации.

При выполнении работы 13 «Обеспечение приемки программных средств» обеспечивается проведение заказчиком приемочных испытаний. Программный продукт поставляется заказчику.

В приведенной последовательности различают ***два вида работ***: системные и программные [8, 15].

Системные работы начинают и завершают процесс разработки. К ним относятся работы с номерами 2, 3, 10, 11.

Работы процесса разработки с 4-й (анализ требований к программным средствам) по 9-ю (квалификационные испытания программных средств) представляют собой *программные работы*. Они выполняются над ПС, выделенными из системы.

Таким образом, системные работы являются расширением совокупности программных работ.

Процесс эксплуатации определяет работы и задачи оператора. Данный процесс включает эксплуатацию программного продукта и поддержку пользователей в процессе эксплуатации.

Процесс сопровождения определяет работы и задачи персонала сопровождения и реализуется при модификациях программного продукта. Назначением процесса является изменение существующего программного продукта при сохранении его целостности. Процесс охватывает вопросы переносимости и снятия программного продукта с эксплуатации.

Вспомогательные процессы жизненного цикла – это процессы, являющиеся целенаправленными составными частями других процессов и предназначенные для обеспечения успешной реализации и качества выполнения

программного проекта. К *вспомогательным процессам* относится восемь процессов:

- документирование;
- управление конфигурацией;
- обеспечение качества;
- верификация;
- аттестация;
- совместный анализ;
- аудит;
- решение проблем.

Вспомогательные процессы вызываются другими процессами ЖЦ.

Процесс документирования предназначен для формализованного описания информации, созданной в процессе или работе жизненного цикла. Он включает планирование, проектирование, разработку, выпуск, редактирование, распространение и сопровождение документов по программному продукту.

Процесс управления конфигурацией предназначен для определения состояния (базовой линии) программных объектов в системе, управления их изменениями и выпуском.

Процесс обеспечения качества предназначен для обеспечения гарантий того, что программные продукты и процессы в жизненном цикле проекта соответствуют требованиям и планам.

Процесс верификации предназначен для определения соответствия функционирования программных продуктов требованиям и условиям, реализованным в предшествующих работах. В процессе разработки верификация связана с экспертизой результатов конкретной работы с целью определения их соответствия установленным на входе данной работы требованиям.

Процесс аттестации предназначен для определения полноты соответствия установленных требований, созданной системы или программного продукта их функциональному назначению. В процессе разработки аттестация связана с экспертизой промежуточного или конечного продукта в целях определения его соответствия потребностям пользователя (то есть исходным требованиям к проекту).

Процесс совместного анализа предназначен для оценки состояния и результатов работ по проекту. Данный процесс может выполняться двумя сторонами, участвующими в договоре, когда одна сторона (анализирующая) проверяет другую (анализируемую).

Процесс аудита предназначен для определения соответствия требованиям, планам и условиям договора. Данный процесс может выполняться двумя сторонами, участвующими в договоре, когда одна сторона (ревизующая) проверяет другую сторону (ревизуемую).

Процесс решения проблем предназначен для анализа и решения проблем (включая найденные несоответствия), которые обнаружены в ходе выполнения разработки, эксплуатации, сопровождения или других процессов.

Организационные процессы жизненного цикла – это процессы, предназначенные для создания в некоторой организации и совершенствования организационных структур, охватывающих процессы жизненного цикла и соответствующий персонал. К *организационным процессам* относится четыре процесса:

- управление;
- создание инфраструктуры;
- усовершенствование;
- обучение.

Процесс управления состоит из общих работ и задач, которые могут быть использованы стороной, управляющей соответствующим процессом. В данном процессе разрабатываются планы выполнения процессов ЖЦ ПС, осуществляется управление и текущий надзор за ходом процессов ЖЦ, обеспечивается управление оценкой планов, программных продуктов, работ и задач.

Процесс создания инфраструктуры предназначен для создания и сопровождения инфраструктуры, необходимой для любого другого процесса. *Инфраструктура* содержит технические и программные средства, инструментальные средства, методики, стандарты и условия для разработки, эксплуатации или сопровождения.

Процесс усовершенствования предназначен для создания, оценки, измерения, контроля и улучшения любого процесса жизненного цикла программных средств.

Процесс обучения является процессом обеспечения обучения персонала работам по заказу, поставке, разработке, эксплуатации или сопровождению программного проекта.

С понятием ЖЦ ПС и систем тесно связано понятие модели ЖЦ. ***Модель жизненного цикла*** – это совокупность процессов, работ и задач жизненного цикла, отражающая их взаимосвязь и последовательность выполнения.

Очевидно, что существуют тесные взаимосвязи между моделью ЖЦ, выбранной при реализации процесса разработки ПС, используемыми стратегиями и технологиями разработки ПС и уровнем качества разработанного программного продукта.

Резюме

Процессы ЖЦ ПС и систем регламентируются международным стандартом *ISO/IEC 12207*:*2008* и соответствующими национальными стандартами. В Республике Беларусь в настоящее время действует национальный стандарт *СТБ ИСО/МЭК 12207–2003*, являющийся аутентичным переводом стандарта *ISO/IEC 12207*:*1995*. В соответствии со стандартом *СТБ ИСО/МЭК 12207–2003* процессы ЖЦ ПС делятся на три группы: основные, вспомогательные, организационные. К основным процессам относятся процессы заказа, поставки, разработки, эксплуатации и сопровождения. Процесс разработки включает тринадцать работ.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что подразумевается под технологией разработки ПО?
2. Что является целью структурных методов проектирования ПС?
3. Дайте определение программного продукта.
4. Дайте определение системы.
5. Назовите базовый стандарт в области ЖЦ ПС и систем.
6. Определите понятие ЖЦ программного средства или системы.
7. Определите понятие модели ЖЦ программного средства или системы.
8. Определите иерархическую структуру ЖЦ ПС, регламентированную стандартом СТБ ИСО/МЭК 12207–2003.
9. На какие группы делятся процессы ЖЦ – В соответствии с положениями стандарта СТБ ИСО/МЭК 12207–2003?
10. Назовите основные стороны, участвующие в ЖЦ ПС и систем.
11. Перечислите и определите назначение процессов ЖЦ в каждой группе, регламентированной стандартом СТБ ИСО/МЭК 12207–2003.
12. Перечислите работы процесса разработки, регламентированные стандартом СТБ ИСО/МЭК 12207–2003, и опишите их содержание.
13. Назовите системные и программные работы процесса разработки, регламентированного стандартом СТБ ИСО/МЭК 12207–2003.

РАЗДЕЛ 2. СТРАТЕГИИ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ И СИСТЕМ И РЕАЛИЗУЮЩИЕ ИХ МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА

2.1. Стратегии разработки программных средств и систем

2.1.1. Базовые стратегии разработки программных средств и систем

На начальном этапе развития вычислительной техники ПС разрабатывались по принципу «кодирование – устранение ошибок» [33]. Модель такого процесса разработки ПС иллюстрирует рис. 2.1.

Очевидно, что *недостатками* данной модели являются:

- неструктурированность процесса разработки ПС;
- ориентация на индивидуальные знания и умения программиста;
- сложность управления и планирования проекта;
- большая длительность и стоимость разработки;
- низкое качество программных продуктов;
- высокий уровень рисков проекта.

Для устранения или сокращения вышеназванных недостатков к настоящему времени созданы и широко используются ***три базовые стратегии*** разработки ПО:

- каскадная;
- инкрементная;
- эволюционная.

Некоторые характеристики каскадной, инкрементной и эволюционной стратегий разработки ПС и предъявляемые к ним требования приведены в стандарте ***ГОСТ Р ИСО/МЭК ТО 15271–2002 – Информационная технология – Руководство по применению ГОСТ Р ИСО/МЭК 12207 (Процессы жизненного цикла программных средств)*** [8].

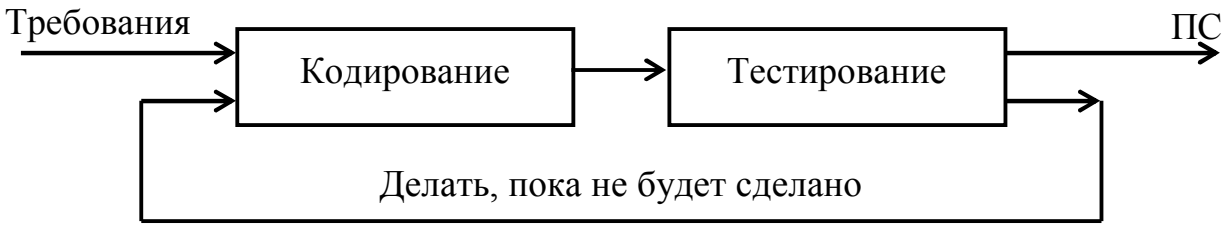


Рис. 2.1. Модель «Делать, пока не будет сделано»

Выбор той или иной стратегии определяется характеристиками:

- проекта;
- требований к продукту;
- команды разработчиков;
- команды пользователей.

Данные характеристики подробно рассмотрены в подразд. 3.1.

Каждая из стратегий разработки имеет как достоинства, так и недостатки, определяемые правильностью выбора данной стратегии для реализации конкретного проекта. Следует подчеркнуть, что одни и те же свойства стратегии могут проявлять себя как достоинства при правильном выборе стратегии и как ее недостатки, если стратегия выбрана неверно [33].

Три базовые стратегии могут быть реализованы с помощью различных моделей ЖЦ. В подразд. 2.2 – 2.5 приведены некоторые из наиболее известных и используемых моделей ЖЦ, большинство из которых рекомендовано к использованию Институтом качества программного обеспечения SQI (Software Quality Institute) [33] или стандартом *ГОСТ Р ИСО/МЭК 15271–2002* [8]. Модели, рекомендованные Институтом SQI, в данном пособии адаптированы с учетом положений стандарта *СТБ ИСО/МЭК 12207–2003* (см. подразд. 1.2).

Резюме

Существуют три базовые стратегии разработки ПС: каскадная, инкрементная, эволюционная. Данные стратегии могут быть реализованы с помощью различных моделей ЖЦ.

2.1.2. Каскадная стратегия разработки программных средств и систем

Каскадная стратегия представляет собой однократный проход этапов разработки. Данная стратегия основана на полном определении всех требований к разрабатываемому программному средству или системе в начале процесса разработки. Каждый этап разработки начинается после завершения предыдущего этапа. Возврат к уже выполненным этапам не предусматривается. Промежуточные продукты разработки в качестве версии программного средства (системы) не распространяются.

Представителями моделей, реализующих каскадную стратегию, являются каскадная и V-образная модели.

Основными достоинствами каскадной стратегии, проявляемыми при разработке соответствующего ей проекта, являются:

1) стабильность требований в течение ЖЦ разработки;

2) необходимость только одного прохода этапов разработки, что обеспечивает простоту применения стратегии;

3) простота планирования, контроля и управления проектом;

4) доступность для понимания заказчиками.

К *основным недостаткам каскадной стратегии*, проявляемым при ее использовании в проекте, ей не соответствующем, следует отнести:

1) сложность полного формулирования требований в начале процесса разработки и невозможность их динамического изменения на протяжении ЖЦ;

2) линейность структуры процесса разработки; разрабатываемые ПС или системы обычно слишком велики и сложны, чтобы все работы по их созданию выполнять однократно; в результате возврат к предыдущим шагам для решения возникающих проблем приводит к увеличению финансовых затрат и нарушению графика работ;

3) непригодность промежуточных продуктов для использования;

4) недостаточное участие пользователя в процессе разработки ПС – только в самом начале (при разработке требований) и в конце (во время приемочных испытаний); это приводит к невозможности предварительной оценки пользователем качества программного средства или системы.

Области применения каскадной стратегии определяются ее достоинствами и ограничены ее недостатками. Использование данной стратегии наиболее эффективно в следующих случаях [33]:

1) при разработке проектов с четкими, неизменяемыми в течение ЖЦ требованиями и понятной реализацией;

2) при разработке проектов невысокой сложности, например:
   - создание программного средства или системы такого же типа, как уже разрабатывались разработчиками;
   - создание новой версии уже существующего программного средства или системы;
   - перенос уже существующего продукта на новую платформу;

3) при выполнении больших проектов в качестве составной части моделей ЖЦ, реализующих другие стратегии разработки (см., например, модели, приведенные на рис. 2.5 и рис. 2.12).

В подразд. 2.2 рассмотрены модели ЖЦ, реализующие каскадную стратегию разработки ПС и систем.

Резюме

Каскадная стратегия представляет собой однократный проход этапов разработки. Данная стратегия основана на полном определении всех требований к

программному средству или системе в начале процесса разработки. Возврат к уже выполненным этапам не предусматривается. Промежуточные результаты в качестве версии программного средства (системы) не распространяются. Каскадная стратегия имеет достоинства и недостатки, определяемые правильностью выбора данной стратегии по отношению к конкретному проекту.

2.1.3. Инкрементная стратегия разработки программных средств и систем

Инкрементная стратегия представляет собой многократный проход этапов разработки с запланированным улучшением результата.

Данная стратегия основана на полном определении всех требований к разрабатываемому программному средству (системе) в начале процесса разработки. Однако полный набор требований реализуется постепенно в соответствии с планом в последовательных циклах разработки.

Результат каждого цикла называется ***инкрементом***.

Первый инкремент реализует базовые функции программного средства. В последующих инкрементах функции программного средства постепенно расширяются, пока не будет реализован весь набор требований. Различия между инкрементами соседних циклов в ходе разработки постепенно уменьшаются.

Результат каждого цикла разработки может распространяться в качестве очередной поставляемой версии программного средства или системы.

Особенностью инкрементной стратегии является большое количество циклов разработки при незначительной продолжительности цикла и небольших различиях между инкрементами соседних циклов. Например, данная стратегия разработки ПС и систем используется в компании Microsoft. Здесь на каждую версию программного средства разрабатывается около тысячи инкрементов. Период разработки инкремента составляет одни сутки (например, днем инкремент разрабатывается, ночью тестируется) [17]. В ряде организаций используется недельный период разработки инкремента (чаще всего пять дней – разработка, два дня – тестирование).

Инкрементная стратегия обычно основана на объединении элементов каскадной модели и прототипирования. При этом использование прототипирования позволяет существенно сократить продолжительность разработки каждого инкремента и всего проекта в целом.

Под ***прототипом*** понимается легко поддающаяся модификации и расширению рабочая модель разрабатываемого программного средства (или системы), позволяющая пользователю получить представление о его ключевых свойствах до полной реализации [33].

Современной реализацией инкрементной стратегии является экстремальное программирование [17, 42]. Различные модификации моделей, реализующих инкрементную стратегию, рассмотрены в подразд. 2.4.

Основными достоинствами инкрементной стратегии, проявляемыми при разработке соответствующего ей проекта, являются:

1) возможность получения функционального продукта после реализации каждого инкремента;

2) короткая продолжительность создания инкремента; это приводит к сокращению сроков начальной поставки, позволяет снизить затраты на первоначальную и последующие поставки программного продукта;

3) предотвращение реализации громоздких спецификаций требований; стабильность требований во время создания определенного инкремента; возможность учета изменившихся требований;

4) снижение рисков по сравнению с каскадной стратегией;

5) включение в процесс пользователей, что позволяет оценить функциональные возможности продукта на более ранних этапах разработки и в конечном итоге приводит к повышению качества программного продукта, снижению затрат и времени на его разработку.

К *основным недостаткам инкрементной стратегии*, проявляющимся в результате ее несоответствующего применения, следует отнести:

1) необходимость полного функционального определения системы или программного средства в начале ЖЦ для обеспечения планирования инкрементов и управления проектом;

2) возможность текущего изменения требований к системе или программному средству, которые уже реализованы в предыдущих инкрементах;

3) сложность планирования и распределения работ;

4) проявление человеческого фактора, связанного с тенденцией к оттягиванию решения трудных проблем на поздние инкременты, что может нарушить график работ или снизить качество программного продукта.

Области применения инкрементной стратегии определяются ее достоинствами и ограничены ее недостатками. Использование данной стратегии наиболее эффективно в следующих случаях [33]:

1) при разработке проектов, в которых большинство требований можно сформулировать заранее, но часть из них могут быть уточнены через определенный период времени;

2) при разработке сложных проектов с заранее сформулированными требованиями; для них разработка системы или программного средства за один цикл связана с большими трудностями;

3) при необходимости быстро поставить на рынок продукт, имеющий базовые функциональные свойства;

4) при разработке проектов с низкой или средней степенью рисков;

5) при выполнении проекта с применением новых технологий.

Резюме

Инкрементная стратегия представляет собой многократный проход этапов разработки с запланированным улучшением результата. Данная стратегия

основана на полном определении всех требований к разрабатываемому программному средству или системе в начале процесса разработки. Однако полный набор требований реализуется постепенно в соответствии с планом в последовательных циклах разработки. При инкрементной стратегии часто используется прототипирование. Инкрементная стратегия имеет достоинства и недостатки, определяемые правильностью выбора данной стратегии по отношению к конкретному проекту.

2.1.4. Эволюционная стратегия разработки программных средств и систем

Эволюционная стратегия представляет собой многократный проход этапов разработки. Данная стратегия основана на частичном определении требований к разрабатываемому программному средству или системе в начале процесса разработки. Требования постепенно уточняются в последовательных циклах разработки. Результат каждого цикла разработки обычно представляет собой очередную поставляемую версию программного средства или системы.

Следует отметить, что в общем случае для эволюционной стратегии характерно существенно меньшее количество циклов разработки при большей продолжительности цикла по сравнению с инкрементной стратегией. При этом результат каждого цикла разработки (очередная версия программного средства или системы) гораздо сильнее отличается от результата предыдущего цикла.

Как и при инкрементной стратегии, при реализации эволюционной стратегии зачастую используется прототипирование.

В данном случае основной целью прототипирования является обеспечение полного понимания требований. Оно позволяет итеративно уточнять требования к продукту при достижении предельно высокой производительности разработки проекта и одновременном снижении затрат. Использование прототипирования наиболее эффективно в тех случаях, когда в проекте применяются новые концепции или новые технологии, так как в этих случаях достаточно сложно полностью и корректно разработать детальные технические требования к системе или программному средству на ранних стадиях цикла разработки.

Для итеративного уточнения требований при применении прототипирования в цикле разработки должен участвовать заказчик.

Представителями моделей, реализующих эволюционную стратегию, являются, например, спиральные модели (см. подразд. 2.5).

Основными достоинствами эволюционной стратегии, проявляемыми при разработке соответствующего ей проекта, являются:

1) возможность уточнения и внесения новых требований в процессе разработки;
2) пригодность промежуточного продукта для использования;
3) возможность управления рисками;

4) обеспечение широкого участия пользователя в проекте, начиная с ранних этапов, что минимизирует возможность разногласий между заказчиками и разработчиками и обеспечивает создание продукта высокого качества;

5) реализация преимуществ каскадной и инкрементной стратегий.

К *недостаткам эволюционной стратегии*, проявляемым при ее несоответствующем выборе, следует отнести:

1) неизвестность точного количества необходимых итераций и сложность определения критериев для продолжения процесса разработки на следующей итерации; это может вызвать задержку реализации конечной версии системы или программного средства;

2) сложность планирования и управления проектом;

3) необходимость активного участия пользователей в проекте, что реально не всегда осуществимо;

4) необходимость в мощных инструментальных средствах и методах прототипирования;

5) возможность отодвигания решения трудных проблем на последующие циклы, что может привести к несоответствию полученных продуктов требованиям заказчиков.

Очевидно, что ряд недостатков эволюционной стратегии (см. недостатки 3 – 5) характерны и для инкрементной стратегии.

Области применения эволюционной стратегии определяются ее достоинствами и ограничены ее недостатками. Использование данной стратегии наиболее эффективно в следующих случаях [33]:

1) при разработке проектов, для которых требования слишком сложны, неизвестны заранее, непостоянны или требуют уточнения;

2) при разработке сложных проектов, в том числе:
   - больших долгосрочных проектов;
   - проектов по созданию новых, не имеющих аналогов ПС или систем;
   - проектов со средней и высокой степенью рисков;
   - проектов, для которых нужна проверка концепции, демонстрация технической осуществимости или промежуточных продуктов;

3) при разработке проектов, использующих новые технологии.

В подразд. 2.5 рассмотрены модели ЖЦ, реализующие эволюционную стратегию разработки ПС и систем.

Резюме

Эволюционная стратегия представляет собой многократный проход этапов разработки. Данная стратегия основана на частичном определении требований к разрабатываемому программному средству или системе в начале процесса разработки. Требования постепенно уточняются в последовательных циклах разработки. Результат каждого цикла разработки обычно представляет собой очередную поставляемую версию программного средства или системы. При эволюционной стратегии часто используется прототипирование. Эволюци-

онная стратегия имеет достоинства и недостатки, определяемые правильностью выбора данной стратегии по отношению к конкретному проекту.

2.2. Модели жизненного цикла, реализующие каскадную стратегию разработки программных средств и систем

2.2.1. Общие сведения о каскадных моделях

Моделью ЖЦ, пришедшей на смену принципу разработки ПС «кодирование – устранение ошибок», явилась классическая каскадная модель. Первые публикации о ней появились в 1970 г. Данная модель впервые формализовала структуру этапов разработки ПС. Она поддерживает *каскадную стратегию однократного прохода этапов разработки ПС* и базируется на полном формулировании требований в начале ЖЦ. К их уточнению или изменению на следующих шагах ЖЦ возврата не происходит.

Процесс разработки ПС и систем реализуется с помощью упорядоченной последовательности независимых шагов. Модель предусматривает, что каждый последующий шаг начинается после полного завершения выполнения предыдущего шага.

Существуют различные варианты каскадной модели ЖЦ.

Резюме

Каскадные модели реализуют каскадную стратегию однократного прохода этапов разработки ПС. Каждый последующий шаг разработки начинается после полного завершения выполнения предыдущего шага.

2.2.2. Классическая каскадная модель

В общем случае каскадные модели могут представлять процесс разработки с различной степенью детализации. При этом в качестве шага модели могут быть приняты некоторые фазы, этапы или работы процесса разработки.

Рис. 2.2 представляет вариант классической каскадной модели, ориентированный на работы процесса разработки, структура которого определена в *СТБ ИСО/МЭК 12207–2003* (см. подразд. 1.2). Для данного варианта модели понятие шага разработки программного средства совпадает с понятием одной или нескольких работ процесса разработки вышеназванного стандарта.

С учетом внешнего вида каскадной модели (см. рис. 2.2) ее называют также *водопадной моделью*.

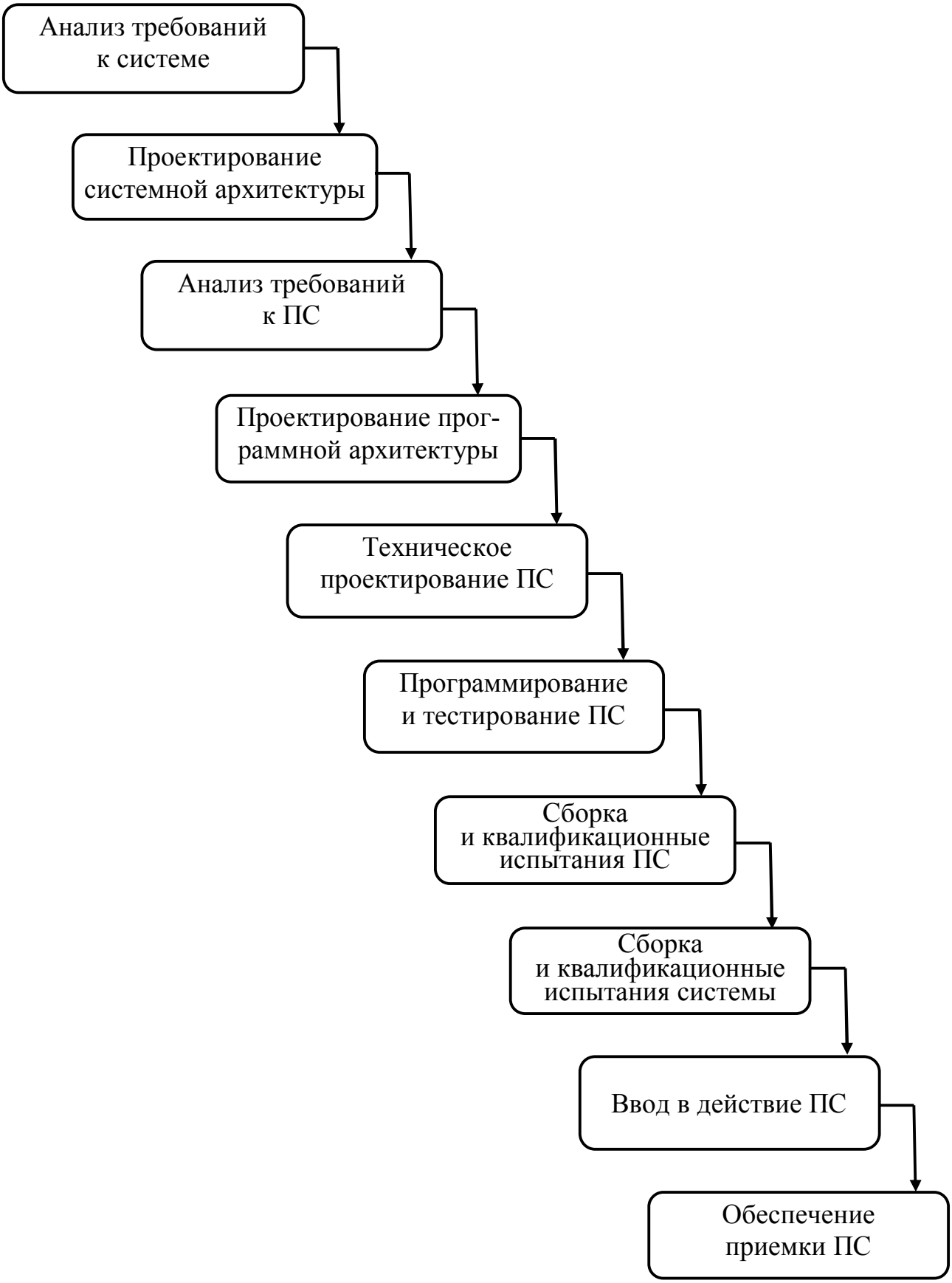


Рис. 2.2. Классическая каскадная модель,
ориентированная на работы процесса разработки СТБ ИСО/МЭК 12207–2003

На всех шагах модели по необходимости выполняются вспомогательные и организационные процессы, например, управление проектом, обеспечение качества, верификация, аттестация, управление конфигурацией, документирование (см. подразд. 1.2).

Результатами завершения шагов модели являются промежуточные продукты разработки, которые не могут изменяться на последующих шагах и не могут сдаваться заказчику в качестве версий программного средства.

Классическая каскадная модель обладает всеми достоинствами и недостатками, характерными для каскадной стратегии разработки (см. п. 2.1.2).

Резюме

В каскадных моделях возможна различная степень детализации процесса разработки ПС. Рассмотренный вариант классической каскадной модели базируется на работах процесса разработки, определенного в стандарте *СТБ ИСО/МЭК 12207–2003*. В каскадной модели продукты промежуточных шагов разработки не могут изменяться на последующих шагах и не могут сдаваться заказчику.

2.2.3. Каскадная модель с обратными связями

Реализовать классическую каскадную модель ЖЦ в чистом виде затруднительно ввиду сложности разработки ПС без возвратов к предыдущим шагам и изменения их результатов для устранения возникающих проблем. В этой связи разработаны варианты каскадной модели с обратными связями между ее отдельными шагами.

Рис. 2.3 отражает организацию обратных связей между соседними шагами модели, ориентированной на работы стандарта *СТБ ИСО/МЭК 12207–2003*.

Возможна организация обратных связей между любыми шагами каскадной модели. Рис. 2.4 схематично иллюстрирует фрагмент каскадной модели с возможностью возврата от некоторого шага процесса разработки к различным шагам, выполненным ранее.

Достоинством каскадной модели с обратными связями по сравнению с классической каскадной моделью является возможность исправления продуктов предыдущих шагов процесса разработки.

К *недостаткам* каскадной модели с обратными связями по сравнению с классической каскадной моделью относятся сложности планирования и финансирования проекта, достаточно высокий риск нарушения графика разработки.

Резюме

Возможна организация обратных связей между любыми шагами каскадной модели. Это позволяет выполнять исправление промежуточных продуктов предыдущих шагов процесса разработки. При этом возрастает сложность пла нирования и финансирования проекта, достаточно высок риск нарушения графика разработки.

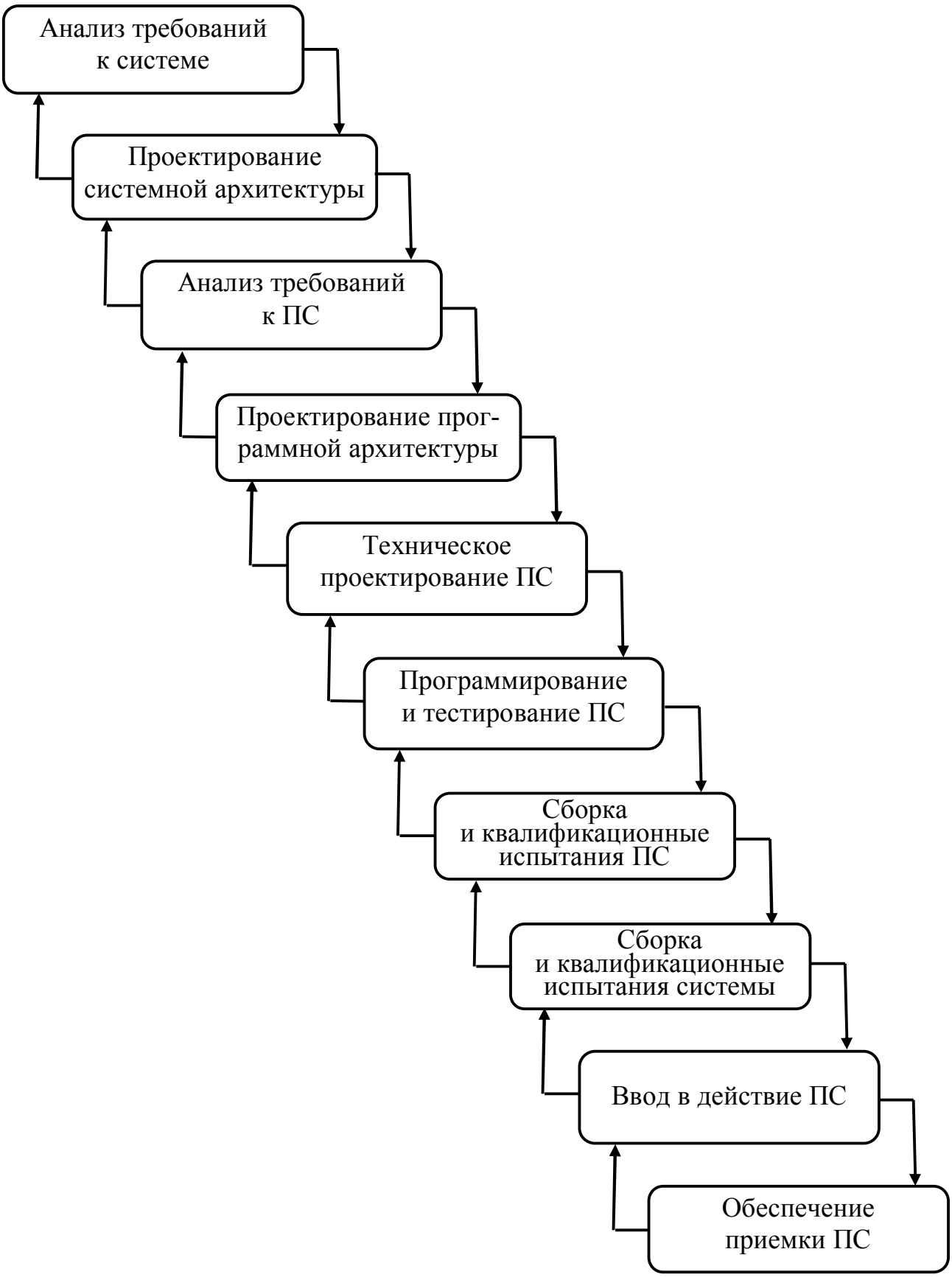


Рис. 2.3. Каскадная модель с обратными связями,
ориентированная на работы процесса разработки СТБ ИСО/МЭК 12207–2003

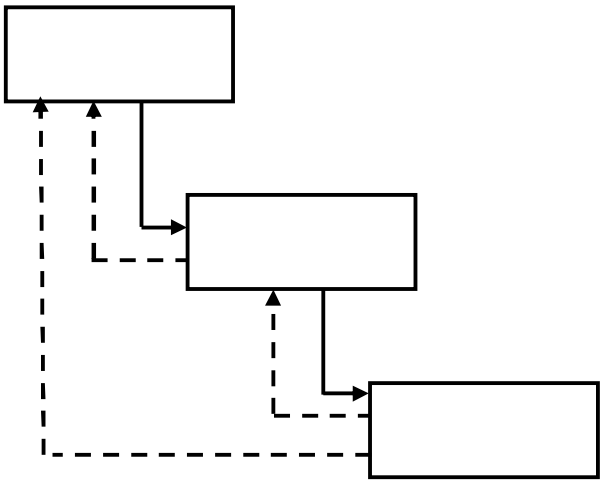


Рис. 2.4. Структура фрагмента каскадной модели
с возможностью возврата к различным шагам

2.2.4. Вариант каскадной модели по ГОСТ Р ИСО/МЭК ТО 15271–2002

В *ГОСТ Р ИСО/МЭК ТО 15271–2002* [8] приведен вариант каскадной модели, ориентированный на коллективную разработку систем (рис. 2.5). В данном варианте, как и в рассмотренных ранее вариантах модели, понятие шага разработки совпадает с понятием работы процесса разработки, определенного в стандарте *СТБ ИСО/МЭК 12207–2003* (см. подразд. 1.2).

Приведенный вариант базируется на возможности параллельной разработки ПС, входящих в состав системы, различными командами разработчиков, а также выбора программных объектов из существующих или разработанных ранее.

Данный вариант учитывает также необходимость разработки или выбора технических компонентов системы.

Резюме

На основе каскадной модели возможна организация параллельной разработки ПС, входящих в состав системы, различными командами разработчиков.

2.2.5. V-образная модель

Основное назначение V-образной модели – обеспечение планирования тестирования (испытаний) системы и программного средства на ранних стадиях проекта.

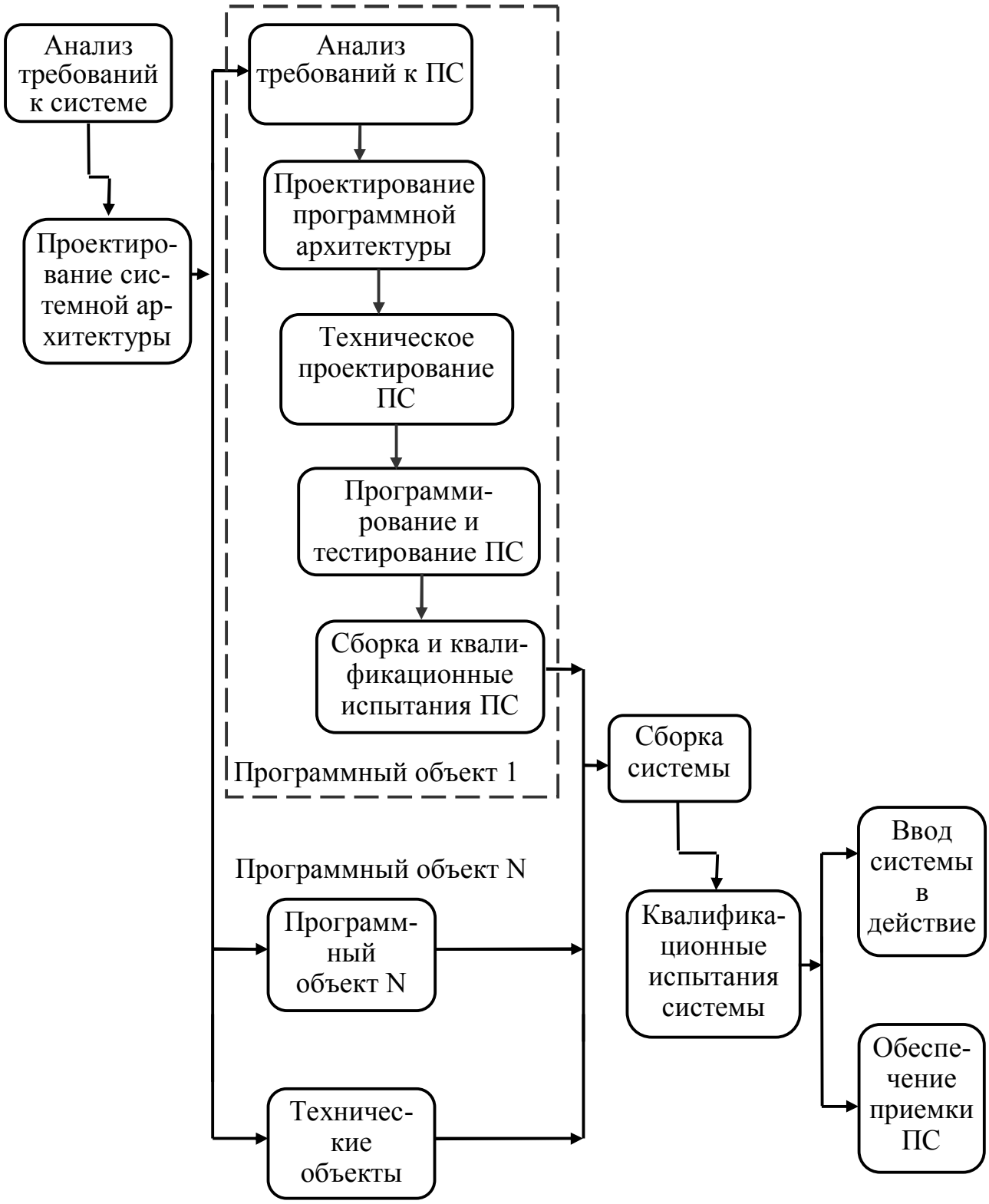


Рис. 2.5. Вариант каскадной модели
по ГОСТ Р ИСО/МЭК ТО 15271–2002

V-образная модель представляет собой разновидность каскадной модели. Данная модель поддерживает каскадную стратегию однократного выполнения этапов процесса разработки ПС или систем и базируется на предварительном полном формировании требований. В классической V-образной модели каждый шаг начинается после завершения предыдущего шага.

Отличием V-образной модели от каскадной является то, что в ней выделены связи между шагами, предшествующими программированию, и соответствующими видами тестирования и испытаний.

Рис. 2.6 иллюстрирует вариант V-образной модели, адаптированный к работам процесса разработки, определенного в *СТБ ИСО/МЭК 12207–2003* (см. подразд. 1.2). Данная модель состоит из последовательных этапов.

Этапы *подготовки процесса разработки, анализа требований к системе* и *программирования и тестирования программных средств* соответствуют работам 1, 2 и 7 процесса разработки (см. подразд. 1.2).

Кроме того, на этапе *анализа требований к системе* разрабатывается план ввода в действие и обеспечения приемки системы.

На этапе проектирования системы выполняются работы 3 (проектирование системной архитектуры) и 4 (анализ требований к программным средствам) процесса разработки. На данном этапе, помимо этого, разрабатываются планы сборки и квалификационных испытаний системы.

На этапе проектирования программных средств выполняются работы 5 (проектирование программной архитектуры) и 6 (техническое проектирование программных средств). Одновременно составляются план сборки ПС и план квалификационных испытаний ПС.

На этапе сборки и квалификационных испытаний программных средств выполняются работы 8 (сборка программных средств) и 9 (квалификационные испытания программных средств). Данный этап выполняется в соответствии с планами, разработанными на этапе проектирования программных средств, и имеет одной из основных целей подтверждение результатов названного этапа.

На этапе сборки и квалификационных испытаний системы выполняются работы 10 (сборка системы) и 11 (квалификационные испытания системы).

Данный этап выполняется в соответствии с планами, разработанными на этапе проектирования системы. Подтверждение результатов последнего является одной из основных целей данного этапа.

На этапе ввода в действие и обеспечения приемки осуществляются работы 12 (ввод в действие программных средств), 13 (обеспечение приемки программных средств), а также при необходимости работы по вводу в действие и обеспечению приемки всей системы в целом. Этот этап выполняется в соответствии с планом, разработанным на этапе анализа требований к системе. На данном этапе выполняются приемочные испытания, целью которых является проверка пользователем соответствия системы исходным требованиям.

Связи между деятельностью по разработке планов испытаний и тестирования и деятельностью по подтверждению результатов соответствующих этапов на рис. 2.6 обозначены пунктирными линиями.

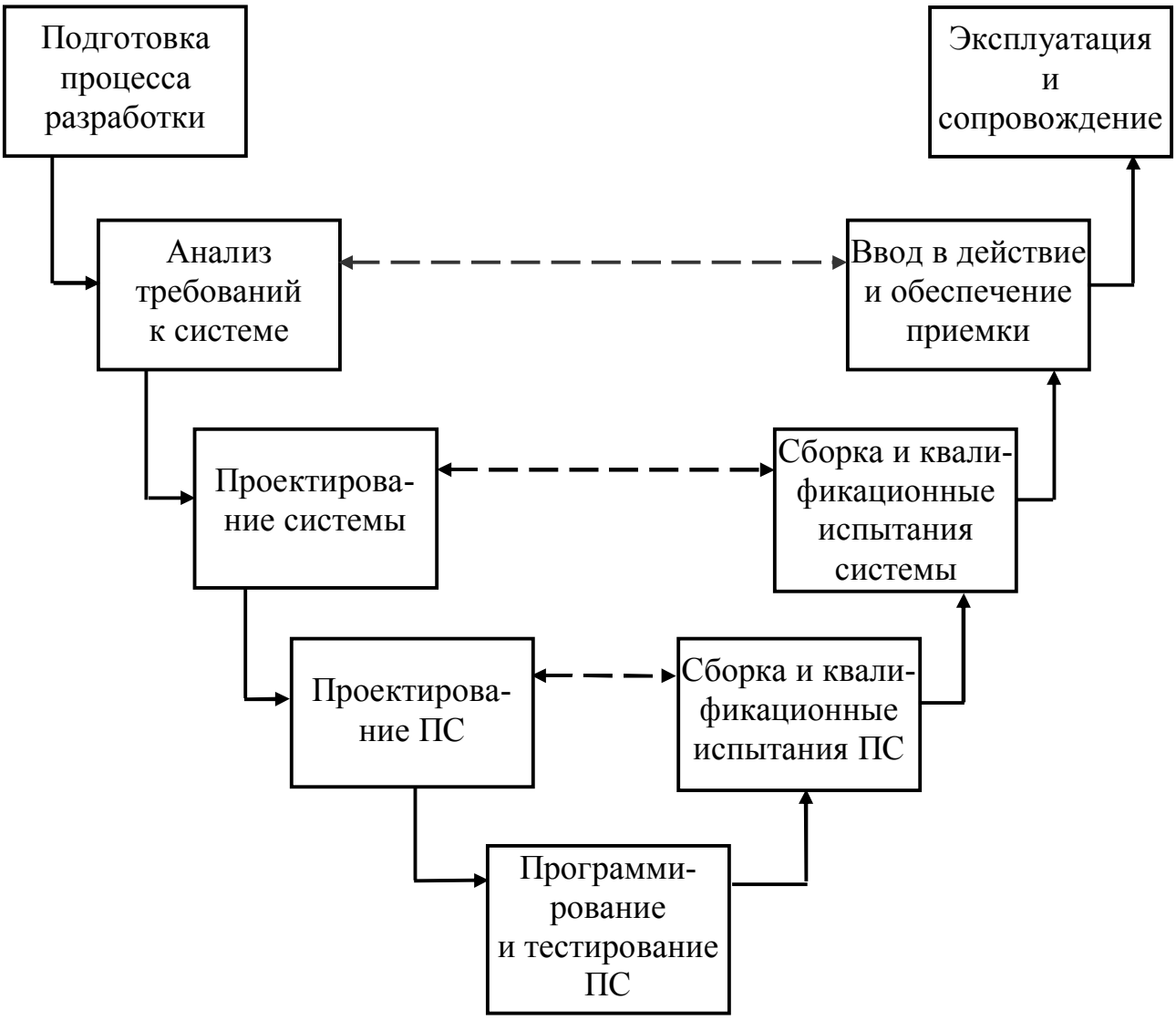


Рис. 2.6. V-образная модель жизненного цикла

Как и в каскадной модели, на всех этапах V-образной модели выполняются необходимые вспомогательные процессы ЖЦ ПС (см. подразд. 1.2), например, управление проектом, оценка качества, верификация, аттестация, управление конфигурацией, документирование.

С целью сокращения недостатков каскадной стратегии разработан ряд модификаций V-образной модели, обусловленных разновидностью обратных связей, которые обеспечивают возможность изменения результатов предыдущих этапов разработки.

Например, рис. 2.7 иллюстрирует V-образную модель с организацией обратных связей между соседними этапами процесса разработки.

Поскольку V-образная модель поддерживает каскадную стратегию разработки ПС и систем, то она обладает всеми достоинствами данной стратегии.

Кроме того, при подходящем использовании V-образная модель обладает следующими дополнительными ***достоинствами***:

1) планирование тестирования и испытаний на ранних стадиях разработки системы и программного средства;

2) упрощение аттестации и верификации промежуточных результатов разработки;

3) упрощение управления и контроля хода процесса разработки.

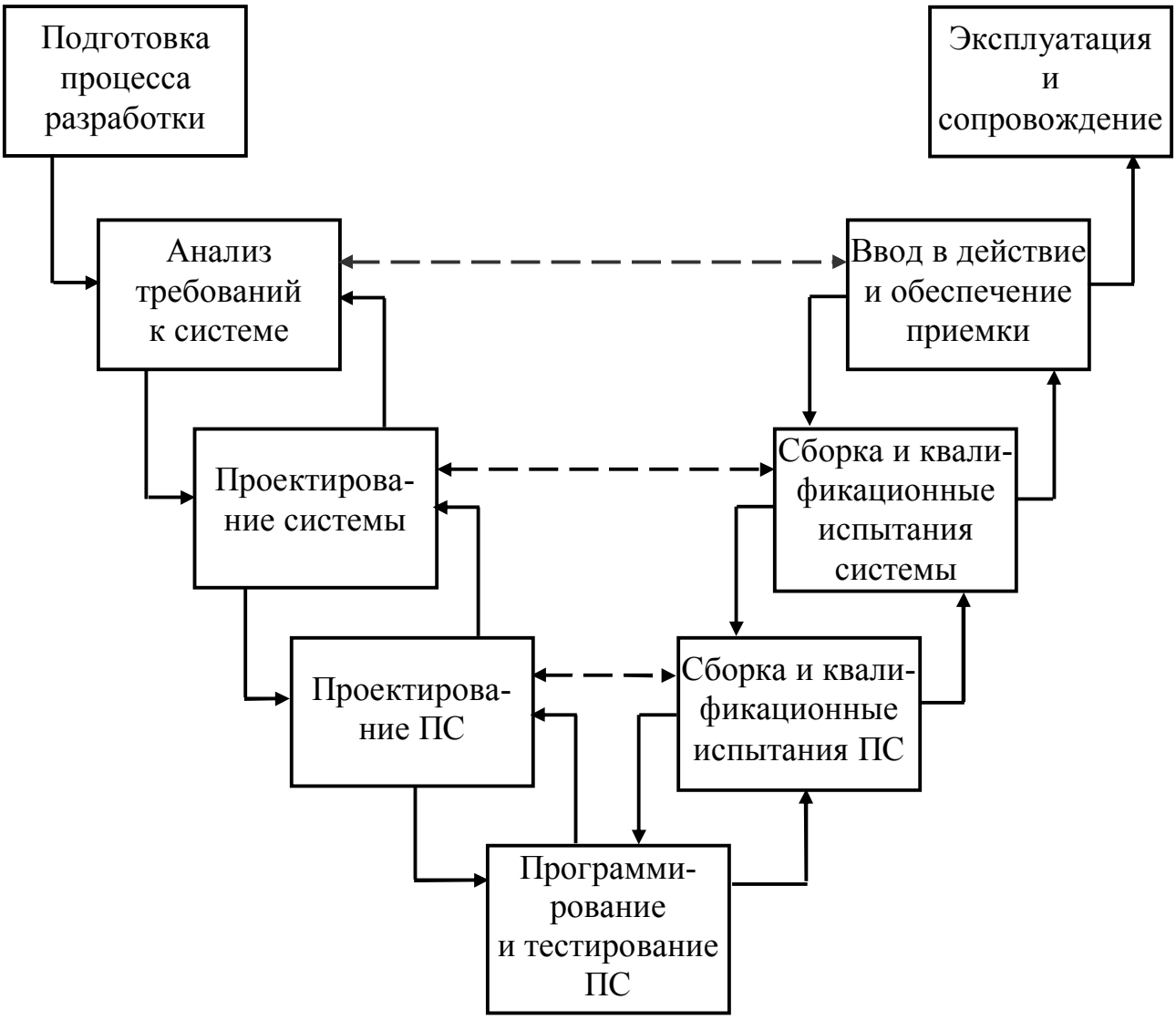


Рис. 2.7. V-образная модель жизненного цикла
с организацией обратных связей между соседними этапами

При использовании V-образной модели для несоответствующего ей проекта выявляются следующие ее ***недостатки***:

1) поздние сроки тестирования требований в жизненном цикле, что оказывает существенное влияние на график выполнения проекта при необходимости изменения требований;

2) отсутствие, как и в остальных каскадных моделях, действий, направленных на анализ рисков.

Резюме

V-образная модель поддерживает каскадную стратегию разработки. В ней выделены связи между шагами, предшествующими программированию, и соответствующими видами тестирования и испытаний. Данные связи увязывают деятельность по разработке планов испытаний и тестирования с деятельностью по подтверждению результатов соответствующих этапов. В V-образной модели возможна организация обратных связей между этапами модели.

2.3. Модели быстрой разработки приложений

Модель быстрой разработки приложений (Rapid Application Development, RAD) появилась в 80-е гг. XX в. в связи с бурным развитием мощных технологий и инструментальных средств разработки программных продуктов. Данная модель, исходя из особенностей ее реализации и целей ее использования, может поддерживать как инкрементную, так и эволюционную стратегию разработки систем или ПС. Как правило, RAD-модели используются в составе другой модели для ускорения цикла разработки прототипа (версии) системы или программного средства (см. пп. 2.5.3, 2.5.4). При невысокой сложности проектов RAD-модели могут применяться как независимые модели.

Как было отмечено в подразд. 2.1, модели ЖЦ, реализующие инкрементную или эволюционную стратегию разработки, широко применяют понятие быстрого прототипирования. RAD-модель представляет собой модель, на использовании которой прототипирование базируется.

Основу RAD-модели составляет использование мощных *инструментальных средств разработки*. Такими средствами являются языки четвертого поколения *4GL* (Fourth Generation Language – язык программирования четвертого поколения) и *CASE-средства* (Computer Aided Software Engineering – компьютерная поддержка проектирования ПО), благодаря наличию в них сред визуальной разработки и кодогенераторов (подробно понятия CASE-средств и CASE-технологий рассмотрены в подразд. 5.1, 7.4, 7.5). Поэтому в процессе быстрой разработки приложений основное внимание уделяется не программированию и тестированию, а анализу требований и проектированию.

Использование инструментальных средств позволяет задействовать пользователя, а следовательно, дать оценку продукту на всех этапах его разработки.

Характерной чертой RAD-модели является короткое время перехода от анализа требований до создания полной системы или программного средства. Разработка прототипа, как правило, ограничивается четко определенным периодом времени (*временным блоком*; обычно 60 дней) [30, 33]. При полностью

определенных требованиях и не очень сложной проектной области использование RAD-модели позволяет за временной блок создать полную функциональную систему.

Резюме

Модель быстрой разработки приложений может поддерживать как инкрементную, так и эволюционную стратегию разработки систем или ПС. Обычно RAD-модель применяется в составе другой модели для ускорения цикла разработки прототипа системы или программного средства. Основу RAD-модели составляет использование мощных инструментальных средств разработки. Разработка прототипа ограничивается временным блоком.

2.3.1. Базовая RAD-модель

Рис. 2.8 представляет вариант базовой модели быстрой разработки приложений. Данный вариант отражает зависимость трудозатрат по разработке и участия пользователя от этапов RAD-модели [33].

На этапе анализа требований к системе совместно с заказчиком (пользователем) выполняется анализ предметной области, сбор и разработка требований к системе (работа 2 – го процесса разработки, определенного в стандарте *СТБ ИСО/МЭК 12207–2003*, см. подразд. 1.2). При этом используются соответствующие инструментальные средства (см. подразд. 7.4, 7.5).

На этапе проектирования выполняются работы 3 – 5 процесса разработки (проектирование системной архитектуры, анализ требований к программным средствам, проектирование программной архитектуры). При этом используются инструментальные средства, обеспечивающие сбор и уточнение пользовательской информации, визуальные анализ и проектирование.

На этапе конструирования выполняются работы 6 – 11 процесса разработки (техническое проектирование программных средств, программирование и тестирование программных средств, сборка и квалификационные испытания программных средств и системы). При этом используются соответствующие инструментальные средства проектирования, автоматической кодогенерации и тестирования.

На этапе ввода в действие и обеспечения приемки выполняются установка системы или программного средства, приемочные испытания и обучение пользователей (работы 12, 13 процесса разработки).

Данный вариант RAD-модели является обобщенным и не учитывает специфику выполнения отдельных этапов процесса разработки. Этот вариант может использоваться в отдельных итерациях некоторой эволюционной модели и в качестве независимой модели в небольших проектах.

Резюме

Базовая RAD-модель отражает укрупненные этапы процесса разработки (анализ требований, проектирование, конструирование, ввод в действие и обес-

печение приемки). Наибольшие трудозатраты разработчика и наименьшее участие пользователя приходятся на этап конструирования.

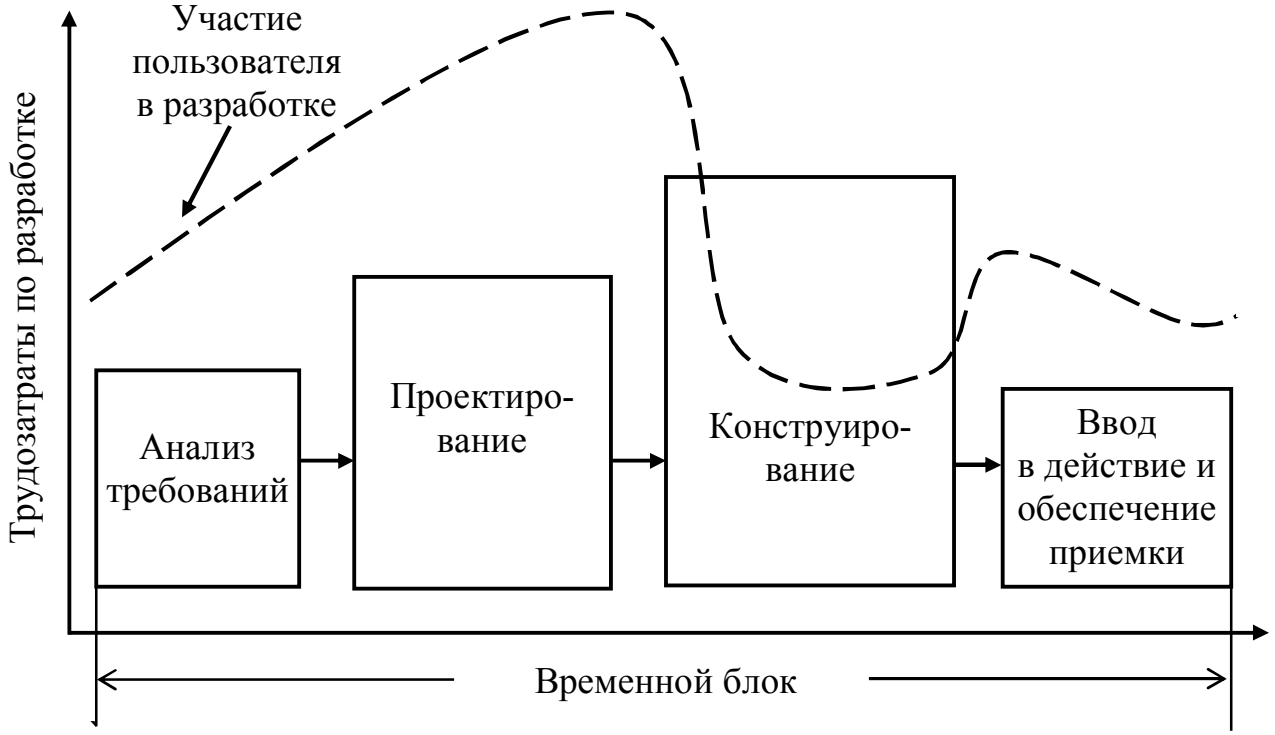


Рис. 2.8. Базовая модель быстрой разработки приложений

2.3.2. RAD-модель, основанная на моделировании предметной области

В RAD-модели для ускорения процесса разработки обычно используются различные виды моделирований предметной области, например, функциональное моделирование, моделирование данных, моделирование процесса (поведения). С этой целью широко используются CASE-средства (см. подразд. 7.4, 7.5). Затем на основе созданных моделей выполняется автоматическая кодогенерация программного средства. С учетом этого разработаны варианты RAD-модели, базирующиеся на моделировании предметной области. Один из вариантов приведен на рис. 2.9.

В данной модели выделяется пять этапов.

На *этапе функционального моделирования* определяются и анализируются функции и информационные потоки предметной области. В подразд. 5.2 и 5.3 рассмотрены методологии функционального моделирования IDEF0 и DFD.

На этапе моделирования данных на базе информационных потоков, определенных на предыдущем этапе, разрабатывается информационная модель предметной области. В подразд. 5.4 рассмотрена методология информационного моделирования IDEF1X.

На этапе моделирования поведения выполняется динамическое (поведенческое) моделирование предметной области. Одной из известных методологий динамического моделирования является методология IDEF3.

На этапе автоматической кодогенерации на основе информационной, функциональной и поведенческой моделей выполняется генерация текстов программных компонентов. При этом используются языки программирования четвертого поколения (Fourth Generation Language – 4GL) и CASE-средства. Широко применяются повторно используемые программные компоненты.

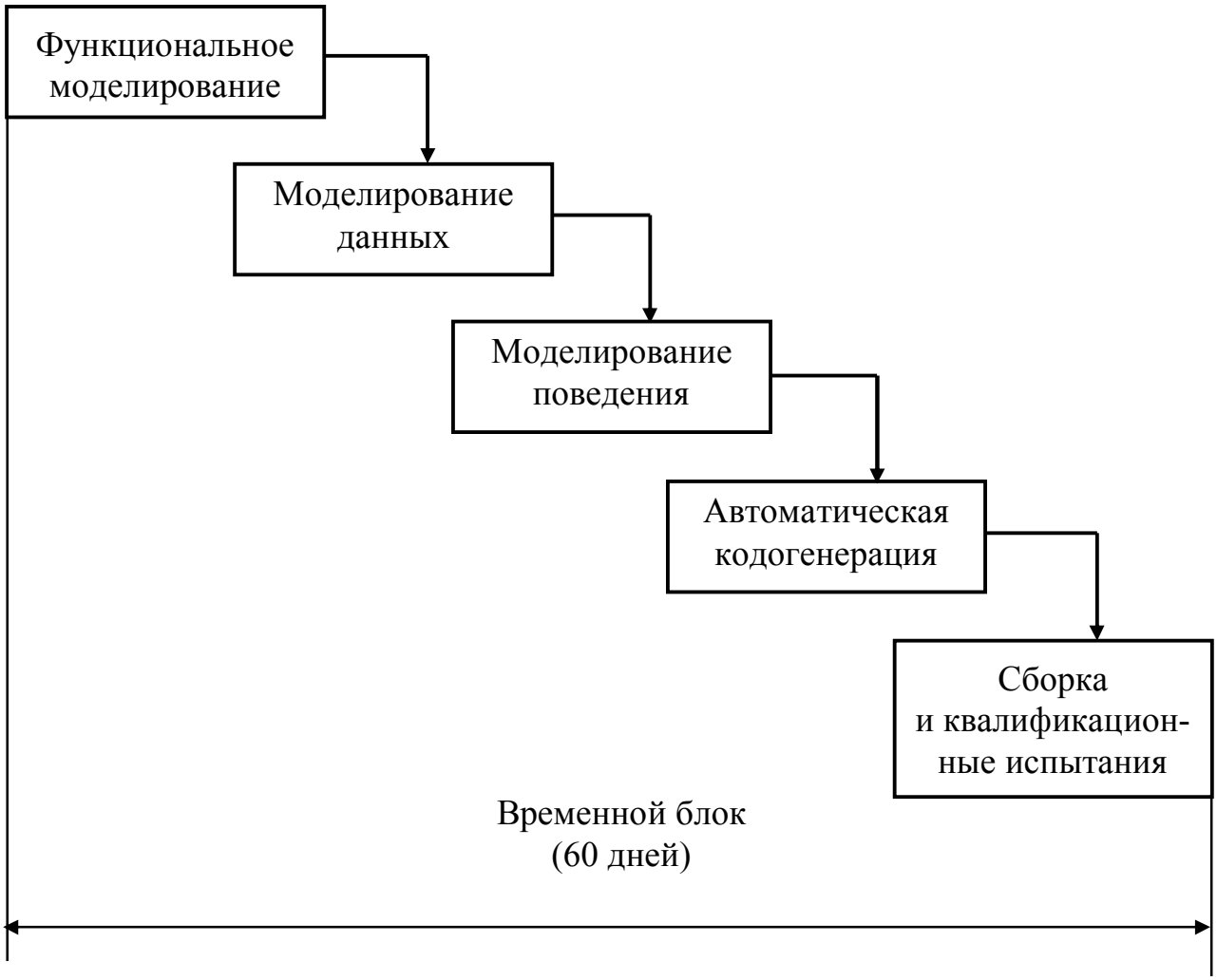


Рис. 2.9. Вариант модели быстрой разработки приложений, основанной на моделировании предметной области

На этапе сборки и квалификационных испытаний выполняется сборка и испытания результирующей системы, подсистемы или программного средства.

При независимом использовании рассмотренный вариант RAD-модели поддерживает стратегию однократного прохода этапов процесса разработки, то есть фактически каскадную стратегию. В этом случае он может применяться для проектов невысокой сложности.

В проектах более высокой сложности данный вариант может встраиваться в инкрементные и эволюционные модели как основа реализации отдельных итераций.

Резюме

В RAD-модели, основанной на моделировании предметной области, этапы анализа требований, проектирования и программирования (работы 2 – 7 процесса разработки, см. подразд. 1.2) заменены этапами функционального моделирования, моделирования данных, моделирования поведения разрабатываемого программного средства или системы и автоматической кодогенерации. Это существенно ускоряет процесс разработки.

2.3.3. RAD-модель параллельной разработки приложений

При разработке сложных проектов с организацией коллективной разработки ПС могут использоваться различные варианты RAD-модели [30]. Один из вариантов представлен на рис. 2.10.

Данный вариант модели поддерживает параллельную разработку программных компонентов, реализующих базовые функции программного средства различными группами разработчиков, с дальнейшей интеграцией разработанных компонентов в единую систему или программное средство.

2.3.4. Модель быстрой разработки приложений по ГОСТ Р ИСО/МЭК ТО 15271–2002

На рис. 2.11 представлен вариант RAD-модели, приведенный в стандарте *ГОСТ Р ИСО/МЭК ТО 15271–2002* и адаптированный под требования стандарта *ISO/IEC 12207*:*1995* (*СТБ ИСО/МЭК 12207–2003*) [8, 3, 9]. Числами в скобках обозначены номера работ процесса разработки, используемые на соответствующих этапах модели (см. подразд. 1.2), * обозначает процесс эксплуатации.

На этапе осуществимости выполняется анализ проекта на основе критических факторов успешности реализации RAD-модели. К данным факторам могут быть отнесены, например, области применения RAD-модели (см. п. 2.3.5).

На этапе анализа деловой деятельности определяется область применения проекта и разрабатывается план прототипирования.

Назначением *цикла функциональной модели* является уточнение функциональных требований к разрабатываемой системе (программному средству). В данном цикле с помощью инструментальных средств быстрой разработки (CASE-средств) разрабатываются функциональные прототипы. Кроме того, формулируются нефункциональные требования и стратегия реализации полного прототипа.

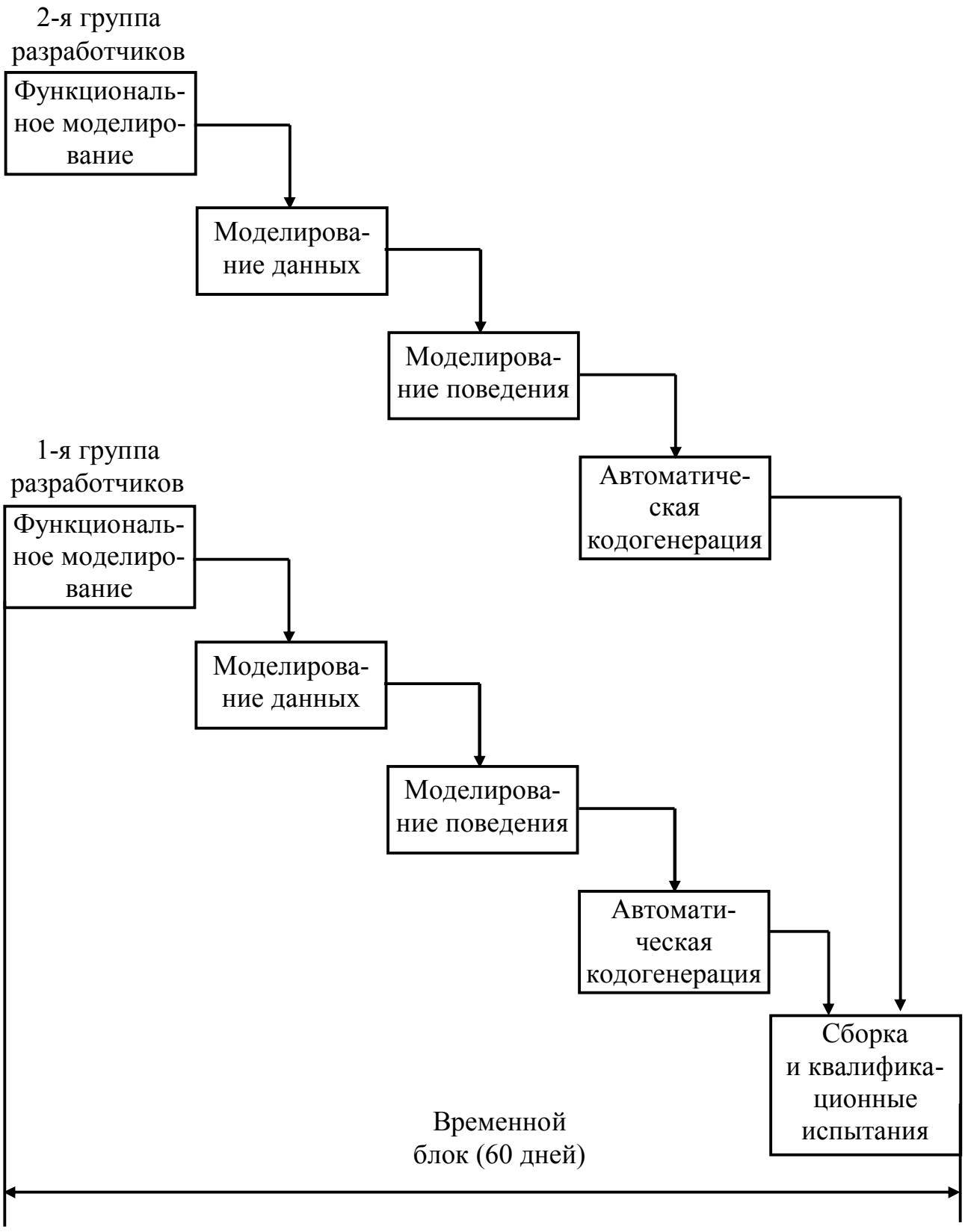


Рис. 2.10. Вариант RAD-модели,
основанный на независимой работе групп разработчиков

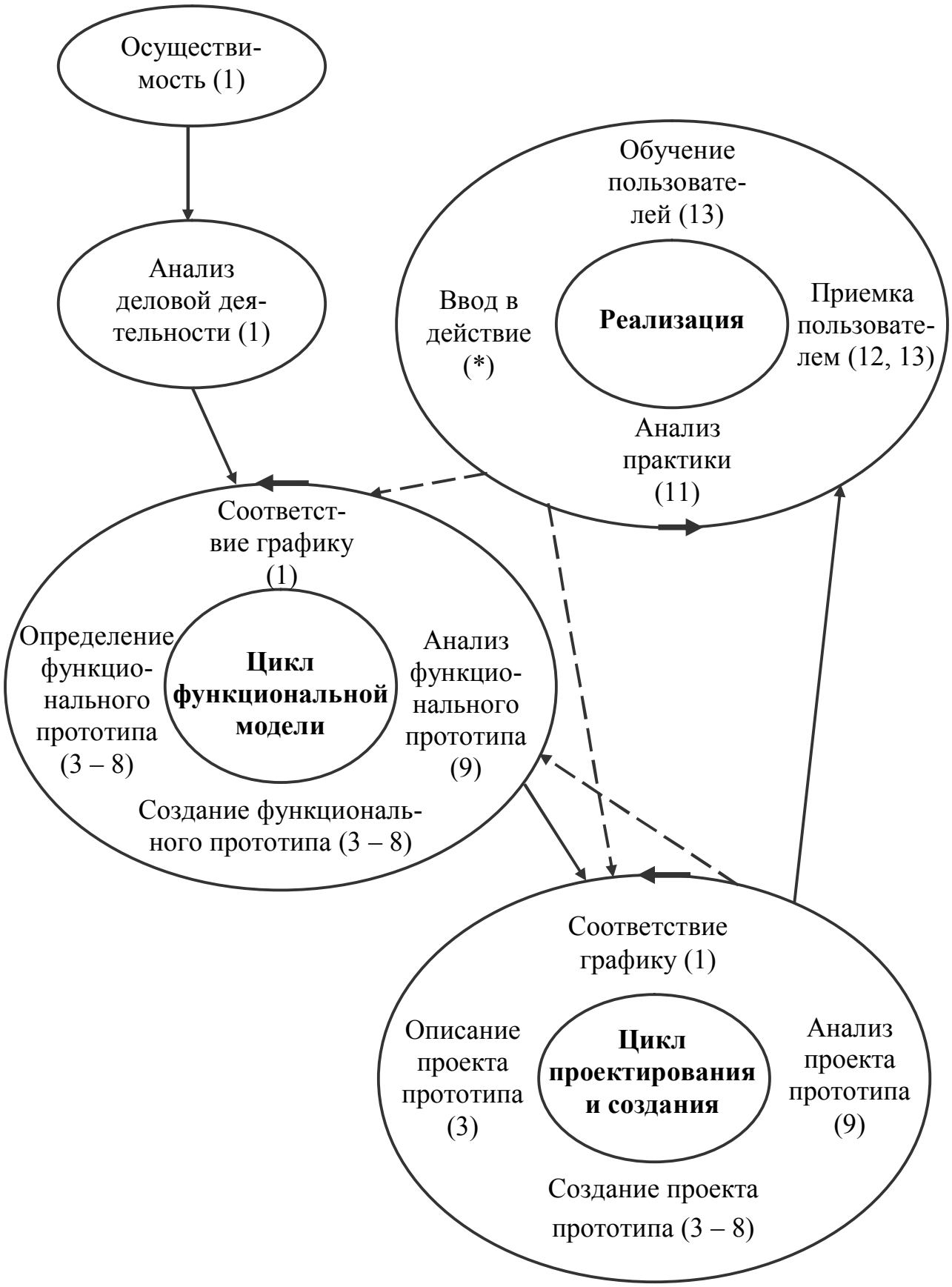


Рис. 2.11. Вариант модели быстрой разработки приложений
по ГОСТ Р ИСО/МЭК ТО 15271–2002

В цикле проектирования и создания на основе функционального прототипа последовательно проектируются, реализуются и анализируются заказчиком прототипы, все более полно учитывающие нефункциональные требования к разрабатываемой системе или программному средству. В конечном итоге создаются ПС, прошедшие квалификационные испытания и удовлетворяющие всем функциональным и нефункциональным требованиям заказчика.

На этапе реализации выполняется внедрение ПС в среде пользователя и обучение соответствующего персонала.

Цикл функциональной модели и цикл проектирования и создания предусматривают последовательную итерацию работ 3 – 9 процесса разработки, регламентированного стандартом *СТБ ИСО/МЭК 12207–2003* (см. подразд. 1.2). При этом каждая итерация прототипа должна быть создана в соответствии с графиком работ за определенный временной интервал. Конкретное время реализации каждого шага итерационного цикла, как правило, определяется набором *трех итераций* – предварительное исследование, уточнение и утверждение (принятие) каждого прототипа.

Из описания приведенного варианта RAD-модели следует, что он фактически реализует эволюционную стратегию разработки ПС. В каждом цикле разработки реализуется уточнение функциональных и нефункциональных требований, разработка функционального прототипа, разработка версии программного средства, ввод в действие и эксплуатация данной версии.

Резюме

Приведенный вариант RAD-модели адаптирован под требования стандарта *ISO/IEC 12207*:*1995* (*СТБ ИСО/МЭК 12207–2003*). Данный вариант базируется на использовании цикла функциональной модели и цикла проектирования и создания. Данные циклы выполняются итерационно.

2.3.5. Достоинства, недостатки и области использования RAD-моделей

При использовании RAD-модели в соответствующем ей проекте проявляются следующие ее *основные достоинства*:

1) сокращение продолжительности цикла разработки и всего проекта в целом, сокращение количества разработчиков, а следовательно, и стоимости проекта за счет использования мощных инструментальных средств;

2) сокращение риска несоблюдения графика за счет использования принципа временного блока и связанное с этим упрощение планирования;

3) сокращение риска, связанного с неудовлетворенностью заказчика (пользователя) разработанным программным продуктом, за счет его привлечения на постоянной основе к циклу разработки; возрастание уверенности, что продукт будет соответствовать требованиям;

4) возможность повторного использования существующих компонентов; это достоинство проявляется при использовании RAD-модели в составе инкрементной или эволюционной модели; в этом случае наращивание функциональных возможностей осуществляется на базе разработанных ранее компонентов.

Основными недостатками RAD-модели при использовании в неподходящем для нее проекте являются:

1) необходимость в постоянном участии пользователя в процессе разработки, что часто невыполнимо и в итоге сказывается на качестве конечного продукта;

2) необходимость в высококвалифицированных разработчиках, умеющих работать с инструментальными средствами разработки;

3) возможность применения только для систем или ПС, для которых отсутствует требование высокой производительности;

4) жесткость временных ограничений на разработку прототипа;

5) сложность ограничения затрат и определения сроков завершения работы над проектом в случаях, когда не удается разработать продукт за временной интервал;

6) неприменимость в условиях высоких технических рисков, при использовании новых технологий.

Области применения RAD-модели определяются ее достоинствами и ограничены ее недостатками. RAD-модель может эффективно применяться в следующих случаях:

1) при разработке систем и продуктов, для которых характерно хотя бы одно из следующих свойств:

- поддаются моделированию;
- предназначены для концептуальной проверки;
- являются некритическими;
- имеют небольшой размер;
- имеют низкую производительность;
- относятся к известной разработчикам предметной области;
- являются информационными системами;
- требования для них хорошо известны;
- имеются пригодные к повторному использованию в них компоненты;

2) если пользователь может принимать постоянное участие в процессе разработки;

3) если в проекте заняты разработчики, обладающие достаточными навыками в использовании инструментальных средств разработки;

4) при выполнении проектов в сокращенные сроки (как правило, не более чем за 60 дней);

5) при разработке ПС, для которых требуется быстрое наращивание функциональных возможностей на последовательной основе;

6) при невысокой степени технических рисков;

7) в составе других моделей жизненного цикла.

Резюме

Использование RAD-модели сокращает количество разработчиков, продолжительность и стоимость разработки ПС и систем, сокращает риски. В то же время ее применение требует высокой квалификации разработчиков и постоянного участия пользователей в процессе разработки ПС и систем.

2.4. Модели жизненного цикла, реализующие инкрементную стратегию разработки программных средств и систем

2.4.1. Общие сведения об инкрементных моделях

Инкрементные модели поддерживают инкрементную стратегию разработки ПС и систем. Данная стратегия представляет собой запланированное улучшение продукта в процессе его ЖЦ (см. п. 2.1.3). При использовании инкрементных моделей осуществляется изначальная частичная реализация всей системы (или программного средства). При этом в первую очередь реализуются базовые требования. За этим следует медленное наращивание функциональных возможностей или характеристик качества системы или программного средства в реализуемых последовательно прототипах (инкрементах).

Применение инкрементных моделей ускоряет создание программного средства или системы за счет применяемого принципа компоновки из стандартных или разработанных ранее программных компонентов. Это позволяет существенно уменьшить затраты на разработку.

Существуют различные варианты реализации инкрементных моделей.

В классических вариантах модель основана на использовании полного заранее сформированного набора требований, реализуемых последовательно в виде небольших инкрементов.

Существуют варианты модели, начинающиеся с формулирования общих требований. Требования постепенно уточняются в процессе разработки прототипов. Данные варианты инкрементных моделей похожи на эволюционные модели. Однако от последних они отличаются существенно большим количеством инкрементов при гораздо меньших различиях между соседними инкрементами. К таким вариантам моделей относится, например, модель ЖЦ, реализущая современную реализацию инкрементной стратегии – экстремальное программирование (см. п. 2.4.4).

При использовании инкрементной модели различия между последовательными прототипами постепенно уменьшаются. В каждой последующей вер-

сии системы или программного средства добавляются к предыдущей версии определенные программные компоненты, реализующие соответствующие функциональные возможности до тех пор, пока не будут реализованы все требования к системе (программному средству). При этом каждая версия системы или программного средства может сдаваться в эксплуатацию.

Резюме

В классических вариантах инкрементная модель основана на использовании полного заранее сформированного набора требований, реализуемых последовательно в виде небольших проектов. Существуют варианты инкрементной модели, начинающиеся с формулирования общих целей, которые постепенно уточняются в процессе разработки прототипов.

2.4.2. Инкрементная модель с уточнением требований на начальных этапах разработки

В [33] приведен вариант инкрементной модели с возможностью изменения или уточнения требований на начальных этапах процесса разработки системы или программного средства. На рис. 2.12 представлена аналогичная модель, адаптированная к структуре процесса разработки, определенного в стандарте *СТБ ИСО/МЭК 12207–2003* (см. подразд. 1.2).

На ранних этапах процесса разработки (анализ требований к системе, проектирование системной архитектуры, анализ требований к программным средствам) выполняется проектирование системы в целом. При этом используется каскадная модель с обратными связями между этапами. Применение обратных связей позволяет производить уточнение требований к системе, выполнять проектирование ее архитектуры с учетом изменившихся требований, уточнять требования к ПС. На этих этапах определяются инкременты и реализуемые ими функции.

Каждый инкремент затем проходит через остальные этапы жизненного цикла (проектирование ПС, программирование и тестирование ПС, сборка и квалификационные испытания ПС и системы, ввод в действие и обеспечение приемки ПС и содержащей их системы, эксплуатация и сопровождение ПС). Выполнение данных этапов соответствует каскадной модели ЖЦ и может быть распределено согласно календарному графику.

В модели в первую очередь реализуется набор функциональных и нефункциональных требований, формирующих основу продукта. Последующие инкременты улучшают функциональные возможности или характеристики про граммного средства. Каждый инкремент верифицируется в соответствии с набором требований, предъявляемых к данному инкременту.

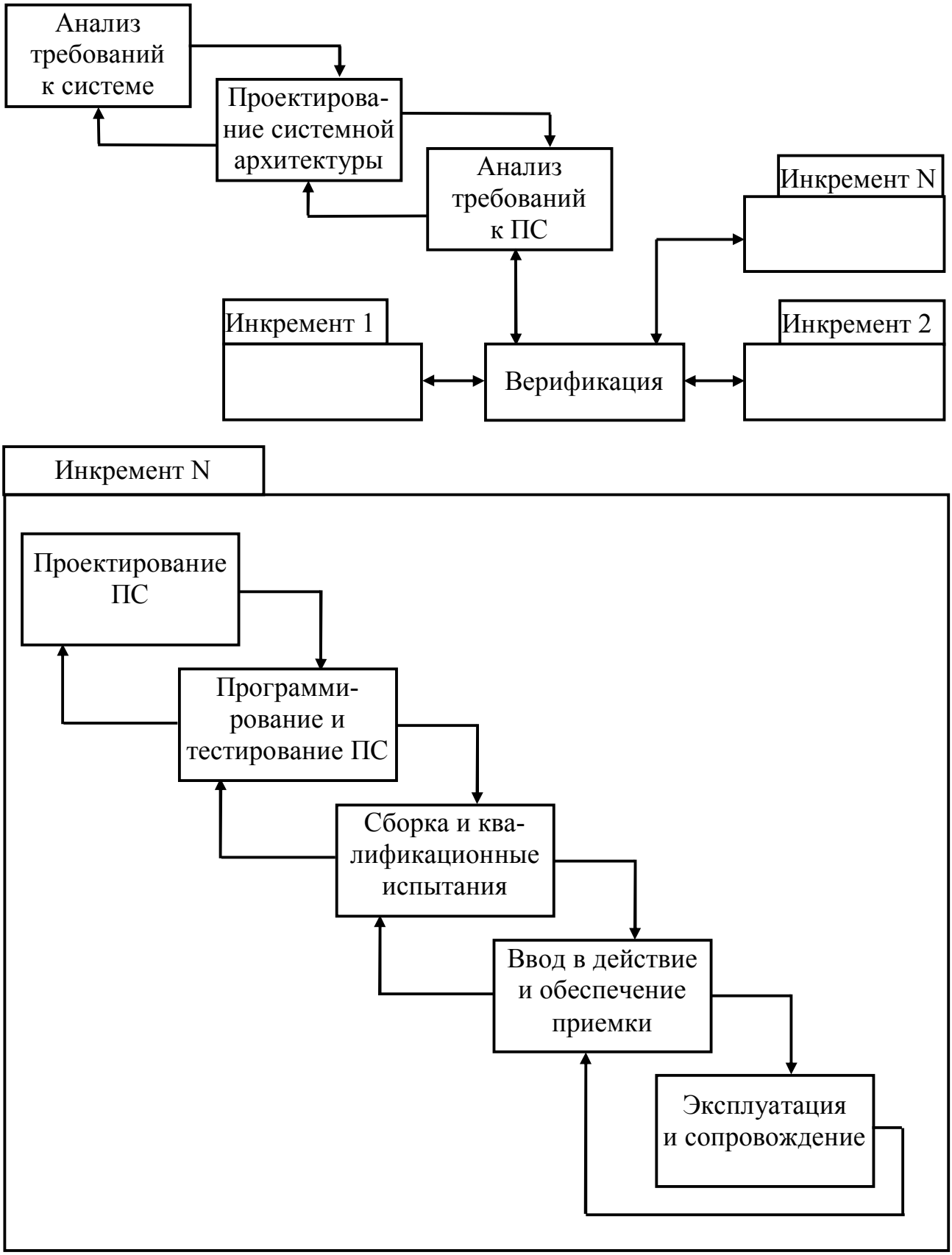


Рис. 2.12. Инкрементная модель жизненного цикла

Резюме

В рассмотренном варианте инкрементной модели возможно уточнение требований на начальных этапах процесса разработки системы. Разработка каждого инкремента реализуется на базе каскадной стратегии.

2.4.3. Вариант инкрементной модели по ГОСТ Р ИСО/МЭК ТО 15271–2002

В стандарте *ГОСТ Р ИСО/МЭК ТО 15271–2002* [8] приводится другой вариант реализации инкрементной модели (рис. 2.13). Он основан на использовании полного заранее сформированного набора требований и их постепенной реализации в отдельных инкрементах. Данный вариант модели учитывает как возможность частично параллельной разработки инкрементов различными группами разработчиков (на рис. 2.13 это возможные информационные потоки между инкрементами 1 и 2), так и возможность последовательной разработки (информационные потоки между инкрементами 2 и N).

В инкременте 1 проектируется архитектура программного средства (или системы) и реализуются его базовые функции. Результаты проектирования инкремента 1 могут быть использованы для проектирования инкремента 2, не дожидаясь окончания реализации инкремента 1.

В дальнейшем различия между инкрементами уменьшаются, что позволяет сократить время их разработки. Поэтому с целью упрощения управления проектом обычно выполняется последовательная разработка инкрементов.

Разработка каждого инкремента состоит из трех укрупненных этапов: проектирование, программирование и тестирование, ввод в действие и обеспечение приемки. Каждый этап объединяет соответствующие работы процесса разработки. Например, при разработке программного средства этап проектирования объединяет работы 5, 6 процесса разработки, этап программирования и тестирования – работы 7, 8, 9, этап ввода в действие и обеспечения приемки – работы 12 и 13 (см. подразд. 1.2).

Инкрементные модели можно комбинировать с другими моделями. Часто их объединяют со спиральной или V-образной моделью [33]. Например, разработка каждого инкремента может выполняться в соответствии с V-образной моделью. Это позволяет снизить затраты и риски при разработке ПС.

Резюме

Рассмотренный вариант инкрементной модели базируется на предварительном полном определении требований и их последовательной планомерной реализации. В модели возможна как последовательная, так и частично параллельная разработка инкрементов различными группами разработчиков. Возможна комбинация модели с другими моделями. Все это позволяет сократить общие сроки и затраты на разработку ПС и системы в целом.

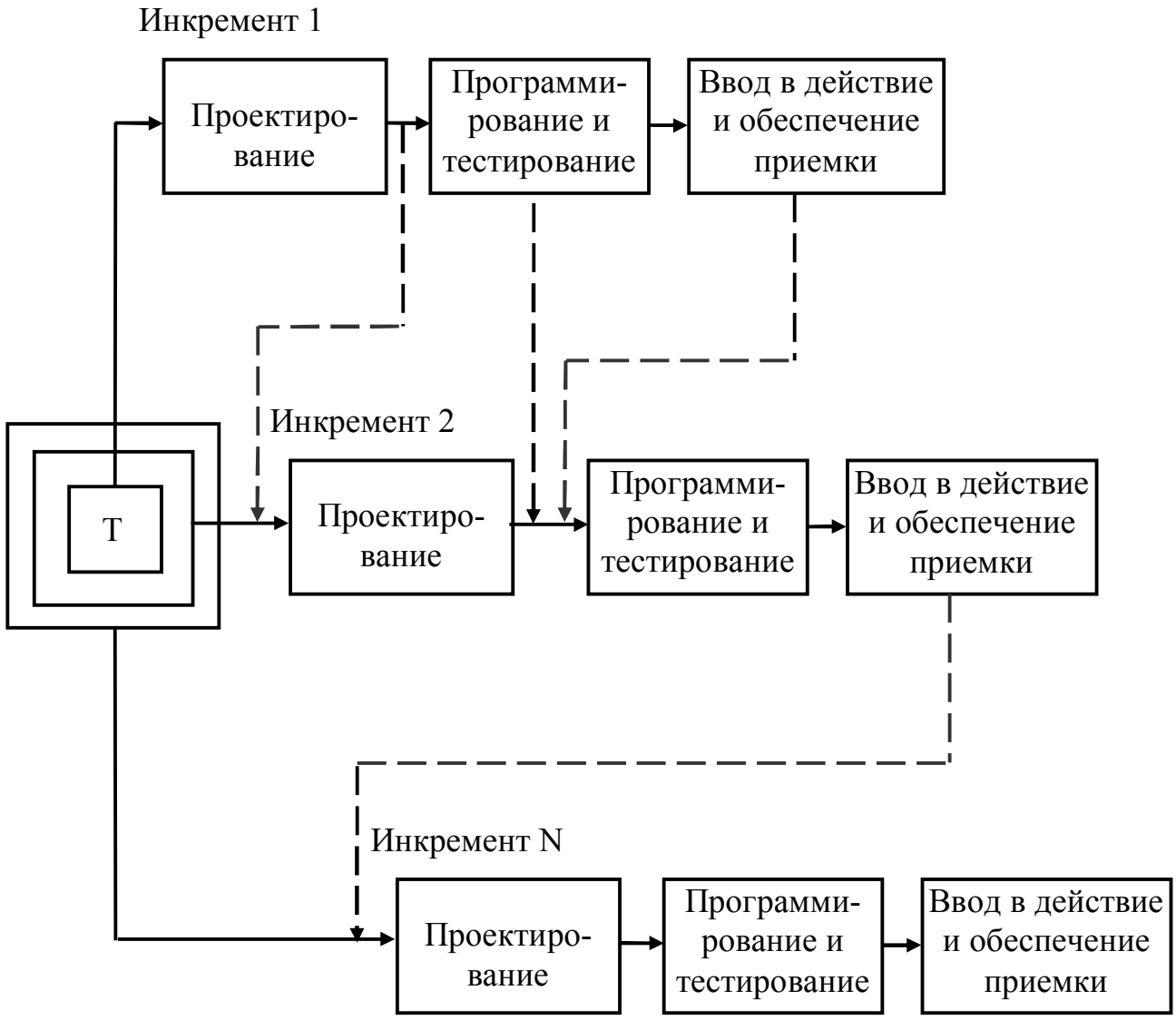


Рис. 2.13. Вариант инкрементной модели
по ГОСТ Р ИСО/МЭК ТО 15271–2002

2.4.4. Инкрементная модель экстремального программирования

На рис. 2.14 представлена инкрементная модель жизненного цикла, реализуемая при экстремальном программировании [42]. Данная модель может использоваться в том случае, когда требования заказчика неопределенны или постоянно изменяются. Модель отличается гибкостью, поскольку она ориентирована на высокую степень неопределенности спецификации требований.

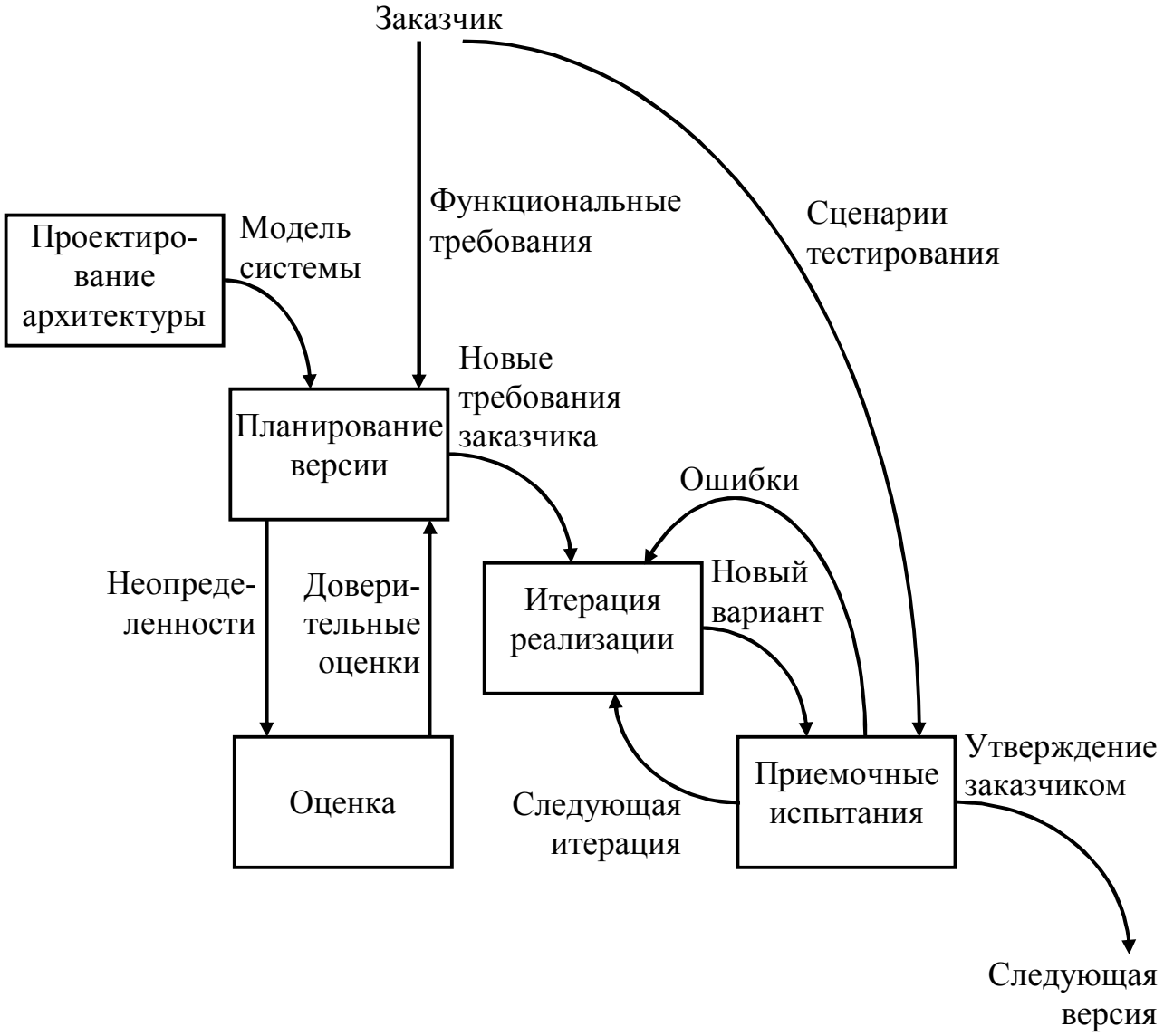


Рис. 2.14. Вариант инкрементной модели
экстремального программирования

В соответствии с данной моделью на первом этапе разработки (проектирование архитектуры) усилия разработчиков затрачиваются на тщательное проектирование архитектуры системы и/или программного средства. В дальнейшем уточнение или изменение архитектуры не предусматривается. Результатом данного этапа является модель системы.

На втором этапе разработки выполняется планирование очередной версии системы (программного средства). Каждое новое функциональное требование, поступившее к текущему моменту от заказчика, оценивается с точки зрения стоимости и времени его реализации. При этом учитываются результаты оценки неопределенности данных требований и рисков их реализации. С учетом выполненных оценок заказчик выбирает и утверждает новые требования, кото-

рые будут реализовываться в очередной версии системы (программного средства).

Следующие этапы разработки выполняются итерационно. Новые требования заказчика реализуются в новом варианте системы (программного средства), выполняются приемочные испытания данного варианта. Если в очередном варианте системы (программного средства) найдены ошибки, то данный вариант возвращается на следующую итерацию реализации. Процесс продолжается, пока результаты очередных приемочных испытаний не будут утверждены заказчиком. Утвержденный вариант системы (программного средства) поступает в эксплуатацию в качестве очередной версии.

Резюме

Рассмотренный вариант инкрементной модели реализуется при экстремальном программировании. Данный вариант ориентирован на высокую степень неопределенности спецификации требований заказчика. В модели выполняется однократная разработка архитектуры системы и ПС. Реализация каждого требования заказчика выполняется с учетом его стоимости и целесообразности. Каждая версия системы реализуется итерационно.

2.5. Модели жизненного цикла, реализующие эволюционную стратегию разработки программных средств и систем

2.5.1. Общие сведения об эволюционных моделях

Эволюционные модели поддерживают эволюционную стратегию разработки ПС и систем, при которой в начале жизненного цикла определяются не все требования. Система или программное средство строится в виде последовательности версий. Каждая из версий реализует некоторое подмножество требований. После реализации каждой версии требования уточняются (см. п. 2.1.4).

Как правило, эволюционные модели базируются на использовании прототипирования.

2.5.2. Эволюционная модель по ГОСТ Р ИСО/МЭК ТО 15271–2002

Один из простейших классических вариантов эволюционной модели приведен в стандарте *ГОСТ Р ИСО/МЭК ТО 15271–2002* (рис. 2.15) [8].

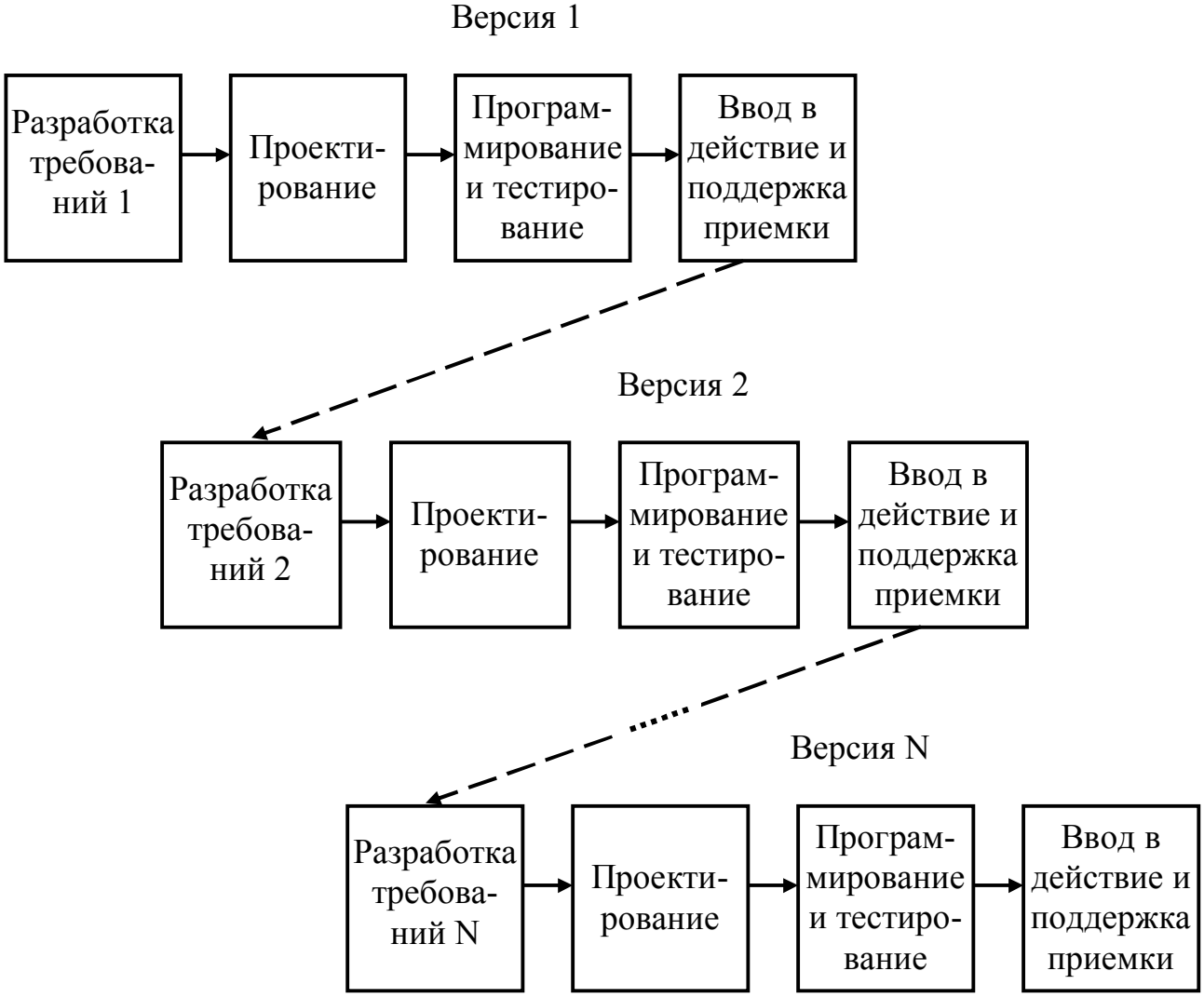


Рис. 2.15. Вариант эволюционной модели
по ГОСТ Р ИСО/МЭК ТО 15271–2002

В данном случае разработка каждой версии системы (программного средства) выполняется на основе каскадной модели, содержащей четыре этапа: разработка требований, проектирование, программирование и тестирование, ввод в действие и поддержка приемки.

При разработке первой версии формулируются наиболее важные (базовые) требования к продукту. На основе данных требований разрабатывается и вводится в действие первая версия системы (программного средства).

При разработке каждой очередной версии требования уточняются, при необходимости вводятся новые или изменяются уже реализованные требования. На основе уточненных требований разрабатывается и вводится в действие очередная версия продукта.

Процесс продолжается, пока все требования не будут окончательно уточнены и полностью реализованы.

Резюме

Рассмотренный вариант эволюционной модели поддерживает классическую эволюционную стратегию разработки ПС и систем. Для разработки отдельных версий продукта используется каскадная модель.

2.5.3. Структурная эволюционная модель быстрого прототипирования

При использовании структурной эволюционной модели быстрого прототипирования система (программное средство) строится в виде последовательности эволюционных прототипов [33]. Данная модель представлена на рис.2.16.

Начало ЖЦ разработки находится в центре модели. С учетом предварительных требований пользователями и разработчиками разрабатывается предварительный план проекта.

Затем выполняется быстрый анализ требований к системе (программному средству), во время которого совместно с пользователями разрабатываются *умышленно* неполные требования. На их основе выполняется *укрупненное* проектирование, программирование и тестирование программных компонентов, тестирование системы в целом. Таким образом, реализуется построение исходного прототипа. Данный прототип оценивается пользователем.

После этого начинается итерационный цикл быстрого прототипирования, содержание которого аналогично циклу построения исходного прототипа (уточнение требований, *укрупненное* проектирование, программирование и тестирование программных компонентов, тестирование системы в целом).

Пользователь оценивает функционирование очередного прототипа. По результатам оценки уточняются требования, на основании которых разрабатывается новый прототип. Этот процесс продолжается до тех пор, пока быстрый прототип не окажется удовлетворительным и не будет принят пользователем.

Затем осуществляется детализированная разработка программного средства, во время которой реализуются его несущественные функции, пользовательские интерфейсы и т.п. После этого выполняется ввод в действие программного средства в среде системы и поддержка его приемки. В результате ускоренный прототип становится полностью действующей системой, которая сдается в эксплуатацию.

Из описания модели быстрого прототипирования видны ее отличия от других эволюционных моделей. Основное отличие заключается в том, что с целью ускорения разработки результаты промежуточных циклов в данной модели представляются в виде прототипов, не доводятся до уровня рабочих версий программного средства или системы и поэтому не сдаются в эксплуатацию. Прототипы предназначены только для уточнения требований.

Зачастую данная модель применяется в комбинации с каскадной моделью. На начальных этапах разработки используется прототипирование, а на по-

следнем (детальная разработка) – этапы каскадной модели с целью обеспечения качества программного средства.

Основным *достоинством* структурной эволюционной модели быстрого прототипирования является ускорение процесса разработки ПС и систем.

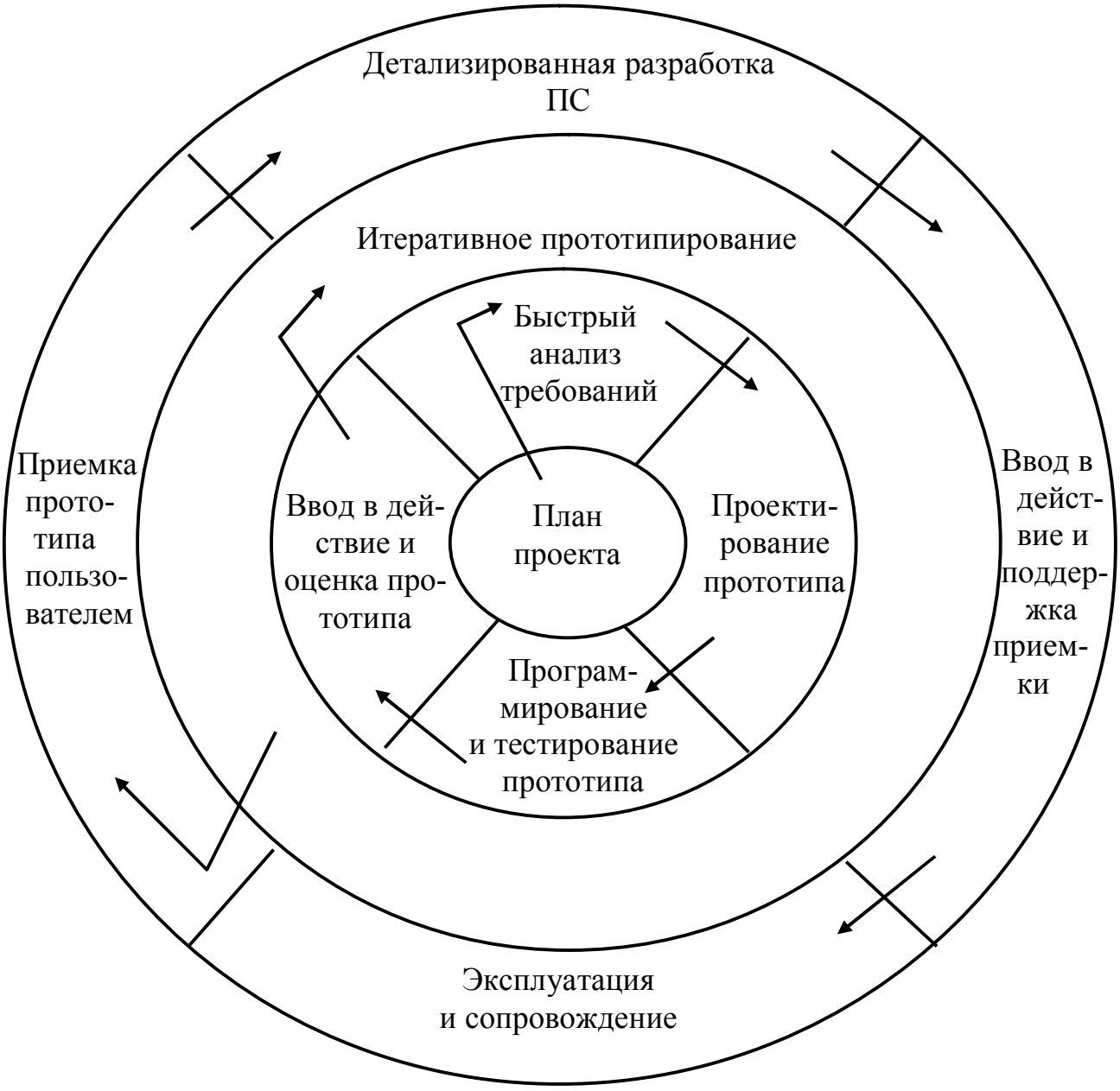


Рис. 2.16. Структурная эволюционная модель
быстрого прототипирования

К основным *недостаткам* данной модели можно отнести следующее:

1) существует вероятность недостаточного качества результирующего программного средства (и системы в целом) за счет его создания из рабочего прототипа (при детальной разработке программного средства из последнего

прототипа может оказаться сложной или невозможной реализация функций, не реализованных при итерационном прототипировании);

2) возможна задержка реализации конечной версии системы (программного средства) при несочетании языка или среды прототипирования с рабочим языком или средой программирования.

С учетом особенностей и отличий от классической эволюционной стратегии (см. п. 2.1.4) быстрое прототипирование иногда выделяют в отдельную стратегию разработки – так называемую *стратегию быстрого прототипирования*. Существуют различные модели быстрого прототипирования. Общим для них является то, что прототипирование в этих моделях предназначено только для уточнения требований к разрабатываемому продукту.

Резюме

Эволюционная модель быстрого прототипирования используется для ускорения разработки систем или ПС. В данной модели результаты промежуточных циклов представляются в виде прототипов, не доводятся до уровня рабочих версий продукта и не сдаются в эксплуатацию. Прототипы предназначены для уточнения требований. После принятия пользователем (заказчиком) конечного прототипа создается рабочая система или программное средство.

2.5.4. Эволюционная модель прототипирования по ГОСТ Р ИСО/МЭК ТО 15271–2002

В стандарте *ГОСТ Р ИСО/МЭК ТО 15271–2002* приведен пример эволюционной модели, основанной на прототипировании (рис. 2.17) [8]. Данная модель адаптирована под требования стандарта *ISO/IEC 12207*:*1995* (*СТБ ИСО/МЭК 12207–2003*, см. подразд. 1.2). Модель предназначена для разработки небольших коммерческих систем.

Основой эффективного применения данной модели жизненного цикла является максимально возможная детализация на ранних этапах процесса разработки (анализ требований к системе и проектирование системной архитектуры). Данные этапы выполняются в модели один раз. Однократное выполнение этих этапов достигается за счет тесных связей разработчиков с пользователями проекта. Требования к системе, в первую очередь, функции системы и внешние интерфейсы определяются пользователями в начале ЖЦ, процессы обработки уточняются при проведении пользователем серии оценок прототипов системы.

В данной модели при создании версий программного средства используется прототипирование. При разработке каждого прототипа уточняются требования к нему. Затем выполняется программирование прототипа в среде языка программирования четвертого поколения 4GL. При выполнении данного этапа инструментальная среда 4GL используется в первую очередь для быстрого проектирования и сборки программного средства, а также его оперативного нара-

щивания, изменения и уточнения. Языки 4GL осуществляют частичную автоматическую кодогенерацию программного средства.

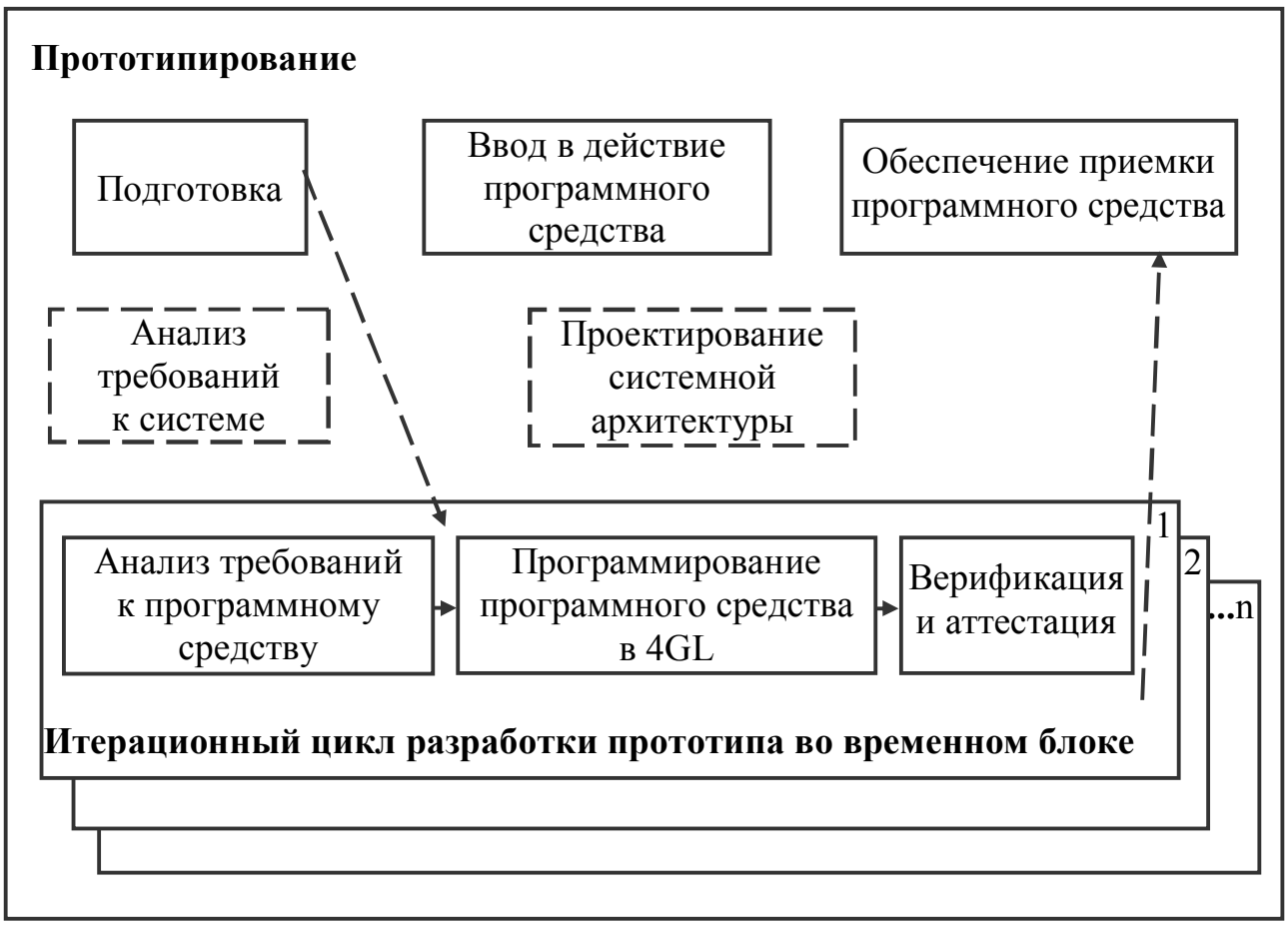


Рис. 2.17. Вариант эволюционной модели прототипирования по ГОСТ Р ИСО/МЭК ТО 15271–2002

Следует отметить, что в данной модели при разработке прототипов помимо языков 4GL могут эффективно применяться CASE-средства (см. разд. 7).

В данной модели ЖЦ определен фиксированный период проведения прототипирования и произвольное количество итераций.

Проверка и оценка каждого прототипа осуществляется пользователем в реальной эксплуатационной среде.

Разработчик программного средства может *контролировать прототипирование* с помощью:

1) установления приоритетов требований к программному средству;
2) ужесточения ограничений временного интервала;
3) привлечения конечного пользователя.

Из описания данной модели видно, что при разработке прототипов фактически используется RAD-модель жизненного цикла, базирующаяся на использовании временного блока (см. подразд. 2.3).

Резюме

В рассмотренном варианте эволюционной модели анализ требований к системе и проектирование системной архитектуры выполняются один раз. Для разработки версий программного средства используется прототипирование. При этом применяется RAD-модель ЖЦ с фиксированным периодом проведения прототипирования.

2.5.5. Спиральная модель Боэма

Спиральные модели ЖЦ поддерживают эволюционную стратегию разработки ПС и систем.

Данные модели объединяют в себе преимущества других видов моделей. Кроме того, в них включен ряд вспомогательных и организационных процессов, предусмотрены анализ и управление рисками, возможности использования прототипирования и быстрой разработки систем и ПС на основе RAD-модели [33].

Базовая концепция спиральной модели заключается в следующем. Каждый цикл разработки (итерация) представляет собой набор операций, соответствующий шагам в каскадной модели. Шагам каскадной модели соответствует и последовательность витков спиральной модели.

Следует отметить, что большинство спиральных моделей было разработано ранее принятия международного стандарта *ISO/IEC 12207*:*1995* [3]. В связи с этим необходимо выполнять адаптацию этих моделей с учетом положений действующих национальных аналогов данного стандарта [9] и его новой редакции *ISO/IEC 12207*:*2008* [4].

Базовой в семействе спиральных моделей является классическая модель Боэма, разработанная в 1988 г. [33]. На рис. 2.18 изображен вариант классической спиральной модели Боэма, адаптированный с учетом структуры процесса разработки и основных положений стандарта *СТБ ИСО/МЭК 12207–2003*.

Для удобства описания модели этапы разработки сгруппированы в фазы, соответствующие виткам спирали. Каждая фаза может выполняться итерационно за один или несколько циклов. На рис. 2.18 использованы следующие обозначения этапов разработки.

А. ***Фаза разработки концепции*** (соответствует первому витку спирали).

В состав данной фазы входят следующие этапы:

1 – определение потребности;

2 – анализ рисков фазы разработки концепции;

3 – концептуальное прототипирование;

4 – разработка концепции требований к системе/программному продукту (концепции эксплуатации);

5 – планирование проекта и процесса разработки.

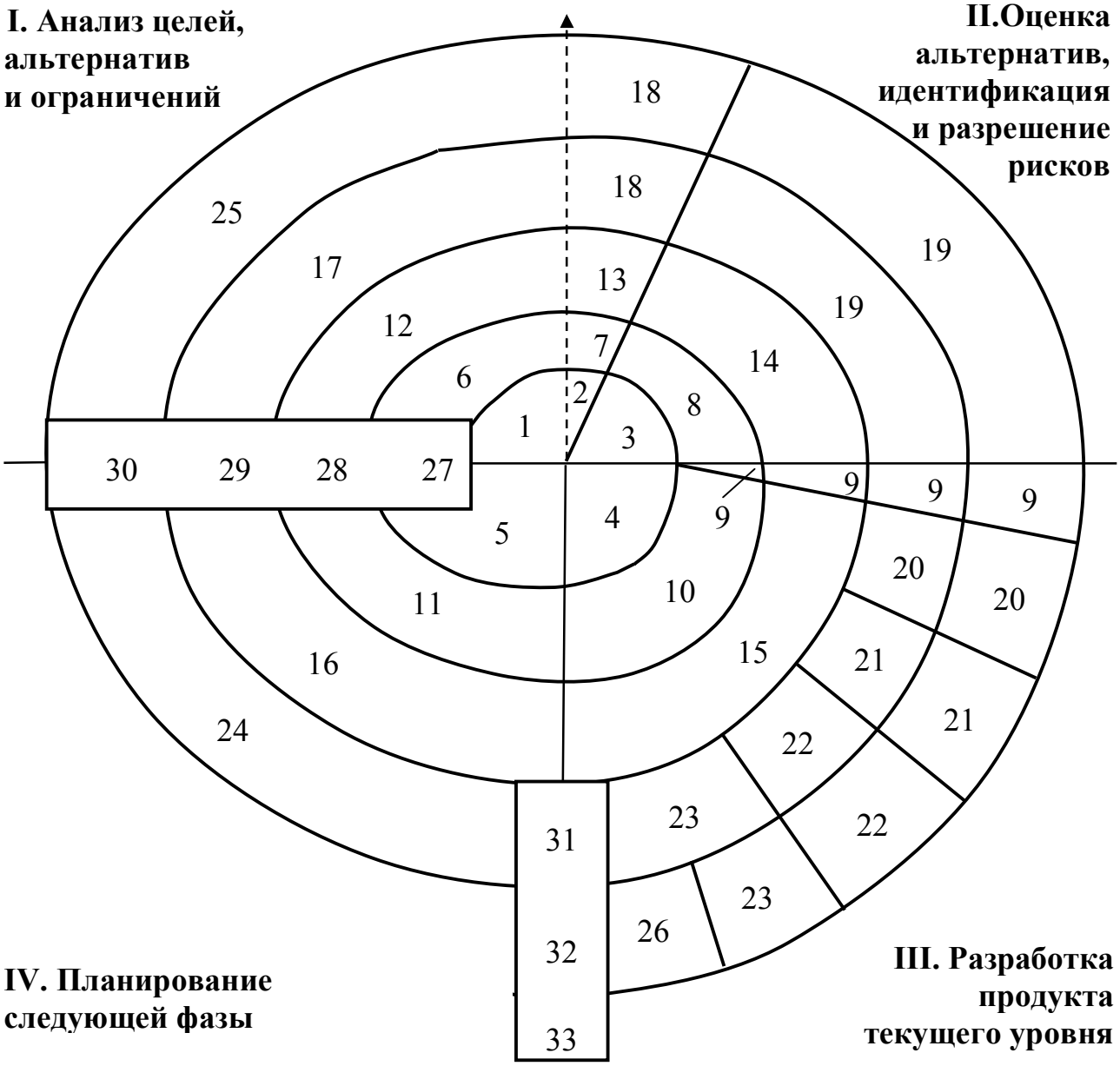


Рис. 2.18. Вариант спиральной модели Боэма,
адаптированный к положениям стандарта *СТБ ИСО/МЭК 12207–2003*

В. ***Фаза анализа требований*** (соответствует второму витку спирали).

В состав данной фазы входят следующие этапы:

6 – анализ целей, альтернатив и ограничений, связанных с системой/программным продуктом;

7 – анализ рисков фазы анализа требований;

8 – прототипирование требований (называемое демонстрационным прототипированием);

9 – оценка характеристик системы/программного продукта;

10 – разработка требований к системе/программному продукту и их аттестация;

11 – планирование перехода на фазу проектирования системы/программного продукта.

C. ***Фаза проектирования системы/программного продукта*** (соответствует третьему витку спирали).

В состав данной фазы входят следующие этапы:

12 – анализ целей, альтернатив и ограничений, связанных с текущим циклом проектирования;

13 – анализ рисков фазы проектирования;

14 – прототипирование проектирования системы/программного продукта (оценочное прототипирование);

9 – оценка характеристик системы/ программного продукта;

15 – проектирование системной/программной архитектуры, верификация и аттестация результатов проектирования;

16 – планирование перехода на фазу реализации.

D. ***Фаза реализации (технического проектирования, программирования и сборки)***. Соответствует четвертому витку спирали.

В состав данной фазы входят следующие этапы:

17 – анализ возможности реализации в текущем цикле целей, альтернатив и ограничений, связанных с техническим проектированием, программированием и сборкой системы/продукта;

18 – анализ рисков фазы реализации;

19 – прототипирование реализации (операционное прототипирование);

9 – оценка характеристик системы/продукта;

20 – техническое проектирование программного средства;

21 – программирование и тестирование программного средства;

22 – сборка и квалификационные испытания программного средства;

23 – сборка и квалификационные испытания системы;

24 – планирование перехода на фазу расширения функциональных возможностей.

E. ***Фаза сопровождения и расширения функциональных возможностей*** (соответствует пятому витку спирали).

В состав данной фазы входят следующие этапы:

25 – анализ целей, альтернатив и ограничений, связанных с сопровождением и расширением функциональных возможностей;

18 – анализ рисков реализации;

19 – прототипирование реализации (операционное прототипирование);

9 – оценка характеристик системы/программного продукта;

20 – техническое проектирование программного средства;

21 – программирование и тестирование программного средства;

22 – сборка и квалификационные испытания программного средства;
23 – сборка и квалификационные испытания системы;
26 – приемочные испытания.

Выполнение этапов 9, 18 – 23 данной фазы аналогично соответствующим этапам фазы реализации.

Результаты каждого цикла разработки подвергаются соответствующим ***оценочным действиям*** (см. рис. 2.18, левый прямоугольник):

27 – оценка концепции;
28 – оценка требований;
29 – оценка проектирования;
30 – оценка версии системы/продукта.

В нижний прямоугольник модели сведены действия, связанные с ***поставкой результатов*** текущего уровня разработки:

31 – поставка первой пригодной версии;
32 – поставка очередной пригодной версии;
33 – аудит конфигурации версии.

Модель Боэма поделена на *четыре квадранта*. В каждый квадрант модели входят основные и вспомогательные действия по разработке продукта или системы.

В квадранте I – анализ целей, альтернативных вариантов и ограничений – определяются рабочие характеристики, выполняемые функции, стабильность (возможность внесения изменений), аппаратно/программный интерфейс продукта разработки данной фазы или цикла. Формулируются требования к данному продукту.

Определяются альтернативные способы реализации системы или программного продукта (разработка, повторное использование компонент, покупка, договор подряда и т.п.). Определяются ограничения, налагаемые на применение альтернативных вариантов (затраты, график выполнения, интерфейс, ограничения среды и др.).

Определяются риски, связанные с недостатком опыта в данной предметной области, применением новых технологий, жесткими графиками, недостаточно хорошо организованными процессами.

В квадранте II – оценка альтернативных вариантов, идентификация и разрешение рисков – выполняется оценка альтернативных вариантов, рассмотренных в предыдущем квадранте; оценка возможных вариантов сокращения или устранения рисков. Выполняется прототипирование как основа для работ следующего квадранта.

В квадрант III – разработка продукта текущего уровня – включаются действия по непосредственной разработке системы или программного продукта: разработка требований, проектирование системы и ее программных компонентов,

разработка и тестирование исходных текстов программ, сборка, тестирование и квалификационные испытания продукта или системы и т.п.

В квадранте IV – планирование следующей фазы – выполняются действия, связанные с решением о переходе на цикл следующей фазы разработки или выполнении еще одного цикла текущей фазы разработки, в частности, оценка заказчиком (пользователем) результатов текущего цикла, уточнение требований, разработка или коррекция планов проекта и следующего цикла, управление конфигурацией.

Работа над проектом в соответствии со спиральной моделью начинается с определения заказчиком потребности в разработке системы или программного продукта (этап 1 квадранта I в центре спирали, см. рис. 2.18).

Первая создаваемая версия системы основывается на предварительных требованиях заказчика (пользователя). Затем начинается планирование следующего цикла, учитывающее требования и пожелания заказчика (пользователя), сформулированные им по результатам работ соответствующего уровня квадранта III. Каждая последующая версия более точно реализует требования заказчика (пользователя). Степень вносимых изменений от одной версии системы (программного продукта) к следующей уменьшается с каждой новой версией. В результате получается окончательный вариант системы/программного продукта.

Для каждого цикла модели анализируются требования, альтернативные варианты и ограничения; определяются, сокращаются или устраняются риски; разрабатывается версия продукта или системы этого цикла спирали и подтверждается ее правильность; планируется следующий цикл и выбираются методы его осуществления. В конце каждого цикла осуществляется оценка его результатов, по итогам которой выполняется либо переход к следующей фазе, либо в случае необходимости выполнение еще одного цикла данной фазы.

Количество итераций модели зависит от сложности проекта. Работы каждой итерации должны бать адаптированы под конкретный проект.

Следует обратить внимание на то, что программирование в спиральной модели выполняется значительно позже, чем в других моделях. Это позволяет минимизировать риски посредством последовательных уточнений требований, выдвигаемых заказчиком (пользователем). На каждой итерации рассматривается один или несколько главных факторов риска, начиная с фактора наивысшего риска. *Типичные риски* включают в себя неправильно истолкованные требования, неправильно спроектированную архитектуру, потенциальные проблемы эксплуатации продукта или системы, проблемы в технологии и т.д.

При использовании спиральной модели при выполнении соответствующего ей проекта проявляются следующие ее *достоинства*:

1) сокращение и заблаговременное определение непреодолимых рисков, благодаря наличию действий по их анализу;

2) усовершенствование управления процессом разработки, затратами, соблюдением графика и кадровым обеспечением, что достигается путем выполнения анализа в конце каждой итерации.

При использовании спиральной модели применительно к неподходящему ей проекту проявляются следующие ее *недостатки*:

1) усложненность структуры модели, что приводит к сложности ее использования разработчиками, администраторами проекта и заказчиками;

2) необходимость высокопрофессиональных знаний для оценки рисков;

3) высокая стоимость модели за счет стоимости и дополнительных временных затрат на планирование, определение целей, выполнение анализа рисков и прототипирование при прохождении каждого цикла спирали; неоправданно высокая стоимость модели для проектов, имеющих низкую степень риска или небольшие размеры.

Применение спиральной модели Боэма целесообразно при разработке проектов в организации, обладающей навыками адаптации модели с учетом его сложности и критичности.

Резюме

Спиральная модель Боэма является одной из самых сложных моделей ЖЦ. Данная модель состоит из фаз разработки концепции, анализа требований, проектирования системы/продукта, реализации (технического проектирования, программирования и сборки), сопровождения и расширения функциональных возможностей. Модель поделена на четыре квадранта. В каждый квадрант входят основные и вспомогательные действия по разработке продукта или системы. Особое внимание в модели уделяется анализу и путям разрешения рисков.

2.5.6. Упрощенные варианты спиральной модели

Как было отмечено, основным недостатком модели Боэма является сложность. С учетом этого разработан ряд упрощенных спиральных моделей ЖЦ. Ниже приведены некоторые из данных моделей.

Модель Института качества SQI

Институтом качества SQI предлагается к использованию один из простейших вариантов спиральной модели. Данный вариант представлен на рис. 2.19. В этой модели жизненный цикл проекта разделен на четыре квадранта: «Планирование», «Анализ рисков», «Разработка», «Оценивание заказчиком». В пределах квадрантов выделяются только основные действия различного уровня.

Как и в других спиральных моделях, разработка начинается в центре модели с анализа начальных требований. Анализируются риски, связанные с требованиями, ищутся пути их устранения. На основе результатов разрешения рисков принимается решение о продолжении разработки. Создается и оценива-

ется заказчиком (пользователем) начальный прототип. По результатам оценки уточняются требования к разрабатываемому продукту (системе).

В каждом цикле разработки (витке спирали) выполняются аналогичные действия. При этом в каждом последующем витке спирали разрабатывается более полная версия продукта (системы) до получения полнофункционального продукта с характеристиками, удовлетворяющими заказчика (пользователя).

Таким образом, в данной модели устранена чрезмерная детализация процесса. Необходимый уровень детализации предусматривается при выполнении работ этапов планирования конкретного проекта.

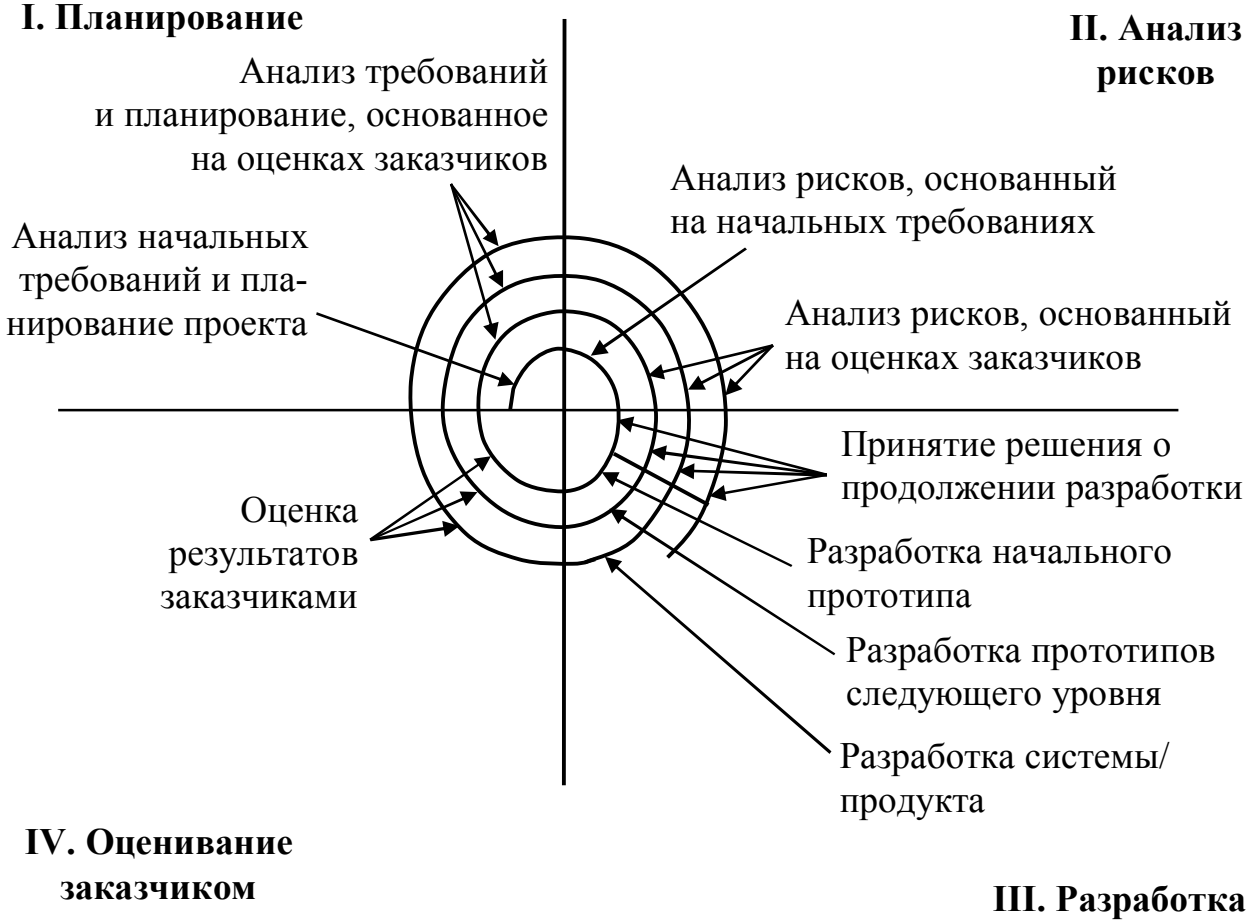


Рис. 2.19. Вариант спиральной модели Института качества SQI

Модель Института Управления проектами PMI

Институтом Управления проектами – PMI (Project Management Institute, США) предложен к использованию свой вариант упрощенной спиральной модели [36]. На рис. 2.20 данный вариант представлен в адаптированной к положениям стандарта *СТБ ИСО/МЭК 12207–2003* форме, учитывающей структуру процесса разработки (см. подразд. 1.2).

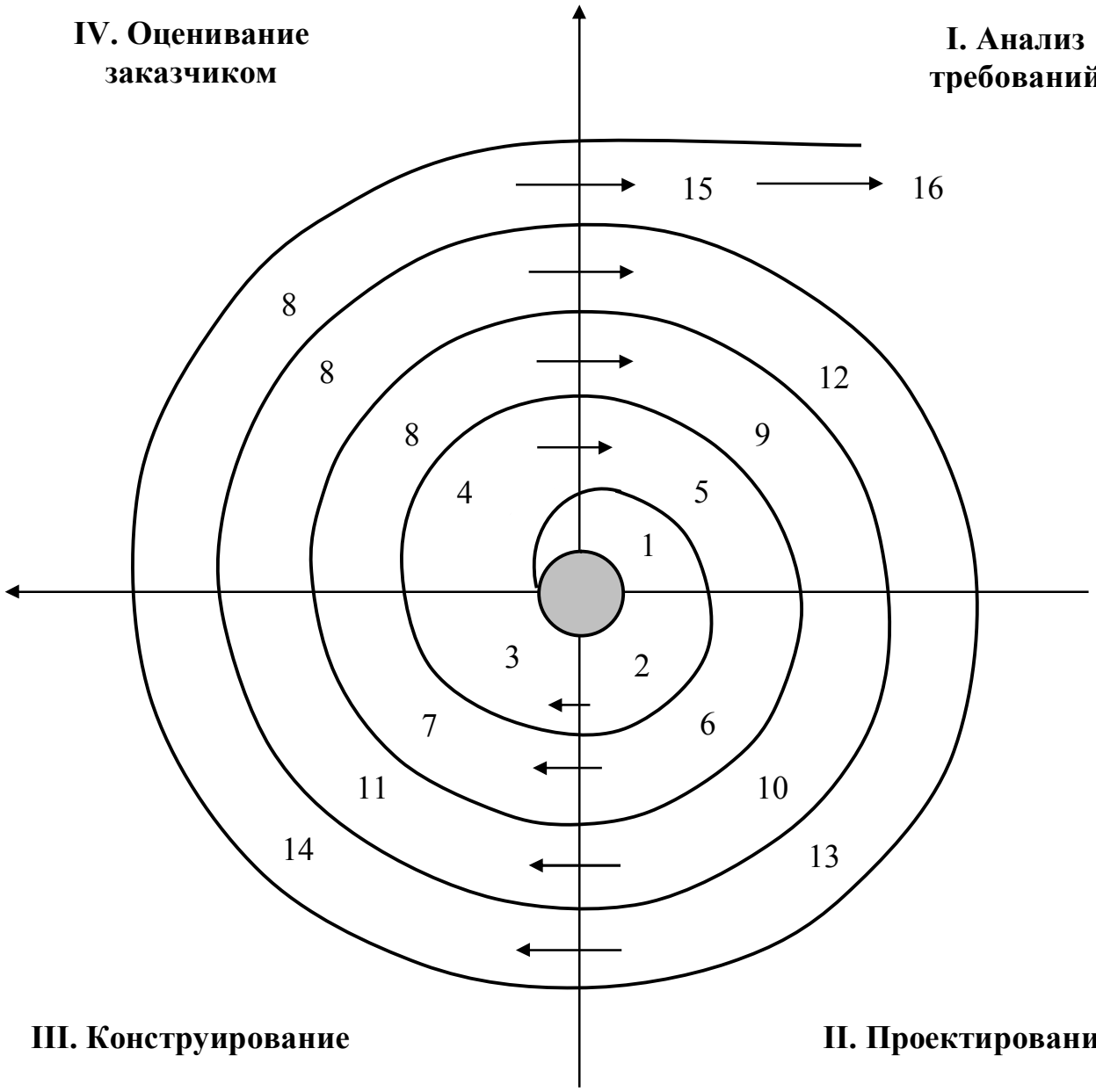


Рис. 2.20. Модель Института управления проектами PMI, адаптированная к требованиям стандарта СТБ ИСО/МЭК 12207–2003

В рассматриваемом варианте модели процесс разработки проекта разделен на четыре квадранта: «Анализ требований», «Проектирование», «Конструирование», «Оценивание заказчиком».

Модель базируется на применении четырех итерационных циклов:

- цикл доказательства концепции (внутренний виток спирали);
- цикл первой версии (второй виток спирали);
- цикл очередной версии (третий виток спирали);
- цикл конечной версии (внешний виток спирали).

В цикле доказательства концепции выполняются следующие этапы работ (см. рис. 2.20):

1 – анализ начальных требований (требований заказчика);

2 – концептуальное проектирование;

3 – конструирование концептуального прототипа;

4 – анализ рисков.

В цикле первой версии выполняются следующие этапы работ:

5 – анализ требований к системе;

6 – проектирование системы;

7 – конструирование первой версии;

8 – квалификационные испытания и оценка.

В цикле очередной версии выполняются следующие этапы работ:

9 – анализ требований к ПС системы;

10 – проектирование ПС;

11 – конструирование очередной версии,

а также этап 8 квалификационных испытаний и оценки.

В цикле конечной версии выполняются следующие этапы работ:

12 – анализ требований к программным модулям;

13 – проектирование программных модулей;

14 – конструирование конечной версии,

а также этап 8 квалификационных испытаний и оценки.

На рис. 2.20 также обозначены:

15 – ввод в действие и обеспечение приемки;

16 – эксплуатация и сопровождение.

Как и в других спиральных моделях действия, выполняемые в каждом витке спирали, осуществляются итерационно за один или несколько циклов разработки.

В квадранте «Анализ требований» выполняются действия, связанные с разработкой требований к результатам очередного цикла модели.

В квадранте «Проектирование» выполняются концептуальное проектирование, проектирование системы, проектирование ПС, программных компонентов и программных модулей.

В квадранте «Конструирование» выполняются кодирование, тестирование и сборка компонентов текущей версии программного средства (или системы) различного уровня.

В квадранте «Оценивание заказчиком» осуществляется оценка рисков проекта (в начале ЖЦ), квалификационные испытания промежуточных версий системы или программного продукта и оценка их результатов заказчиком с целью определения необходимости в переходе к следующему циклу разработки или повторению предыдущего цикла, квалификационные испытания конечного программного средства или системы в целом.

Модель «win-win»

На рис. 2.21 представлен модифицированный вариант спиральной модели под названием «win-win» (взаимный выигрыш), разработанный Боэмом в 1998 г. [33]. Целью данной модели является обеспечение таких условий выполнения проекта, при которых все его стороны оказываются в выигрыше. С учетом этого модель уделяет повышенное внимание участникам проекта (пользователям, заказчикам, разработчикам, тестировщикам и т.д.). Модель основана на постоянном согласовании всех работ ЖЦ разработки.

Рис. 2.21. Спиральная модель «win-win»

Каждый цикл в модели разделен на *шесть этапов*:

I – планирование работ уровня (цикла), обновление всего плана разработки и определение участников работ планируемого уровня;

II – определение условий, необходимых для успешного выполнения работ участниками уровня;

III – согласование условий успешного выполнения работ; анализ целей, ограничений и альтернативных вариантов уровня;

IV – оценка альтернативных вариантов уровня (в отношении как продукта, так и процесса), устранение или сокращение рисков;

V – разработка продукта текущего уровня;

VI – аттестация продукта и процесса текущего уровня; анализ и утверждение результатов уровня.

В каждом цикле разработки, исходя из запланированных работ данного цикла, определяется соответствующий состав исполнителей и оптимальные условия, необходимые для их работы. С учетом существующих ограничений анализируются и оцениваются возможные альтернативные варианты реализации цикла, оцениваются и разрешаются риски, связанные с каждым вариантом.

Исходя из результатов разрешения рисков выполняется разработка продукта текущего уровня, его аттестация и анализ заказчиком (пользователем). По результатам анализа пользователь готовит предложения по уточнению требований к продукту следующего уровня (цикла).

На основе данных предложений на следующем цикле выполняется разработка обновленных требований, реализуемых на данном уровне, с учетом чего планируются работы этого уровня.

К *достоинствам спиральной модели «win-win»* по отношению к другим спиральным моделям можно отнести:

1) ускорение разработки продуктов проекта благодаря содействию, оказываемому участникам проекта;

2) уменьшение стоимости продуктов проекта благодаря уменьшению объема переделок и ускорению разработки;

3) более высокий уровень удовлетворенности со стороны участников проекта;

4) как результат – более высокое качество разработанных продуктов.

Спиральная модель Консорциума по вопросам разработки программного обеспечения

Рис. 2.22 иллюстрирует структуру одного цикла спиральной модели, созданной Консорциумом по вопросам разработки программного обеспечения (Software Productivity Consortium) [33].

В данной модели каждый цикл разделен на *пять секторов*:

I – Определение проекта;

II – Анализ рисков;

III – Разработка плана проекта;

IV – Разработка продукта текущего уровня;

V – Управление и планирование.

Рис. 2.22. Спиральная модель Консорциума по вопросам разработки программного обеспечения

Результаты выполнения каждого этапа на рис. 2.22 изображены в прямоугольниках. Содержание работ каждого этапа указано в соответствующем секторе спирали.

На данном рисунке приняты следующие обозначения этапов проекта:

1 – Определение планируемого цикла проекта, в том числе:

- определение количества и состава участников работ;
- анализ целей;
- анализ альтернатив;
- анализ ограничений;

2 – Определение рисков;
3 – Оценка рисков;
4 – Планирование разрешения (сокращения и устранения) рисков;
5 – Оценка разрешения рисков;
6 – Оценка альтернатив с учетом результатов оценки разрешения рисков;
7 – Планирование и разработка графиков проекта;
8 – Разработка и верификация продукта;
9 – Мониторинг (надзор) и оценка продукта;
10 – Анализ необходимости изменений версии;
11 – Оценка изменений;
12 – Обновление плана проекта для следующего цикла (витка спирали).

В каждом цикле проекта на основе целей цикла анализируются альтернативные варианты реализации продукта данного уровня и ограничения, связанные с каждым вариантом. Оцениваются риски, связанные с альтернативами.

С учетом этого выбирается конкретный альтернативный вариант и создается план разработки. Разрабатывается текущая версия продукта. По результатам ее оценки определяется необходимость изменений версии, уточняются требования к продукту следующего цикла, обновляется план следующего цикла.

Таким образом, в рассматриваемой модели из пяти секторов витка спирали непосредственной разработке продукта посвящен один сектор. Остальные направлены на усовершенствование управления и планирования проекта, оценку продукта, своевременное выявление, ранжирование и разрешение рисков. Это приводит к улучшению условий выполнения проекта, сокращению потерь, связанных с внесением изменений в продукт и как следствие – улучшению качества работ проекта и конечного продукта.

Компонентно-ориентированная спиральная модель

На рис. 2.23 представлен вариант спиральной модели, называемый компонентно-ориентированной моделью [30]. В этой модели основное внимание уделяется процессу разработки. Модель ориентирована на повторное использование существующих программных компонентов.

Программные компоненты, созданные в предыдущих проектах или предыдущих циклах текущего проекта, хранятся в специальной библиотеке. В каждом цикле текущего проекта, исходя из требований, идентифицируются и ищутся в библиотеке кандидаты в компоненты. Они используются в проекте повторно. Если таких кандидатов не выявлено, разрабатываются и включаются в библиотеку новые компоненты.

К *достоинствам* компонентно-ориентированной спиральной модели относятся сокращение длительности и стоимости разработки конечного продукта за счет повторного использования компонентов.

Очевидно, что данную модель можно совместить с другими видами спиральных моделей.

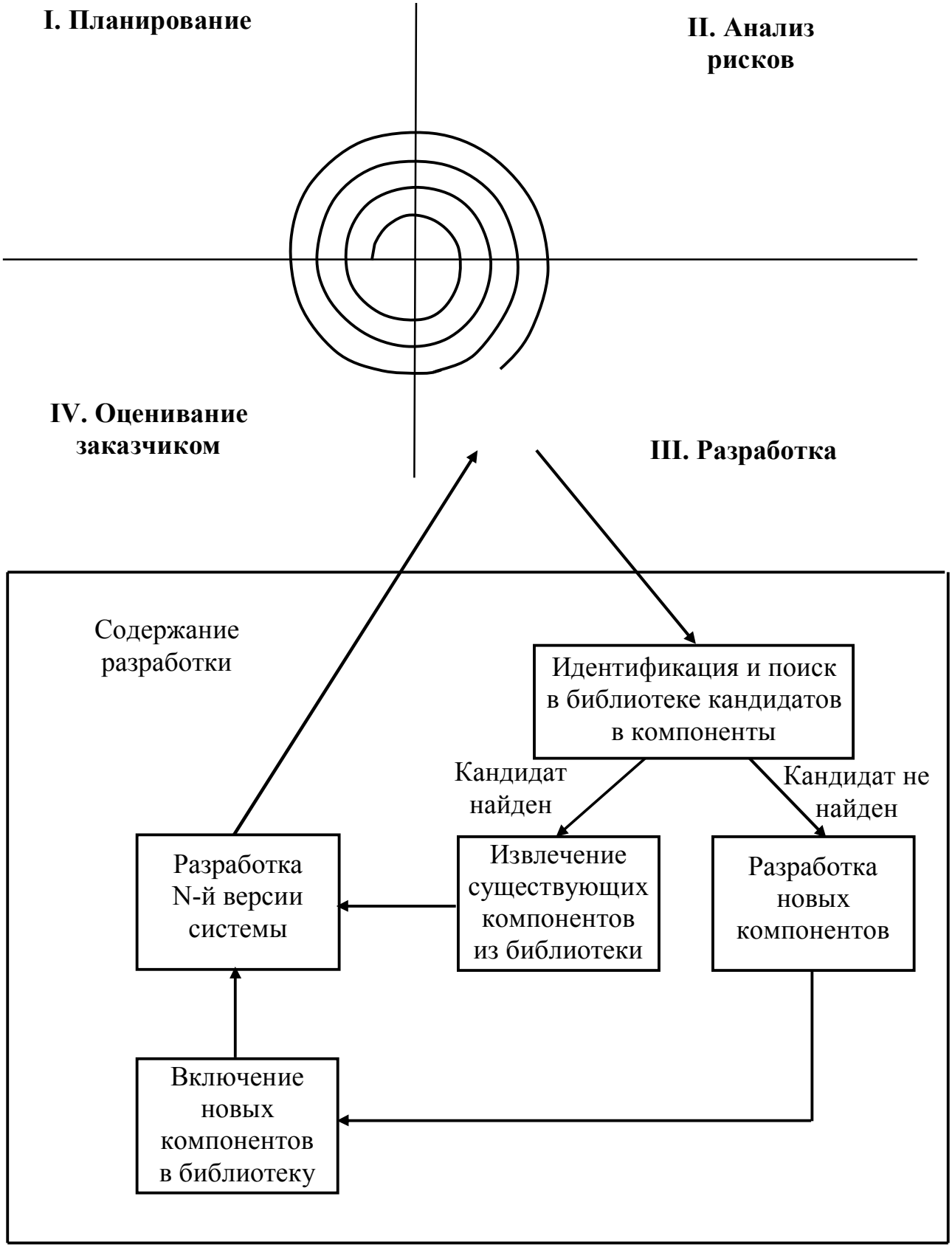


Рис. 2.23. Вариант компонентно-ориентированной спиральной модели

Достоинства упрощенных спиральных моделей

Обобщая написанное выше об упрощенных спиральных моделях, можно выделить общие для них *достоинства*:

1) существенное упрощение по сравнению с базовой спиральной моделью Боэма делает данные модели интуитивно более понятными заказчикам, разработчикам и администраторам проекта, а также уменьшает стоимость проектов;

2) повышенное внимание во всех упрощенных спиральных моделях уделяется действиям, непосредственно не связанным с разработкой (анализ рисков, планирование, работа с персоналом и т.п.). Непосредственная разработка в большинстве моделей занимает не более одного-двух секторов модели. Это обеспечивает более высокое качество процессов и работ проекта и продукта в целом, упрощает прогнозирование сроков и стоимости разработки, повышает удовлетворенность заказчика результатами работ.

Резюме

Упрощенные варианты спиральной модели созданы для снижения сложности базовой спиральной модели Боэма за счет устранения излишней детализации модели. В данных моделях процесс разделен на 4 – 6 секторов, из которых только один-два посвящены непосредственно разработке продукта. Остальные сектора направлены на анализ рисков, планирование, оценки результатов заказчиками, обеспечение условий успешного выполнения работ – то есть действия, влияющие как на качество процессов ЖЦ ПС и систем, так и на качество продуктов разработки.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Назовите базовые стратегии разработки ПС и систем.
2. Охарактеризуйте сущность каскадной стратегии разработки ПС и систем, перечислите достоинства, недостатки и области применения данной стратегии.
3. Охарактеризуйте сущность инкрементной стратегии разработки ПС и систем, перечислите достоинства, недостатки и области применения данной стратегии.
4. Охарактеризуйте сущность эволюционной стратегии разработки ПС и систем, перечислите достоинства, недостатки и области применения данной стратегии.
5. Дайте сравнительную характеристику каскадной, инкрементной и эволюционной стратегий разработки ПС и систем.
6. Назовите общие черты каскадных моделей жизненного цикла.
7. Изобразите и охарактеризуйте классическую каскадную модель ЖЦ.
8. Изобразите и охарактеризуйте каскадную модель ЖЦ с обратными связями. В чем заключаются ее преимущества и недостатки по сравнению с классической каскадной моделью?
9. Изобразите и охарактеризуйте каскадную модель ЖЦ, рекомендуемую ГОСТ Р ИСО/МЭК ТО 15271–2002. В чем заключаются ее особенности по сравнению с классической каскадной моделью?
10. Изобразите и охарактеризуйте V-образную модель ЖЦ. В чем заключаются ее отличия, преимущества и недостатки по сравнению с классической каскадной моделью?
11. Изобразите и охарактеризуйте V-образную модель ЖЦ с обратными связями. В чем заключаются ее преимущества и недостатки по сравнению с V-образной моделью без обратных связей?
12. Назовите основные черты RAD-моделей ЖЦ.
13. Изобразите и охарактеризуйте базовую RAD-модель ЖЦ. В чем заключаются ее отличия, преимущества и недостатки по сравнению с классической каскадной моделью?
14. Изобразите и охарактеризуйте RAD-модель ЖЦ, основанную на моделировании предметной области. В чем заключаются ее отличия, преимущества и недостатки по сравнению с базовой RAD-моделью?
15. Изобразите и охарактеризуйте RAD-модель параллельной разработки приложений. В чем заключаются ее особенности по сравнению с базовой RAD-моделью?
16. Изобразите и охарактеризуйте RAD-модель ЖЦ, рекомендованную ГОСТ Р ИСО/МЭК ТО 15271–2002. В чем заключаются ее особенности по сравнению с базовой RAD-моделью?
17. Перечислите основные достоинства, недостатки и области использования RAD-моделей.

18. Назовите общие черты инкрементных моделей ЖЦ.
19. Изобразите и охарактеризуйте инкрементную модель ЖЦ с уточнением требований на начальных этапах разработки. В чем заключаются ее особенности по сравнению с классической каскадной моделью?
20. Изобразите и охарактеризуйте инкрементную модель ЖЦ, рекомендованную ГОСТ Р ИСО/МЭК ТО 15271–2002. В чем заключаются ее особенности по сравнению с классической каскадной моделью?
21. Изобразите и охарактеризуйте инкрементную модель экстремального программирования. В чем заключаются ее особенности по сравнению с классической каскадной моделью?
22. Назовите общие черты эволюционных моделей ЖЦ.
23. Изобразите и охарактеризуйте эволюционную модель ЖЦ, рекомендованную ГОСТ Р ИСО/МЭК ТО 15271–2002. В чем заключаются ее особенности по сравнению с инкрементной моделью, приведенной в данном стандарте?
24. Изобразите и охарактеризуйте структурную эволюционную модель быстрого прототипирования. В чем заключаются ее особенности по сравнению с другими эволюционными моделями жизненного цикла?
25. Изобразите и охарактеризуйте эволюционную модель прототипирования, рекомендованную ГОСТ Р ИСО/МЭК ТО 15271–2002. В чем заключаются ее особенности по сравнению с другими эволюционными моделями ЖЦ?
26. Изобразите и охарактеризуйте спиральную модель Боэма. Перечислите фазы и квадранты данной модели. Назовите достоинства и недостатки данной модели ЖЦ.
27. Изобразите и охарактеризуйте спиральную модель Института качества SQI. В чем заключаются ее особенности, достоинства и недостатки по сравнению с базовой спиральной моделью ЖЦ Боэма?
28. Изобразите и охарактеризуйте спиральную модель Института Управления проектами PMI. В чем заключаются ее особенности, достоинства и недостатки по сравнению с базовой спиральной моделью ЖЦ Боэма?
29. Изобразите и охарактеризуйте спиральную модель «win-win». В чем заключаются ее особенности, достоинства и недостатки по сравнению с базовой спиральной моделью ЖЦ Боэма?
30. Изобразите и охарактеризуйте спиральную модель Консорциума по вопросам разработки программного обеспечения. В чем заключаются ее особенности, достоинства и недостатки по сравнению с базовой спиральной моделью ЖЦ Боэма?
31. Изобразите и охарактеризуйте компонентно-ориентированную спиральную модель ЖЦ. В чем заключаются ее особенности, достоинства и недостатки по сравнению с другими спиральными моделями?

РАЗДЕЛ 3. ВЫБОР МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА ДЛЯ КОНКРЕТНОГО ПРОЕКТА

3.1. Классификация проектов по разработке программных средств и систем

Существуют различные схемы классификации проектов, связанных с разработкой ПС и систем.

В ***ГОСТ Р ИСО/МЭК ТО 12182–2002 – Информационная технология. Классификация программных средств*** [7] – приведена схема классификации ПС по 16 видам, каждый из которых подразделяется на классы. Данная классификация имеет общий характер и в целом не может использоваться для обоснования выбора модели ЖЦ ПС и систем.

Институтом качества программного обеспечения SQI (Software Quality Institute, США) специально для выбора модели ЖЦ предложена схема классификации проектов по разработке ПС и систем [33]. Основу данной классификации составляют *четыре категории критериев*. По каждому критерию проекты подразделяются на *два альтернативных класса*.

Критерии классификации проектов, предложенные Институтом SQI, объединены в следующие категории.

1. Характеристики требований к проекту

Критерии данной категории классифицируют проекты в зависимости от свойств требований пользователя (заказчика) к разрабатываемой системе или программному средству. Например, известны ли требования к началу проекта, сложны ли они, будут ли они изменяться.

2. Характеристики команды разработчиков

Чтобы иметь возможность пользоваться критериями данной категории классификации проектов, состав команды разработчиков необходимо сформировать до выбора модели ЖЦ. Характеристики команды разработчиков играют важную роль при выборе модели ЖЦ, поскольку разработчики несут основную ответственность за успешную реализацию проекта. В первую очередь следует

учитывать квалификацию разработчиков, их знакомство с предметной областью, инструментальными средствами разработки и т.п.

3. Характеристики пользователей (заказчиков)

Чтобы иметь возможность пользоваться критериями данной категории классификации проектов, до выбора модели ЖЦ необходимо определить возможную степень участия пользователей (заказчиков) в процессе разработки и их взаимосвязь с командой разработчиков на протяжении проекта.

Это важно, поскольку отдельные модели ЖЦ требуют усиленного участия пользователей в процессе разработки. Выбор таких моделей при отсутствии реальной возможности пользователей участвовать в проекте приведет к существенному снижению качества результатов разработки.

4. Характеристики типов проектов и рисков

В некоторых моделях в достаточно высокой степени предусмотрено управление рисками. В других моделях управление рисками вообще не предусматривается. Поэтому при выборе модели ЖЦ следует учитывать реальные риски проекта, критичность и сложность продуктов разработки.

Критерии данной категории отражают различные виды рисков, в том числе связанные со сложностью проекта, достаточностью ресурсов для его исполнения, учитывают график проекта и т.д. С учетом этого обеспечивается выбор модели, минимизирующей выявленные риски.

Примеры критериев каждой из категорий приведены в подразд. 3.2.

Следует отметить, что данная классификация проектов, направленная на обоснованный выбор модели ЖЦ, применима для *достаточно масштабных проектов* по разработке ПС и систем. Использование данной классификации и основанной на ней процедуры выбора модели ЖЦ (см. подразд. 3.2) для небольших проектов может привести к чрезмерному увеличению графика проекта, дополнительным затратам и дополнительным рискам.

Таким образом, руководитель проекта должен чётко представлять себе масштаб проекта и необходимость его классификации для выбора модели ЖЦ в соответствии с критериями, предложенными Институтом SQI.

Резюме

Институтом качества программного обеспечения SQI (США) специально для выбора модели ЖЦ разработана схема классификации проектов по разработке ПС и систем. Основу данной классификации составляют четыре категории критериев: характеристики требований к проекту, характеристики команды разработчиков, характеристики пользователей (заказчиков), характеристики типов проектов и рисков. По каждому из критериев проекты подразделяются на два альтернативных класса. Данная классификация предназначена для достаточно крупных проектов.

3.2. Процедура выбора модели жизненного цикла программных средств и систем

Для выбора подходящей к условиям конкретного проекта модели ЖЦ ПС и систем Институтом качества программного обеспечения SQI рекомендуется использовать специальную процедуру [33]. Данная процедура базируется на применении четырех таблиц вопросов. Примеры вопросов приведены в табл. 3.1 – 3.4.

Таблица 3.1

Выбор модели жизненного цикла на основе характеристик требований

№ критерия	Критерии категории требований	Каскадная	V-образная	RAD	Инкрементная	Быстрого прототипирования	Эволюционная
1.	Являются ли требования к проекту легко определимыми и реализуемыми?	Да	Да	Да	Нет	Нет	Нет
2.	Могут ли требования быть сформулированы в начале ЖЦ?	Да	Да	Да	Да	Нет	Нет
3.	Часто ли будут изменяться требования на протяжении ЖЦ?	Нет	Нет	Нет	Нет	Да	Да
4.	Нужно ли демонстрировать требования с целью их определения?	Нет	Нет	Да	Нет	Да	Да
5.	Требуется ли проверка концепции программного средства или системы?	Нет	Нет	Да	Нет	Да	Да
6.	Будут ли требования изменяться или уточняться с ростом сложности системы (программного средства) в ЖЦ?	Нет	Нет	Нет	Да	Да	Да
7.	Нужно ли реализовать основные требования на ранних этапах разработки?	Нет	Нет	Да	Да	Да	Да

Таблица 3.2

Выбор модели жизненного цикла
на основе характеристик команды разработчиков

№ критерия	Критерии категории команды разработчиков проекта	Каскадная	V-образная	RAD	Инкрементная	Быстрого прототипирования	Эволюционная
1.	Являются ли проблемы предметной области проекта новыми для большинства разработчиков?	Нет	Нет	Нет	Нет	Да	Да
2.	Являются ли инструментальные средства, используемые в проекте, новыми для большинства разработчиков?	Да	Да	Нет	Нет	Нет	Да
3.	Изменяются ли роли участников проекта на протяжении ЖЦ?	Нет	Нет	Нет	Да	Да	Да
4.	Является ли структура процесса разработки более значимой для разработчиков, чем гибкость?	Да	Да	Нет	Да	Нет	Нет
5.	Важна ли легкость распределения человеческих ресурсов проекта?	Да	Да	Да	Да	Нет	Нет
6.	Приемлет ли команда разработчиков оценки, проверки, стадии разработки?	Да	Да	Нет	Да	Да	Да

Каждая из табл. 3.1 – 3.4 представляет одну из категорий классификации проектов (см. подразд. 3.1). Каждый из вопросов (строка в таблице) предназначен для классификации анализируемого проекта по определенному критерию категории. Столбцы данных таблиц соответствуют обобщенным моделям ЖЦ, фактически представляющим стратегии разработки ПС. При этом под RAD-моделью подразумевается независимая RAD-модель, не встроенная в другие модели жизненного цикла (см. пп. 2.3.1, 2.3.2).

Рассматриваемая процедура состоит из следующей последовательности шагов.

Таблица 3.3

Выбор модели жизненного цикла
на основе характеристик коллектива пользователей

№ критерия	Критерии категории коллектива пользователей	Каскадная	V-образная	RAD	Инкрементная	Быстрого прототипирования	Эволюционная
1.	Будет ли присутствие пользователей ограничено в ЖЦ разработки?	Да	Да	Нет	Да	Нет	Да
2.	Будут ли пользователи оценивать текущее состояние программного продукта (системы) в процессе разработки?	Нет	Нет	Нет	Да	Да	Да
3.	Будут ли пользователи вовлечены во все фазы ЖЦ разработки?	Нет	Нет	Да	Нет	Да	Нет
4.	Будет ли заказчик отслеживать ход выполнения проекта?	Нет	Нет	Нет	Нет	Да	Да

1-й шаг. Проанализировать отличительные черты проекта по критериям категорий, представленным в виде вопросов.

2-й шаг. Ответить на вопросы по анализируемому проекту, отметив слова «да» или «нет» в соответствующих строках табл. 3.1 – 3.4. Если слов «да» или «нет» в строке несколько, необходимо отметить все из них (все «да» или все «нет»).

В качестве примера в табл. 3.1 выделены варианты ответов для проекта разработки сложного и критичного программного средства, требования к которому заранее не известны и будут уточняться по ходу разработки.

3-й шаг. Расположить по степени важности категории (таблицы) и/или критерии, относящиеся к каждой категории (вопросы внутри таблиц), относительно проекта, для которого выбирается модель ЖЦ.

4-й шаг. Выбрать из моделей (см. табл. 3.1 – 3.4) ту модель, которая соответствует столбцу с наибольшим количеством отмеченных ответов с учетом их степени важности (с наибольшим количеством отмеченных ответов в верхней части приоритетных таблиц). Выбранная модель ЖЦ является наиболее приемлемой для анализируемого проекта.

Таблица 3.4

Выбор модели жизненного цикла
на основе характеристик типа проектов и рисков

№ криерия	Критерии категории типов проекта и рисков	Каскадная	V-образная	RAD	Инкре-ментная	Быстрого прототипи-рования	Эволюци-онная
1.	Разрабатывается ли в проекте продукт нового для организации направления?	Нет	Нет	Нет	Да	Да	Да
2.	Будет ли проект являться расширением существующей системы?	Да	Да	Да	Да	Нет	Нет
3.	Будет ли проект крупно- или среднемасштабным?	Нет	Нет	Нет	Да	Да	Да
4.	Ожидается ли длительная эксплуатация продукта?	Да	Да	Нет	Да	Нет	Да
5.	Необходим ли высокий уровень надежности продукта проекта?	Нет	Да	Нет	Да	Нет	Да
6.	Предполагается ли эволюция продукта проекта в течение ЖЦ?	Нет	Нет	Нет	Да	Да	Да
7.	Велика ли вероятность изменения системы (продукта) на этапе сопровождения?	Нет	Нет	Нет	Да	Да	Да
8.	Является ли график сжатым?	Нет	Нет	Да	Да	Да	Да
9.	Предполагается ли повторное использование компонентов?	Нет	Нет	Да	Да	Да	Да
10.	Являются ли достаточными ресурсы (время, деньги, инструменты, персонал)?	Нет	Нет	Нет	Нет	Да	Да

Для рассмотренного примера на основе результатов заполнения табл. 3.1 наиболее подходящими являются модели быстрого прототипирования и эволюционные модели. Уточнение выбора модели следует производить по результатам анализа табл. 3.1 – 3.4.

Резюме

Процедура выбора модели ЖЦ ПС и систем Института SQI базируется на применении четырех таблиц вопросов, соответствующих предложенной данным институтом классификации проектов. По каждому из вопросов необходимо выделить ответы, соответствующие конкретному проекту, и выбрать модель с наибольшим количеством отмеченных ответов.

3.3. Адаптация модели жизненного цикла разработки программных средств и систем к условиям конкретного проекта

Как отмечалось в подразд. 1.2, основным стандартом в области ЖЦ ПС и систем в Республике Беларусь является стандарт *СТБ ИСО/МЭК 12207–2003*.

В соответствии с данным стандартом выбор модели ЖЦ должен осуществляться при выполнении первой работы процесса разработки (подготовка процесса, см. подразд. 1.2). В соответствии с положениями стандарта, если модель ЖЦ ПС не определена в договоре, то разработчик должен определить или выбрать модель, соответствующую области реализации, величине и сложности проекта.

Выбор подходящей модели ЖЦ – это *первая стадия* применения модели в конкретном проекте. На этой стадии может использоваться процедура выбора модели ЖЦ ПС и систем Института SQI, рассмотренная в подразд. 3.2.

Вторая стадия заключается в адаптации выбранной модели к потребностям данного проекта, к процессу разработки, принятому в данной организации, и к требованиям действующих стандартов.

С учетом этого должны быть выбраны и структурированы в модель ЖЦ ПС работы и задачи процесса разработки из стандарта *СТБ ИСО/МЭК 12207–2003*. Данные работы и задачи могут пересекаться, взаимодействовать, выполняться итерационно или рекурсивно.

В соответствии с положениями стандартов *СТБ ИСО/МЭК 12207–2003* и *ГОСТ Р ИСО/МЭК ТО 15271–2002* вопросы структурирования в выбранной модели работ и задач процесса разработки должны решаться с учетом следующих ***характеристик проекта***, определенных в данных стандартах [9, 8]:

1) *организационные подходы*, принятые в организации (например связанные с защитой, безопасностью, конфиденциальностью, управлением рисками, использованием независимого органа по верификации и аттестации, использованием конкретного языка программирования, обеспечением техническими ресурсами); действия, связанные с особенностями реализации данных подходов, должны быть структурированы в модель ЖЦ;

2) *политика заказа* (например типы договора, необходимость использования процесса поставки, использование услуг сторонних разработчиков или субподрядчиков, которых необходимо контролировать);

3) *политика сопровождения программных средств* (например ожидаемые период сопровождения и периодичность внесения изменений, критичность применения, квалификация персонала сопровождения, необходимая для сопровождения среда);

4) *вовлеченные стороны* (например заказчик, поставщик, разработчик, субподрядчик, посредники по верификации и аттестации, персонал сопровождения; численность сторон); большое количество сторон вызывает необходимость структурирования в модель ЖЦ работ, связанных с усиленным административным контролем;

5) *работы жизненного цикла системы* (например подготовка проекта заказчиком, разработка и сопровождение поставщиком); в модель ЖЦ должны быть структурированы работы соответствующих процессов;

6) *характеристики системного уровня* (например количество подсистем и объектов конфигурации, межсистемные и внутрисистемные интерфейсы, интерфейсы пользователя, оценка временных ограничений, наличие реализованных техническими средствами программ); в модели ЖЦ должны быть учтены работы процесса разработки, относящиеся к системному уровню;

7) *характеристики программного уровня* (например, количество программных объектов, типы документов, характеристики качества ПС по *ISO/IEC 9126–1:2001* [6] и СТБ ИСО/МЭК 9126–2003 [10], типы и объемы программных продуктов); выделяются следующие ***типы программных продуктов***:

- новая разработка; при адаптации модели ЖЦ должны учитываться все требования к процессу разработки;
- использование готового программного продукта; при выборе и адаптации модели ЖЦ следует учесть, что на ее первом этапе должна быть выполнена оценка функциональных характеристик, документации, применимости, возможность поддержки готового продукта; процесс разработки может не понадобиться;
- модификация готового программного продукта; при выборе и адаптации модели ЖЦ следует учесть, что на ее первом этапе должна быть выполнена оценка функциональных характеристик, документации, применимости, возможность поддержки готового продукта; процесс разработки реализуется с учетом критичности продукта и величины изменений;
- программный или программно-аппаратный продукт, встроенный или подключенный к системе; в модели ЖЦ необходимо учитывать работы процесса разработки, связанные с системой (работы 2, 3, 10, 11, см. подразд. 1.2);
- отдельно поставляемый программный продукт; не требуется учитывать работы процесса разработки, связанные с системой;

- непоставляемый программный продукт; требования стандарта *СТБ ИСО/МЭК 12207–2003* можно не учитывать;

8) *объем проекта* (в больших проектах, в которые вовлечены десятки или сотни лиц, необходим тщательный административный надзор и контроль с применением процессов совместного анализа, аудита, верификации, аттестации, обеспечения качества; данные процессы следует учесть в модели ЖЦ; для малых проектов такие методы контроля могут быть излишними);

9) *критичность проекта* (значительная зависимость работы системы от правильного функционирования ПС и своевременности выдачи результатов); для таких продуктов характерны повышенные требования к их качеству, поэтому необходим более тщательный надзор и контроль хода выполнения проекта; в модели жизненного цикла следует учитывать процессы верификации, аттестации, обеспечения качества;

10) *технические риски* (например, создание уникального или сложного программного средства, которое трудно сопровождать и использовать, неправильное, неточное или неполное определение требований); в таких случаях в модель ЖЦ следует структурировать действия по разрешению рисков или использовать модели, в которых анализ и разрешение рисков являются их составными частями (например, спиральные модели, см. пп. 2.5.5, 2.5.6);

11) *другие характеристики* (например, усиленный административный контроль за критичными или большими программными продуктами, требующий применения постоянных оценок).

В стандарте *ГОСТ Р ИСО/МЭК ТО 15271–2002* [8] приведен пример модели ЖЦ из со структурированными в нее работами процесса разработки. Данная модель рассмотрена в п. 2.3.4 (см. рис. 2.11).

Резюме

Стандарт *СТБ ИСО/МЭК 12207–2003* определяет, что выбор модели ЖЦ должен осуществляться при выполнении первой работы процесса разработки (подготовка процесса) программного средства или системы. Выбранная модель должна быть адаптирована к потребностям данного проекта, к процессу разработки, принятому в данной организации, и к требованиям действующих стандартов. На адаптацию модели влияют характеристики проекта, определенные в стандартах *СТБ ИСО/МЭК 12207–2003* и *ГОСТ Р ИСО/МЭК ТО 15271–2002*.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Охарактеризуйте схему классификации проектов по разработке ПС и систем, предложенную Институтом качества программного обеспечения SQI для выбора модели ЖЦ.
2. Перечислите категории критериев, положенных в основу схемы классификации проектов Института SQI.
3. Перечислите шаги процедуры выбора модели ЖЦ ПС и систем, предложенной Институтом SQI.
4. Назовите критерии категории характеристик требований к проекту.
5. Назовите критерии категории характеристик команды разработчиков.
6. Назовите критерии категории характеристик пользователей (заказчиков).
7. Назовите критерии категории характеристик типов проектов и рисков.
8. В чем заключается суть адаптации выбранной модели ЖЦ к потребностям конкретного проекта, определенная в стандарте *СТБ ИСО/МЭК 12207–2003*?
9. Перечислите характеристики проекта, влияющие на адаптацию выбранной модели ЖЦ к потребностям данного проекта.
10. Перечислите типы программных продуктов, влияющие на адаптацию выбранной модели ЖЦ к потребностям конкретного проекта.

РАЗДЕЛ 4. КЛАССИЧЕСКИЕ МЕТОДОЛОГИИ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ

4.1. Структурное программирование

В 70-х гг. XX в. возник новый подход к разработке алгоритмов и программ, который получил название *структурного программирования*. Одним из первых инициаторов структурного программирования был профессор Э. Дейкстра. В 1965 г. он высказал предположение, что оператор GoTo (оператор безусловного перехода) вообще может быть исключён из языков программирования. По мнению Дейкстры, «квалификация программиста обратно пропорциональна числу операторов GoTo в его программах».

Достоинства структурного программирования по сравнению с интуитивным неструктурным программированием [23]:

1) уменьшение трудностей тестирования программ;
2) повышение производительности труда программистов;
3) повышение ясности и читабельности программ, что упрощает их сопровождение;
4) повышение эффективности объектного кода программ как с точки зрения времени их выполнения, так и с точки зрения необходимых затрат памяти.

4.1.1. Основные положения структурного программирования

К *концепциям* структурного программирования относятся:

- отказ от использования оператора безусловного перехода (GoTo);
- применение фиксированного набора управляющих конструкций;
- использование метода нисходящего проектирования (данный метод рассмотрен в подразд. 4.3).

В основу структурного программирования положено *требование*, чтобы каждый модуль алгоритма (программы) проектировался с единственным входом и единственным выходом. Программа представляется в виде множества *вложенных* модулей, каждый из которых имеет один вход и один выход [23].

Основой реализации структурированных программ является *принцип*

Бома и Джакопини, в соответствии с которым любая программа может быть разработана с использованием лишь *трех базовых структур*:

1) функционального блока;
2) конструкции принятия двоичного (дихотомического) решения;
3) конструкции обобщенного цикла.

Функциональный блок – это отдельный вычислительный оператор или любая последовательность вычислений с единственным входом и единственным выходом. Изображается с помощью символа «Процесс» (рис. 4.1).

Рис. 4.1. Изображение функционального блока
в структурном программировании

Конструкция принятия двоичного (дихотомического) решения называется также конструкцией If-Then-Else (если-то-иначе), разветвлением или ветвлением. Это структура, обеспечивающая выбор между двумя альтернативными путями вычислительного процесса в зависимости от выполнения некоторого условия. Изображается с помощью символов «Решение» и «Процесс» (рис. 4.2).

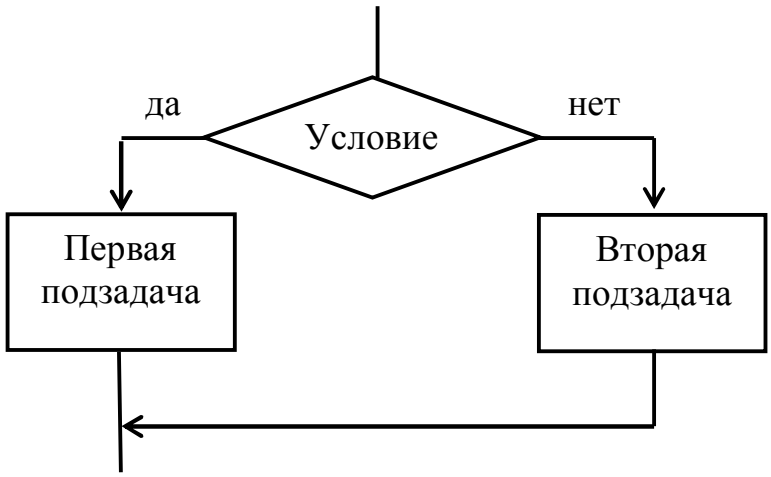


Рис. 4.2. Изображение конструкции If-Then-Else
в структурном программировании

Конструкция обобщенного цикла – в качестве базовой конструкции структурного программирования используется цикл с предусловием, называе-

мый циклом «Пока» (пока условие истинно, тело цикла выполняется). Изображается с помощью символов «Решение» и «Процесс» (рис. 4.3).

Из рис. 4.2, 4.3 видно, что логические конструкции принятия двоичного решения и обобщенного цикла имеют только один вход и один выход. Поэтому они могут рассматриваться как функциональные блоки. С учётом этого вводится *преобразование логических блоков в функциональный блок.*

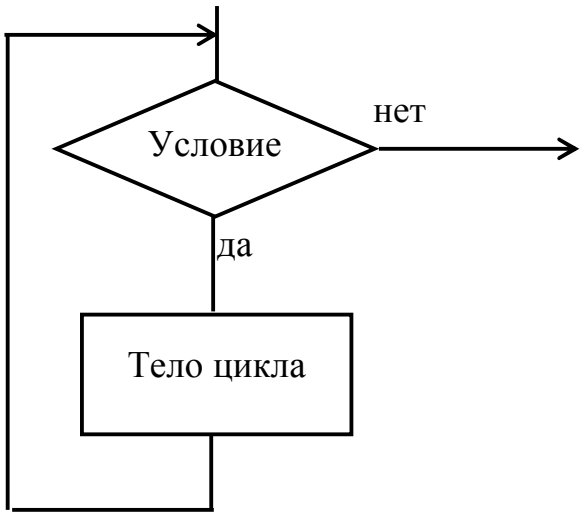


Рис. 4.3. Изображение конструкции обобщенного цикла
в структурном программировании

Кроме того, всякая последовательность функциональных блоков, называемая *конструкцией следования* (рис. 4.4), также может быть приведена к одному функциональному блоку.

Рис. 4.4. Конструкция следования

Преобразования логических блоков и конструкции следования в один функциональный блок называются *преобразованиями Бома–Джакопини.* Их основу составляет принцип «чёрного ящика» (часть алгоритма или программы, реализующая некоторую функцию, с одним входом и одним выходом).

Таким образом, всякая программа, состоящая из функциональных блоков, операторов цикла с предусловием и операторов If-Then-Else, поддаётся последовательному преобразованию к единственному функциональному блоку. Эта последовательность преобразований может быть использована как средство понимания программы, подход к доказательству ее правильности и *структури-*

рованности. Обратная последовательность преобразований может быть использована в процессе проектирования алгоритма (программы) по методу *нисходящего проектирования* – алгоритм (программа) разрабатывается, исходя из единственного функционального блока, который постепенно детализируется в сложную структуру основных элементов (см. подразд. 4.3).

Резюме

Структурное программирование базируется на концепциях отказа от использования оператора безусловного перехода, применения фиксированного набора управляющих конструкций; использования метода нисходящего проектирования. Программа или схема алгоритма представляется в виде совокупности вложенных модулей, каждый из которых имеет один вход и один выход. В соответствии с принципом Бома–Джакопини любая программа может быть разработана с использованием лишь трех базовых структур: функционального блока, конструкции принятия двоичного решения, конструкции обобщенного цикла. Преобразования Бома–Джакопини могут быть использованы в качестве средства доказательства структурированности программ.

4.1.2. Реализация основ структурного программирования в языках программирования

Реализация теоретических основ структурного программирования при разработке программ на конкретных языках программирования базируется на следующих *правилах*: все операторы в программе должны представлять собой либо непосредственно исполняемые в линейном порядке функциональные операторы, либо следующие *управляющие конструкции* [23]:

1) вызовы подпрограмм – любое допустимое на конкретном языке программирования обращение к замкнутой подпрограмме с одним входом и одним выходом;

2) вложенные на произвольную глубину операторы If-Then-Else;

3) циклические операторы (цикл с предусловием).

Этих средств достаточно для составления структурированных программ. Однако иногда допускаются *расширения* данных конструкций:

1) дополнительные конструкции организации цикла:
   - цикл с параметром как вариант цикла с предусловием;
   - цикл с постусловием, называемый в структурном программировании циклом «До», в котором тело цикла выполняется перед проверкой условия выхода из цикла и повторяется до выполнения условия; изображается, как показано на рис. 4.5;

2) использование оператора Case как расширения конструкции If-Then-Else; в структурном программировании конструкция Case представляется в соответствии с рис. 4.6.

3) подпрограммы с несколькими входами или несколькими выходами (например один выход нормальный, второй – по ошибке), если это допускается отраслевыми стандартами или стандартами предприятия;

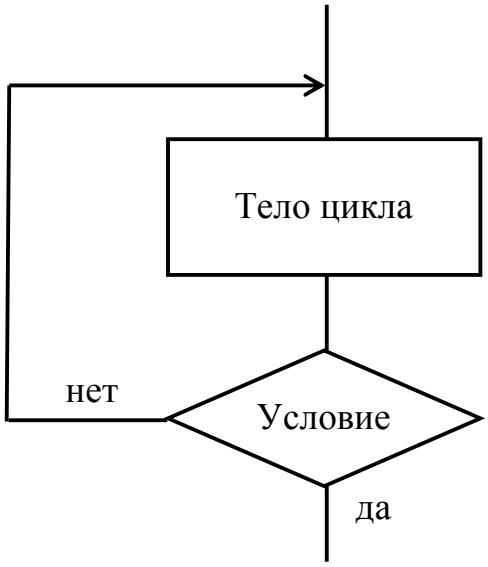


Рис. 4.5. Изображение цикла с постусловием
в структурном программировании

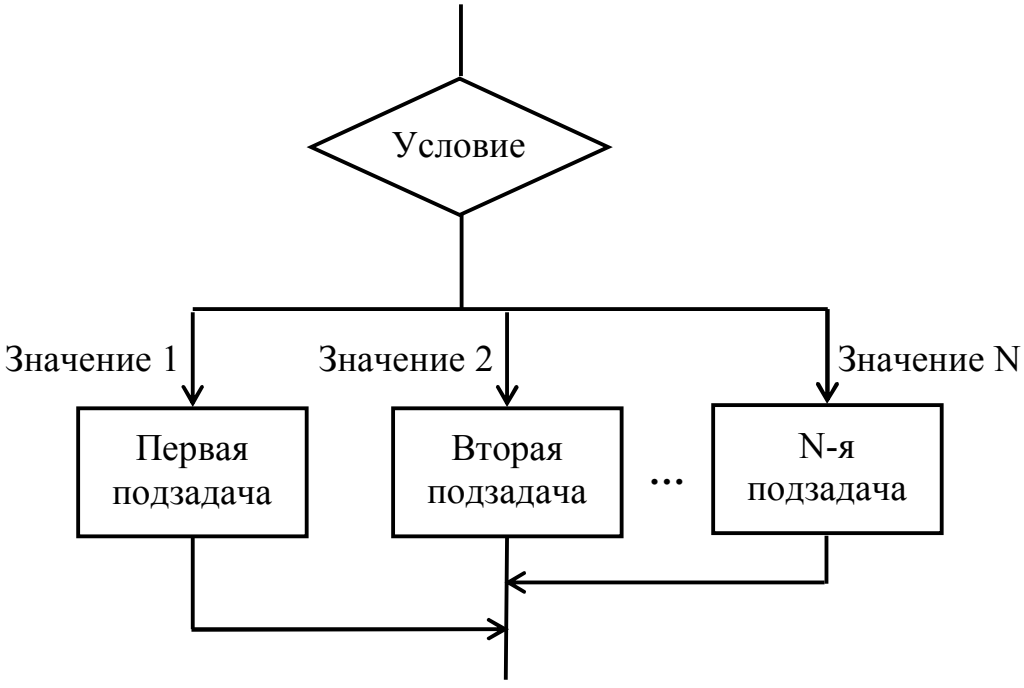


Рис. 4.6. Изображение конструкции Case
в структурном программировании

4) применение оператора GoTo с жёсткими ограничениями (например,

передача управления не далее чем на десять операторов или только вперёд по программе в соответствии с положениями отраслевых стандартов или стандартов предприятия).

Принципы структурного программирования используются, как правило, в совокупности с идеями нисходящего проектирования на нижних уровнях модульной структуры ПС (см. подразд. 4.2, 4.3) и при разработке отдельных программных модулей.

Резюме

Основные положения структурного программирования в языках программирования реализуются с помощью вызовов подпрограмм, вложенных на произвольную глубину операторов If-Then-Else, циклических операторов, оператора Case. В качестве исключений допускаются операторы GoTo с жёсткими ограничениями и подпрограммы с несколькими входами или несколькими выходами.

4.1.3. Графическое представление структурированных схем алгоритмов

Для графического представления структурированных схем алгоритмов разработан ряд специальных методов. Ниже рассмотрены два из них – метод Дамке и схемы Насси–Шнейдермана [22].

Метод Дамке

М. Дамке предложил для конструкций структурированных схем алгоритмов специальные обозначения, основанные на идеях нисходящего проектирования (см. подразд. 4.3). *Основные конструкции* структурного программирования по методу Дамке изображаются следующим образом:

1. *Функциональный блок*, как обычно, обозначается прямоугольником (рис. 4.7).

2. *Конструкция If-Then-Else* изображается в соответствии с рис. 4.8.

Элементы с выполняемыми действиями находятся справа от символа «Решение». Вход и выход из конструкции находятся соответственно сверху и снизу символа «Решение».

3. *Конструкция цикла с предусловием* («Пока») представляется в соответствии с рис. 4.9. В шестиугольнике записывается условие входа в цикл. Для повышения понятности алгоритма перед условием может быть записано слово «Пока» или соответствующее служебное слово оператора цикла с предусловием целевого языка программирования (например While в языке Паскаль).

Тело цикла выполняется до тех пор, пока условие истинно. Условие проверяется первым. Графически это изображается положением шестиугольника *выше* выполняемого тела цикла.

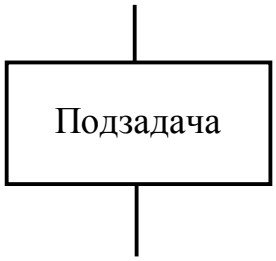


Рис. 4.7. Представление функционального блока
по методу Дамке

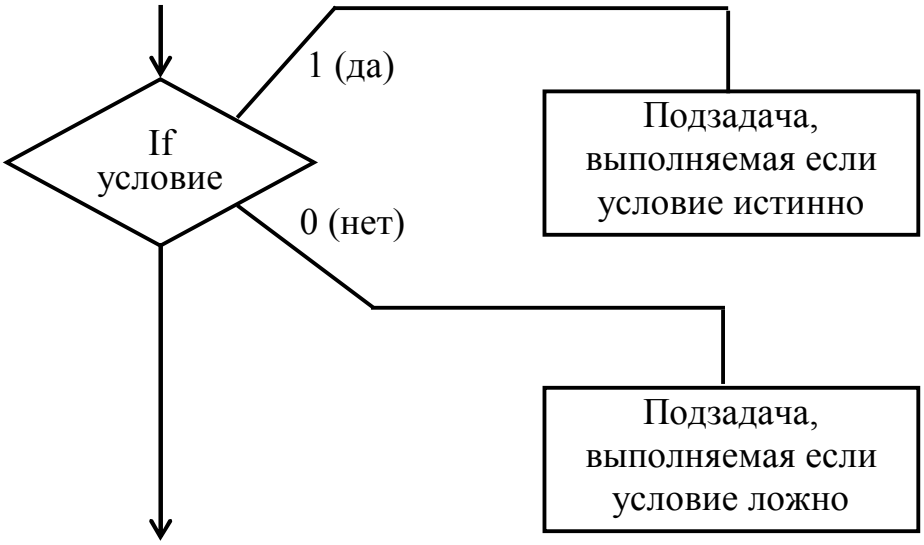


Рис. 4.8. Представление конструкции If-Then-Else
по методу Дамке

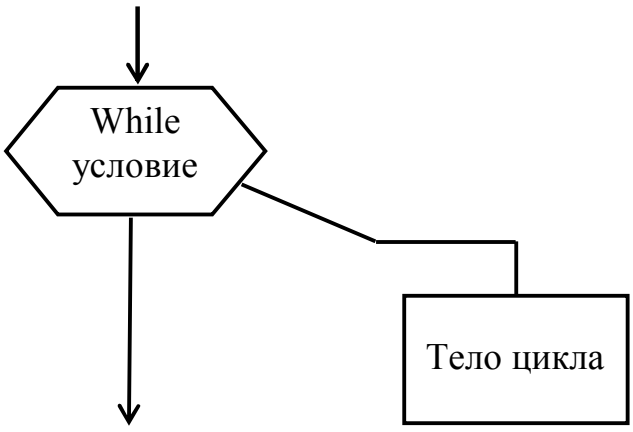


Рис. 4.9. Представление цикла с предусловием
по методу Дамке

Следует обратить внимание на то, что входы и выходы из всех конструкций метода Дамке находятся в левой части (сверху и снизу) графического представления конструкций. Расширения конструкций в правой части представления выходов не имеют.

Дополнительные конструкции структурного программирования изображаются следующим образом.

Конструкция цикла с постусловием («До») представляется в соответствии с рис. 4.10. В шестиугольнике записывается условие выхода из цикла. Перед условием может быть помещена фраза «До тех пор, пока не» или соответствующее служебное слово оператора цикла с постусловием целевого языка программирования (например Until в языке Паскаль).

Если условие истинно, осуществляется выход из цикла. Тело цикла выполняется до проверки условия. Графически это изображается положением шестиугольника *ниже* тела цикла.

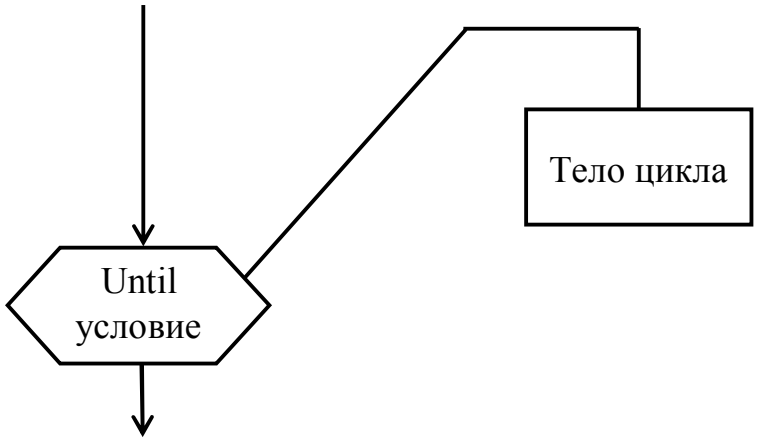


Рис. 4.10. Представление цикла с постусловием
по методу Дамке

Конструкция цикла с параметром изображается с учетом того, что она является частным случаем цикла с предусловием (рис. 4.11). В шестиугольнике записываются начальное и конечное значения параметра цикла, перед которыми помещается слово «Для» или соответствующее служебное слово оператора цикла с параметром целевого языка программирования (например, For в ряде языков).

Конструкция Case представляется в соответствии с рис. 4.12.

Основным принципом при разработке структурированных схем алгоритмов по методу Дамке является *принцип декомпозиции* (пошагового уточнения, см. подразд. 4.3). В соответствии с данным принципом любой элемент алгоритма, реализующий некоторую функцию (задачу), можно разделить на несколько элементов, реализующих необходимые подфункции (подзадачи).

Элементы в самой левой части схемы представляют укрупнённую струк-

туру алгоритма. Затем элементы расширяются вправо по мере разделения каждого элемента на подэлементы.

Чтобы исследовать любую подзадачу, достаточно анализировать только те элементы и управляющие структуры, которые находятся справа от нее.

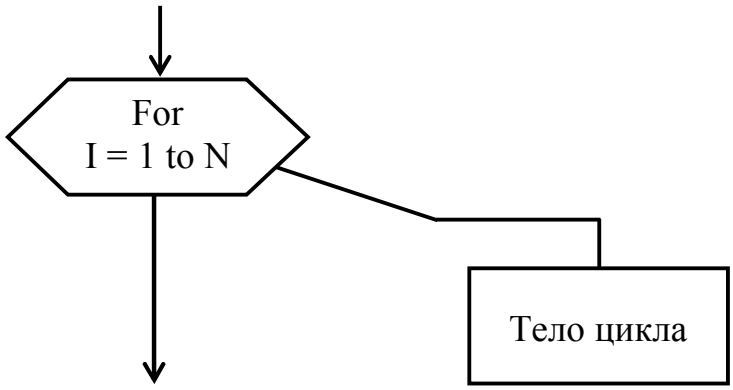


Рис. 4.11. Представление конструкции цикла с параметром
по методу Дамке

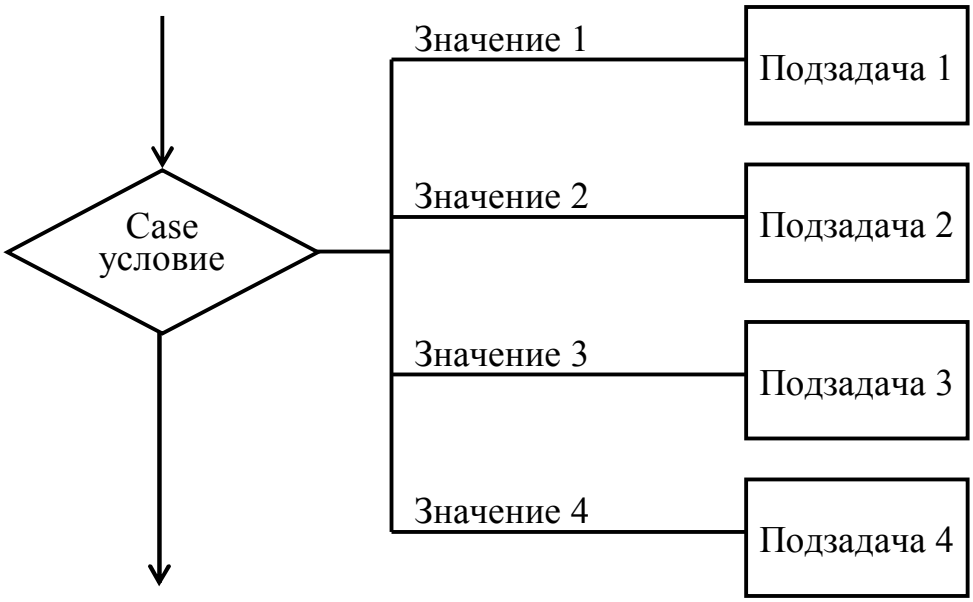


Рис. 4.12. Представление конструкции Case
по методу Дамке

Пример 4.1

Дан массив **A**, состоящий из **N** элементов. Найти наибольший из элементов массива (**Amax**) и его номер (**Imax**).

Схему алгоритма решения данной задачи, представленную по методу Дамке, иллюстрирует рис. 4.13.

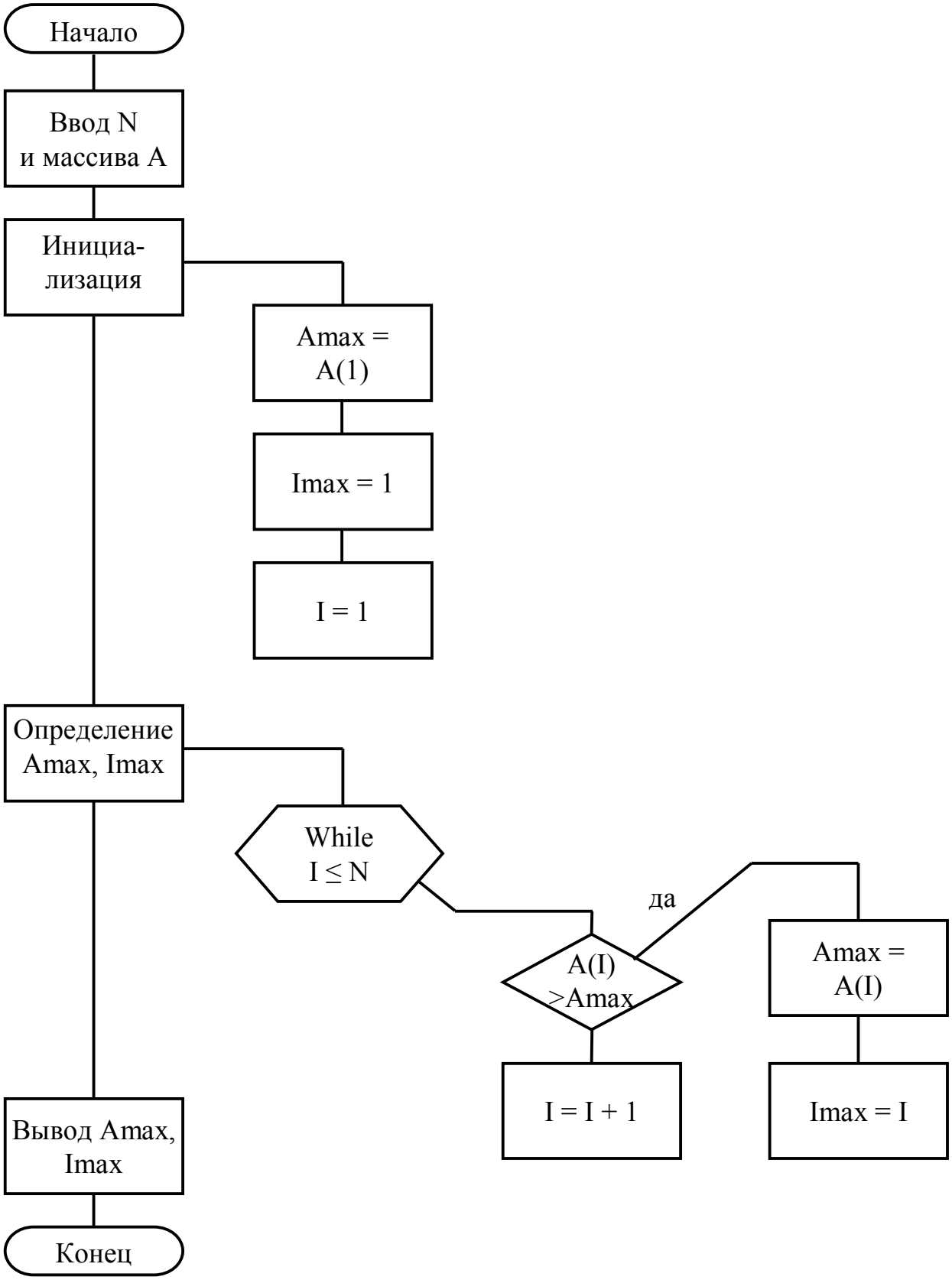


Рис. 4.13. Схема алгоритма поиска максимального элемента массива и его номера, представленная по методу Дамке

Достоинства метода Дамке:

- схема алгоритма, представленная с помощью данного метода, нагляднее, чем классическая, особенно для больших программ;
- метод Дамке удобно использовать при разработке алгоритма по методу нисходящего проектирования;
- метод Дамке удобен при коллективной разработке ПС, так как позволяет независимо разрабатывать отдельные функциональные части программы.

Схемы Насси–Шнейдермана

Схемы Насси–Шнейдермана – это схемы, иллюстрирующие структуру передач управления внутри модуля с помощью вложенных друг в друга блоков.

Схемы используются для изображения структурированных схем и позволяют уменьшить громоздкость схем за счёт отсутствия явного указания линий перехода по управлению.

Схемы Насси–Шнейдермана называют ещё *структурограммами.*

Изображение основных элементов структурного программирования в схемах Насси–Шнейдермана организовано следующим образом. Каждый блок имеет форму прямоугольника и может быть вписан в любой внутренний прямоугольник любого другого блока. Информация в блоках записывается по тем же правилам, что и в структурированных схемах алгоритмов (на естественном языке или языке математических формул).

Конструкции структурированных алгоритмов в схемах Насси–Шнейдермана изображаются следующим образом.

1. Функциональный блок (блок обработки) представляется прямоугольником без входов и выходов (рис. 4.14).

Каждый символ схем Насси–Шнейдермана представляет собой блок обработки. Каждый прямоугольник внутри любого символа также является блоком обработки.

Рис. 4.14. Представление функционального блока
в схемах Насси–Шнейдермана

2. Блок следования изображается так, как представлено на рис. 4.15.

Данный блок представляет собой объединение ряда следующих друг за другом процессов обработки.

3. *Блок решения* изображается в соответствии с рис. 4.16.

Блок решения используется для представления конструкции принятия

двоичного решения If-Then-Else. Условие записывается в центральном треугольнике, варианты исполнения условия – в боковых треугольниках (варианты исполнения могут быть записаны в виде: *1, 0; да, нет; +, –*). Процессы обработки обозначаются прямоугольниками.

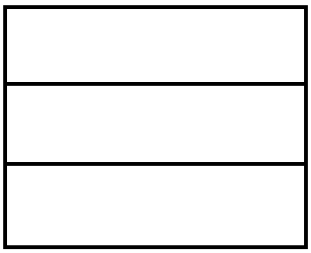


Рис. 4.15. Представление блока следования
в схемах Насси–Шнейдермана

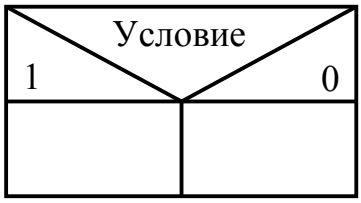


Рис. 4.16. Представление блока решения
в схемах Насси–Шнейдермана

4. Блок Case представляется в соответствии с рис. 4.17.

Данный блок является расширением блока решения. Те варианты выхода из этого блока, которые можно точно сформулировать, размещаются слева от нижней вершины центрального треугольника. Остальные выходы объединяются в один, называемый выходом по несоблюдению условий (выход «иначе»). Данный выход располагается справа от нижней вершины треугольника.

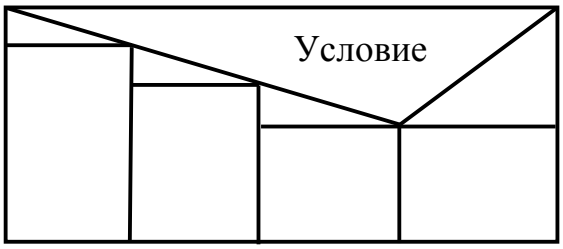


Рис. 4.17. Представление блока Case
в схемах Насси–Шнейдермана

Если можно перечислить все возможные случаи, правую часть можно оставить незаполненной или совсем опустить, а выходы разместить по обе стороны центрального треугольника.

По аналогии с конструкцией If-Then-Else условие записывается в центральном треугольнике, варианты выполнения условия – в боковых треугольниках. Процессы обработки помещаются в прямоугольниках.

5. Цикл с предусловием изображается так, как показано на рис. 4.18.

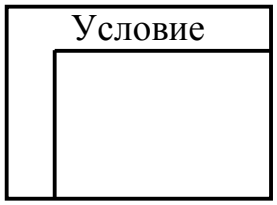


Рис. 4.18. Представление цикла с предусловием
в схемах Насси–Шнейдермана

Данный блок обозначает циклическую конструкцию с проверкой условия в начале цикла. Условие выполнения цикла размещается в верхней полосе.

6. Цикл с постусловием представляется в соответствии с рис. 4.19.

Данный блок обозначает циклическую конструкцию с проверкой условия после выполнения тела цикла. Условие выхода из цикла размещается в нижней полосе.

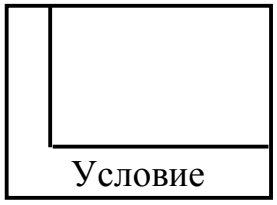


Рис. 4.19. Представление цикла с постусловием
в схемах Насси–Шнейдермана

Основным *достоинством* схем Насси–Шнейдермана является компактность, обусловленная отсутствием линий, которые отображают потоки управления (линий перехода по управлению) между блоками. Благодаря данному достоинству схемы Насси–Шнейдермана зачастую используются в CASE-средствах для представления схем алгоритмов (см. подразд. 7.3) [43].

Пример 4.2

Дан массив **A**, состоящий из **N** элементов. Найти наибольший из элементов массива (**Amax**) и его номер (**Imax**).

Схема алгоритма решения данной задачи, представленная по методу Дамке, приведена в примере 4.1.

Укрупненная и детализированная схемы Насси–Шнейдермана для решения данной задачи представлены на рис. 4.20.

Укрупненная схема Насси–Шнейдермана

Детализированная схема Насси–Шнейдермана

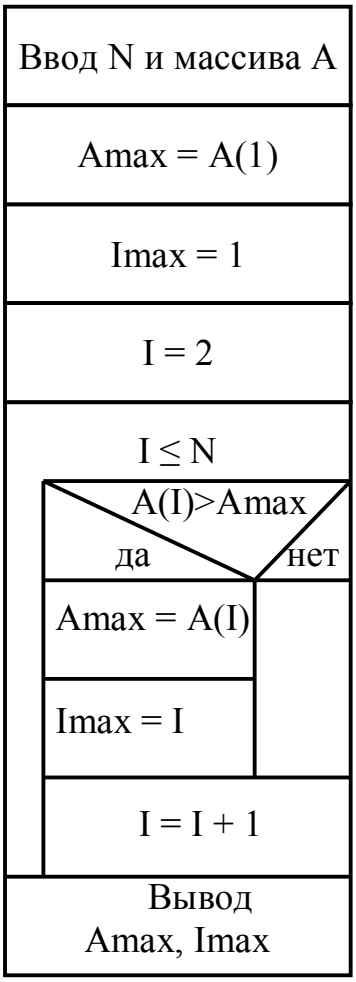


Рис. 4.20. Пример диаграммы Насси–Шнейдермана для алгоритма поиска максимального элемента массива и его номера

Резюме

Для графического представления структурированных схем алгоритмов разработан ряд специальных методов. Метод Дамке реализует идеи нисходящего проектирования структурированных схем алгоритмов за счет детализации компонентов программ вправо. Схемы Насси–Шнейдермана изображают структуру передач управления внутри модуля с помощью вложенных друг в друга блоков без явного указания линий перехода по управлению.

4.2. Модульное проектирование программных средств

Модульное проектирование является одним из первых подходов к разработке структуры ПС и уже несколько десятилетий сохраняет свои позиции как в качестве классического подхода, так и в качестве основы для современных технологий разработки ПС.

При разработке модульных ПС могут использоваться *методы структурного проектирования* или *методы объектно-ориентированного проектирования*. Их целью является формирование структуры создаваемой программы – ее разделение по некоторым установленным правилам на структурные компоненты (***модуляризация***) с последующей иерархической организацией данных компонентов. Для различных языков программирования такими компонентами могут быть подпрограммы, внешние модули, объекты и т.п.

Обзор методов объектно-ориентированного анализа и проектирования приведен в разд. 6. В данном разделе рассмотрены методы структурного проектирования. Такие методы ориентированы на формирование структуры программного средства по функциональному признаку.

Классическое определение *идеальной* модульной программы формулируется следующим образом. ***Модульная программа*** – это программа, в которой любую часть логической структуры можно изменить, не вызывая изменений в ее других частях [22].

Признаки модульности программ:

1) программа состоит из модулей. Данный признак для модульной программы является очевидным;

2) модули являются независимыми. Это значит, что модуль можно изменять или модифицировать без последствий в других модулях;

3) условие «один вход – один выход». Модульная программа состоит из модулей, имеющих одну точку входа и одну точку выхода. В общем случае может быть более одного входа, но важно, чтобы точки входов были определены и другие модули не могли входить в данный модуль в произвольной точке.

Достоинства модульного проектирования:

1) упрощение разработки ПС;
2) исключение чрезмерной детализации обработки данных;
3) упрощение сопровождения ПС;
4) облегчение чтения и понимания программ;
5) облегчение работы с данными, имеющими сложную структуру.

Недостатки модульности:

1) модульный подход требует большего времени работы центрального процессора (в среднем на 5 – 10 %) за счет времени обращения к модулям;

2) модульность программы приводит к увеличению ее объема (в среднем на 5 – 10 %);

3) модульность требует дополнительной работы программиста и определенных навыков проектирования ПС.

Классические методы структурного проектирования модульных ПС делятся на три основные группы [22]:

1) методы нисходящего проектирования;
2) методы расширения ядра;
3) методы восходящего проектирования.

На практике обычно применяются различные сочетания этих методов.

Резюме

В идеальной модульной программе любую часть логической структуры можно изменить, не вызывая изменений в ее других частях. Идеальная модульная программа состоит из независимых модулей, имеющих один вход и один выход. Модульные программы имеют достоинства и недостатки. Существует три группы классических методов проектирования модульных ПС.

4.3. Методы нисходящего проектирования

Основное *назначение* нисходящего проектирования – служить средством разбиения большой задачи на меньшие подзадачи так, чтобы каждую подзадачу можно было рассматривать независимо.

Суть метода нисходящего проектирования заключается в следующем.

На начальном шаге в соответствии с общими функциональными требованиями к программному средству разрабатывается его укрупненная структура без детальной проработки его отдельных частей. Затем выделяются функциональные требования более низкого уровня и в соответствии с ними разрабатываются отдельные компоненты программного средства, не детализированные на предыдущем шаге. Эти действия являются рекурсивными, то есть каждый из компонентов детализируется до тех пор, пока его составные части не будут окончательно уточнены. В последнем случае принимается решение о прекращении дальнейшего проектирования.

На каждом шаге нисходящего проектирования делается оценка правильности вносимых уточнений в контексте правильности функционирования разрабатываемого программного средства в целом.

Компоненты нижнего уровня ПС называются *программными модулями*. Для модулей характерны достаточная простота и прозрачность, позволяющие выполнять их непосредственное программирование.

Таким образом, на каждом шаге разработки уточняется реализация фрагмента алгоритма, то есть решается более простая задача.

Следует отметить, что метод нисходящего проектирования положен в основу стандартного процесса разработки, регламентированного стандартом *СТБ ИСО/МЭК 12207–2003* (см. подразд. 1.2).

Основными классическими *стратегиями,* на которых основана реализация метода нисходящего проектирования, являются [22]:

1) пошаговое уточнение; данная стратегия разработана Э. Дейкстрой;

2) анализ сообщений; данная стратегия базируется на работах группы авторов (Йодана, Константайна, Мейерса).

Эти стратегии отличаются способами определения начальных спецификаций требований, методами разбиения задачи на части и правилами записи (*нотациями*), положенными в основу проектирования ПС.

Резюме

Нисходящее проектирование служит средством разбиения большой задачи на меньшие подзадачи так, чтобы каждую подзадачу можно было рассматривать независимо. Существуют различные стратегии реализации нисходящего проектирования. Основные из них – пошаговое уточнение и анализ сообщений.

4.3.1. Пошаговое уточнение

Пошаговое уточнение является одной из классических стратегий, реализующих метод нисходящего проектирования ПС [22].

При пошаговом уточнении на каждом следующем этапе декомпозиции детализируются программные компоненты очередного более низкого уровня. При этом результаты каждого этапа являются уточнением результатов предыдущего этапа лишь с небольшими изменениями.

Существуют различные способы реализации пошагового уточнения. Рассмотрим два классических способа:

1) проектирование программного средства с помощью псевдокода и управляющих конструкций структурного программирования;

2) использование комментариев для описания обработки данных.

Пошаговое уточнение требует, чтобы взаимное расположение строк текста программы обеспечивало ее читабельность. *Общее правило записи текста разрабатываемой программы*: служебные слова, которыми начинается и заканчивается та или иная управляющая конструкция, записываются на одной вертикали; все вложенные в данную конструкцию псевдокоды (или комментарии, или операторы программы) и управляющие конструкции записываются с отступом вправо.

Преимущества метода пошагового уточнения:

1) основное внимание при его использовании обращается на проектирование корректной структуры программы, а не на ее детализацию;

2) так как каждый последующий этап является уточнением предыдущего лишь с небольшими изменениями, то легко может быть выполнена проверка корректности процесса разработки на всех этапах.

Недостаток метода пошагового уточнения: на поздних этапах проектирования программного средства может обнаружиться необходимость в структурных изменениях, требующих пересмотра более ранних решений.

Резюме

При пошаговом уточнении на каждом следующем этапе декомпозиции детализируются программные компоненты очередного более низкого уровня. К классическим способам реализации пошагового уточнения относятся проектирование программы с помощью псевдокода и управляющих конструкций структурного программирования, а также использование комментариев для описания обработки данных.

4.3.2. Проектирование программных средств с помощью псевдокода и управляющих конструкций структурного программирования

Одним из классических способов реализации пошагового уточнения является проектирование ПС с помощью псевдокода и управляющих конструкций структурного программирования [22].

При использовании данного способа разбиение программы на модули осуществляется эвристическим способом. На каждом этапе проектирования осуществляется *выбор необходимых управляющих конструкций*, но *операции с данными по возможности не уточняются* (это откладывается на возможно более поздние сроки). Таким образом, фактически проектируется управляющая структура программы.

На этапе, когда принимается решение о прекращении дальнейшего уточнения, оставшиеся неопределенными функции становятся вызываемыми модулями или подпрограммами, а проектируемый модуль – управляющим модулем.

Пример 4.3

Рассмотрим пример проектирования программы с использованием псевдокода и управляющих конструкций структурного программирования. Пусть программа обрабатывает файл дат. Необходимо отделить правильные даты от неправильных, отсортировать правильные даты, перенести летние и зимние даты в выходной файл, вывести неправильные даты.

В данном примере в качестве псевдокода используются предложения, состоящие из русских слов, соединенных между собой символом подчеркивания.

Первый этап пошагового уточнения

Задается заголовок программы, соответствующий ее назначению.

Program Обработка_дат.

Второй этап пошагового уточнения

Определяются основные структурные компоненты программы в соответствии с ее основными функциями.

```
Program Обработка_дат;
    Отделить_правильные_даты_от_неправильных {*}
    Сортировать_правильные_даты
    Выделить_зимние_и_летние_даты
    Обработать_неправильные_даты
End.
```

Фрагмент программы, детализированный на втором этапе, располагается правее детализированного на первом этапе.

Третий этап пошагового уточнения

Дальнейшая детализация программы. Детализация фрагмента {*}. Возможно появление необходимости в использовании управляющих конструкций структурного программирования.

```
Program Обработка_дат;
    While не_конец_входного_файла Do
        Begin
            Прочитать_дату
            Проанализировать_правильность_даты
        End
    Сортировать_правильные_даты
    Выделить_зимние_и_летние_даты
    Обработать_неправильные_даты
End.
```

Жирным шрифтом здесь выделены служебные слова цикла с предусловием **While,** представляющего собой одну из управляющих конструкций структурного программирования.

На некоторых этапах уточнения можно приостановить определение некоторых функций, выделяя их в вызываемые модули в том случае, если они функционально независимы от основной обработки (например, в примере это могут быть функции «Проанализировать правильность даты» и «Сортировать правильные даты»).

Процесс детализации продолжается, пока не будет принято решение о прекращении дальнейшей детализации. В этом случае все действия, записанные в последней программе в виде предложений, необходимо оформить в виде подпрограмм. Такие подпрограммы могут быть реализованы, например, с использованием принципа структурного программирования.

Резюме

При использовании псевдокода и управляющих конструкций структурного программирования проектируется управляющая структура программы. На каждом этапе проектирования осуществляется выбор необходимых управляю-

щих конструкций. Уточнение операций с данными по возможности откладывается на поздние сроки.

4.3.3. Использование комментариев для описания обработки данных

Еще одним классическим способом реализации пошагового уточнения является использование комментариев для описания обработки данных [22].

При этом способе на каждом этапе уточнений используются *управляющие конструкции структурного программирования*, а *правила обработки данных не детализируются*. Они описываются в виде комментариев.

На каждом этапе уточнений блоки, представленные комментариями, частично детализируются. Но сами комментарии при этом не выбрасываются. В результате после окончания проектирования получается хорошо прокомментированный текст программы.

Наиболее часто используются следующие ***виды комментариев***:

1) *заголовки*. Объясняют назначение основных компонентов программы на отдельных этапах пошаговой детализации;

2) *построчные*. Описывают мелкие фрагменты программы;

3) *вводные*. Помещаются в начале текста программы и задают общую информацию о ней (например, назначение программы, сведения об авторах, дата написания, используемый метод решения, время выполнения, требуемый объем памяти).

Считается, что один из самых больших недостатков программы – отсутствие в ней комментариев. В среднем нужно руководствоваться следующими ***нормами комментариев***:

- 4 – 5 строк комментария-заголовка на каждый компонент программы (подпрограмму, блок, модуль и т.п.);
- по одному комментарию на каждые две строки исходного текста для построчных комментариев.

Пример 4.4

Рассмотрим тот же пример проектирования программы обработки файла дат, который был рассмотрен в п. 4.3.2. Необходимо отделить правильные даты от неправильных, отсортировать правильные даты, перенести летние и зимние даты в выходной файл, вывести неправильные даты.

Первый этап пошагового уточнения

Записываются вводный комментарий и комментарий к заголовку программы.

```
{Программа обработки дат. Разработчик Иванов И. И.}
{Заголовок программы}
```

Второй этап пошагового уточнения

Определяются основные шаги обработки дат. Сохраняются комментарии предыдущего этапа и добавляются комментарии текущего этапа.

```
{Программа обработки дат. Разработчик Иванов И. И.}
{Заголовок программы}
Program {Обработка дат}
    {Отделение правильных дат от неправильных} {*}
    {Сортировка правильных дат}
    {Выделение зимних и летних дат}
    {Обработка неправильных дат}
End.
```

Третий этап пошагового уточнения

Дальнейшая детализация программы. Детализация фрагмента *. Сохраняются комментарии предыдущих этапов и добавляются комментарии текущего этапа. Возможно появление необходимости в использовании управляющих конструкций структурного программирования.

```
{Программа обработки дат. Разработчик Иванов И. И.}
{Заголовок программы}
Program {Обработка дат}
    {Отделение правильных дат от неправильных} {*}
    While {не конец входного файла} Do
        Begin
            {Чтение даты}
            {Анализ правильности даты}
        End
    {Сортировка правильных дат}
    {Выделение зимних и летних дат}
    {Обработка неправильных дат}
End.
```

Процесс продолжается до принятия решения о прекращении дальнейшей детализации. После этого все действия, записанные в последней программе в виде комментариев, которые не были реализованы, необходимо оформить в виде подпрограмм.

Резюме

При использовании комментариев для описания обработки данных проектируется управляющая структура программы. На каждом этапе проектирования осуществляется выбор необходимых управляющих конструкций. Операции обработки данных представляются в виде коментариев. Существуют комментарии-заголовки, построчные комментарии и вводные комментарии. Имеются общепринятые нормы использования комментариев.

4.3.4. Анализ сообщений

Анализ сообщений является второй из рассматриваемых классических стратегий, реализующих метод нисходящего проектирования. Анализ сообщений используется в первую очередь для структуризации ПС обработки информации и основывается на анализе потоков данных, обрабатываемых программным средством [22].

Пример 4.5

Рассмотрим тот же пример проектирования программы обработки файла дат, который был рассмотрен в пп. 4.3.2, 4.3.3. Необходимо отделить правильные даты от неправильных, отсортировать правильные даты, перенести летние и зимние даты в выходной файл, вывести неправильные даты.

Рис. 4.21, 4.22 иллюстрируют возможные варианты укрупненного и детализированного представления потоков информации и обрабатывающих их процессов для данной программы, представленные с помощью диаграмм потоков данных (Data Flow Diagram, DFD) в нотации Йодана–ДеМарко (см. подразд. 5.3).

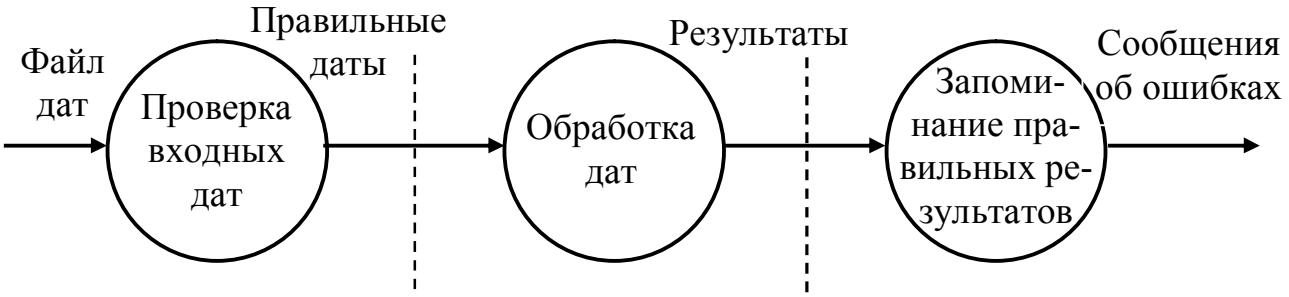


Рис. 4.21. Укрупненная диаграмма потоков данных для программы обработки файла дат

В соответствии со стратегией анализа сообщений первоначальный поток данных разбивается на три потока: первый содержит непреобразованные входные данные, второй – потоки преобразования, третий – только выходную информацию. Границы, разделяющие эти потоки, на рис. 4.21, 4.22 показаны штриховыми линиями. Они делят диаграмму на три части.

Данные, подлежащие обработке с помощью процесса «Обработка дат» (см. рис. 4.21), могут не включать все входные даты, но это еще часть входных данных. Процесс «Обработка дат» в общем случае может включать различные виды преобразования данных (например, кодирование, декодирование, вычисления и др.). Результаты данного процесса представляют собой выходные данные, хотя они могут быть еще неотформатированными, неотредактированными, возможно, неверными.

Три части программы, соответствующие трем потокам данных, принято называть соответственно ***истоком, преобразователем и стоком*** [22].

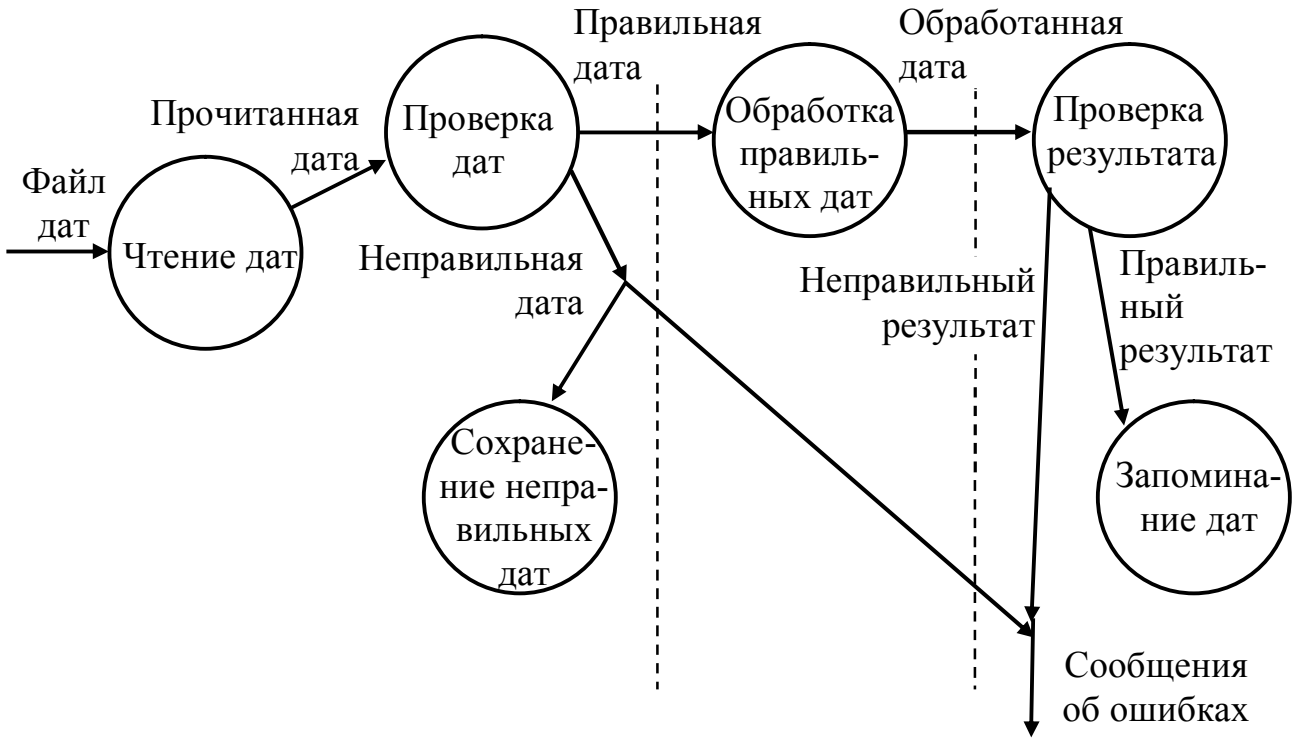


Рис. 4.22. Детализированная диаграмма потоков данных
для программы обработки файла дат

Преобразователь – это основная часть программы, *исток* выполняет функцию управления входным потоком данных, *сток* выполняет функцию управления выходным потоком данных.

На рис. 4.23 представлен общий вид DFD-диаграммы разбиения любой программы на исток – преобразователь – сток. Линии на диаграмме показывают потоки передачи данных между процессами.

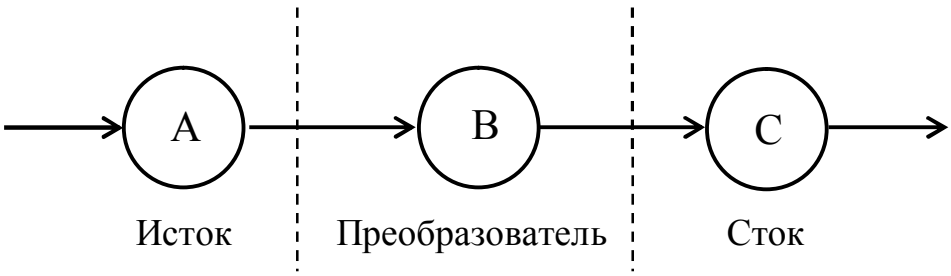


Рис. 4.23. DFD-диаграмма разбиения программы
на исток-преобразователь-сток

Рис. 4.24 содержит соответствующую схему иерархии компонентов программы. В схеме иерархии линии указывают связи по управлению между компонентами, а также изображают отношения типа вызывающий – вызываемый.

Процесс декомпозиции программы заключается в рекурсивном использовании метода разбиения на исток – преобразователь – сток на отдельных ветвях ее древовидной структуры. На нижнем уровне в результате формируется совокупность программных модулей – наименьших единиц проектирования программы. Каждый из модулей может быть реализован в зависимости от назначения, сложности и размера как независимый модуль, внутренняя подпрограмма или некоторая часть основной программы.

Следует отметить, что не все компоненты программы обязательно должны быть разбиты на три компонента более низкого уровня. Результат декомпозиции компонента-стока должен обязательно содержать сток, компонента-преобразователя – преобразователь, компонента-истока – исток. Вызывающий компонент – это главный сток для компонента-истока и главный исток для компонента-стока.

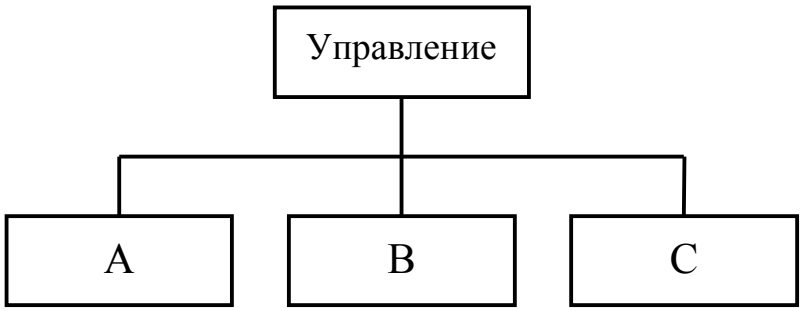


Рис. 4.24. Схема иерархии компонентов программы

Если структуры данных имеют большие размеры, управление ими осуществляется на уровне детализации данных для компонентов истока и стока.

На рис. 4.25 представлена иерархическая структура компонентов для первого уровня декомпозиции программы обработки файла дат, на рис. 4.26 – для второго уровня декомпозиции.

На данных рисунках приняты следующие обозначения:

1 – проверка входных дат (исток);

2 – обработка дат (преобразователь);

3 – запоминание правильных результатов (сток);

4 – чтение дат (исток истока 1);

5 – проверка дат (преобразователь истока 1);

6 – сохранение неправильных дат (сток истока 1);

7 – обработка правильных дат (преобразователь преобразователя 2);

8 – проверка результата (преобразователь стока 3);

9 – запоминание дат (сток стока 3).

Количество уровней декомпозиции определяется сложностью задачи и необходимой степенью детализации.

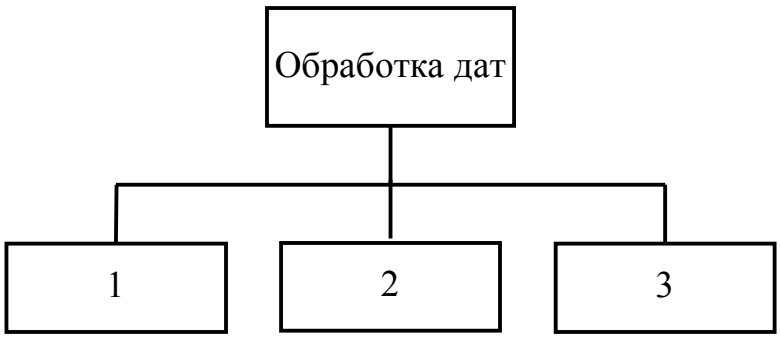


Рис. 4.25. Схема иерархии программы обработки дат
(декомпозиция первого уровня)

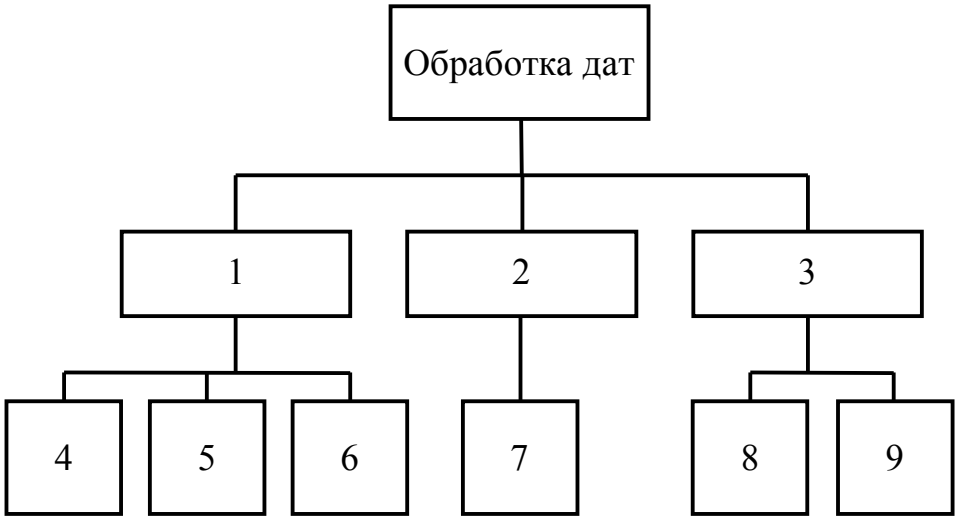


Рис. 4.26. Схема иерархии программы обработки дат
(декомпозиция второго уровня)

Каждый компонент программы при информационном обмене использует определенную часть данных. Однако в иерархической структуре программы информационные связи между компонентами не отражены. Поэтому описание иерархической структуры должно содержать *таблицу взаимодействия компонентов*, показывающую передачу данных между ними. В этой таблице должны быть определены все способы информационного обмена, задаваемые как при помощи формальных параметров, так и с помощью глобальных переменных.

В табл. 4.1 представлены информационные связи между компонентами программы обработки дат. В таблице отражены те данные, которые передаются каждому компоненту в момент его вызова. Входные данные управляющего компонента изображаются как выходные для вызываемых им компонентов.

Выходные данные управляющего компонента отображаются как входные для вызываемых им компонентов. Поэтому компонент-исток имеет выходные данные, сток – входные данные, преобразователь – и те, и другие.

Резюме

Анализ сообщений основывается на анализе потоков данных, обрабатываемых программным средством. Первоначальный поток данных разбивается на три потока: исток, преобразователь и сток. Процесс декомпозиции заключается в рекурсивном использовании метода разбиения на исток – преобразователь – сток на отдельных ветвях иерархической структуры программы. Описание иерархической структуры должно содержать таблицу взаимодействия компонентов, показывающую передачу данных между ними.

Таблица 4.1

Таблица связей между компонентами

Компонент	Вход	Выход
1	—	Правильные даты
2	Правильные даты	Результаты
3	Результаты	—
4	—	Прочитанная дата
5	Прочитанная дата	Правильная дата Неправильная дата
6	Неправильная дата	—
7	Правильная дата	Обработанная дата
8	Обработанная дата	Правильный результат Неправильный результат
9	Правильный результат	—

4.4. Методы восходящего проектирования

При использовании восходящего проектирования в первую очередь выделяются функции нижнего уровня, которые должно выполнять программное средство. Эти функции реализуются с помощью программных модулей самых нижних уровней. Затем на основе этих модулей проектируются программные компоненты более высокого уровня. Данные компоненты реализуют функции

более высокого уровня. Процесс продолжается, пока не будет завершена разработка всего программного средства.

В чистом виде метод восходящего проектирования используется крайне редко. Основным его *недостатком* является то, что программисты начинают разработку программного средства с несущественных, вспомогательных деталей. Это затрудняет проектирование программного средства в целом.

Метод восходящего проектирования *целесообразно* применять в следующих случаях:

- существуют разработанные модули, которые могут быть использованы для выполнения некоторых функций разрабатываемой программы;
- заранее известно, что некоторые простые или стандартные модули потребуются нескольким различным частям программы (например, подпрограмма анализа ошибок, ввода-вывода и т.п.).

Обычно используется *сочетание* методов нисходящего и восходящего проектирования [22]. Такое сочетание возможно различными *способами*. Ниже рассмотрены два из них.

Первый способ сочетания

Выделяются ключевые (наиболее важные) модули промежуточных уровней разрабатываемой программы. Затем проектирование ведется нисходящим и восходящим методами одновременно (рис. 4.27).

Второй способ сочетания

Проектируются модули нижнего уровня (те, которые необходимо спроектировать заранее). Затем программа проектируется одновременно нисходящим и восходящим методами (рис. 4.28).

При таком способе проектирования наиболее важной задачей является согласование интерфейса между верхними и нижними уровнями программы, выполняемое в последнюю очередь. Это является существенным недостатком данного способа сочетания. Разработчики должны обладать достаточно высокой квалификацией, чтобы не оказалось, что верхняя и нижняя части программы несовместимы между собой.

Резюме

При использовании метода восходящего проектирования в первую очередь реализуются функции нижнего уровня программы. На основе полученных модулей проектируются программные компоненты более высокого уровня. Часто используется сочетание методов нисходящего и восходящего проектирования. Такое сочетание возможно различными способами.

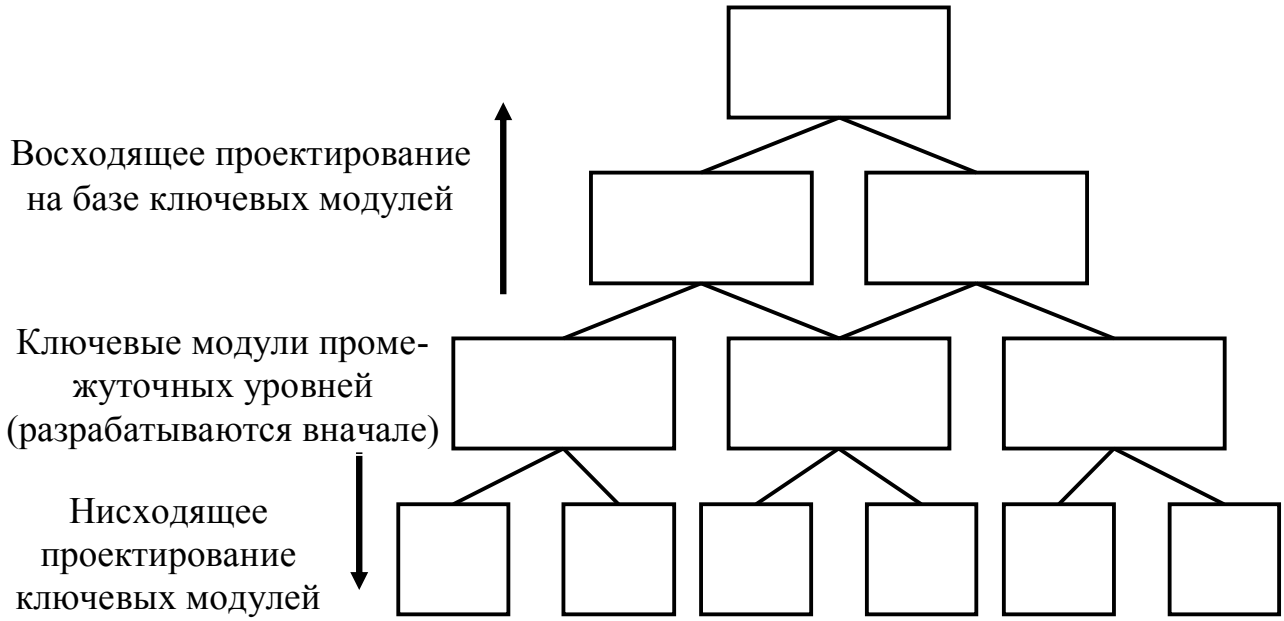


Рис. 4.27. Проектирование программы нисходящим и восходящим методами на базе ключевых модулей

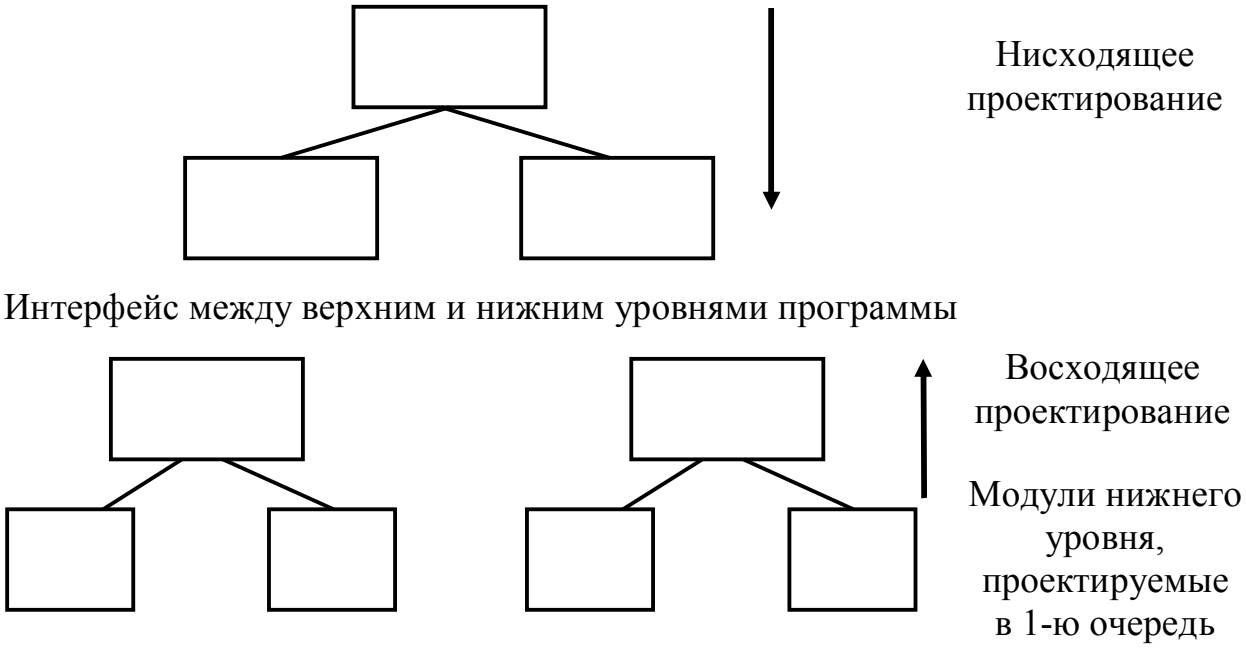


Рис. 4.28. Проектирование программы нисходящим и восходящим методами на базе модулей нижнего уровня

4.5. Методы расширения ядра

При использовании данных методов в первую очередь создается ядро (основная часть) программы. Затем данное ядро постепенно расширяется, пока не будет полностью сформирована управляющая структура разрабатываемой программы.

Существует два подхода к реализации методов расширения ядра.

Первый подход основан на методах проектирования *структур данных*, используемых при иерархическом проектировании модулей. Данный подход применяется в методах JSP и JSD, разработанных Майклом Джексоном [26, 38].

Второй подход основан на определении областей хранения данных с последующим анализом связанных с ними функций. Данный подход использует метод определения спецификаций модуля, разработанный Парнасом [22].

В данном разделе рассмотрен метод JSP, реализующий первый подход. Метод JSD, являющийся развитием метода JSP, кратко рассмотрен в п. 5.5.1.

4.6. Метод JSP Джексона

Метод структурного программирования JSP (Jackson Structured Programming) разработан М. Джексоном в 70-х гг. XX в. Данный метод наиболее эффективен в случае высокой степени структуризации данных. Это характерно, например, для класса планово-экономических задач.

Метод JSP (называемый также методом Джексона) базируется на *исходном положении*, состоящем в том, что структура программы зависит от структуры подлежащих обработке данных. Поэтому *структура данных может использоваться для формирования структуры программы*.

4.6.1. Основные конструкции данных

Метод JSP основывается на возможности представления структур данных и структур программ единым набором основных конструкций. М. Джексоном предложены *четыре основные конструкции данных* [26].

1. Конструкция последовательности данных

Эта конструкция используется, когда два или более компонента данных следуют друг за другом строго последовательным образом и образуют единый компонент данных.

Графически конструкция последовательности данных (последовательность данных) изображается в соответствии с рис. 4.29. На данном рисунке компоненты **B, C, D, E** объединяются в указанном порядке и образуют последовательность **A**.

В конструкции последовательности данных должно быть не менее двух подкомпонентов, причем каждый из них должен встречаться строго один раз и обязательно в предписанном порядке.

Рис. 4.30 представляет пример последовательности данных – запись даты **D**, состоящая из трех последовательных частей – поля **N** числа, поля **M** месяца и поля **Y** года.

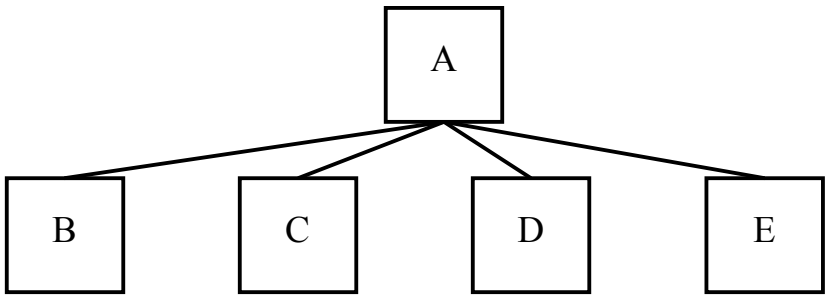


Рис. 4.29. Конструкция последовательности данных

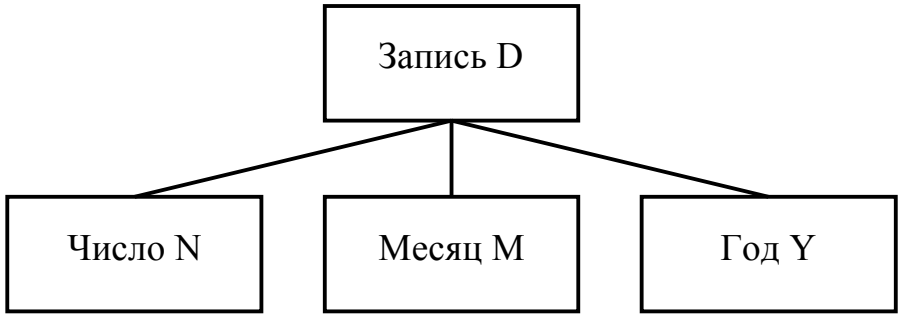


Рис. 4.30. Пример последовательности данных

2. Конструкция выбора данных

Конструкцией выбора данных (выбором данных) называется конструкция сведения результирующего компонента данных к одному из двух или более выбираемых подкомпонентов.

На рис. 4.31 представлена конструкция выбора данных, результирующий компонент **S** которой сводится к подкомпоненту **P**, **Q** или **R**. Внешне конструкция выбора данных отличается от конструкции последовательности наличием символа «**o**» в верхнем правом углу каждого из выбираемых подкомпонентов.

Очевидно, что в конструкции выбора должно быть не менее двух подкомпонентов.

На практике выбор может заключаться в том, что подкомпонент выбора либо присутствует, либо отсутствует. Рис. 4.32 иллюстрирует некорректное изображение такой конструкции выбора. Данная конструкция должна быть представлена в соответствии с рис. 4.33.

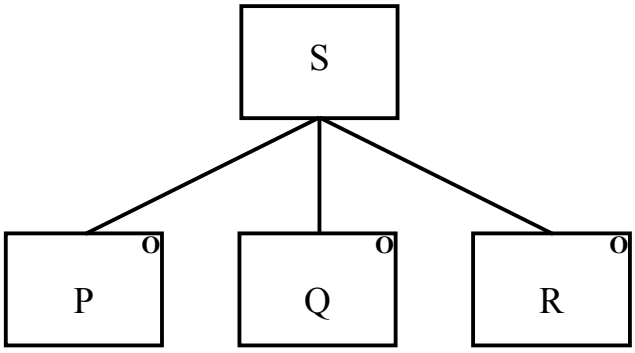


Рис. 4.31. Конструкция выбора данных

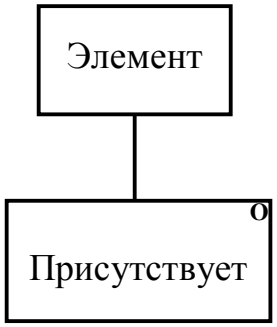


Рис. 4.32. Пример некорректного изображения конструкции выбора данных

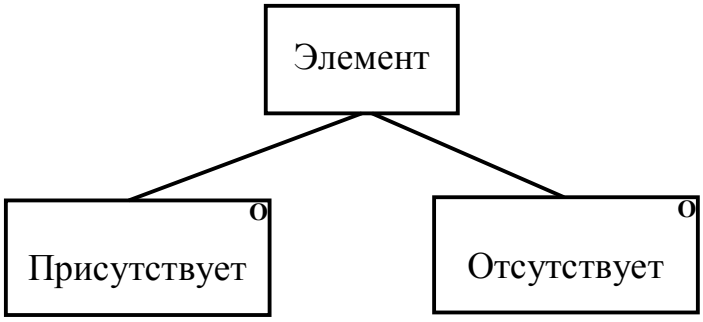


Рис. 4.33. Пример правильного представления
конструкции выбора данных

3. Конструкция повторения данных

Данная конструкция применяется тогда, когда конкретный элемент данных может повторяться от нуля до неограниченного числа раз.

Представление конструкции повторения данных (повторения данных) в нотации структур Джексона иллюстрирует рис. 4.34. На данном рисунке компонент **I** состоит из повторяющихся подкомпонентов **X**.

У конструкции повторения только один подкомпонент. Признаком повторяемой части конструкции является символ * в верхнем правом углу.

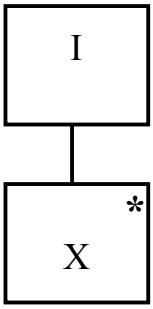


Рис. 4.34. Конструкция повторения данных

Пример конструкции повторения данных представлен на рис. 4.35. На данном рисунке файл **F** состоит из нуля или более записей **D**.

Если некоторый компонент должен включать одно или более появлений повторяемого подкомпонента (ситуация нуля повторений подкомпонента возникнуть не может), то используется конструкция, представленная на рис. 4.36.

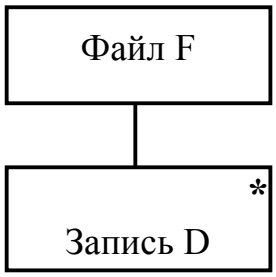


Рис. 4.35. Пример конструкции повторения данных

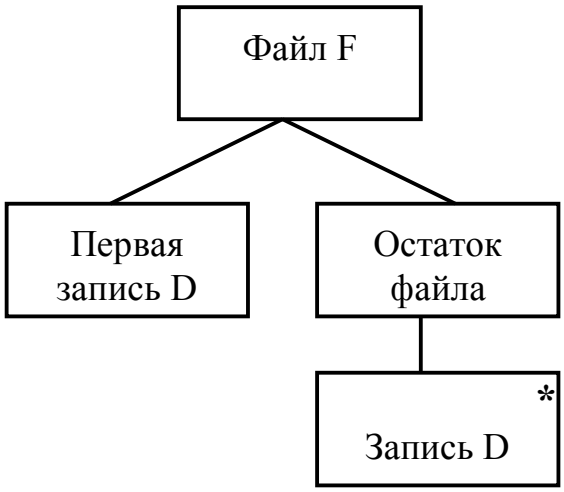


Рис. 4.36. Пример конструкции повторения данных
с не менее чем одним появлением

В данном случае файл **F** изображается последовательностью из двух подкомпонентов. Подкомпонент «Остаток файла» представляет собой повторение записи **D**.

При необходимости конкретное количество повторений может быть указано в круглых скобках рядом с повторяемой частью конструкции (рис. 4.37). Указание числа повторений является расширением нотации метода JSP.

4. Элементарная конструкция

Элементарными являются те компоненты, которые не разбиваются далее на подкомпоненты. Примерами элементарных конструкций являются, например, первая запись **D** и запись **D** на рис. 4.36, компоненты число **N**, месяц **M**, год **Y** на рис. 4.30.

Компонент может являться элементарным, потому что его нельзя разложить дальше или потому, что с практической точки зрения отсутствует необходимость в его дальнейшем разбиении.

Резюме

Метод JSP основывается на возможности представления структур данных и структур программ единым набором основных конструкций. Существует четыре основных конструкции данных: конструкция последовательности, конструкция выбора, конструкция повторения и элементарная конструкция.

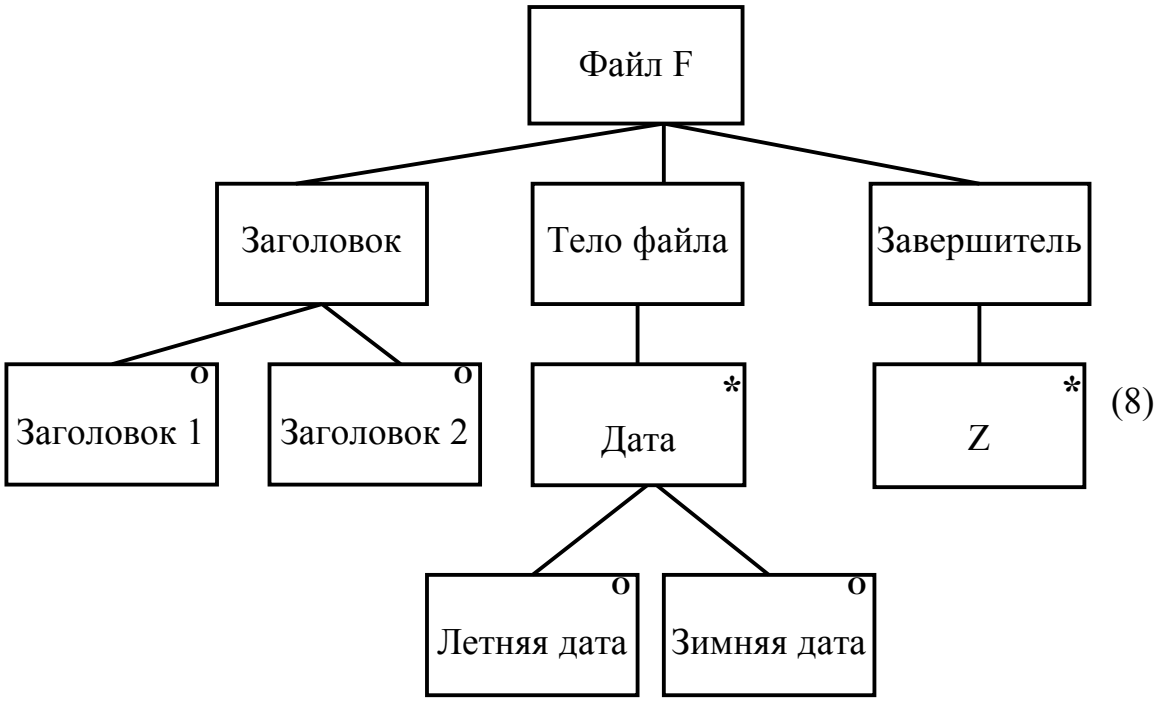


Рис. 4.37. Иерархическая структура данных

4.6.2. Построение структур данных

Рассмотрим возможность формирования сложных структур данных на основе комбинации четырех описанных в п. 4.6.1 конструкций данных.

Пусть имеется некоторый файл **F**, состоящий из заголовка, за которым следует совокупность дат, завершающаяся признаком окончания. Заголовок может иметь один из двух типов, каждая дата представляет собой летнюю или зимнюю дату, завершитель состоит из восьми символов **Z**. Структуру файла **F** можно представить в виде *иерархической структуры данных*, базируясь на основных конструкциях данных Джексона (см. рис. 4.37).

На данном рисунке файл **F** представлен в виде конструкции последовательности данных, заголовок – в виде конструкции выбора, тело файла и завершитель – конструкции повторения, дата – конструкции выбора.

На первый взгляд кажется, что в вышеприведенной структуре компонент «Тело файла» является излишним (см. рис. 4.37). Однако если данный компонент убрать, то последовательность «Файл F» будет состоять из трех подкомпонентов, вторым среди которых будет являться «Дата». Подкомпонент «Дата» представляет собой повторяемый подкомпонент. Следовательно, «Дата» может присутствовать в файле любое число раз. Но в конструкции последовательности данных любой подкомпонент должен встретиться ровно один раз. Поэтому наличие компонента «Тело файла» в структуре данных, представленной на рис. 4.37, является обязательным.

Из вышеприведенного примера видно, что если данные можно представить в виде иерархической структуры, то их можно изобразить с помощью набора из четырех основных конструкций Джексона.

Помимо иерархической структуры данных широко используется сетевая и реляционная структуры данных.

В *сетевой структуре данных* имеются связи между отдельными компонентами, включая компоненты самых разных уровней структуры. Проектирование программы для обработки таких данных связано со значительными трудностями. Поэтому сетевые структуры данных обычно преобразуются к иерархическому виду. В этом случае один компонент сетевой структуры обычно принимается в качестве основного (ключевого). Для преобразования сетевой структуры в иерархическую упрощают сложные взаимосвязи между данными, не существенные для конкретного вида данных.

Пример сетевой структуры данных иллюстрирует рис. 4.38 [26]. На рис. 4.39 показан возможный вариант преобразования этих данных к иерархическому виду.

При *реляционном подходе* структура данных представляется в виде совокупности таблиц. На рис. 4.40 представлена таблица, состоящая из пяти строк и шести столбцов. Очевидно, что сами таблицы имеют иерархическую структуру. На рис. 4.41 показаны два способа иерархического представления таблицы, состоящей из пяти строк и шести столбцов.

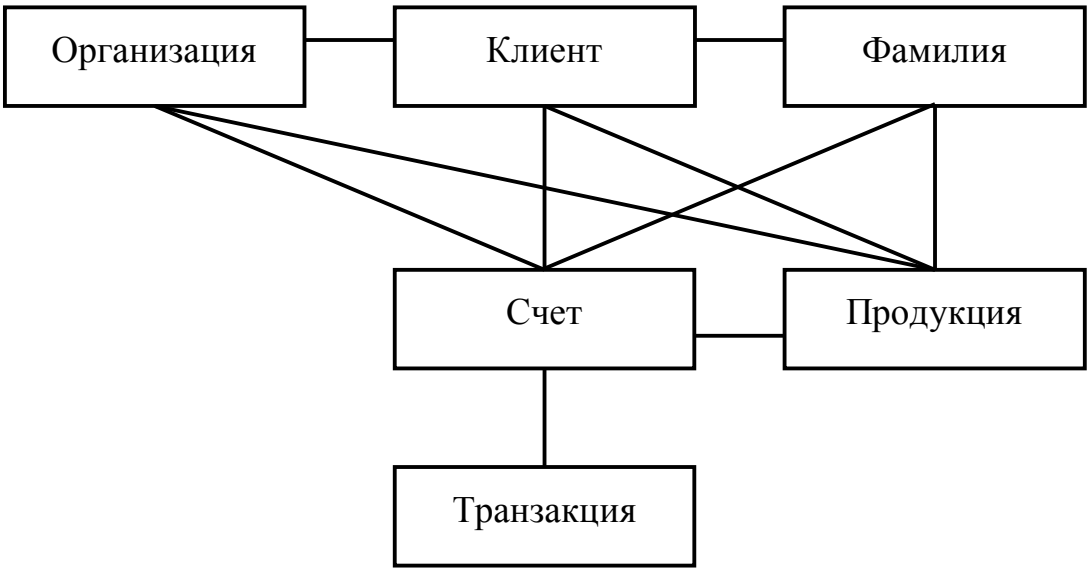


Рис. 4.38. Пример сетевой структуры данных

Рис. 4.39. Иерархический вид сетевой структуры данных

Столбцы

Строки	1	2	3	4	5	6
1						
2						
3						
4						
5						

Рис. 4.40. Пример представления структуры данных
при использовании реляционного подхода

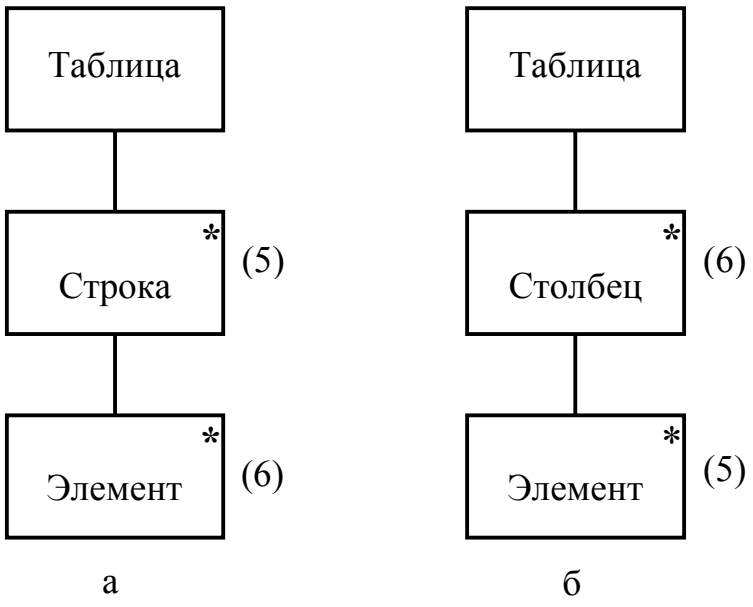


Рис. 4.41. Иерархическая структура таблиц:
а – представление таблицы в виде набора строк;
б – представление таблицы в виде набора столбцов

Таким образом, реальные наборы данных, как правило, могут быть представлены в виде иерархических структур.

Представление структур данных в виде иерархий является первым этапом проектирования программы по методу Джексона (методу JSP). На следующих этапах описывается взаимосвязь между структурами данных и программой.

Резюме

Большинство структур (иерархическая, сетевая, реляционная) реальных наборов данных может быть сведено к иерархическим структурам, которые могут быть представлены в нотации структур метода JSP Джексона.

4.6.3. Проектирование структур программ

В соответствии с методом JSP конструкции, используемые для построения структур данных, применяются и для построения структур программ. Так же, как и данные, программы могут быть составлены из конструкций последовательности, выбора и повторения.

Большинство ПС предназначено для обработки некоторых входных данных и получения некоторых выходных данных. Структура выходных данных формируется программой в результате некоторого преобразования структуры входных данных. Таким образом, для проектирования структуры программы необходимо определить взаимосвязь между входными данными, выходными данными и процессом преобразования [26].

Пример 4.6

Рассмотрим простейший пример [26]. Пусть необходимо найти суммы элементов строк в массиве, состоящем из 15 строк и 10 столбцов.

Рис. 4.42 иллюстрирует структуры данных, представляющие входные и выходные данные разрабатываемой программы.

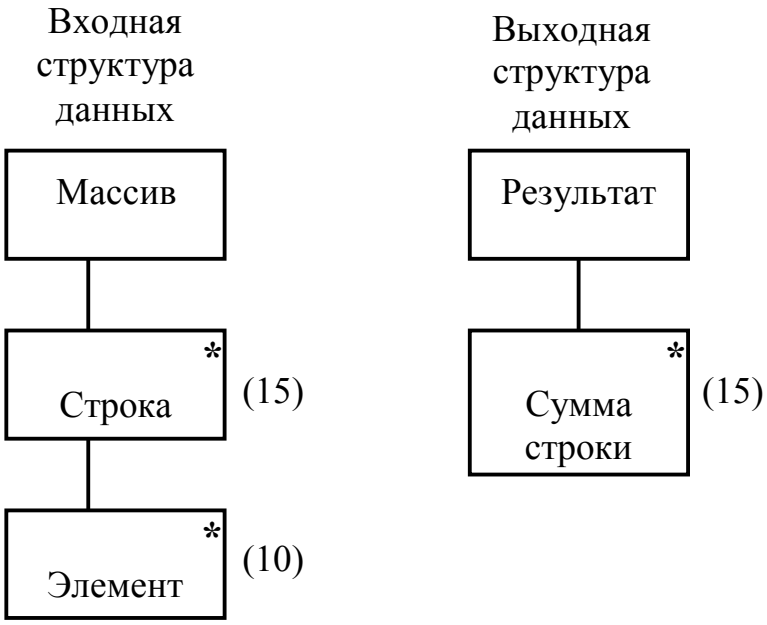


Рис. 4.42. Структуры входных и выходных данных

Очевидно, что между обеими структурами имеется непосредственная взаимосвязь. Компонент «Сумма строки» в структуре «Результат» появляется

столько раз, сколько раз компонент «Строка» появляется во входной структуре «Массив», и в том же порядке.

Соответствия между компонентами входной и выходной структур данных принято изображать жирными линиями с двусторонними стрелками. Соответствие между структурами входных и выходных данных для рассматриваемого примера представлено на рис. 4.43.

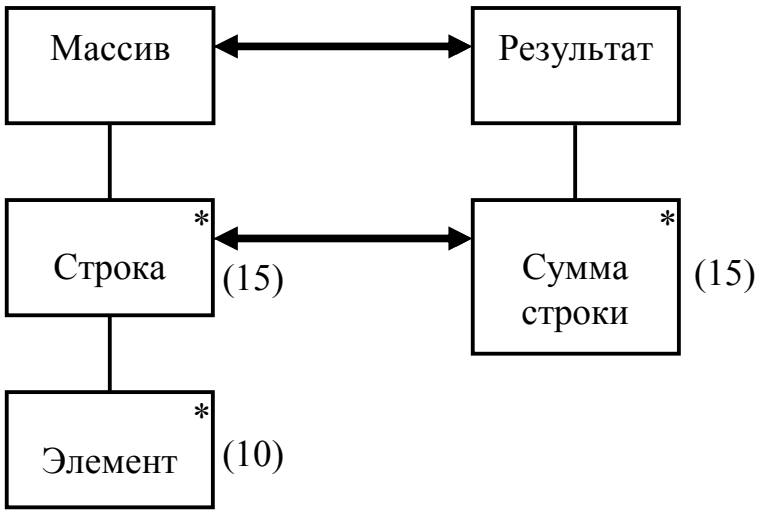


Рис. 4.43. Соответствие между структурами входных и выходных данных

Каждая пара соответствующих компонентов входных и выходных данных образует основу одного компонента программы. Поэтому проектирование упрощенной структуры программы (ее ядра) выполняется путем «слияния» соответствующих компонентов структур входных и выходных данных. Это значит, что на месте линий соответствия размещаются компоненты программы, которым присваивается соответствующее имя (рис. 4.44). Слияние двух повторяемых **N** раз компонентов данных сформирует один повторяющийся **N** раз компонент программы.

Пример 4.7

Рассмотрим более сложный пример [26]. Пусть результат предыдущего примера должен сопровождаться некоторым заголовком и некоторым завершителем. Структуры входных и выходных данных и соответствия между ними для данного случая представлены на рис. 4.45.

На первом подэтапе формирования структуры программы на основании линий соответствия формируется упрощенный вариант структуры (ядро программы, рис. 4.46).

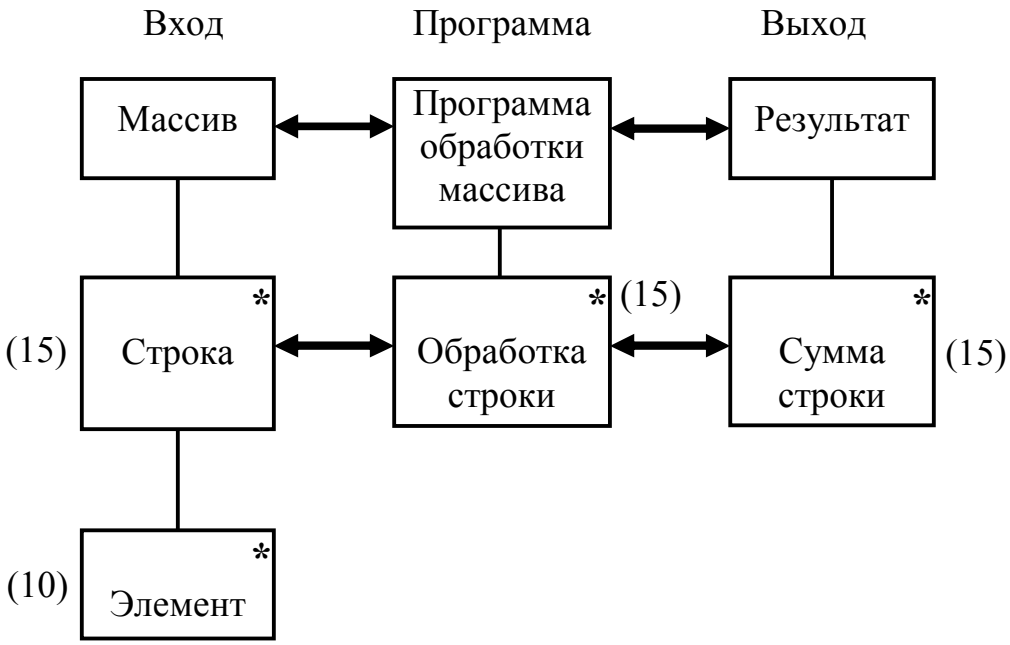


Рис. 4.44. Соответствие данных и программы

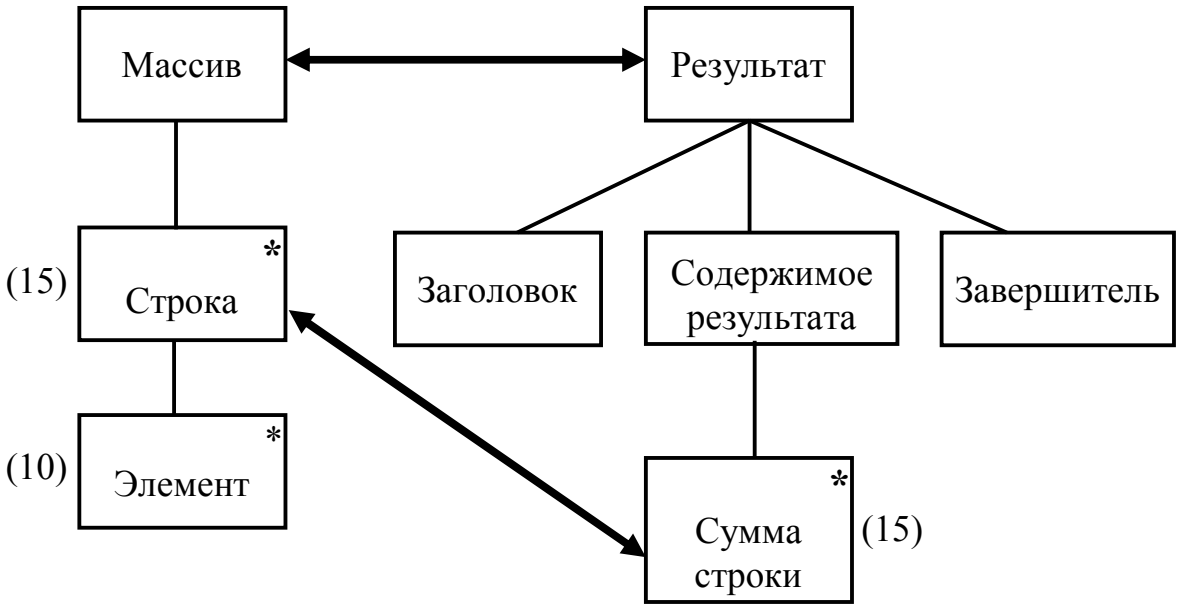


Рис. 4.45. Расширенные структуры входных и выходных данных

На следующих подэтапах ядро программы расширяется. При этом рассматриваются те компоненты структур данных, к которым не подходят линии соответствия.

На втором подэтапе анализируются компоненты во входной структуре данных, к которым не подходят линии соответствия.

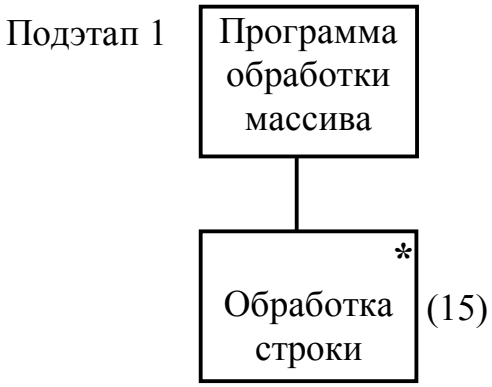


Рис. 4.46. Первый подэтап формирования структуры программы

Таким компонентом во входной структуре данных является «Элемент». Ему не соответствует ни один компонент в структуре выходных данных.

Но «Элемент» образует подкомпоненты, составляющие компонент «Строка». А «Строка» сливается с компонентом «Сумма строки», формируя компонент «Обработка строки». Поэтому компонент «Элемент» можно трактовать как повторяемый подкомпонент компонента «Обработка строки» и переименовать его в компонент «Обработка элемента» (рис. 4.47).

На третьем подэтапе рассматриваются компоненты в выходной структуре данных, к которым не подходят линии соответствия. Такими компонентами являются «Заголовок», «Содержимое результата», «Завершитель» (см. рис. 4.45).

Рис. 4.47. Второй подэтап формирования структуры программы

По аналогии с компонентом «Элемент» каждый из этих компонентов помещается в то относительное положение в структуре программы, в котором он появляется в структурах данных. И соответственно меняется имя данных компонентов (добавляется слово «Получение», «Формирование», «Обработка» и т. п.). Рис. 4.48 иллюстрирует полученную в результате структуру программы.

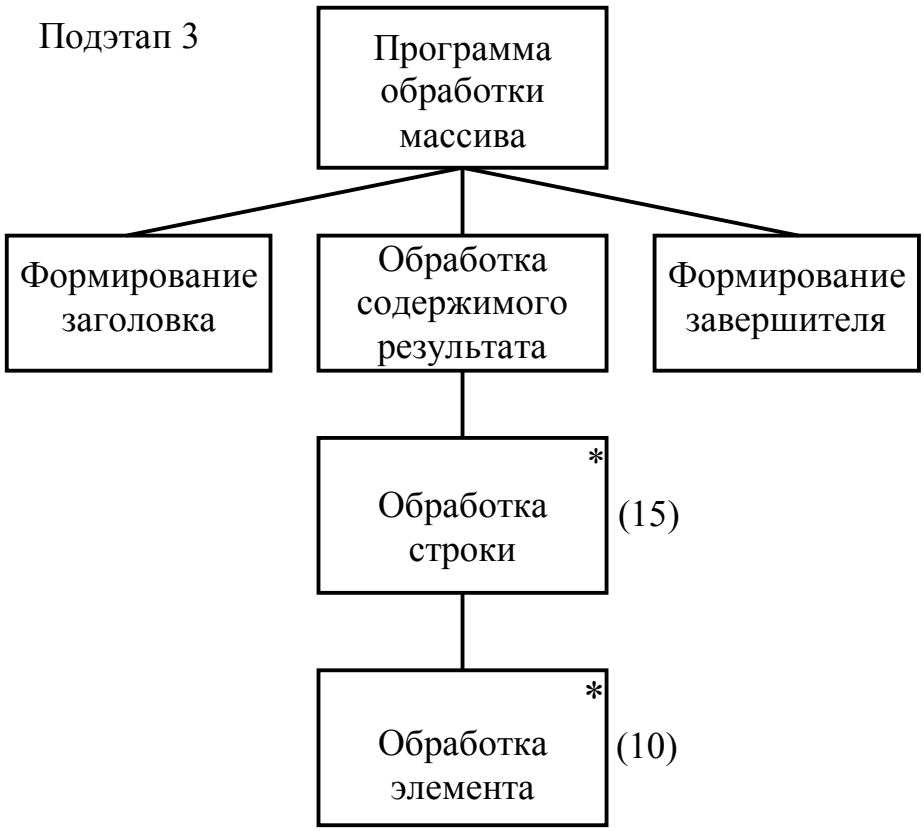


Рис. 4.48. Формирование полной структуры программы

Таким образом, выше на простых примерах показан процесс генерации структуры программы из структур входных и выходных данных на основе метода Джексона. Следующий подраздел посвящен *формализованному* описанию данного процесса.

Резюме

Для проектирования структуры программы по методу Джексона необходимо разработать структуры ее входных и выходных данных, определить взаимосвязь между данными структурами и процессом преобразования входных данных в выходные, сформировать ядро программы с учетом соответствий между входной и выходной структурами данных, поместить не нашедшие соответствия компоненты входных и выходных данных в нужные места структуры программы.

4.6.4. Этапы проектирования программного средства

Метод JSP реализуется *пятью этапами* [26]:

1. Проектирование структур входных и выходных данных.
2. Идентификация соответствий между структурами данных.
3. Проектирование структуры программы.
4. Перечисление и распределение выполняемых операций.
5. Создание текста программы на метаязыке структурированного описания.

Рассмотрим правила выполнения данных этапов на конкретном примере.

Пример 4.8

Пусть имеется входной файл, состоящий из заголовка, за которым следует совокупность дат, завершающаяся признаком окончания (завершителем). Заголовок начинается символом **H** (Heading), за которым следует содержимое заголовка, состоящее из типа файла и даты создания файла. Тип файла может иметь одно из двух значений – «Тип 1» или «Тип 2». Каждая дата представляет собой летнюю или зимнюю дату. Завершитель состоит из восьми символов **Z**.

Необходимо получить выходной файл, состоящий из заголовка и тела файла. Заголовок состоит из типа файла и даты создания файла, причем данные компоненты должны извлекаться из входного файла. Тело файла должно содержать результат подсчета количества летних дат и количества зимних дат, имеющихся во входном файле.

Этап 1. Проектирование структур входных и выходных данных

Данный этап является самым важным этапом метода JSP, так как структуры данных образуют основу формируемой структуры программы.

При создании структур данных обычно принято использовать *три вида документов*:

А. Таблицы для идентификации компонентов данных.

Б. Графическое представление структур данных.

В. Контрольные перечни вопросов для анализа структур данных.

А. Таблицы для идентификации компонентов данных

Формат таблицы для идентификации компонентов входных данных иллюстрирует табл. 4.2. Эта таблица содержит структуру входных данных, соответствующую примеру 4.8. Данная структура представлена в иерархическом виде и построена на базе конструкций Джексона. Аналогичная таблица заполняется для полного набора выходных данных (табл. 4.3).

Таблица 4.2

Структура входных компонентов данных

Ссылочный номер	Тип компонента	Ссылочный номер старшего компонента	Имя компонента данных (условие появления)	Ссылочные номера составляющих компонентов			
				Повто-рение	Последо-ватель-ность	Выбор	Элемен-тарная
1	Последова-тельность	—	Входной файл	1.2, 1.3	1.1	—	—
1.1	Последова-тельность	1	Заголовок	—	1.1.2	—	1.1.1
1.2	Повторение	1	Тело файла	—	—	1.2.1	—
1.3	Повторение	1	Завершитель	—	—	—	1.3.1
1.1.1	Элементарная	1.1	Символ «H»	—	—	—	—
1.1.2	Последова-тельность	1.1	Содержимое заголовка	—	—	1.1.2.1	1.1.2.2
1.2.1	Выбор	1.2	Дата (пока не за-вершитель)	—	—	—	1.2.1.1, 1.2.1.2
1.3.1.	Элементарная	1.3	«Z» (пока <= 8)	—	—	—	—
1.1.2.1	Выбор	1.1.2	Тип файла	—	—	—	1.1.2.1.1, 1.1.2.1.2
1.1.2.2	Элементарная	1.1.2	Дата создания файла	—	—	—	—
1.2.1.1	Элементарная	1.2.1	Летняя дата	—	—	—	—
1.2.1.2	Элементарная	1.2.1	Зимняя дата	—	—	—	—
1.1.2.1.1	Элементарная	1.1.2.1	Тип 1	—	—	—	—
1.1.2.1.2	Элементарная	1.1.2.1	Тип 2	—	—	—	—

Таблица 4.3

Структура выходных компонентов данных

Ссылочный номер	Тип компонента	Ссылочный номер старшего компонента	Имя компонента данных (с условием появления)	Ссылочные номера составляющих компонентов			
				Повто-рение	Последо-ватель-ность	Выбор	Элемен-тарная
1	Последова-тельность	—	Выходной файл		1.1, 1.2	—	—
1.1	Последова-тельность	1	Заголовок	—		—	1.1.1, 1.1.2
1.2	Последова-тельность	1	Тело файла	—	—	—	1.2.1, 1.2.2
1.1.1	Элементарная	1.1	Тип файла	—	—	—	—
1.1.2	Элементарная	1.1	Дата создания файла	—	—	—	—
1.2.1	Элементарная	1.2	Количество летних дат	—	—	—	—
1.2.2	Элементарная	1.2	Количество зимних дат	—	—	—	—

Для заполнения таблиц необходимо выполнить следующие действия.

1-й шаг. Представить совокупность всех входных и выходных данных в виде компонентов самого высокого уровня (например, «Файл», «Отчет» и т.д.).

2-й шаг. Перечислить подкомпоненты данных, которые содержит компонент из 1-го или 4-го шагов. Этот компонент должен быть либо последовательностью, либо повторением, либо выбором.

3-й шаг. Снабдить иерархическими номерами все подкомпоненты. Указать имя, тип, номер и условие появления.

4-й шаг. Для каждого подкомпонента определить, можно ли его обрабатывать при каждом появлении одним и тем же набором действий независимо от условий. Если да, то компонент можно считать элементарным и исследовать следующий подкомпонент компонента более высокого уровня. Если нет, то необходим возврат ко 2-му шагу с целью дальнейшего разложения этого подкомпонента.

Б. Графическое представление структур данных

На основе разработанных таблиц, идентифицирующих компоненты входных и выходных данных, выполняется графическое построение их структур.

На рис. 4.49 содержится результат графического представления структуры входных данных. Основу для данного представления составляет табл. 4.2.

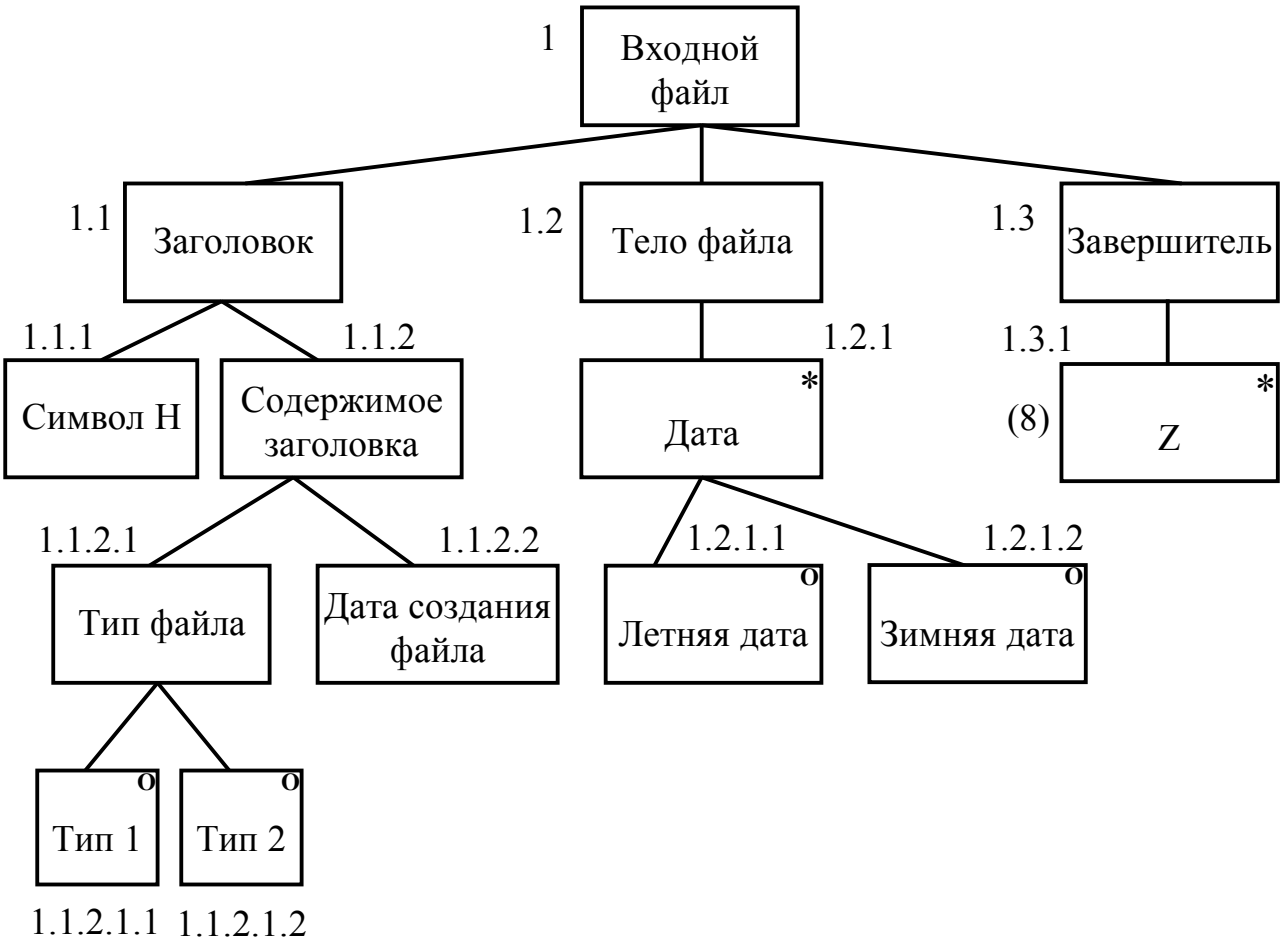


Рис. 4.49. Графическое представление структуры входного файла

Рис. 4.50 содержит графическое представление структуры выходного файла. Данное представление получено на основе табл. 4.3.

В. Контрольный перечень вопросов для анализа структур данных

Когда проектирование структур данных завершено, возникает необходимость в проверке их правильности. С этой целью используется *контрольный перечень*, представляющий собой достаточно большой стандартный набор вопросов для оценки полноты и корректности структур данных. Контрольный перечень состоит из двух частей.

Первая часть перечня предназначена для проверки полноты структуры данных. Примеры вопросов, относящихся к данной части:

- идентифицирован ли каждый вход и выход?

• представляет ли компонент данных самого высокого уровня весь вход или выход?

• идентифицирован ли каждый компонент данных именем?

• идентифицированы ли все объекты прикладного уровня (заголовки, типы строк отчетов и т.п.)?

• можно ли идентифицировать дополнительные структуры данных?

Вторая часть перечня предназначена для оценки корректности структур данных. Примеры вопросов, относящихся к данной части:

• является ли каждый подкомпонент конструкции выбора взаимоисключающим среди других соответствующих подкомпонентов?

• изображена ли структура данных сверху вниз и слева направо с точки зрения появления компонентов данных?

• является ли каждый компонент данных корректной последовательностью, выбором, повторением или элементарным компонентом?

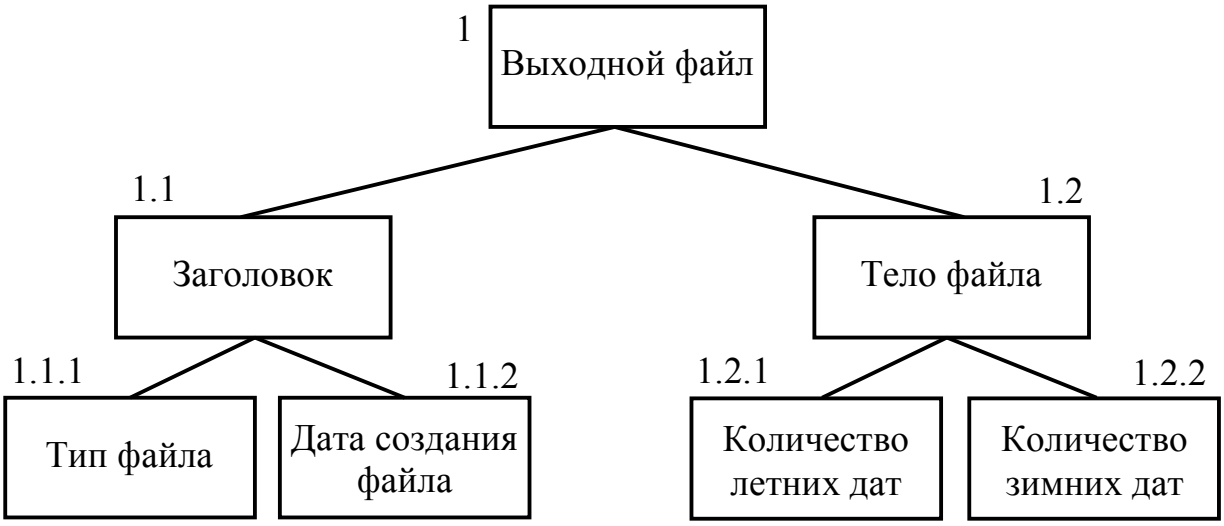


Рис. 4.50. Графическое представление структуры выходного файла

Этап 2. Идентификация соответствий между структурами данных

Второй этап метода JSP Джексона заключается в идентификации соответствий между структурами входных и выходных данных, созданными на первом этапе. *Общие правила* установления соответствий между компонентами входной и выходной структур данных:

1) должно совпадать количество соответствующих друг другу компонентов данных;

2) соответствующие компоненты должны появляться в одинаковом порядке;

3) должна быть возможной совместная обработка каждого набора соответствующих компонентов.

Наиболее эффективный способ идентификации всех соответствий состоит в том, чтобы начать со структуры данных с наименьшим числом компонентов. В рассматриваемом примере такой структурой является структура выходных данных (сравните рис. 4.49 и рис. 4.50). Используя три приведенных выше правила, необходимо начинать с вершины структуры и при работе продвигаться вниз.

На рис. 4.51 представлен результат идентификации соответствий рассмотренных структур входных и выходных данных.

Этап 3. Создание структуры программы

Можно выделить *три подэтапа* создания структур программ.

Подэтап 1. Слияние соответствующих компонентов входных и выходных данных для формирования компонентов программы.

В результате получается упрощенная структура (ядро) программы. Каждый из ее сформированных компонентов снабжается соответствующим именем (например, вида «Обработка X для создания Y»).

На рис. 4.52 показан результат данного подэтапа для рассматриваемого примера. В этом случае имеется пять линий соответствия между входными и выходными данными (см. рис. 4.51). Каждая линия заменяется соответствующим компонентом программы. Поэтому упрощенная структура программы также состоит из пяти компонентов.

Подэтап 2. Включение не имеющих соответствия компонентов структуры входных данных в формируемую структуру программы на те же относительные иерархические места и присвоение им соответствующих имен.

На рис. 4.53 содержится результат выполнения данного подэтапа. На данном рисунке двойной линией справа выделены добавленные на данном подэтапе компоненты программы. Повторяемые компоненты входной структуры стали повторяемыми компонентами программы. Выбираемые компоненты входной структуры стали выбираемыми компонентами программы. Однако, как будет видно при рассмотрении подэтапа 3, такое непосредственное преобразование возможно не всегда.

Подэтап 3. Включение не имеющих соответствия компонентов структуры выходных данных в формируемую структуру программы на те же относительные иерархические места и присвоение им соответствующих имен.

На рис. 4.54 представлен результат выполнения данного подэтапа. На данном рисунке двойной линией справа выделены добавленные на данном подэтапе компоненты программы.

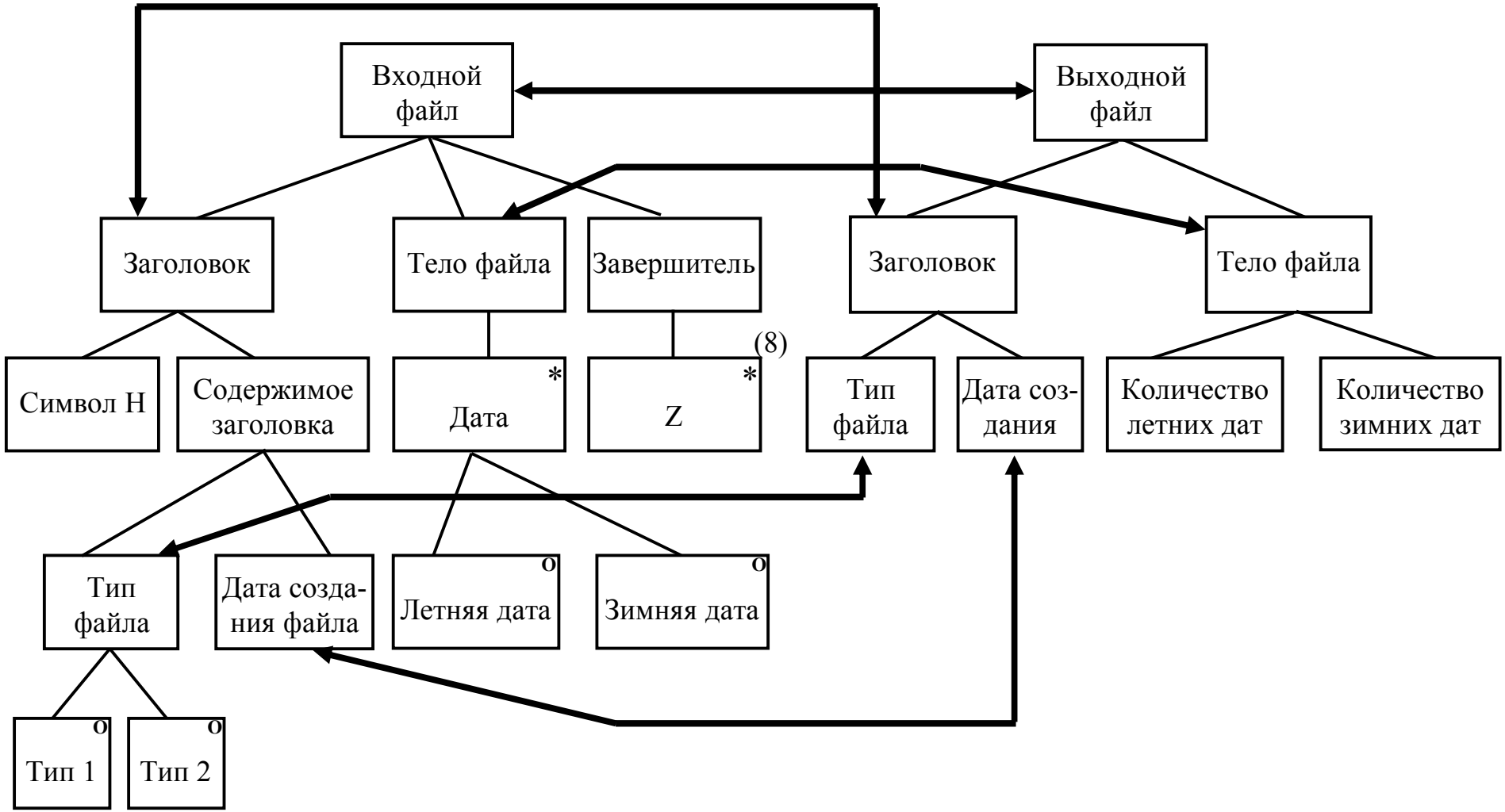


Рис. 4.51. Идентификация соответствий между входной и выходной структурами данных

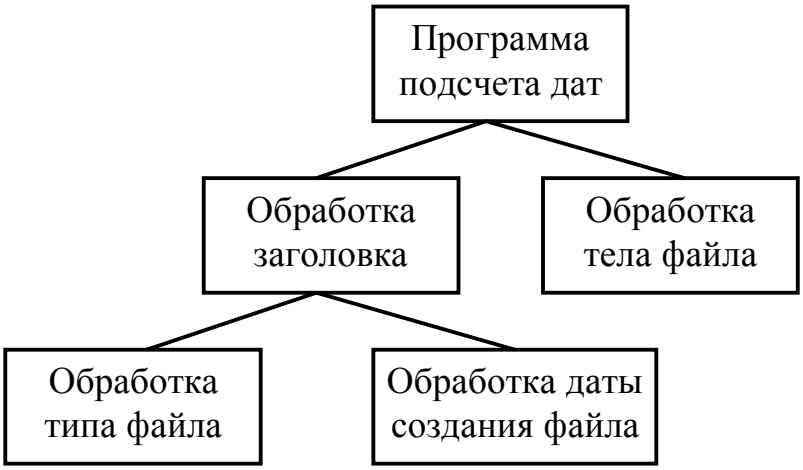


Рис. 4.52. Упрощенная структура программы

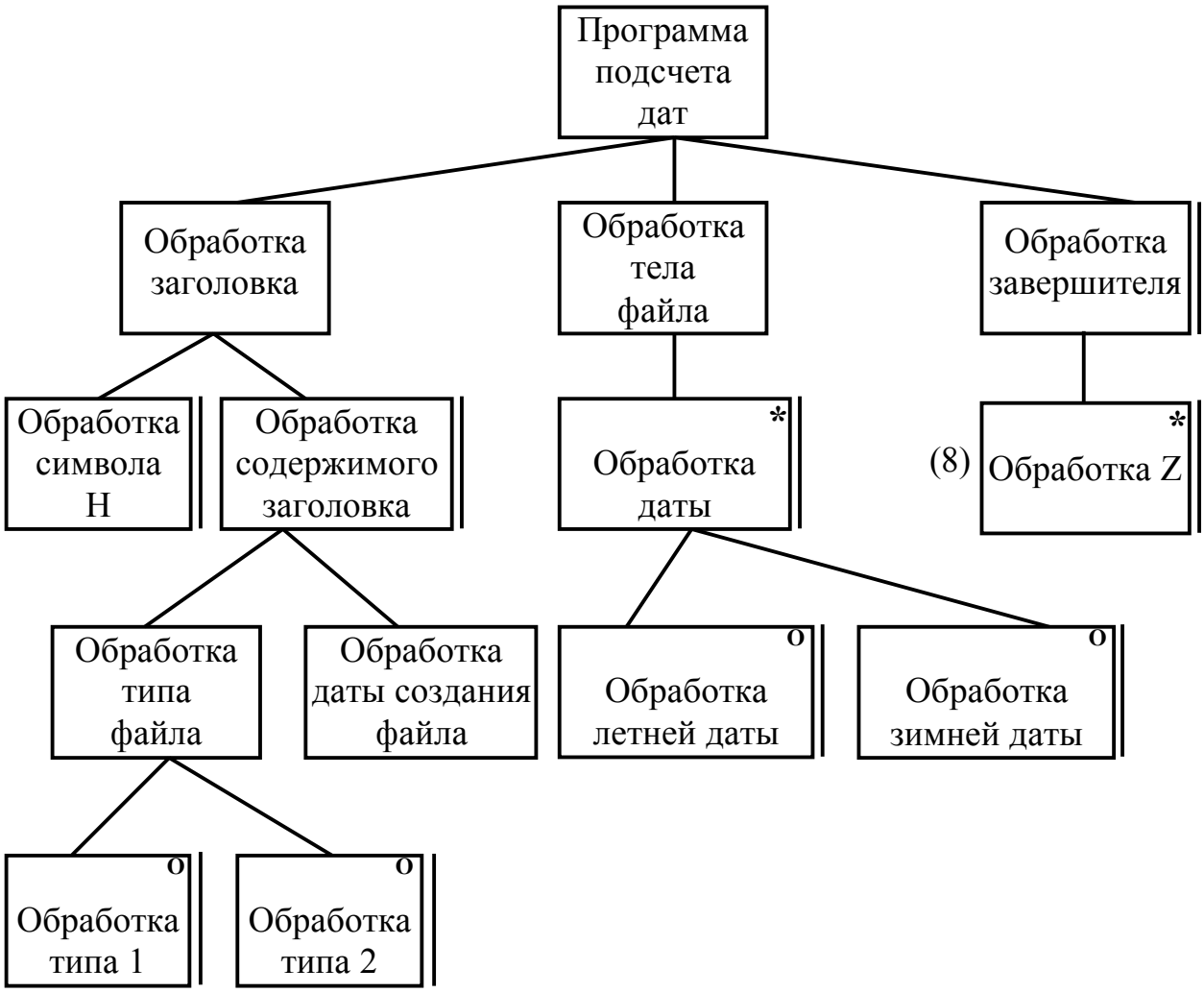


Рис. 4.53. Добавление не имеющих соответствия компонентов структуры входных данных

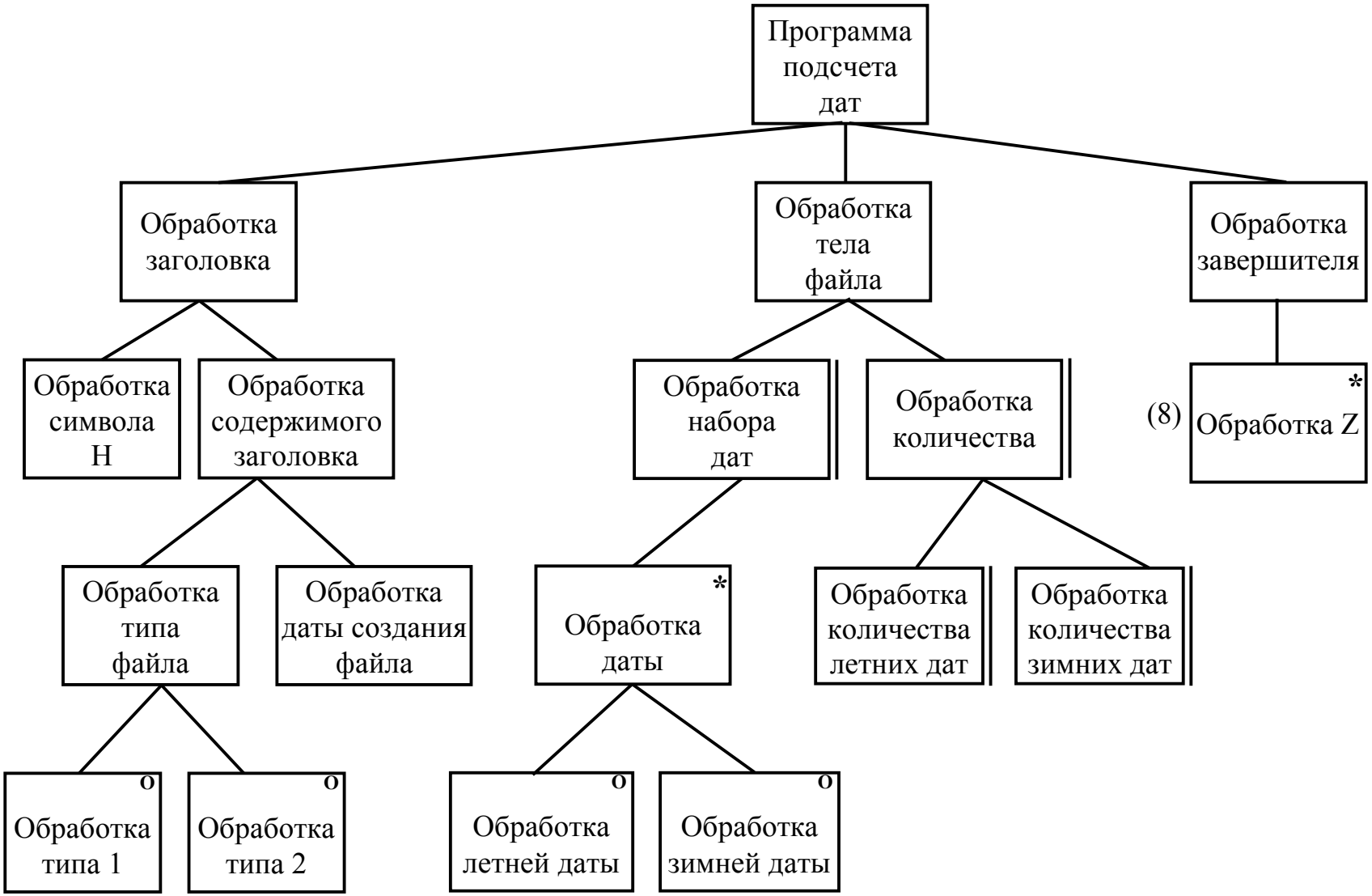


Рис. 4.54. Добавление не имеющих соответствия компонентов структуры выходных данных

В структуре выходных данных не имеют соответствия компоненты «Количество летних дат» и «Количество зимних дат». Это подкомпоненты компонента «Тело файла» выходных данных (см. рис. 4.50 и рис. 4.51). Следовательно, в проектируемой структуре программы компоненты, обрабатывающие «Количество летних дат» и «Количество зимних дат», должны стать подкомпонентами компонента «Обработка тела файла». Однако они не должны содержаться в подкомпоненте «Обработка даты» последнего (см. рис. 4.51 и рис. 4.53). Поэтому их нужно поместить вне этого подкомпонента. Так как подкомпоненты «Количество летних дат» и «Количество зимних дат» могут возникнуть только после обработки всего набора дат, то соответствующие компоненты программы нужно поместить за компонентом «Обработка даты». Поэтому в это место структуры программы добавляется компонент «Обработка количества».

Добавление изменяет структуру компонента «Обработка тела файла». Ранее этот компонент был повторением с повторяемой частью «Обработка даты». Теперь он стал последовательностью. Напомним, что среди последовательных подкомпонентов не должно быть повторяемых. Поэтому в структуру компонента «Обработка тела файла» добавляется подкомпонент «Обработка набора дат».

Таким образом, в результате трех подэтапов третьего этапа сформирована управляющая структура программы (см. рис. 4.54).

На следующем этапе необходимо снабдить управляющую структуру программы выполняемыми операциями.

Этап 4. Перечисление и распределение выполняемых операций

Для составления точного списка операций, которые должна выполнять проектируемая программа, необходимо знать:

- спецификацию того, что должно делать программное средство;
- язык программирования, на котором должна быть реализована программа (для определения уровня детализации описания операций).

Для составления списка операций рекомендуется пользоваться *контрольным перечнем операций*. В состав данного перечня входят следующие *группы операций*.

I. Операции завершения, служащие для прекращения работы программы – по одной на программу (например «Стоп», «Конец» и т.п.).

II. Операции открытия и закрытия (например для файлов).

III. Операции вывода результатов (например «Писать»).

IV. Вычисления.

V. Операции ввода входных данных (например «Читать»).

VI. Управление внутренними переменными (например запоминание, инициализация и т.п.).

Ниже описан перечень операций для программы, управляющая структура которой представлена на рис. 4.54. При этом используется один из метаязыков, называемый *метаязыком структурированного изложения* [26].

I. *Операции завершения*

Как уже отмечалось, для проектируемой программы необходима одна операция данной группы:

1. Стоп.

II. *Операции открытия и закрытия*

В рассматриваемой программе используется два файла – входной и выходной. Поэтому из операций данной группы для проектируемой программы необходимы следующие операции:

2. Открыть входной файл.
3. Открыть выходной файл.
4. Закрыть входной файл.
5. Закрыть выходной файл.

III. *Операции вывода результатов*

Для определения операций вывода исследуется структура выходных данных (см. рис. 4.50). Каждому элементарному компоненту данной структуры соответствует своя операция вывода:

6. Писать тип файла.
7. Писать дату создания файла.
8. Писать количество летних дат.
9. Писать количество зимних дат.

IV. *Вычисления*

Вычисления требуются для формирования выходных данных. Поэтому анализируется каждая из операций вывода и определяется, какие вычисления или обработка нужны для получения соответствующих выходных данных.

В рассматриваемом примере для операций 6 и 7 вычислений не нужно – информация по условию задачи должна просто переписываться из входного файла в выходной (см. условие примера 4.8).

Для операций 8, 9 необходим подсчет количества летних (Кл) и зимних (Кз) дат. Поэтому появляются операции:

10. Кл := Кл + 1.
11. Кз := Кз + 1.

V. *Операции ввода входных данных*

В рассматриваемом примере используется один входной файл. Поэтому необходима одна операция ввода:

12. Читать из входного файла.

VI. *Управление внутренними переменными*

Внутренние переменные используются, как правило, в вычислениях. Переменными, участвующими в вычислениях, в рассматриваемом примере явля-

ются счетчики Кл и Кз (см. операции 10, 11). Вначале их нужно обнулить:

13. Кл := 0.

14. Кз := 0.

Итак, полный набор выполняемых операций в рассматриваемом примере содержит 14 операций.

Для размещения данных операций в нужные места структуры программы (см. рис. 4.54) по каждой из операций необходимо определить следующую информацию:

1) когда и какую часть данных обрабатывает операция (например один раз на файл, один раз на запись и т.п.);

2) где эта часть данных обрабатывается в структуре программы. В результате идентифицируется компонент программы, в который вносится анализируемая операция. Эта операция вносится как его последовательный подкомпонент;

3) на каком последовательном месте должна появиться операция в компоненте программы (слева или справа – в начале или в конце компонента).

Результат размещения выполняемых операций в структуре программы представлен на рис. 4.55. На данном рисунке двойной линией справа выделены добавленные на данном этапе компоненты программы.

Операция 1 (Стоп) встречается один раз на файл. Файл обрабатывается компонентом «Программа подсчета дат». Поэтому операция «Стоп» должна являться последовательным подкомпонентом этого компонента. Данная операция встречается в самом конце программы. Поэтому размещается на правом краю компонента «Программа подсчета дат».

По аналогии *операции 2 – 5 (Открытие и закрытие файлов)* также являются последовательными подкомпонентами компонента «Программа подсчета дат». Но операции 2, 3 (Открытие входного и выходного файлов) необходимо поместить вначале (слева) программы, а операции 4, 5 (Закрытие файлов) – справа, но перед операцией 1 (Стоп).

Операция 6 (Писать тип файла) выполняется один раз на тип файла. Тип файла обрабатывается компонентом «Обработка типа файла». Но у него уже есть два выбираемых подкомпонента «Обработка типа 1», «Обработка типа 2». Операция 6 должна появляться после любого из этих подкомпонентов. Таким образом, компонент «Обработка типа файла» должен превратиться из выбора в последовательность двух подкомпонентов: сначала выбор, затем операция 6. Поэтому в программу вводится дополнительный компонент «Обработка содержимого типа».

Очевидно, что *операция 7 (Писать дату создания файла)* является подкомпонентом компонента «Обработка даты создания файла».

Аналогично, *операции 8, 9 (Писать количество летних дат, Писать количество зимних дат)* распределяются по компонентам «Обработка количества летних дат», «Обработка количества зимних дат».

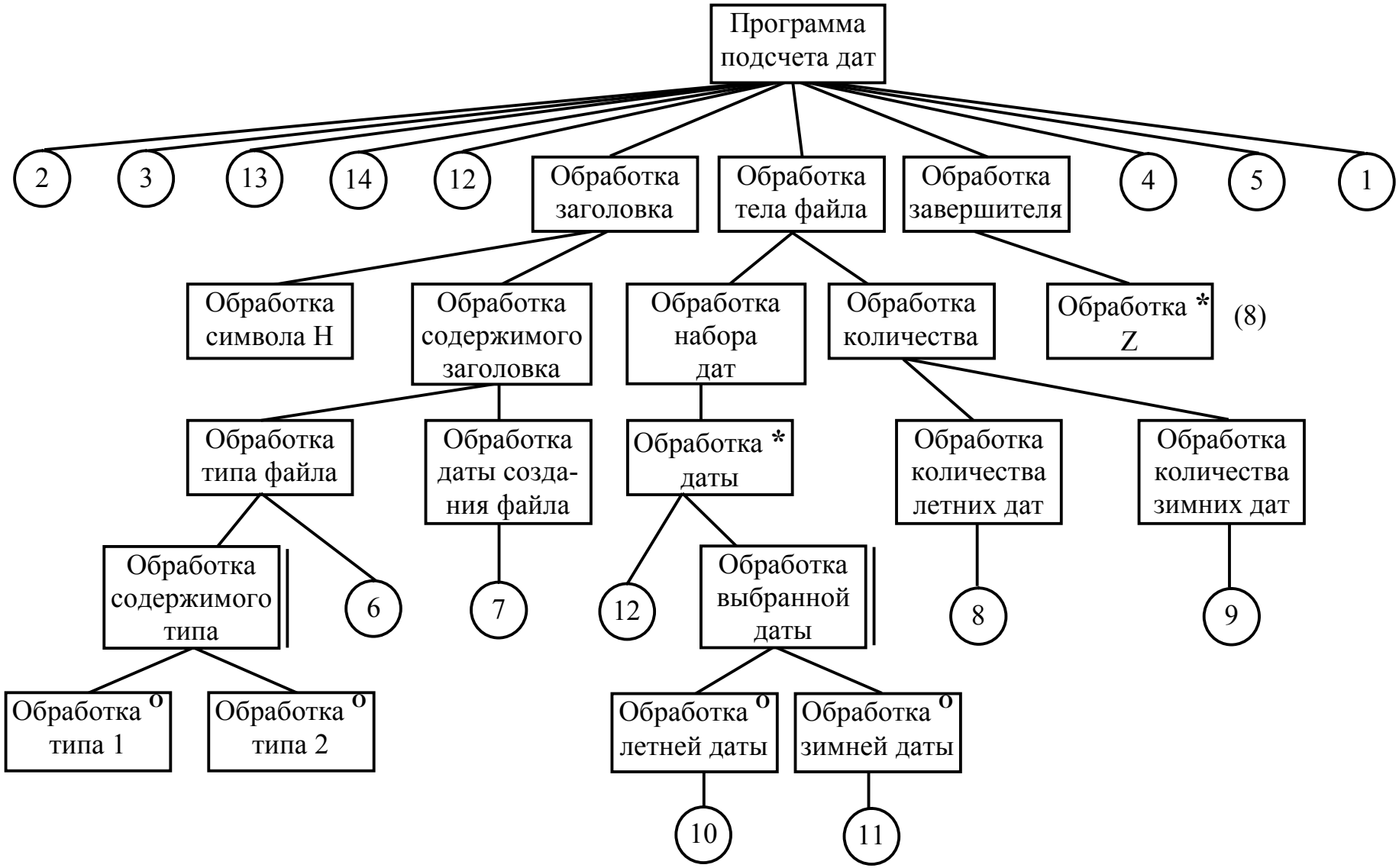


Рис. 4.55. Размещение выполняемых операций

Точно также *операции 10, 11 (Кл := Кл + 1, Кз := Кз + 1)* являются подкомпонентами компонентов «Обработка летней даты», «Обработка зимней даты».

Операции 13, 14 (Кл := 0, Кз := 0) выполняются один раз на всю программу. Поэтому они являются подкомпонентами компонента «Программа подсчета дат» и должны располагаться вначале (слева) программы.

Операция 12 (Читать из входного файла) должна появиться первый раз перед обработкой заголовка (один раз на всю программу). Поэтому операция 12 подключается к компоненту «Программа подсчета дат» перед подкомпонентом «Обработка заголовка» (слева от него). Затем операция 12 должна появляться перед обработкой каждой даты. Поэтому ее нужно подключить как последовательный подкомпонент компонента «Обработка даты». Таким образом, компонент «Обработка даты» из выбора превращается в последовательность и к нему добавляется подкомпонент «Обработка выбранной даты» (см. рис. 4.55).

В результате описанных выше действий в управляющей структуре проектируемой программы размещены все выполняемые операции. На следующем этапе разрабатывается текст программы. При этом используется один из метаязыков структурированного описания программ.

Этап 5. Создание текста программы на метаязыке структурированного описания

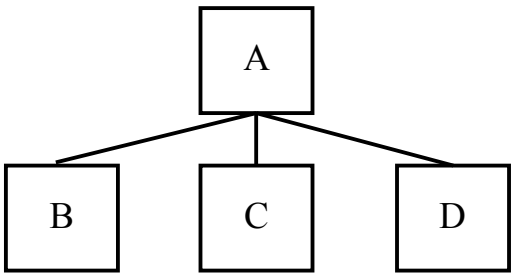
Каждая из основных конструкций, используемых в методе JSP (последовательность, выбор, повторение), может быть записана на метаязыке структурированного описания, который является одной из разновидностей словесного описания алгоритма [26].

Рис. 4.56 иллюстрирует представление конструкции последовательности при использовании структурированного описания. Данное представление состоит из списка подкомпонентов последовательности в порядке слева направо с меткой посл (последовательность) в начале и с меткой конец в конце.

Рис. 4.57 содержит представление конструкции выбора в структурированном описании. Представление выбора начинается меткой выб (выбор) и заканчивается меткой конец. «Условие P» – это условие, при истинности которого выполняется процесс **P**. Аналогичное назначение имеют «Условие Q», «(Условие R)». Последнее условие берется в скобки, так как в программе, написанной на конкретном языке программирования, оно, как правило, не пишется (соответствует ветви «иначе»).

Рис. 4.58 иллюстрирует представление конструкции повторения в структурированном описании. Представление начинается меткой повт (повторение) и заканчивается меткой конец. «Пока условие X» описывает условие, при котором повторяется выполнение процесса **X**.

Последовательность

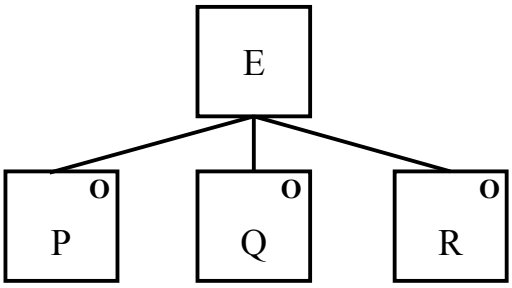


Структурированное описание

A посл
Выполнить B
Выполнить C
Выполнить D
A конец

Рис. 4.56. Конструкция последовательности
и ее структурированное описание

Выбор

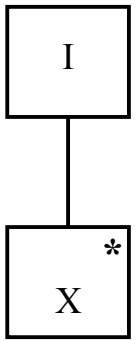


Структурированное описание

E выб Условие P
Выполнить P
E или Условие Q
Выполнить Q
E или (Условие R)
Выполнить R
E конец

Рис. 4.57. Конструкция выбора и ее структурированное описание

Повторение



Структурированное описание

I повт Пока условие X
Выполнить X
I конец

Рис. 4.58. Конструкция повторения и ее структурированное описание

Таким образом, чтобы перейти к структурированному описанию программы на базе ее структуры и размещенных выполняемых операций, необходимо определить условия для конструкций выбора и повторения.

Для обеих конструкций выбора «Обработка содержимого типа» и «Обработка выбранной даты» (см. рис. 4.55) условия непосредственно связаны с проверкой текущего содержимого обрабатываемых данных.

Для повторения «Обработка набора дат» условием повторения является отсутствие завершителя (8 символов **Z**). Таким образом, с завершителем ничего не нужно делать, кроме распознавания его появления.

Еще одним компонентом, не требующим выполнения операций, является компонент «Обработка символа Н».

Таким образом, *структурированное описание программы подсчета дат*, схема которой представлена на рис. 4.55, имеет следующий вид:

```
Программа подсчета дат посл
        Открыть входной файл
        Открыть выходной файл
        Кл := 0
        Кз := 0
        Читать из входного файла
        Обработка заголовка посл
                Писать тип файла
                Писать дату создания файла
        Обработка заголовка конец
        Обработка тела файла посл
                Обработка набора дат повт  пока не завершитель ((8) Z)
                        Обработка даты посл
                                Читать из входного файла
                                Обработка выбранной даты выб условие летней даты
                                        Кл := Кл + 1
                                Обработка выбранной даты или (условие зимней даты)
                                        Кз := Кз + 1
                                Обработка выбранной даты конец
                        Обработка даты конец
                Обработка набора дат конец
                Обработка количеств посл
                        Писать количество летних дат
                        Писать количество зимних дат
                Обработка количеств конец
        Обработка тела файла конец
        Закрыть входной файл
        Закрыть выходной файл
        Стоп
Программа подсчета дат конец.
```

Из приведенного примера видно, что в структурированном описании программы присутствуют только те компоненты ее структуры (сравните с рис. 4.55), которые содержат выполняемую операцию. Отступы в тексте структурированного описания указывают уровень вложенности соответствующей части программы. Например, «Обработка количеств посл» находится на глубине второго уровня вложенности в программной структуре.

Структурированное описание легко преобразуется в код программы, написанной на любом языке программирования.

Очевидно, что применение метода JSP достаточно сложно и громоздко даже для программ небольшой сложности. Поэтому данный метод применяется, как правило, на нижних уровнях проектирования модульных ПС, а также при разработке программных модулей и программ невысокой сложности при условии высокой степени структуризации данных. Таким образом, при проектировании модульной структуры программного средства вначале обычно используются методы нисходящего проектирования, а затем может применяться метод JSP.

Развитием метода JSP является метод JSD (Jackson System Development), также разработанный М. Джексоном. Метод JSD рассмотрен в п. 5.5.1.

Резюме

Предложенный М. Джексоном метод JSP реализуется с помощью пяти этапов: проектирование структур входных и выходных данных, идентификация соответствий между структурами данных, проектирование структуры программы, перечисление и распределение выполняемых операций, создание текста программы на метаязыке структурированного описания.

4.7. Оценка структурного разбиения программы на модули

В предыдущих подразделах разд. 4 рассмотрены классические методы проектирования модульных ПС.

Для оценки корректности и эффективности структурного разбиения программы на модули необходимо оценить характеристики получившихся модулей. Существуют различные меры оценки характеристик модулей. Ниже рассматриваются две из них – связность и сцепление [22].

4.7.1. Связность модуля

Связность модуля определяется как мера независимости его частей. Чем выше связность модуля, тем больше отдельные части модуля зависят друг от друга и тем лучше результат проектирования. Для количественной оценки связности используется понятие ***силы связности модуля***. Типы связности мо-

дулей и соответствующие им силы связности представлены в табл. 4.4 [22].

Модуль с *функциональной связностью* выполняет единственную функцию и реализуется обычно последовательностью операций в виде единого цикла. Если модуль спроектирован так, чтобы изолировать некоторый алгоритм, он имеет функциональную связность. Он не может быть разбит на два других модуля, имеющих связность того же типа. Примером модуля с функциональной связностью является, например, модуль сортировки дат (см. пп. 4.3.2, 4.3.3). Другой пример – модуль, который может быть разбит только на исток, преобразователь и сток, так как он выполняет единую функцию (см. п. 4.3.4).

Таблица 4.4

Типы и силы связности модулей

Связность	Сила связности
1. Функциональная	10 (сильная связность)
2. Последовательная	9
3. Коммуникативная	7
4. Процедурная	5
5. Временная	3
6. Логическая	1
7. Связность по совпадению	0 (слабая связность)

Модуль, имеющий *последовательную связность*, может быть разбит на последовательные части, выполняющие независимые функции, совместно реализующие единую функцию. Модуль с последовательной связностью реализуется обычно как последовательность циклов или операций.

Модуль, имеющий *коммуникативную связность*, может быть разбит на несколько функционально независимых модулей, использующих общую структуру данных. Общая структура данных является основой его организации как единого модуля. Если модуль спроектирован так, чтобы упростить работу со сложной структурой данных, изолировать эту структуру, он имеет коммуникативную связность. Такой модуль предназначен для выполнения нескольких различных и независимо используемых функций над структурой данных (например запоминание некоторых данных, их поиск и редактирование).

Процедурная связность характерна для модуля, управляющие конструкции которого организованы в соответствии со схемой алгоритма, но без выделения его функциональных частей. Такая структура модуля возникает, напри-

мер, при расчленении длинной программы на части в соответствии с передачами управления, но без определения каких-либо функций при выборе разделительных точек; при группировании альтернативных частей программы; если для уменьшения размеров модуль с функциональной связностью делится на два модуля (например, исходный модуль содержит объявления, подпрограммы и раздел операторов для выполнения единой функции; после его разделения один модуль содержит объявления и подпрограммы, а другой – раздел операторов).

Модуль, содержащий функционально не связанные части, необходимые в один и то же момент обработки, имеет *временную связность* (*связность по классу*). Данный тип связности имеет, например, модуль инициализации, реализующий все требуемые в начале выполнения программы функции и начальные установки. Для увеличения силы связности модуля функции инициализации целесообразно разделить между другими модулями, выполняющими обработку соответствующих переменных или файлов или включить их выполнение в управляющий модуль, но не выделять в отдельный модуль.

Если в модуле объединены операторы только по принципу их функционального подобия (например, все они предназначены для проверки правильности данных), а для настройки модуля применяется алгоритм переключения, то модуль имеет *логическую связность*. Его части ничем не связаны, а лишь похожи. Например, модуль, состоящий из разнообразных подпрограмм обработки ошибок, имеет логическую связность. Однако если с помощью этого модуля может быть получена вся выходная информация об ошибках, то он имеет коммуникативную связность, поскольку изолирует данные об ошибках.

Модуль имеет *связность по совпадению*, если его операторы объединяются произвольным образом.

Резюме

Модули верхних уровней иерархической структуры программы должны иметь функциональную или последовательную связность. Для модулей обслуживания предпочтительнее коммуникативная связность. Если модули имеют процедурную, временную, логическую связность или связность по совпадению, это свидетельствует о недостаточно продуманном их проектировании. Необходимо добиваться функциональной связности проектируемых модулей.

4.7.2. Сцепление модулей

Сцепление модулей – это мера относительной независимости модулей. Сцепление влияет на сохранность модулей при модификациях и на понятность их исходных текстов. Слабое сцепление определяет высокий уровень независимости модулей. Независимые модули могут быть модифицированы без переделки других модулей.

Два модуля являются полностью независимыми, если в каждом из них не используется никакая информация о другом модуле. Чем больше информации о другом модуле в них используется, тем менее они независимы и тем более сце-

плены. Чем очевиднее взаимодействие двух связных друг с другом модулей, тем проще определить необходимую корректировку одного модуля, зависящую от изменений, производимых в другом.

В табл. 4.5 содержатся типы сцепления модулей и соответствующие им степени сцепления [22].

Таблица 4.5

Типы и степени сцепления модулей

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

Независимое сцепление возможно только в том случае, если модули не вызывают друг друга и не обрабатывают одну и ту же информацию.

Модули сцеплены *по данным*, если они имеют общие простые элементы данных, передаваемые от одного модуля к другому как параметры. В вызывающем модуле определены только имя вызываемого модуля, типы и значения переменных, передаваемых как параметры. Вызываемый модуль может не содержать никакой информации о вызывающем. В этом случае изменения в структуре данных в одном из модулей не влияют на другой модуль.

Например, в вызывающем модуле определена такая структура данных, как массив. В вызываемый модуль передается в качестве параметра элемент массива. При этом изменения в структуре данных вызывающего модуля не повлияют на вызываемый модуль.

Модули со сцеплением по данным не имеют общей области данных (общих глобальных переменных).

Если модули сцеплены по данным, то по изменениям, производимым в объявленных параметрах, легко можно определить модули, на которые эти изменения повлияют.

Модули сцеплены *по образцу*, если в качестве параметров используются структуры данных (например, в качестве параметра передается массив). Недос-

таток такого сцепления заключается в том, что в обоих модулях должна содержаться информация о внутренней структуре данных. Если модифицируется структура данных в одном из модулей, то необходимо корректировать и другой модуль. Следовательно, увеличивается вероятность появления ошибок при разработке и сопровождении ПС.

Модули сцеплены *по общей области*, если они имеют доступ к общей области памяти (например используют общие глобальные данные). В этом случае возможностей для появления ошибок при модификации структуры данных или одного из модулей намного больше, поскольку труднее определить модули, нуждающиеся в корректировке.

Модули сцеплены *по управлению*, если какой-либо из них управляет решениями внутри другого с помощью передачи флагов, переключателей или кодов, предназначенных для выполнения функций управления. Таким образом, в одном из модулей содержится информация о внутренних функциях другого.

Например, если модуль имеет логическую связность и при его вызове используется переключатель требующейся функции, то вызываемый и вызывающий модули сцеплены по управлению.

Модули сцеплены *по внешним ссылкам*, если у одного из них есть доступ к данным другого модуля через внешнюю точку входа. Таким путем осуществляется неявное влияние на функционирование другого модуля. Сцепление этого типа возникает, например, тогда, когда внутренние процедуры одного модуля оперируют с глобальными переменными другого модуля.

Модули сцеплены *по кодам*, если коды их команд объединены друг с другом. Например, для одного из модулей доступны внутренние области другого модуля без обращения к его точкам входа, то есть модули используют общий участок памяти с командами. Это сцепление возникает, когда модули проектируются как отдельные подпрограммы, путь через которые начинается в различных точках входа, но приводит к общему сегменту кодов. Например, некоторый модуль реализует функции синуса и косинуса с учетом того, что

$$\mathrm{Cos}(x) = \mathrm{Sin}(\pi/2 - x).$$

Путь через точки входа Sin и Cos ведет к общему участку команд модуля.

Следует иметь в виду, что если модули косвенно обращаются друг к другу (например, связь между ними осуществляется через промежуточные модули), то между ними также существует сцепление.

Резюме

Различают независимое сцепление модулей, сцепление по данным, по образцу, по общей области, по управлению, по внешним ссылкам, по кодам. Сцепление модулей зависит от спроектированной структуры данных и способов взаимодействия между модулями. Необходимо использовать простые параметры и не применять общих областей памяти.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Назовите основные достоинства структурного программирования.
2. Перечислите теоретические основы структурного программирования.
3. В чем заключается принцип Бома–Джакопини при реализации структурированных программ?
4. Какие преобразования называются преобразованиями Бома–Джакопини? Поясните назначение данных преобразований.
5. Каким образом теоретические основы структурного программирования реализуются в языках программирования?
6. Назовите основные методы графического представления структурированных схем алгоритмов.
7. Представьте графически конструкции структурного программирования, используемые в методе Дамке.
8. Нарисуйте схему алгоритма решения некоторой конкретной задачи, представленную по методу Дамке.
9. Представьте графически конструкции структурного программирования, используемые в схемах Насси–Шнейдермана.
10. Нарисуйте алгоритм решения некоторой конкретной задачи, представленный с помощью схем Насси–Шнейдермана.
11. Дайте определение идеальной модульной программы.
12. Перечислите признаки модульности программ.
13. Назовите основные достоинства и недостатки модульного проектирования.
14. Дайте классификацию классических методов структурного проектирования модульных программных средств.
15. Поясните сущность методов нисходящего проектирования.
16. Перечислите основные классические стратегии реализации нисходящего проектирования.
17. Поясните сущность и назовите способы реализации стратегии пошагового уточнения.
18. Поясните сущность проектирования программ с помощью псевдокода и управляющих конструкций структурного программирования.
19. Поясните сущность проектирования программ с помощью использования комментариев для описания обработки данных.
20. Поясните сущность стратегии анализа сообщений.
21. Поясните сущность методов восходящего проектирования. Назовите случаи, когда применение данных методов является целесообразным.
22. Перечислите и охарактеризуйте способы сочетания методов нисходящего и восходящего проектирования.
23. Сформулируйте базовое положение метода JSP Джексона.
24. Назовите и изобразите графически основные конструкции построения структур данных, используемые в методе JSP Джексона.

25. Приведите пример иерархической структуры данных, представленной в нотации Джексона.
26. Приведите пример сетевой структуры данных, представленной в нотации Джексона.
27. Приведите пример реляционной структуры данных, представленной в нотации Джексона.
28. Перечислите этапы проектирования программы по методу JSP Джексона.
29. Назовите первый этап метода JSP Джексона и на конкретном примере поясните правила его выполнения.
30. Назовите второй этап метода JSP Джексона и на конкретном примере поясните правила его выполнения.
31. Назовите третий этап метода JSP Джексона и на конкретном примере поясните правила его выполнения.
32. Назовите четвертый этап метода JSP Джексона и на конкретном примере поясните правила его выполнения.
33. Назовите пятый этап метода JSP Джексона и на конкретном примере поясните правила его выполнения.
34. Что такое связность модуля?
35. Назовите и охарактеризуйте типы и силы связности модулей.
36. Что такое сцепление модулей?
37. Назовите и охарактеризуйте типы и степени сцепления модулей.

РАЗДЕЛ 5. CASE-ТЕХНОЛОГИИ СТРУКТУРНОГО АНАЛИЗА И ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СРЕДСТВ

5.1. Общие сведения о CASE-технологиях

В соответствии с положениями стандарта *СТБ ИСО/МЭК 12207–2003* [9] процесс разработки сложных систем и ПС состоит из тринадцати работ (см. подразд. 1.2). Как показывают исследования, большинство ошибок вносится в системы и ПС при выполнении ранних работ процесса разработки (работы 2 – 6, связанные с анализом и проектированием). Существенно меньше ошибок возникает при осуществлении программирования, тестирования и последующих работ, причем устранять такие ошибки гораздо проще по сравнению с ошибками ранних работ.

Как правило, ошибки, возникающие при выполнении ранних работ процесса разработки системы или программного средства, являются следствием неполноты или некорректности функциональной спецификации или несогласованности между спецификацией и результатами проектирования. Очевидно, что основная причина этого кроется в несоответствии методов, используемых при осуществлении ранних работ процесса разработки, целям данных работ.

С учетом этого с 70-х гг. XX в. ведется разработка методов структурного анализа и проектирования, специально предназначенных для использования при выполнении ранних работ процесса разработки сложных систем широкого профиля и позволяющих существенно сократить возможности внесения ошибок в разрабатываемую систему. Напомним, что основной *целью* методов структурного анализа и проектирования является разделение сложных систем на части с последующей иерархической организацией этих частей.

Наиболее известными и используемыми из данных методов являются:

- метод структурного анализа и проектирования SADT Росса, в дальнейшем явившийся основой методологии функционального моделирования IDEF0;
- методы, ориентированные на потоки данных (методы Йодана, ДеМарко, Гейна, Сарсона), в дальнейшем явившиеся основой методологии структурного анализа потоков данных DFD; один из таких методов – анализ сообщений – рассмотрен в п. 4.3.4;
- методы структурирования данных (методы JSP Джексона, Орра, Чена), в дальнейшем явившиеся основой методологий JSD Джексона, информацион-

ного моделирования IDEF1 и IDEF1X и др.; метод JSP Джексона подробно рассмотрен в подразд. 4.6.

Появление новых методов анализа и проектирования вызвало необходимость создания ПО, позволяющего автоматизировать их использование при разработке больших систем. С середины 80-х гг. XX в. начал формироваться рынок ПС, названных CASE-средствами.

Первоначально *термин CASE* трактовался как Computer Aided Software Engineering (компьютерная поддержка проектирования ПО). В настоящее время данному термину придается более широкий смысл, и он расшифровывается как Computer Aided System Engineering (компьютерная поддержка проектирования систем). Современные CASE-средства ориентируются на моделирование предметной области, разработку спецификаций, проектирование сложных систем широкого назначения. При этом учитывается, что программное средство – это частный случай системы вообще. Считается, что разработка ПС включает в себя практически те же этапы, что и разработка систем общего назначения.

С учетом вышеизложенного введено понятие CASE-технологии.

CASE-технология – это совокупность методологий разработки и сопровождения сложных систем (в том числе ПС), поддерживаемая комплексом взаимосвязанных средств автоматизации.

Основные цели использования CASE-технологий при разработке ПС – отделить анализ и проектирование от программирования и последующих работ процесса разработки, предоставив разработчику соответствующие методологии визуального анализа и проектирования.

С середины 70-х гг. XX в. в США финансировался ряд проектов, ориентированных на разработку методов описания и моделирования сложных систем [25]. Один из них – проект ***ICAM*** (Integrated Computer-Aided Manufacturing). Его целью являлась разработка подходов, обеспечивающих повышение эффективности производства благодаря систематическому внедрению компьютерных технологий. В соответствии с проектом ICAM было разработано семейство трех методологий ***IDEF*** (ICAM DEFinition), позволяющих моделировать различные аспекты функционирования производственной среды или системы:

- ***IDEF0*** – методология функционального моделирования производственной среды или системы; отображает структуру и функции системы, а также потоки информации и материальных объектов, связывающие эти функции; основана на методе SADT Росса;
- ***IDEF1*** – методология информационного моделирования производственной среды или системы; отображает структуру и содержание информационных потоков, необходимых для поддержки функций системы; основана на реляционной теории Кодда и использовании ER-диаграмм (диаграмм «Сущность –Связь») Чена;
- ***IDEF2*** – методология динамического моделирования производственной среды или системы; отображает изменяющееся во времени поведение функций, информации и ресурсов системы.

В дальнейшем семейство методологий IDEF было дополнено следующими методологиями:

- ***IDEF1X*** – методология семантического моделирования данных; в стандарте [2] названа методологией концептуального моделирования; представляет собой расширение методологии IDEF1, поэтому в литературе часто называется методологией информационного моделирования;
- ***IDEF3*** – методология моделирования сценариев процессов, происходящих в производственной среде или системе; отображает состояния объектов и потоков данных, связи между ситуациями и событиями; используется для документирования процессов, происходящих в производственной среде или системе;
- ***IDEF4*** – методология объектно-ориентированного анализа и проектирования;
- ***IDEF5*** – методология моделирования онтологии производственной среды или системы (***онтология*** – это завершенный словарь для определенной области, имеющий набор точных определений или аксиом, которые накладывают на значения терминов в данном словаре ограничения, достаточные для непротиворечивой интерпретации данных); использует утверждения о реальных объектах, их свойствах и их взаимосвязях в производственной среде или системе.

Из вышеназванных методологий наибольшее распространение нашли методологии IDEF0, IDEF1X и IDEF3.

Методологии IDEF0 и IDEF1X являются стандартизированными. В США приняты национальные стандарты ***IEEE 1320.1–1998 – Стандарт IEEE по языку функционального моделирования – Синтаксис и семантика IDEF0*** [1] и ***IEEE 1320.2–1998 – Стандарт IEEE по языку концептуального моделирования – Синтаксис и семантика IDEFIX97 (IDEF Object)*** [2]. В России действуют руководящий документ ***РД IDEF0–2000. Методология функционального моделирования IDEF0*** и рекомендации по стандартизации ***Р 50.1.028–2001. Информационные технологии поддержки жизненного цикла продукции. Методология функционального моделирования*** [11, 12].

Резюме

В настоящее время при выполнении ранних работ процесса разработки (работ, связанных с анализом предметной области и проектированием систем и ПС) широко используются CASE-технологии. Термин CASE расшифровывается как Computer Aided System Engineering (компьютерная поддержка проектирования систем). При структурном анализе и проектировании широко применяется семейство методологий IDEF.

5.2. Методология функционального моделирования IDEF0

5.2.1. Общие сведения о методологии SADT

Как уже отмечалось, основой для методологии функционального моделирования IDEF0 является методология структурного анализа и проектирования SADT. Методология ***SADT*** (Structured Analysis And Design Technique) [29] сформулирована в общих чертах Дугласом Т.Россом (компания SofTech) в 70-х гг. XX в. На рынке SADT появилась в 1975 г. К 1981 году SADT уже использовали более чем в 50 компаниях.

Основным назначением методологии SADT является моделирование предметной области с целью определения требований к разрабатываемой системе или программному средству и с целью их проектирования. Методология SADT может применяться при выполнении ранних работ процесса разработки системы или программного средства (работы 2 – 5, см. подразд. 1.2).

К *достоинствам методологии SADT* можно отнести:

1) универсальность – SADT может использоваться для проектирования не только ПС, но и сложных систем любого назначения (например управление и контроль, аэрокосмическое производство, телефонные сети, учет материально-технических ресурсов и др.);

2) SADT – методология, достаточно просто отражающая такие системные характеристики, как управление, обратная связь и исполнители;

3) SADT имеет развитые процедуры поддержки коллективной работы;

4) SADT предназначена для использования на ранних этапах создания систем или ПС (при выполнении работ, связанных с анализом предметной области, разработкой требований и проектированием);

5) SADT сочетается с другими структурными методами проектирования.

В моделировании SADT определены два направления: *функциональные модели* выделяют события в системе, *модели данных* выделяют объекты (данные) системы, связывающие функции между собой и с их окружением. В обоих случаях используется один и тот же графический язык блоков и дуг (но блоки и дуги меняются ролями) [19]. При наиболее полном моделировании возможно использование взаимодополняющих моделей обоих типов.

Однако наибольшее распространение и дальнейшее развитие получил функциональный вариант методологии SADT, на базе которого разработана и в дальнейшем стандартизирована методология функционального моделирования IDEF0 [1, 12].

Резюме

Методология структурного анализа и проектирования SADT предназначена для моделирования предметной области с целью определения требований

к разрабатываемой системе и ее проектирования. Методология SADT применяется при выполнении ранних работ процесса разработки системы или программного средства. В первую очередь это такие работы, как анализ требований к системе, проектирование системной архитектуры, анализ требований к программным средствам, проектирование программной архитектуры (см. подразд. 1.2). Развитием методологии SADT является методология IDEF0.

5.2.2. Основные понятия IDEF0-модели

При IDEF0-моделировании используются следующие понятия и определения [29].

Под ***системой*** подразумевается совокупность взаимодействующих компонентов и взаимосвязей между ними.

Моделированием называется процесс создания точного описания системы. IDEF0-методология предназначена для создания описания систем и основана на концепциях системного моделирования.

Под ***моделью IDEF0*** подразумевается графическое и текстовое представление результатов анализа предметной области, разработанное с определенной целью и с выбранной точки зрения и идентифицирующее функции системы [1]. IDEF0-модель дает полное и точное описание, адекватное системе и имеющее конкретное назначение.

Назначение описания называют ***целью модели.***

Формальное определение IDEF0-модели имеет следующий вид:

> **M** есть модель системы **S**, если **M** может быть использована для получения ответов на вопросы относительно **S** с точностью **A**.

Таким образом, *целью модели* является получение ответов на некоторую совокупность вопросов. Обычно вопросы для IDEF0-модели формируются на самом раннем этапе разработки (еще нет технического задания и спецификации). Затем основная суть этих вопросов должна быть выражена в одной-двух фразах.

С определением модели тесно связан выбор точки зрения, с которой наблюдается система и создается ее модель. ***Точка зрения*** – это позиция человека или объекта, в которую надо встать, чтобы увидеть систему в действии. Методология IDEF0 требует, чтобы модель рассматривалась все время с одной и той же позиции [11, 29].

Например, при разработке автоматизированной обучающей системы (АОС) точкой зрения может быть позиция неквалифицированного пользователя, квалифицированного пользователя, программиста и т.п.

Пример 5.1

Пусть необходимо разработать программное средство, предназначенное для автоматизации процесса выполнения студентами лабораторных работ. Разработку функциональной спецификации программного средства следует начать

с разработки IDEF0-модели процесса выполнения лабораторных работ [13].

На первом этапе разработки IDEF0-модели формулируются вопросы к ней, формируется цель модели, определяются претенденты на точку зрения, выбирается точка зрения.

Например, в *перечень вопросов* к IDEF0-модели в рассматриваемом примере могут входить такие вопросы:

- Какие этапы лабораторной работы необходимо выполнить студенту?
- Какие сотрудники участвуют в процессе выполнения студентом лабораторной работы?
- Какие виды работ должен осуществлять преподаватель во время выполнения студентом лабораторной работы?
- Какие виды работ должен осуществлять лаборант во время выполнения студентом лабораторной работы?
- Какая информация является входной при выполнении студентами лабораторных работ?
- Как влияют результаты отдельных этапов на итоги выполнения лабораторной работы?
- Что необходимо для защиты лабораторной работы?

На основании перечня вопросов формулируется *цель модели*: определить основные этапы процесса выполнения лабораторной работы, их влияние друг на друга и на результаты защиты работы с целью обучения студентов методологии IDEF0.

Претенденты на точку зрения: преподаватель, студент, лаборант. С учетом цели модели предпочтение следует отдать точке зрения преподавателя, так как она наиболее полно охватывает все этапы лабораторной работы и только с этой точки зрения можно показать взаимосвязи между отдельными этапами и обязанности участников лабораторной работы.

Субъектом моделирования является сама система. Но система не существует изолированно, она связана с окружающей средой. Иногда трудно сказать, где кончается система и начинается среда. Поэтому в методологии IDEF0 подчеркивается необходимость точного определения ***границ системы***, чтобы избежать включения в модель посторонних субъектов. IDEF0-модель должна иметь *единственный субъект*.

Таким образом, субъект определяет, что включить в модель, а что исключить из нее. Точка зрения диктует автору модели выбор нужной информации о субъекте и форму ее подачи. Цель становится критерием окончания моделирования.

Конечным результатом моделирования является набор тщательно взаимоувязанных описаний, начиная с описания самого верхнего уровня всей системы и кончая подробным описанием деталей или операций системы. Каждое из таких описаний называется ***диаграммой***.

IDEF0-модель – это древовидная структура диаграмм, где верхняя диаграмма является наиболее общей, а нижние наиболее детализированы. Каждая из диаграмм какого-либо уровня представляет собой декомпозицию некоторого компонента диаграммы предыдущего уровня.

Резюме

Методология IDEF0 создана специально для представления сложных систем путем построения моделей. IDEF0-модель – это описание системы, разработанное для единственного субъекта с определенной целью и с выбранной точки зрения. Целью служит набор вопросов, на которые должна ответить модель. Точка зрения – позиция, с которой описывается система. Цель и точка зрения – это основополагающие понятия IDEF0. Описание модели IDEF0 организовано в виде иерархии взаимосвязанных диаграмм. Вершина этой древовидной структуры представляет самое общее описание системы, а ее основание состоит из наиболее детализированных описаний.

5.2.3. Синтаксис IDEF0-диаграмм

Диаграммы являются основными рабочими элементами IDEF0-модели. Диаграммы представляют входные-выходные преобразования и указывают правила и средства этих преобразований. Каждая IDEF0-диаграмма содержит блоки (работы) и дуги (линии со стрелками). Блоки изображают функции моделируемой системы. Дуги связывают блоки вместе и отображают взаимодействия и взаимосвязи между ними.

Синтаксис блоков

Функциональные блоки на диаграмме изображаются прямоугольниками (рис. 5.1).

Блок представляет функцию или активную часть системы.

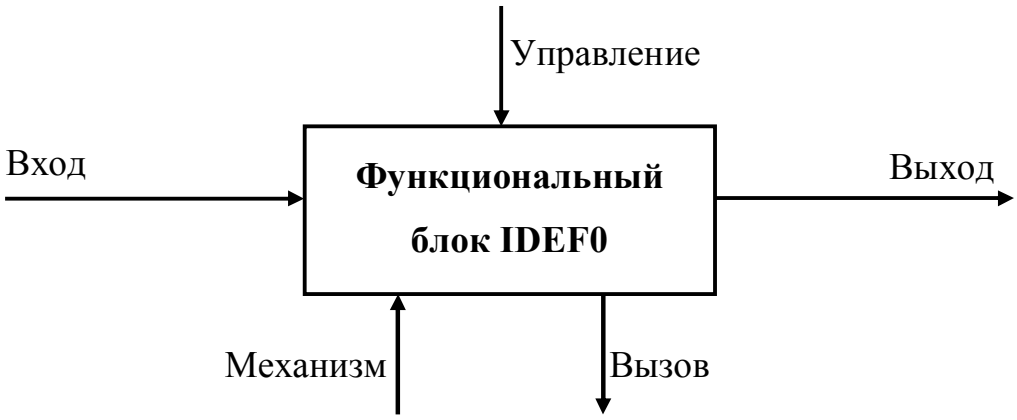


Рис. 5.1. Основная конструкция IDEF0-модели

Каждая сторона блока имеет определенное назначение. Левая сторона предназначена для входов, верхняя – для управления, правая – для выходов, нижняя – для механизмов и вызовов [1, 11, 12].

Назначение дуг

В IDEF0 различают ***пять типов дуг:*** вход (input), управление (control), выход (output), механизм (mechanism), вызов (call) [11].

В основе методологии IDEF0 лежат следующие *правила*:

1) *вход* представляет собой входные данные, используемые или преобразуемые функциональным блоком для получения результата (выхода); блок может не иметь ни одной входной дуги (например блок, выполняющий генерацию случайных чисел);

2) *выход* представляет собой результат работы блока; наличие выходной дуги для каждого блока является обязательным;

3) *управление* ограничивает или определяет условия выполнения преобразований в блоке; в качестве дуг управления могут использоваться некоторые условия, правила, стратегии, стандарты, которые влияют на выполнение функционального блока; наличие управляющей дуги для каждого блока является обязательным;

4) *механизмы* показывают, кто, что и как выполняет преобразования в блоке; механизмы определяют ресурсы, непосредственно осуществляющие эти преобразования (например, денежные средства, персонал, оборудование и т.п.); механизмы представляются стрелками, подключенными к нижней стороне блока и направленными вверх к блоку; наличие дуг механизмов для блока не является обязательным;

5) *вызовы* представляют собой специальный вид дуги и обозначают обращение из данной модели или из данной части модели к блоку, входящему в состав другой модели или другой части модели, обеспечивая их связь; с помощью дуги вызова разные модели или разные части одной модели могут совместно использовать один и тот же блок; вызовы не являются компонентом собственно методологии SADT [29], они является расширением IDEF0-методологии [1, 11, 12] и предназначены для организации коллективной работы над моделью, разделения модели на независимые модели и объединения различных моделей предметной области в одну модель; вызовы представляются стрелками, подключенными к нижней стороне блока и направленными вниз от блока; наличие дуги вызова для блока не является обязательным.

Представление блоков и дуг на диаграмме

Рассмотрим синтаксис IDEF0-диаграмм на примере IDEF0-диаграммы, содержащей основные этапы процесса выполнения лабораторной работы (см. пример 5.1 в п. 5.2.2). Данная диаграмма представлена на рис. 5.2.

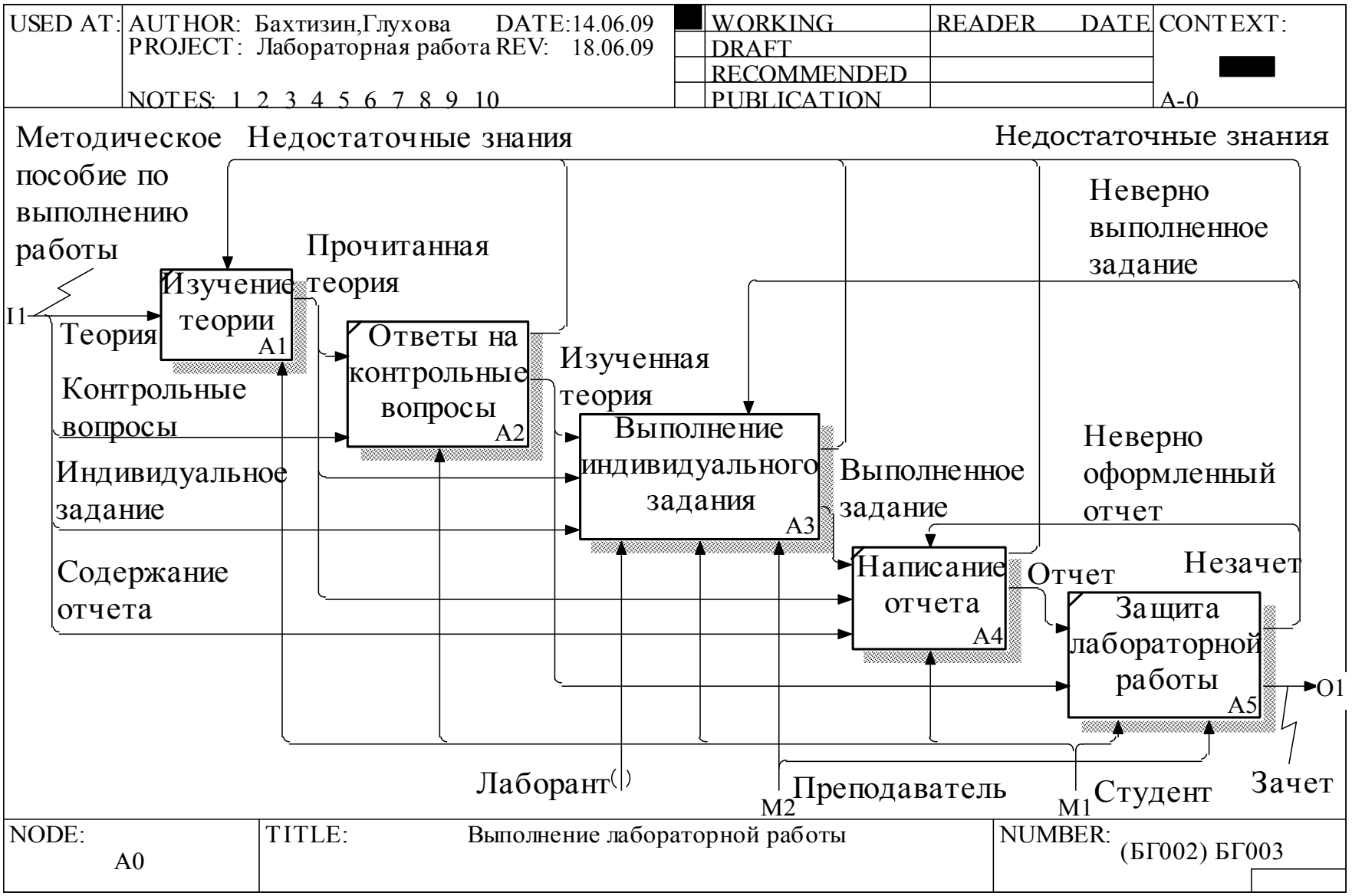


Рис. 5.2. Стандартный IDEF0-бланк и IDEF0-диаграмма, содержащая основные этапы процесса выполнения лабораторной работы

В русскоязычной литературе название IDEF0-блока принято основывать на использовании отглагольного существительного, обозначающего действие (вычисление того-то, определение того-то, обработка того-то и т.д.) [28, 29]. Блоки на рис. 5.2 имеют названия «Изучение теории», «Ответы на контрольные вопросы», «Выполнение индивидуального задания», «Написание отчета», «Защита лабораторной работы». В [1, 11, 12] рекомендуется для названий блоков использовать глаголы в неопределенной форме (в этом случае в применении к рассматриваемому примеру названия блоков будут следующими: «Изучить теорию», «Ответить на контрольные вопросы», «Выполнить индивидуальное задание», «Написать отчет», «Защитить лабораторную работу»).

Методология IDEF0 требует, чтобы в диаграмме было *не менее трех и не более шести блоков*. Это ограничение поддерживает сложность диаграмм на уровне, доступном для чтения, понимания и использования.

Блоки на IDEF0-диаграмме размещаются в порядке степени их важности. В IDEF0 этот относительный порядок называется ***доминированием***. Доминирование понимается как влияние, которое один блок оказывает на другие блоки диаграммы. В методологии IDEF0 принято располагать блоки по диагонали диаграммы (см. рис. 5.2). Наиболее доминирующий блок обычно размещается в левом верхнем углу диаграммы, наименее доминирующий – в правом нижнем углу.

Таким образом, топология диаграмм показывает, какие функции оказывают большее влияние на остальные.

Блоки на IDEF0-диаграмме должны быть пронумерованы. Нумерация блоков выполняется в соответствии с порядком их доминирования (1 – наибольшее доминирование, 2 – следующее и т. д.). Номер блока может содержать префикс **A** (Activity). Номер располагается в правом нижнем углу функционального блока.

Дуги на IDEF0-диаграмме изображаются линиями со стрелками. Для функциональных IDEF0-диаграмм дуга представляет множество объектов. Под ***объектом*** в общем случае понимаются некоторые данные (планы, машины, информация и т.п.). Основу названия дуги на IDEF0-диаграммах составляют существительные. Названия дуг называются *метками.*

Например, на диаграмме, представленной на рис. 5.2, дуги имеют названия «Индивидуальное задание», «Выполненное задание», «Отчет» и т.д.

Типы взаимосвязей между блоками

Дуги определяют, как блоки влияют друг на друга. Это влияние может выражаться:

- в передаче выходной информации к другому блоку для дальнейшего преобразования;
- в выработке управляющей информации, предписывающей, что именно должен выполнить другой блок;

- в передаче информации, определяющей средство достижения цели для другого блока.

С учетом этого в методологии IDEF0 используется ***пять типов взаимосвязей между блоками*** для описания их отношений: управление, вход, обратная связь по управлению, обратная связь по входу, выход-механизм [29].

Отношение управления возникает тогда, когда выход одного блока непосредственно влияет на работу блока с меньшим доминированием (рис. 5.3).

Отношение входа возникает тогда, когда выход одного блока становится входом для блока с меньшим доминированием (рис. 5.4).

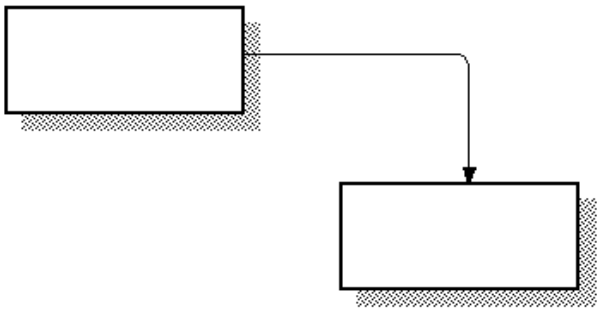


Рис. 5.3. Отношение управления

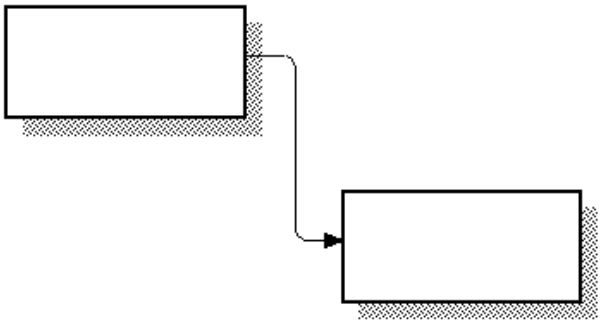


Рис. 5.4. Отношение входа

Обратные связи по управлению и по входу предназначены для представления итерации или рекурсии.

Обратная связь по управлению возникает тогда, когда выход некоторого блока влияет на работу блока с большим доминированием (рис. 5.5).

Обратная связь по входу имеет место тогда, когда выход одного блока становится входом другого блока с большим доминированием (рис. 5.6).

Связь «выход-механизм» встречается нечасто и отражает ситуацию, при которой выход одного блока становится средством достижения цели для другого блока (рис. 5.7). Данная связь характерна при распределении источников ресурсов (например, физическое пространство, оборудование, финансирование, материалы, инструменты, обученный персонал и т. п.).

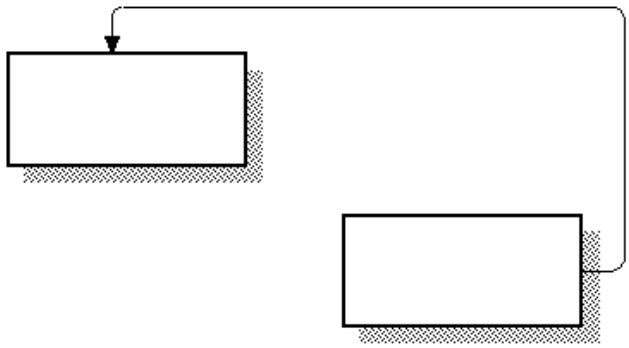


Рис. 5.5. Обратная связь по управлению

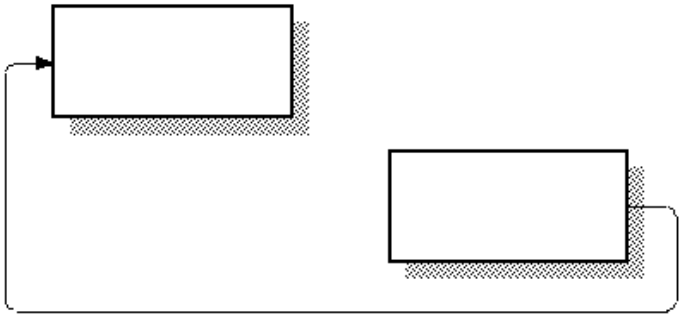


Рис. 5.6. Обратная связь по входу

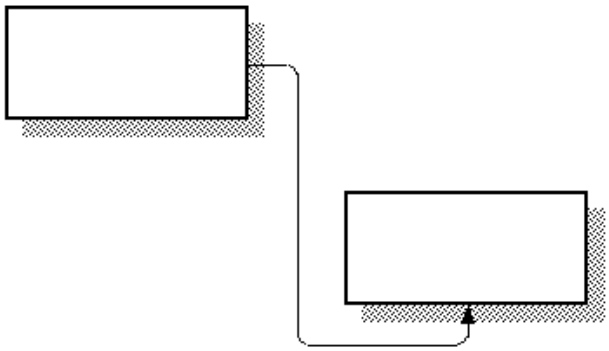


Рис. 5.7. Связь «выход-механизм»

Декомпозиция дуг

Дуга в IDEF0 редко изображает один объект. Обычно она символизирует набор объектов. Поэтому дуги могут разъединяться и соединяться.

Разветвления дуг обозначают, что все содержимое дуг или его часть может появиться в каждом ответвлении дуги. Дуга всегда помечается до разветвления, чтобы дать название всему набору (см. например входную дугу I1 «Методическое пособие по выполнению работы» диаграммы и выходную дугу

«Прочитанная теория» блока «Изучение теории» на рис. 5.2). Каждая ветвь дуги помечается или не помечается в соответствии со следующими *правилами*:

- непомеченные ветви содержат все объекты, указанные в метке дуги перед разветвлением (см. ветви выходной дуги «Прочитанная теория» блока «Изучение теории» на рис. 5.2);
- каждая метка ветви указывает, какие именно объекты исходного набора содержит ветвь (см. входную дугу I1 «Методическое пособие по выполнению работы» и ее ветви «Теория», «Контрольные вопросы», «Индивидуальное задание» и «Содержание отчета» на рис. 5.2).

При ***слиянии дуг*** результирующая дуга всегда помечается для указания нового набора объектов, возникшего после объединения (см. дугу управления «Недостаточные знания» блока «Изучение теории» на рис. 5.2). Каждая ветвь перед слиянием помечается или не помечается в соответствии со следующими *правилами*:

- непомеченные ветви содержат все объекты, указанные в общей метке дуги после слияния (см. ветви дуги управления «Недостаточные знания» блока «Изучение теории» на рис. 5.2);
- метка ветви указывает, какие конкретно объекты результирующего набора содержит ветвь.

Разветвления дуг и их соединения – это синтаксис, который позволяет описывать декомпозицию (разделение на структурные части) содержимого дуг. Разветвляющиеся и соединяющиеся дуги отражают иерархию объектов, представленных этими дугами. Следует обратить внимание на то, что синтаксис дуг позволяет одной и той же ветви участвовать как в разветвлении, так и в слиянии дуг. На рис. 5.2 такой дугой, образованной в результате разветвления и слияния ветвей, является дуга управления «Недостаточные знания» блока «Изучение теории». В данном случае выходная ветвь «Незачет» блока «Защита лабораторной работы» разветвляется на три ветви. Слияние одной из этих ветвей («Недостаточные знания») вместе с тремя аналогичными выходными дугами блоков A2, A3, A4 образовало новый набор объектов с тем же названием.

Из отдельной диаграммы редко можно понять полную иерархию дуги. Для этого требуется рассмотрение некоторой части модели. Поэтому методология IDEF0 предусматривает дополнительное описание полной иерархии объектов системы посредством формирования *глоссария* для каждой диаграммы модели и объединения этих глоссариев в *Словарь данных*. Таким образом, ***Словарь данных*** – это основное хранилище полной иерархии объектов системы.

Хронологические номера диаграмм

Для систематизации информации о диаграммах и модели в целом используется *стандартный IDEF0-бланк* (см. рис. 5.2). Каждое поле бланка имеет конкретное назначение и заполняется по определенным правилам. Данные поля будут описаны ниже по ходу изложения материала.

При создании IDEF0-модели одна и та же диаграмма может перечерчиваться несколько раз, что приводит к появлению различных ее вариантов. Чтобы различать их, в методологии IDEF0 используется *схема контроля конфигурации диаграмм*, основанная на *хронологических номерах*, или ***С-номерах***. С-номерные коды образуются из инициалов автора (авторов) и последовательных номеров. Эти коды записываются в нижнем правом углу IDEF0-бланка (БГ003, см. рис. 5.2). Если диаграмма заменяет более старый вариант, предыдущий С-номер помещается в скобках (например БГ002, см. рис. 5.2). Каждый автор проекта IDEF0 ведет реестр (список) всех созданных им диаграмм, нумеруя их последовательными целыми числами. Для этого используется специальный *бланк реестра С-номеров* IDEF0.

Резюме

Основой IDEF0-диаграмм является блок. Каждая сторона блока имеет определенное назначение (вход, управление, выход, механизм, вызов). Диаграмма в IDEF0 содержит 3 – 6 блоков, связанных дугами, и может иметь несколько версий. Чтобы различить данные версии, используются С-номера. Блоки на диаграмме представляют функции моделируемой системы, дуги – множество различных объектов системы. Блоки изображаются на диаграмме в соответствии с порядком их доминирования. Дуги могут разветвляться и объединяться.

5.2.4. Синтаксис IDEF0-моделей

Диаграммы, собранные и связанные вместе, представляют собой IDEF0-модель проектируемой или анализируемой системы. В методологии IDEF0 дополнительно к правилам синтаксиса диаграмм существуют правила синтаксиса моделей. Синтаксис IDEF0-моделей позволяет автору проекта определить границу модели, связать диаграммы в одно целое и обеспечить точное согласование между диаграммами.

Декомпозиция блоков

IDEF0-модель представляет собой иерархически организованную совокупность диаграмм. Диаграмма содержит 3 – 6 блоков. Каждый из блоков потенциально может быть детализирован на другой диаграмме. Разделение блока на его структурные части (блоки и дуги) называется ***декомпозицией***.

Декомпозиция формирует границы, то есть блок и касающиеся его дуги определяют точную границу диаграммы, представляющей декомпозицию этого блока. Эта диаграмма называется ***диаграммой-потомком***. Декомпозируемый блок называется ***родительским блоком***, а содержащая его диаграмма – ***родительской диаграммой***. Название диаграммы-потомка совпадает с функцией родительского блока [1, 11, 12, 29]. Таким образом, IDEF0-диаграмма является декомпозицией некоторой ограниченной функции (субъекта). Принцип ограничения субъекта встречается на каждом уровне.

Контекстная диаграмма

Один блок и несколько дуг на самом верхнем уровне модели используются для определения границы всей системы. Этот блок описывает общую функцию, выполняемую системой. Дуги, касающиеся этого блока, описывают главные входы, выходы, управления и механизмы этой системы.

Диаграмма, определяющая границу системы и состоящая из одного блока и его дуг, называется ***контекстной диаграммой модели***. Все, что лежит внутри блока, является частью описываемой системы, а все, лежащее вне его, образует *среду системы*.

Рис. 5.8 представляет контекстную диаграмму процесса выполнения лабораторной работы.

Общая функция модели записывается на контекстной диаграмме в виде названия блока (для рассматриваемого процесса – это выполнение лабораторной работы). Блок самого верхнего уровня модели всегда нумеруется нулем.

С контекстной диаграммой связывается цель модели и точка зрения.

Декомпозицией контекстной диаграммы (ее диаграммой-потомком) является диаграмма, содержащаяся на рис. 5.2.

Название (поле TITLE IDEF0-бланка) диаграммы декомпозиции совпадает с названием декомпозируемого блока родительской диаграммы.

Для диаграмм, которые представлены на рис. 5.2 и рис. 5.8, таким названием является «Выполнение лабораторной работы».

Название контекстной диаграммы определяется общей функцией моделируемой системы, то есть совпадает с названием блока контекстной диаграммы (см. рис. 5.8).

Таким образом, две диаграммы IDEF0-модели имеют одно и то же название. Это контекстная диаграмма и ее диаграмма-потомок. Названия всех остальных диаграмм модели уникальны.

Номер узла

Каждая диаграмма модели идентифицируется ***номером узла*** (NODE), расположенным на IDEF0-бланке в левом нижнем углу.

Номер узла для контекстной диаграммы имеет следующий вид: заглавная буква **A** (Activity в функциональных диаграммах), дефис и ноль (A-0, см,рис. 5.8). Номер узла диаграммы, декомпозирующей контекстную диаграмму, – тот же номер узла, но без дефиса (A0, см. рис. 5.2).

Все другие номера узлов образуются посредством добавления к номеру узла родительской диаграммы номера декомпозируемого блока. Например, номер узла родительской диаграммы – A0. Тогда номер узла диаграммы, декомпозирующей первый блок родительской диаграммы, – A01.

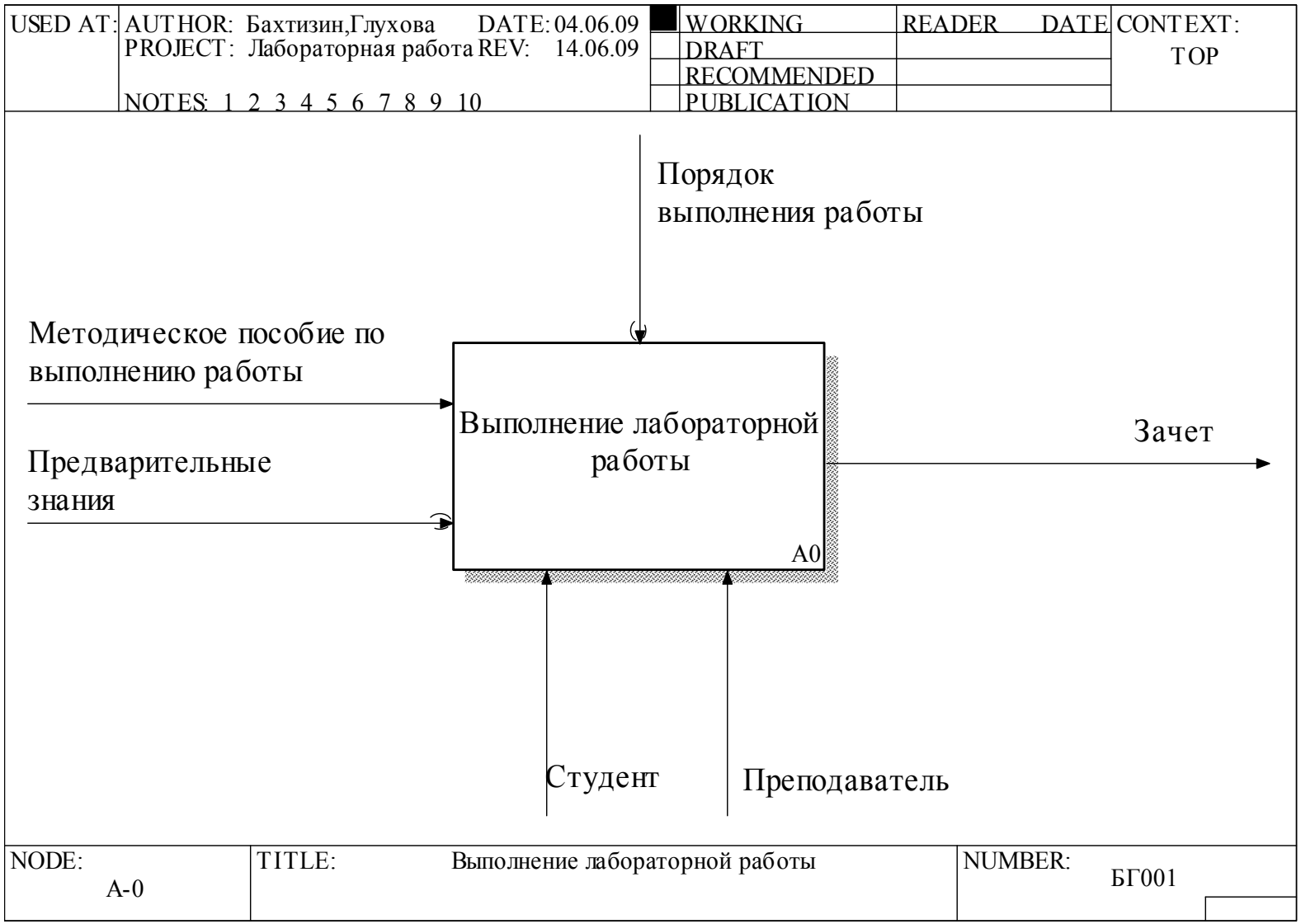


Рис. 5.8. Контекстная диаграмма процесса выполнения лабораторной работы

Первый ноль при образовании номера узла принято опускать. Таким образом, номер узла запишется в виде A1. При декомпозиции третьего блока родительской диаграммы A1 номер узла диаграммы-потомка будет соответствовать значению A13.

Для связи диаграмм при движении вверх по иерархии модели могут применяться С-номера или номера узлов. После декомпозиции родительского блока на диаграмме-потомке формируется ссылка на родительскую диаграмму. Для этого используется поле «КОНТЕКСТ» (CONTEXT), расположенное в правом верхнем углу IDEF0-бланка. В данном поле маленькими прямоугольниками изображается каждый блок родительской диаграммы (с сохранением их относительного положения), заштриховывается квадратик декомпозированного блока и размещается С-номер или номер узла родительской диаграммы.

Например, на диаграмме-потомке, декомпозирующей третий блок A3 родительской диаграммы A0, заполнение поля «КОНТЕКСТ» будет соответствовать представленному на рис. 5.9.

Связь между диаграммами посредством С-номеров или номеров узлов позволяет осуществлять тщательный контроль за введением новых диаграмм в иерархию модели.

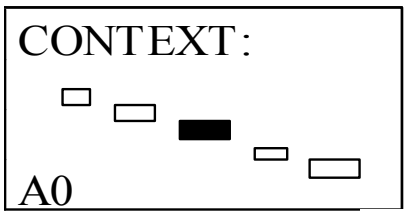


Рис. 5.9. Заполнение поля «КОНТЕКСТ» диаграммы-потомка

Организация связей по дугам между диаграммами

IDEF0-диаграммы имеют ***внешние дуги*** – это дуги, выходящие к краю диаграммы. Эти дуги являются интерфейсом между диаграммой и остальной частью модели. Диаграмма должна быть состыкована со своей родительской диаграммой, то есть внешние дуги должны быть согласованы по числу и наименованию с дугами, касающимися декомпозированного блока родительской диаграммы. Последние называются ***граничными дугами***.

В IDEF0 принята система обозначений, позволяющая авторам модели точно идентифицировать и проверять связи по дугам между диаграммами. Эта схема кодирования дуг называется ***ICOM*** (Input-Conrol-Output-Mechanism).

Правила стыковки и обозначения внешних дуг диаграммы-потомка с граничными дугами родительского блока могут быть сформулированы следующим образом [29]:

• зрительно соединяется каждая внешняя дуга диаграммы-потомка с соответствующей граничной дугой родительского блока;

• каждой зрительной связи присваивается код (**I** – для входных дуг, **C** – для связей между дугами управления, **O** – для связей между выходными дугами, **M** – для связей между дугами механизма, см. рис. 5.2);

• после каждой буквы добавляется цифра, соответствующая положению данной дуги среди других дуг того же типа, касающихся родительского блока. Входные и выходные дуги пересчитываются сверху вниз, а дуги управлений и механизмов – слева направо (в том порядке, как они расположены на родительской диаграмме по отношению к родительскому блоку). Например, внешние дуги M1, M2 (см. рис. 5.2) пронумерованы в соответствии с расположением граничных дуг на родительском блоке (см. рис. 5.8).

Тоннельные дуги

Особые ситуации возникают, когда дуги «входят в тоннель» между диаграммами. ***Дуга «входит в тоннель»*** в следующих случаях:

1) она является внешней дугой, которая отсутствует на родительской диаграмме (дуга имеет *скрытый источник*);

2) она касается родительского блока, но не появляется на диаграмме, которая его декомпозирует (дуга имеет *скрытый приемник*).

Тоннельные дуги от скрытого источника начинаются круглыми скобками, чтобы указать, что эти дуги идут из какой-то другой части модели, прямо извне модели или они не важны для родительской диаграммы и поэтому на ней не изображаются. Например, дуга механизма «Лаборант» является тоннельной дугой со скрытым источником (см. рис. 5.2). Данная дуга отсутствует на родительской диаграмме (см. рис. 5.8), поскольку для родительской диаграммы она является маловажной.

Тоннельные дуги со скрытым приемником заканчиваются круглыми скобками, чтобы отразить тот факт, что такая дуга идет к какой-то другой части модели, выходит из нее или не будет более в этой модели рассматриваться. Тоннельные дуги со скрытым приемником часто используются в том случае, если они должны связываться с каждым блоком диаграммы-потомка. Изображение таких дуг может привести к существенному загромождению данной диаграммы и ее потомков. Например, дуги «Предварительные знания» и «Порядок выполнения работы» (см. рис. 5.8) должны связываться с каждым блоком диаграммы декомпозиции. Их изображение на диаграмме-потомке является малоинформативным. Поэтому данные дуги реализованы в виде тоннельных дуг со скрытым приемником и на диаграмме декомпозиции (см. рис. 5.2) не показаны.

Таким образом, «вхождение в тоннель» для дуг используется чаще всего для упрощения описания системы – тогда, когда диаграммы в модели становятся слишком сложными для чтения и понимания.

Диаграмма дерева узлов

Разработанная IDEF0-модель со всеми уровнями структурной декомпозиции может быть представлена в виде единственной диаграммы дерева узлов (рис. 5.10). На данном виде диаграмм представляется иерархия функций в модели без указания взаимосвязей (дуг) между функциями. Для изображения этого дерева нет стандартного формата. Единственное требование состоит в том, чтобы вся иерархия узлов модели была представлена наглядно [16, 28].

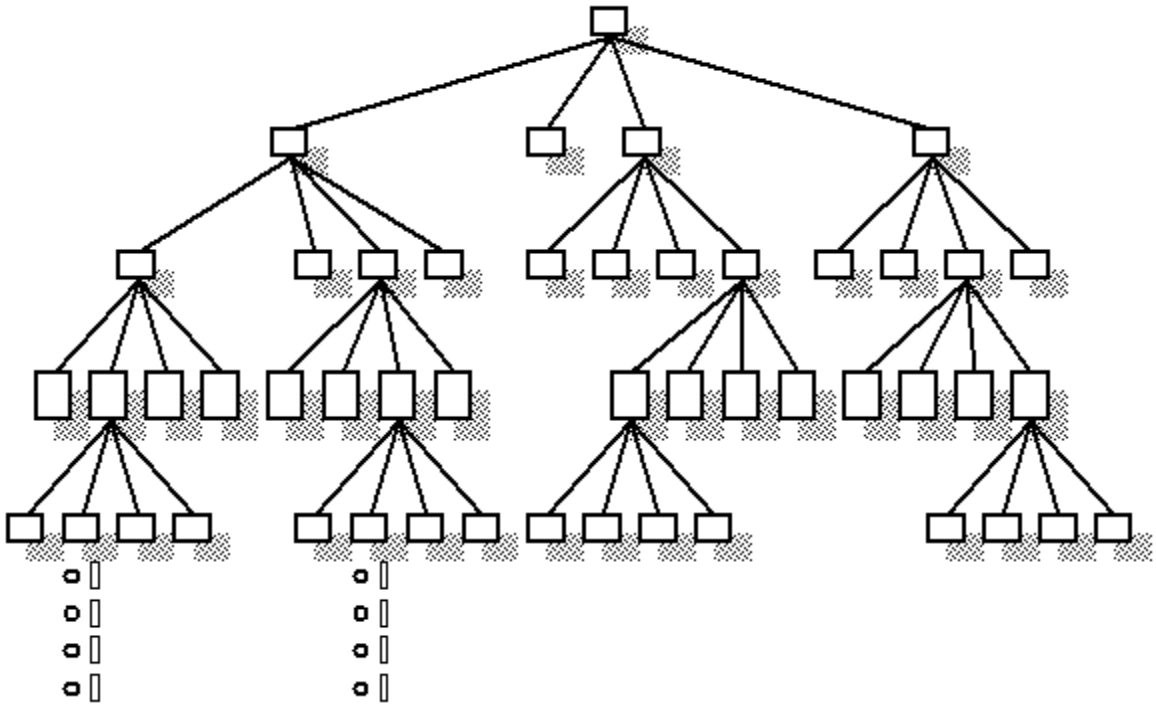


Рис. 5.10. Пример диаграммы дерева узлов

Резюме

IDEF0-диаграммы являются декомпозициями ограниченных субъектов. Субъект ограничивается блоком и касающимися его дугами. Диаграмма, содержащая границу, называется родительской диаграммой. Диаграмма, декомпозирующая блок родительской диаграммы, называется диаграммой-потомком. Для связывания родительской диаграммы и диаграммы-потомка используются С-номера, номера узлов и коды ICOM. Номер узла идентифицирует уровень диаграммы в иерархии модели. Для упрощения описания системы используется специальный технический прием «вхождение дуг в тоннель». IDEF0-модель может быть представлена в виде единственной диаграммы дерева узлов.

5.2.5. Декомпозиция и её стратегии при IDEF0-моделировании

При создании IDEF0-модели предметной области используется *метод декомпозиции ограниченного субъекта* [29].

Как уже отмечалось, декомпозиция – это процесс создания диаграммы, детализирующей определенный блок и связанные с ним дуги. IDEF0-декомпозиция включает в себя *анализ* (начальное разделение элемента на более мелкие части) и *синтез* (последующее их соединение для смыслового описания элемента).

Следуя методу декомпозиции ограниченного субъекта, автор производит *вначале* анализ и синтез системных объектов (напомним, что под объектом в общем случае понимаются некоторые данные, изображаемые на модели дугами). Список данных начинается со всех граничных дуг и их ICOM-кодов с последующей их детализацией на составляющие. После этого некоторые составляющие могут быть объединены для смыслового выделения объектов (например объектов, которые будут выступать в качестве управляющих).

Затем автор выполняет подобный анализ и синтез функций системы, делая это в соответствии со списком данных. В процессе объединения и введения новых управляющих дуг создается список функций для дальнейшей детализации. Эти функциональные части объединяются в наборы из трех–шести блоков.

После определения функциональных частей список данных и список функций используются для чернового варианта диаграммы. Снова выполняются анализ и синтез, в результате чего формируются наборы объектов, которые представляются дугами, соединяющими блоки.

Такая последовательность выполнения декомпозиции имеет большое значение, поскольку в методологии IDEF0 анализ объектов системы оказывает важнейшее влияние на анализ функций.

В процессе создания диаграмм опытный разработчик модели постоянно следит за стратегией декомпозиции и ее влиянием на качество модели. При создании IDEF0-модели наиболее часто используются следующие ***стратегии декомпозиции*** [29]*:*

1. *Функциональная стратегия*. Базируется на функциональных взаимоотношениях в системе. Рекомендуется следовать этой стратегии всегда, когда это возможно.

2. *Декомпозиция в соответствии с функциями, выполняемыми людьми или организациями*. Рекомендуется использовать эту стратегию только в начале работы (так как позже взаимосвязи между исполнителями могут быть очень сложны) над моделью системы, относящейся к разряду P3 (people – люди, paper – бумага, procedures – процедуры). Это помогает собрать исходную информацию о системе. Затем следует применять более обоснованную функциональную декомпозицию в соответствии с первой стратегией.

3. *Декомпозиция в соответствии с уже известными стабильными подсистемами*. Это приводит к созданию набора моделей – по одной модели на каждую подсистему или важный компонент. Затем для описания всей системы строится составная модель, объединяющая все отдельные модели. Стратегия эффективна для систем команд и управления, когда разделение на основные части системы не меняется. Например, создается модель отдельно для торпеды, отдельно для защиты от торпед и отдельно для движения подводной лодки. Затем эти модели объединяются вместе для описания способов защиты подводной лодки от торпед.

4. *Декомпозиция, основанная на отслеживании «жизненного цикла» для ключевых входов системы*. Эффективна для моделирования систем, непрерывно преобразующих свои входы в конечный продукт (например система очистки нефти). Декомпозиция осуществляется в соответствии с этапами преобразования входа (этапами ЖЦ).

5. *Декомпозиция по физическому процессу*, основанная на выделении функциональных стадий, этапов завершения, шагов выполнения и т.п. Результатом стратегии часто является последовательное описание системы, не учитывающее ограничения, накладываемые функциями друг на друга. Поэтому эту стратегию рекомендуется использовать в случаях, если целью модели является описание физического процесса или если разработчик в начале создания IDEF0-модели не понимает, как действовать. В последнем случае по мере накопления информации о моделируемой предметной области следует перейти на применение более обоснованной функциональной декомпозиции в соответствии с первой стратегией.

Резюме

Для построения модели используется метод декомпозиции ограниченного субъекта. Декомпозиция включает этапы анализа и синтеза. Существуют различные стратегии декомпозиции. Их выбор и получение диаграмм высокого качества часто требуют многих итераций и изменений. Декомпозиция прекращается, когда модель достаточно точна, чтобы отвечать на вопросы, составляющие ее цель.

5.2.6. Процесс моделирования в IDEF0

Процесс моделирования в IDEF0 включает сбор информации об исследуемой области, документирование полученной информации с представлением ее в виде модели и уточнение модели посредством итеративного рецензирования. На рис. 5.11 изображен процесс моделирования в IDEF0, описанный с помощью IDEF0-диаграммы [29].

Процесс моделирования в IDEF0 является итерационным. Это позволяет получить точное описание системы или программного средства. Основой процесса IDEF0-моделирования является разделение функций, выполняемых участниками IDEF0-проектов (см. входы механизмов, рис. 5.11).

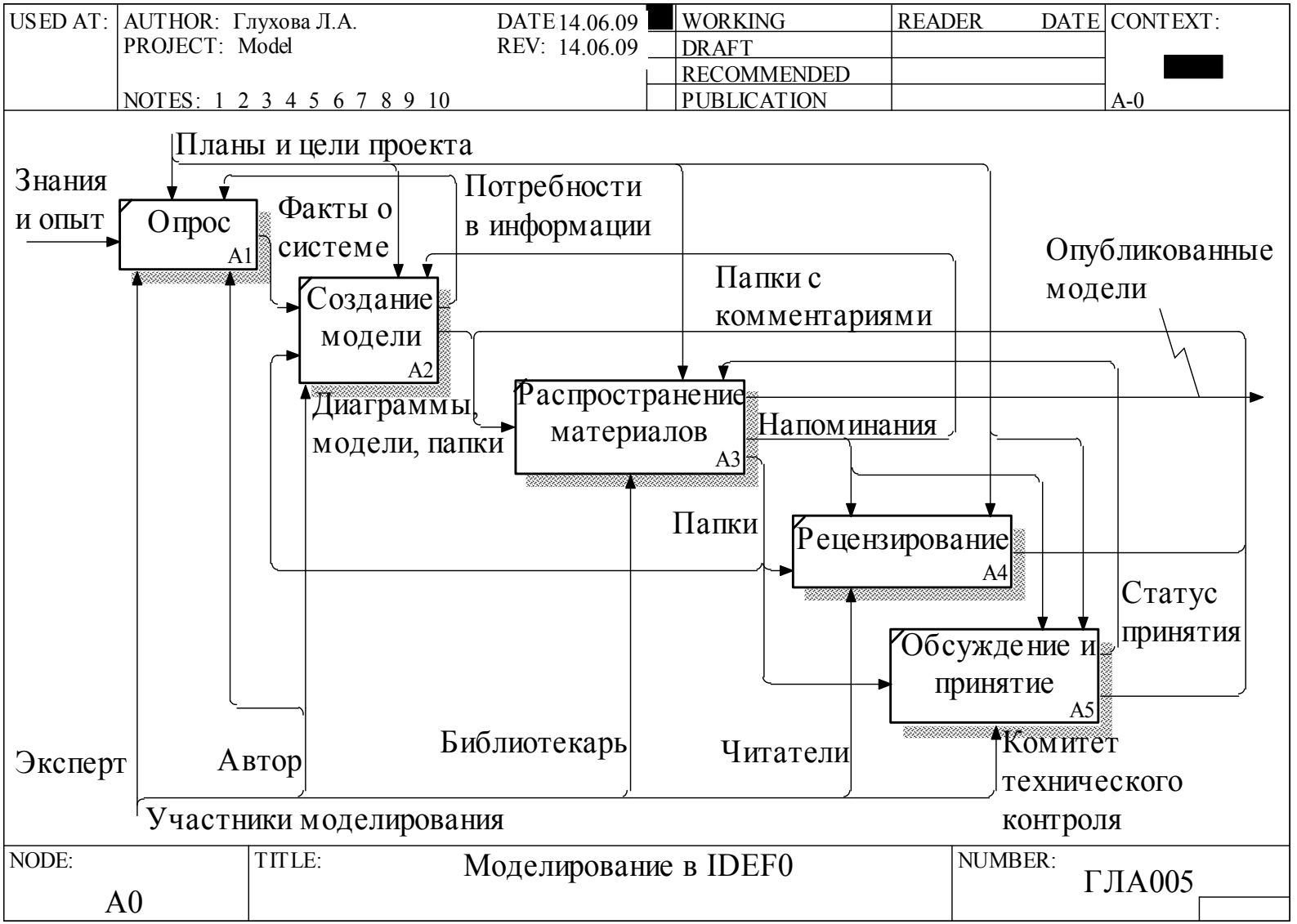


Рис. 5.11. Процесс моделирования в IDEF0

Разделение ролей участников проектов обеспечивает поддержку организации коллективной работы в IDEF0. Выделяются следующие роли участников проектов:

- *эксперты* – специалисты в предметной области, являющиеся источниками информации;
- *авторы* – разработчики диаграмм и моделей;
- *библиотекарь* – координатор своевременного обмена письменной информацией между участниками проекта;
- *читатели* – специалисты в предметной области, которые рецензируют модели;

комитет технического контроля – группа специалистов, принимающих и утверждающих модель.

Как видно из рис. 5.11, процесс IDEF0-моделирования состоит из *пяти этапов*.

Целью *первого этапа* IDEF0-моделирования (блок А1 «Опрос» на рис. 5.11) является получение автором знаний о моделируемой системе (предметной области). Для этого могут быть использованы различные источники информации: собственные знания и опыт автора, изучение документов, опрос экспертов, наблюдение за работой системы и т. п. Результатом данного этапа являются собранные факты о моделируемой системе.

Вторым этапом моделирования является создание модели (блок А2 на рис. 5.11). На данном этапе полученные на предыдущем этапе факты о системе автор представляет в виде одной или нескольких IDEF0-диаграмм. Процесс создания модели осуществляется с помощью *метода декомпозиции ограниченного субъекта.* При его использовании автор модели *вначале* анализирует *объекты* (информацию, данные, механизмы и т.п.), входящие в систему, а *затем* использует полученные знания для анализа *функций* системы (см. п. 5.2.5). На основе этого анализа создается диаграмма, в которой объединяются сходные объекты и функции. Этот путь проведения анализа системы и документирования его результатов является уникальной особенностью методологии IDEF0.

Из рис. 5.11 видно, что этапы «Опрос» и «Создание модели» выполняются многократно (см. обратную связь по управлению «Потребности в информации»). Таким образом, создающиеся IDEF0-модели проходят через серию последовательных улучшений до тех пор, пока они в точности не будут представлять реальную предметную область. Созданные диаграммы и модели оформляются в виде *папок* – небольших пакетов с результатами работы.

Третий этап – «Распространение материалов» (блок А3) предназначен для координации передачи материалов проекта его участникам. Папки, сформированные на этапе «Создание модели» и содержащие некоторую часть работы над моделью, распространяются с целью получения отзыва от читателей.

Для эффективного моделирования важнейшее значение имеет организация своевременной обратной связи между участниками IDEF0-проекта, так как устаревшая информация способна свести на нет все

усилия по разработке модели. Поэтому IDEF0-методология выделяет специальную роль *наблюдателя за процессом рецензирования*. Эту роль выполняет так называемый *библиотекарь*, который является главным координатором процесса моделирования в IDEF0. Он обеспечивает своевременное и согласованное распространение рабочих материалов, контролирует их движение.

На *четвертом этапе* – «Рецензирование» (блок A4) выполняется критическое обсуждение специалистами в предметной области текущих результатов моделирования, представленных в папках. Результаты обсуждаются в течение определенного времени. Сделанные замечания помещаются в папку в виде нумерованных комментариев. К определенному сроку замечания поступают к библиотекарю, а от него – к автору. Автор отвечает на каждое замечание и обобщает критику, содержащуюся в замечаниях.

Этапы «Создание модели» (блок A2), «Распространение материалов» (блок A3) и «Рецензирование» (блок A4, см. рис. 5.11) составляют так называемый *цикл автор/читатель*. Данный цикл является одним из основных компонентов методологии IDEF0. В его ходе читателями выполняется многократное рецензирование достоверности текущих результатов моделирования, на основе которого авторы уточняют создаваемые модели.

Целями пятого этапа «Обсуждение и принятие» (блок A5) являются: оценка того, что создаваемая в процессе анализа модель будет точна и используема в дальнейшем; контроль качества модели; оценка соответствия выполняемой работы конечным целям всего проекта. За выполнение данного этапа отвечает *Комитет технического контроля*. Если модель признана Комитетом применимой, она публикуется. В противном случае авторам направляются замечания для необходимой доработки.

Таким образом, методология IDEF0 поддерживает как асинхронный, так и параллельный просмотры модели. Такая организация процесса является наиболее эффективным способом распределения работы в коллективе.

На практике над различными частями модели работает совместно несколько авторов, так как каждый функциональный блок модели представляет отдельный компонент, который может быть независимо проанализирован и декомпозирован.

В настоящее время существует ряд CASE-средств, поддерживающих методологию IDEF0. Среди недорогих и доступных на нашем рынке инструментальных средств следует отметить CASE-средство AllFusion Process Modeler, известное ранее под названием BPwin. Данное CASE-средство входит в линейку интегрированных CASE-средств CA ERwin Modeling Suite (AllFusion Modeling Suite) компании Computer Associates (см. подразд. 7.6). Процесс функционального моделирования в среде BPwin подробно описывается в [16, 28].

Резюме

Процесс IDEF0-моделирования может быть разделен на несколько этапов: опрос экспертов, создание диаграмм и моделей, распространение документации, рецензирование, принятие диаграмм и моделей. Каждый из исполните-

лей проекта выполняет конкретные обязанности. Среди современных CASE-средств, поддерживающих технологию IDEF0-моделирования, наиболее популярно CASE-средство AllFusion Process Modeler.

5.3. Методология структурного анализа потоков данных DFD

Методология структурного анализа потоков данных ***DFD*** (Data Flow Diagrams) основана на методах, ориентированных на потоки данных (методах Йодана, Де Марко, Гейна, Сарсона). Существуют различные графические нотации данной методологии. Наиболее известными из них являются нотация, предложенная Гейном и Сарсоном (так называемый метод Гейна–Сарсона) [34], и нотация, предложенная Йоданом и ДеМарко (метод Йодана–ДеМарко) [24]. Пример DFD-модели в нотации Йодана–ДеМарко представлен на рис. 4.22. В данном подразделе рассматривается методология DFD в нотации Гейна–Сарсона.

5.3.1. Основные понятия DFD-модели

Методология DFD является одной из методологий функционального моделирования предметной области, поэтому она имеет много общего с методологией IDEF0.

DFD-методология выделяет функции (действия, события, работы) системы. Функции соединяются между собой с помощью потоков данных (объектов). Функции на диаграммах представляются функциональными блоками, потоки данных – дугами.

Аналогично IDEF0-методологии DFD-модель должна иметь единственные цель, точку зрения, субъект и точно определенные границы (см. п. 5.2.2).

Однако если в IDEF0 дуги имеют различные типы и определяют отношения между блоками (см. п. 5.2.3), то в DFD дуги отражают реальное перемещение объектов от одной функции к другой.

Помимо блоков, представляющих собой функции, на DFD-диаграммах используются два типа блоков – хранилища данных и внешние сущности. Данные блоки отражают взаимодействие с частями предметной области, выходящими за границы моделирования.

Резюме

Методология DFD является одной из методологий функционального моделирования предметной области. DFD-модель должна иметь единственные цель, точку зрения, субъект и точно определенные границы. DFD-модель отражает перемещение объектов, их хранение, обработку, внешние источники и потребителей данных.

5.3.2. Синтаксис DFD-диаграмм

Диаграммы являются основными рабочими элементами DFD-модели. Диаграммы отражают перемещение данных, их обработку и хранение. Каждая DFD-диаграмма содержит функциональные блоки и дуги (линии со стрелками). DFD-диаграмма может содержать хранилища данных и внешние сущности.

Функциональный блок отражает некоторую функцию моделируемой системы, преобразующую некоторые входные данные (сырье, материалы, информацию и т. п.) в выходные результаты. Функциональный блок изображается прямоугольником с закругленными углами (рис. 5.12). Все стороны функционального блока в отличие от IDEF0 равнозначны.

Функциональные блоки на диаграмме нумеруются. Номер функционального блока отмечается в его правом верхнем углу с возможным использованием префикса **A** (Activity – работа) перед ним.

Как и в IDEF0-методологии, название функционального блока основывается на использовании отглагольного существительного, обозначающего действие (вычисление того-то, определение того-то, обработка того-то и т.д.) или на использовании глагола в неопределенной форме (вычислить то-то, определить то-то, обработать то-то).

Рис. 5.12. Функциональный блок DFD

Хранилище данных отражает временное хранение промежуточных результатов обработки. Внешний вид блока, представляющего хранилище данных, иллюстрирует рис. 5.13. Название хранилища базируется на использовании существительного.

Хранилища данных на диаграмме нумеруются. Номер хранилища данных записывается слева с возможным префиксом **D** (Data store) перед ним.

Рис. 5.13. Хранилище данных

Внешние сущности являются источниками данных для входов модели и приемниками данных для ее выходов. Внешняя сущность может быть одновременно источником и приемником данных.

Внешние сущности изображаются в соответствии с рис. 5.14 и размещаются, как правило, по краям диаграмм. Название внешней сущности базируется на использовании существительного. Номер внешней сущности записывается в левом верхнем углу с возможным префиксом **E** (External) перед ним.

Одна и та же внешняя сущность (с одним и тем же номером) может быть размещена в нескольких местах диаграммы. Это позволяет в ряде случаев существенно снизить загроможденность диаграмм длинными дугами.

Дуги обозначают передвижение данных в моделируемой системе. С учетом равнозначности сторон блоков диаграммы дуги могут начинаться и заканчиваться на любой их стороне.

Рис. 5.14. Внешняя сущность

В общем случае дуга представляет множество объектов (планы, машины, информация и т.п.). Основу названия дуги на IDEF0-диаграммах составляют существительные. Названия дуг называются *метками.*

Дуги на DFD-диаграмме изображаются линиями со стрелками. На DFD-диаграммах могут использоваться следующие типы дуг:

- однонаправленные сплошные – отражают направление потоков объектов (данных);
- двунаправленные сплошные – обозначают обмен данными между блоками;
- однонаправленные штриховые – обозначают управляющие потоки между блоками.

Как и в IDEF0-методологии, дуги могут разветвляться и соединяться. Синтаксис и семантика разветвления и слияния дуг соответствуют описанным в п. 5.2.3 для методологии IDEF0.

На рис. 5.15 приведен пример DFD-диаграммы процесса выполнения лабораторной работы. Данный пример соответствует предметной области моделирования, подробно рассмотренной в пп. 5.2.2 – 5.2.4 при изучении методологии IDEF0.

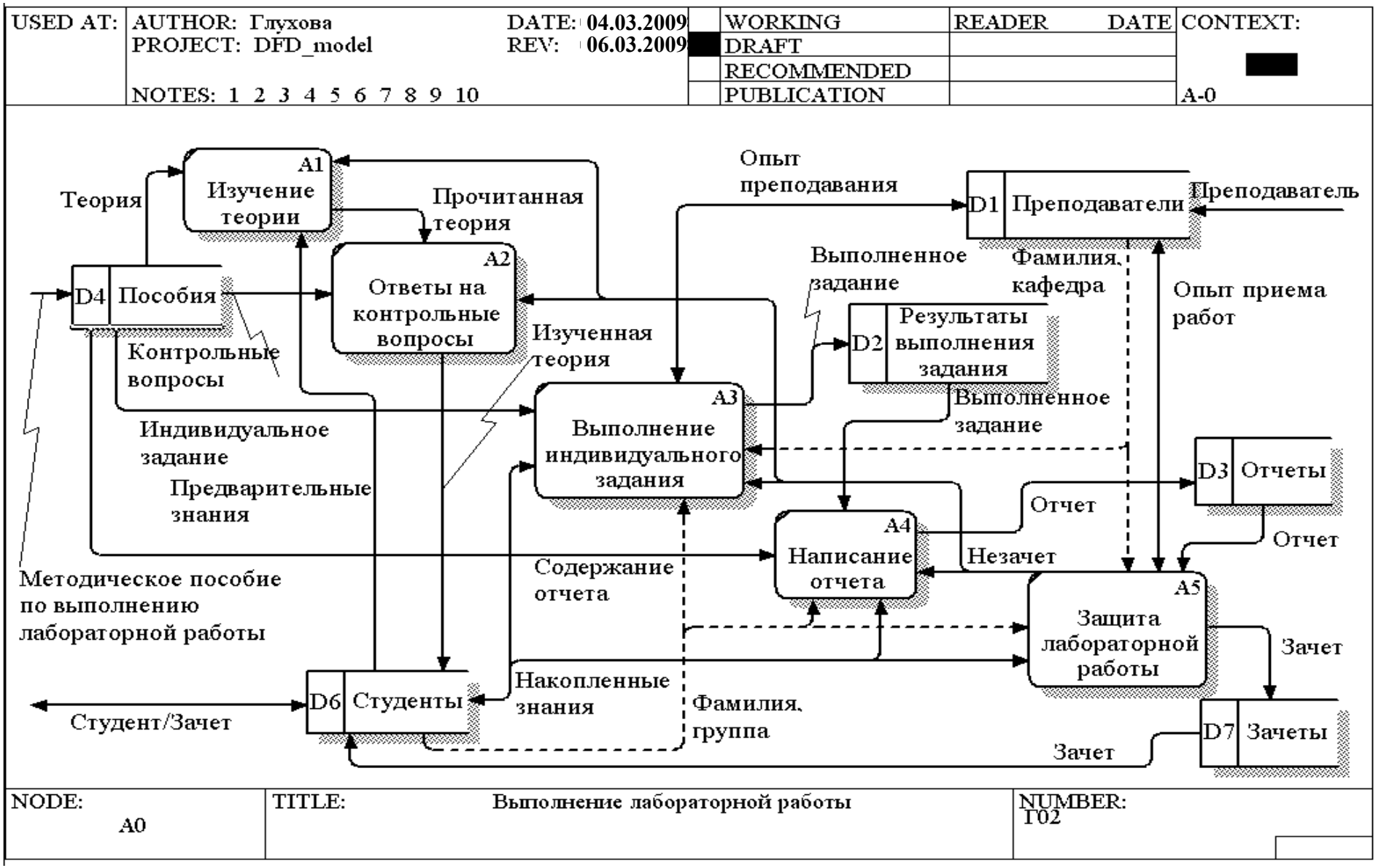


Рис. 5.15. DFD-диаграмма декомпозиции процесса выполнения лабораторной работы

Резюме

DFD-диаграммы отражают перемещение данных, их обработку и хранение. DFD-диаграмма содержит функциональные блоки и дуги, может содержать хранилища данных и внешние сущности. Функциональный блок отражает некоторую функцию моделируемой системы. Хранилище данных отражает временное хранение промежуточных результатов обработки. Внешние сущности являются источниками данных для входов модели и приемниками данных для ее выходов. Дуги могут представлять однонаправленные и двунаправленные потоки данных и управляющие потоки.

5.3.3. Синтаксис DFD-моделей

По аналогии с IDEF0-моделью DFD-модель представляет собой иерархически организованную совокупность диаграмм. Каждый из функциональных блоков диаграммы может быть декомпозирован на другой диаграмме. Функциональный блок и касающиеся его дуги определяют границу диаграммы, представляющей декомпозицию этого блока. Эта диаграмма называется ***диаграммой-потомком***. Декомпозируемый блок называется ***родительским блоком***, а содержащая его диаграмма – ***родительской диаграммой***. Название диаграммы-потомка совпадает с функцией родительского блока.

Один функциональный блок и несколько дуг на самом верхнем уровне модели используются для определения границы всей системы. Этот блок описывает общую функцию, выполняемую системой. Диаграмма, определяющая границу системы и состоящая из одного функционального блока, его дуг и внешних сущностей, называется ***контекстной диаграммой DFD-модели***. Все, что лежит внутри функционального блока, является частью описываемой системы. Внешние сущности определяют *среду системы*, то есть внешние источники и приемники объектов (данных) системы.

Рис. 5.16 представляет контекстную DFD-диаграмму процесса выполнения лабораторной работы. В отличие от контекстной IDEF0-диаграммы (см. рис. 5.8) контекстная DFD-диаграмма содержит внешние сущности «Кафедра» «Студенты», «Методические пособия» и «Правила обучения». В остальном синтаксис этих диаграмм совпадает (см. п. 5.2.4).

Общая функция DFD-модели записывается на контекстной диаграмме в виде названия функционального блока и совпадает с именем данной диаграммы. С контекстной диаграммой связывается цель модели и точка зрения.

Декомпозицией контекстной диаграммы, представленной на рис. 5.16, является диаграмма, содержащаяся на рис. 5.15.

Как и в IDEF0-методологии, дуги в DFD-моделях могут «входить в тоннель» между диаграммами (см. дугу «Лаборант» на рис. 5.16). Синтаксис и семантика тоннелирования дуг соответствуют описанному в п. 5.2.4 для методологии IDEF0.

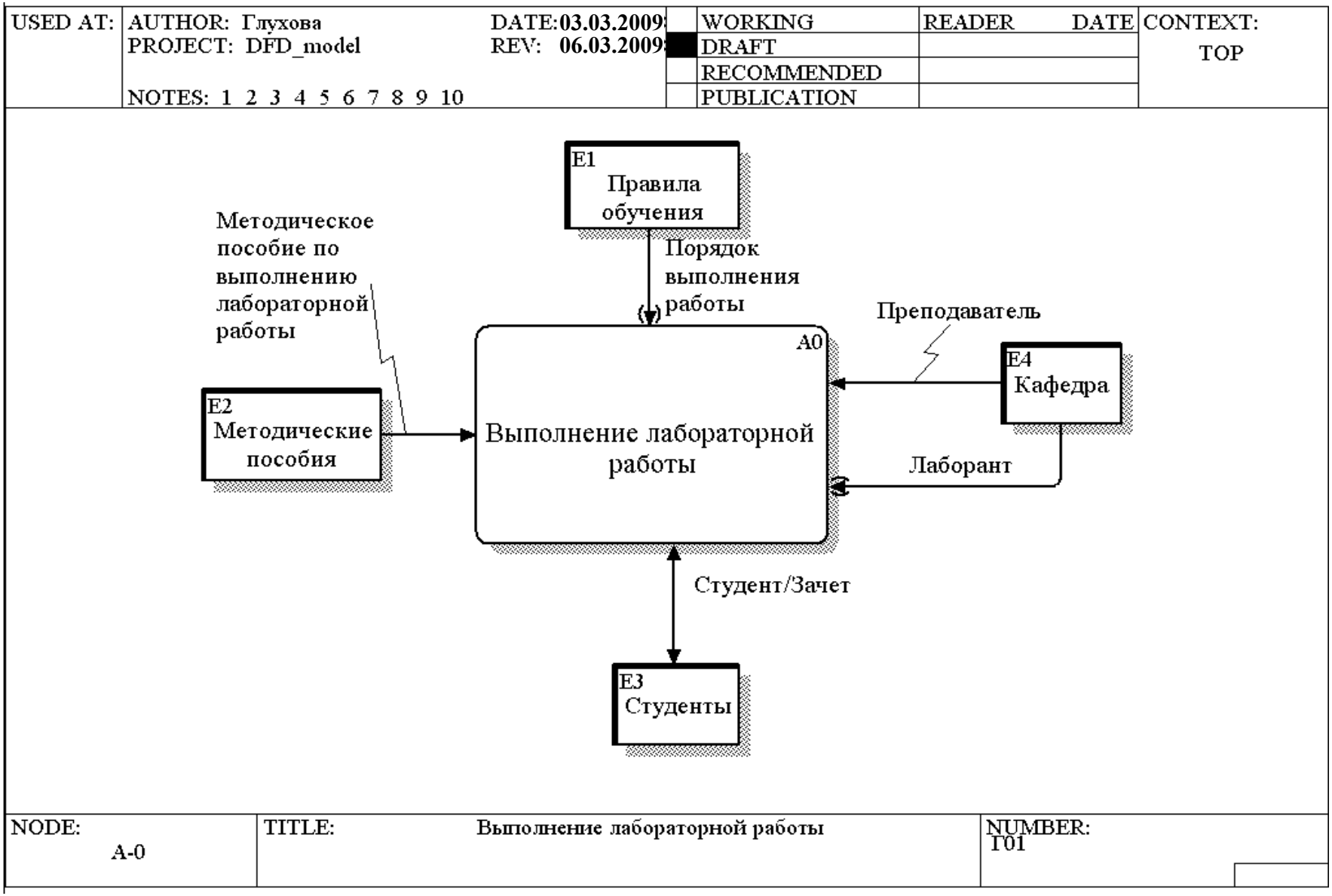


Рис. 5.16. Контекстная DFD-диаграмма процесса выполнения лабораторной работы

Разработанная DFD-модель со всеми уровнями структурной декомпозиции может быть представлена в виде диаграммы дерева узлов (рис. 5.17). На данном виде диаграмм представляется иерархия функций в модели без указания потоков данных между ними. Формат дерева узлов может быть различным.

DFD-моделирование поддерживается рядом CASE-средств, например, CASE-средством AllFusion Process Modeler (Bpwin, см. подразд. 7.6). Процесс функционального DFD-моделирования в среде BPwin описан в [28, 34].

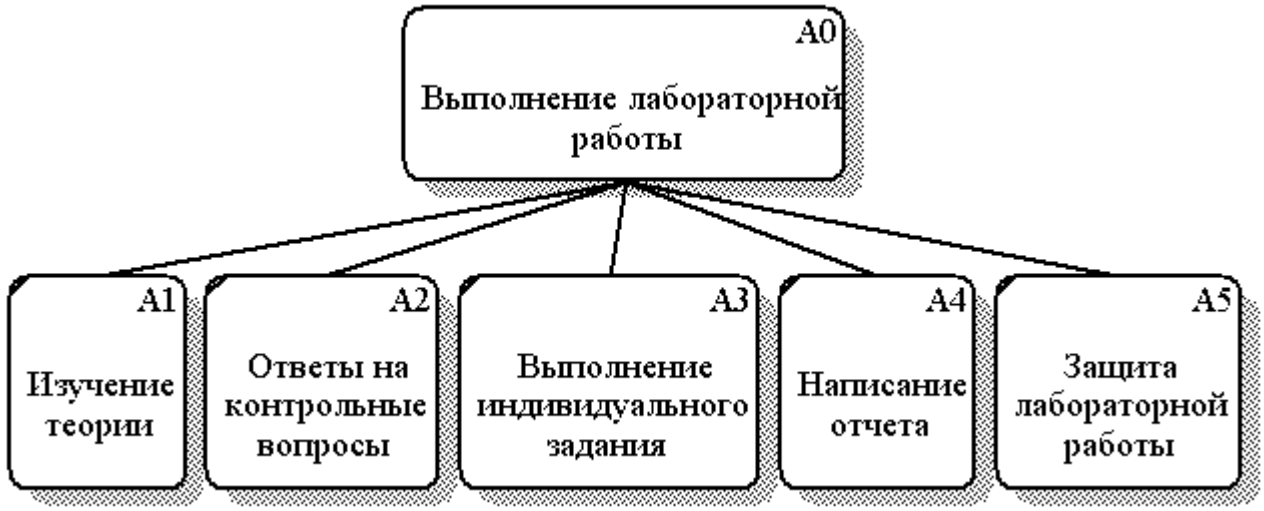


Рис. 5.17. DFD-диаграмма дерева узлов

Резюме

DFD-модель представляет собой иерархически организованную совокупность диаграмм. Диаграмма на верхнем уровне иерархии, состоящая из одного функционального блока, его дуг и внешних сущностей, называется контекстной диаграммой DFD-модели. Все, что лежит внутри функционального блока, является частью моделируемой системы. Внешние сущности определяют среду системы. Иерархию DFD-модели представляет диаграмма дерева узлов.

5.4. Методология информационного моделирования IDEF1X

5.4.1. Основные понятия и определения

Методология информационного моделирования IDEF1X известна также под названиями методологии семантического моделирования данных [25] и методологии концептуального моделирования [2].

Под ***информационной моделью (моделью данных)*** подразумевается графическое и текстовое представление результатов анализа предметной области, которое идентифицирует данные, используемые в организации для достижения своих целей, функций, задач, потребностей и стратегий, а также для управления организацией или ее оценки [2].

Целью информационного моделирования (моделирования данных) является идентификация сущностей, составляющих предметную область, и связей между ними. *Результатом информационного моделирования* является информационная модель предметной области, содержащая сущности, их атрибуты и отражающая взаимосвязи между сущностями.

Наиболее часто информационное моделирование используется при проектировании баз данных.

Общепринятым стандартом представления информационных моделей (моделей данных) в настоящее время является стандарт ***IDEF1X*** [2], разработанный на основе диаграмм «Сущность–Связь» Чена.

В соответствии с данным стандартом компонентами IDEF1X являются:

- сущности (Entities); подразделяются на два вида:
  - независимые сущности (Identifier-Independent Entities);
  - зависимые сущности (Identifier-Dependent Entities);
- связи (Relationships); подразделяются на четыре вида:
  - идентифицирующие соединительные связи (Identifying Connection Relationships);
  - неидентифицирующие соединительные связи (Non-Identifying Connection Relationships);
  - связи категоризации (Categorization Relationships);
  - неспецифические связи (Non-Specific Relationships);
- атрибуты/ключи (Attributes/Keys); подразделяются на четыре вида:
  - атрибуты (Attributes);
  - первичные ключи (Primary Keys);
  - альтернативные ключи (Alternate Keys);
  - внешние ключи (Foreign Keys);
- текстовые комментарии (Notes).

Резюме

Информационная модель выделяет данные предметной области и используется при проектировании баз данных. В соответствии со стандартом IDEF1X компонентами информационной модели являются сущности, их атрибуты, взаимосвязи между сущностями и текстовые комментарии.

5.4.2. Сущности

Под ***сущностью*** в информационном моделировании подразумевается представление множества реальных или абстрактных объектов предметной области (людей, предметов, мест, идей, событий), для которого [2, 35]:

1) все элементы множества (***экземпляры***) имеют одни и те же характеристики;

2) все экземпляры подчинены одному и тому же набору правил и линий поведения и участвуют в одних и тех же связях.

Сущности подразделяются на независимые и зависимые. Сущность называется ***независимой***, если каждый экземпляр данной сущности может быть уникально идентифицирован независимо от ее связей с другими сущностями. Сущность называется ***зависимой***, если уникальная идентификация его экземпляров зависит от связи данной сущности с другими сущностями.

Каждая сущность в информационной модели должна иметь уникальное ***имя***, основанное на использовании существительного. Существительное должно быть представлено в единственном числе. Примеры имен сущностей: Человек, Дом, Студент (но не Люди, Дома, Студенты). Если имя сущности состоит из нескольких слов, они соединяются дефисом или символом подчеркивания, например, Категория-предприятий, Категория_предприятий.

Кроме того, в большой модели для организации документации сущности должны быть пронумерованы. Номер сущности записывается за именем и отделяется от последнего символом «/», например, Студент/21.

Большинство сущностей относится к следующим ***категориям*** [35]:

- реальные объекты;
- роли;
- инциденты;
- взаимодействия;
- спецификации.

Реальные объекты – это представление фактических предметов в физическом мире. Например, к сущностям данной категории относятся сущности Завод, Университет, Аэропорт, Банк.

Роли – это представление цели или назначения человека, оборудования или организации. Например, для университета сущностями-ролями являются Преподаватель, Лаборант и Студент; для магазина – Покупатель, Продавец, Кассир.

Инциденты – это представление какого-либо события. Примерами сущностей-инцидентов могут являться сущности Землетрясение, Запуск-космического-корабля, Выборы.

Взаимодействия – сущности, получаемые из отношений между двумя сущностями. Примерами сущностей-взаимодействий являются сущности Перекресток (место пересечения улиц), Контракт (соглашение между сторонами), Соединение (место соединения некоторой детали с другой).

Спецификации – сущности, используемые для представления правил, стандартов, требований, критериев качества и т.п. Примерами сущностей-спецификаций являются сущности Рецепт (правило приготовления порции пищи), Метрика-качества (метод и шкала измерения некоторого свойства программного средства).

Каждая сущность должна сопровождаться описанием. ***Описание*** – это короткое информативное утверждение, которое позволяет установить, является ли некоторый элемент экземпляром сущности или нет.

Например, для сущности студент описание может выглядеть следующим образом: «Человек, учащийся в ВУЗе».

Резюме

Сущность в информационном моделировании представляет множество реальных или абстрактных объектов с одинаковыми характеристиками, подчиняющихся одному набору правил и линий поведения и участвующих в одних и тех же связях. Сущности подразделяются на независимые и зависимые. Имя сущности базируется на использовании существительного. Существуют следующие категории сущностей: реальные объекты, роли, инциденты, взаимодействия, спецификации.

5.4.3. Атрибуты

Все реальные или абстрактные объекты в мире имеют некоторые характеристики (например, высота, температура, возраст, координаты и т.п.).

Атрибут – это образ характеристики или свойства, которым обладают все экземпляры сущности. Каждый атрибут обеспечивается именем, уникальным в пределах сущности и основанным на использовании существительного. Существительное должно быть представлено в единственном числе [2, 35].

Примеры имен атрибутов: Адрес, Возраст, Фамилия (но не Адреса, Возрасты, Фамилии). Если имя атрибута состоит из нескольких слов, они соединяются дефисом или символом подчеркивания, например, Дата-рождения или Дата_рождения. Для обеспечения уникальности атрибута в пределах модели используется составное имя:

<Имя-сущности>**.**<Имя-атрибута>

Например, для сущности Студент обращение к его атрибуту Фамилия имеет вид

Студент**.**Фамилия

Для определенного экземпляра сущности атрибут принимает конкретное значение. Диапазон допустимых значений, которые атрибут может принимать, называется ***доменом***. Домен должен определяться для каждого атрибута.

При информационном моделировании атрибуты принято подразделять на *указывающие описательные* и *вспомогательные* [35].

Указывающие атрибуты используются для присвоения имени или обозначения экземплярам сущности. Например, Счет.Номер; Студент.Фамилия.

Если значение указывающего атрибута изменяется, то это говорит о том, что изменился экземпляр сущности.

Указывающие атрибуты часто используются как идентификатор или часть идентификатора.

Идентификатор – это атрибут или совокупность нескольких атрибутов, значения которых однозначно определяют каждый экземпляр сущности. Идентификатор, состоящий из нескольких атрибутов, называется ***составным***. Идентификаторы называются также ***первичными ключами (primary keys)***.

Например, для сущности Студент атрибут Фамилия является удовлетворительным идентификатором, если в университете нет однофамильцев. В общем случае идентификатор сущности Студент будет состоять из трех атрибутов (Фамилия, Имя, Отчество), а возможно, и более (например, при наличии полных однофамильцев могут быть добавлены атрибуты Домашний-адрес, Номер-группы или Дата-рождения). Следует отметить, что обработка составных идентификаторов является достаточно сложной и без необходимости их применения нужно избегать.

Сущность может иметь несколько альтернативных идентификаторов. Например, для сущности Аэропорт атрибут Код-аэропорта является идентификатором. Комбинация атрибутов Долгота и Широта является другим идентификатором сущности Аэропорт.

Если сущность имеет несколько альтернативных идентификаторов, один из них выбирается как ***привилегированный*** (первичный ключ). Остальные называются ***альтернативными ключами***.

Для упрощения структуры информационной модели и облегчения работы с ней рекомендуется в качестве идентификатора использовать ***идентификационный номер экземпляра сущности*** (***ID***). Это позволяет исключить необходимость обработки идентификаторов, состоящих из нескольких атрибутов. Наиболее эффективно использование идентификационных номеров целочисленного типа. Значения ID изменяются по порядку, начиная с единицы.

Описательные атрибуты представляют характеристики, внутренне присущие каждому экземпляру сущности.

Примерами описательных атрибутов являются Студент.Домашний-адрес; Собака.Вес; Книга.Название-главы.

Если значение описательного атрибута изменяется, то это говорит о том, что некоторая характеристика экземпляра изменилась, но сам экземпляр остался прежним.

В некоторых случаях описательные атрибуты могут включаться в состав составного идентификатора. Так, в рассмотренном выше примере для сущности Студент в состав идентификатора при наличии полных однофамильцев помимо указывающих атрибутов Фамилия, Имя, Отчество следует ввести некоторый описательный атрибут, например, Домашний-адрес.

Однако в общем случае описательные атрибуты идентификаторами не являются. Такие атрибуты называются ***вторичными ключами*** или ***неключевыми атрибутами***. Например, для сущности Аэропорт вторичным ключом является атрибут Тип-аэропорта, поскольку может существовать достаточно

большое количество аэропортов одного типа (военных, гражданских и т.п.).

Вспомогательные атрибуты используются для связи экземпляра одной сущности с экземпляром другой. Вспомогательные атрибуты называются также ***внешними ключами (foreign keys)*** или ***мигрирующими ключами***.

Если значение вспомогательного атрибута изменяется, то это говорит о том, что другие экземпляры сущностей связываются между собой.

Например, атрибут Собака.Имя-хозяина обозначает человека, которому принадлежит собака; атрибут Счет.Идентификатор-клиента указывает идентификатор клиента, владеющего данным счетом.

Резюме

Атрибут в информационном моделировании представляет образ характеристики или свойства, которым обладают все экземпляры сущности. Имя атрибута базируется на использовании существительного. Существуют описательные, указывающие и вспомогательные атрибуты. Идентификатор – это атрибут или совокупность атрибутов, однозначно определяющих каждый экземпляр сущности. Для каждого атрибута должен быть определен диапазон допустимых значений, называемый доменом.

5.4.4. Способы представления сущностей с атрибутами

При информационном моделировании сущности с атрибутами могут быть представлены различными способами.

1. Графический способ

Графический способ используется в IDEF1X-моделировании [2]. При этом независимая сущность (см. п. 5.4.9) изображается прямоугольником, а зависимая сущность – прямоугольником с закругленными углами (рис. 5.18). Имя сущности и ее номер записываются над прямоугольником.

Внутри прямоугольника записываются имена атрибутов. Атрибуты, составляющие привилегированный идентификатор сущности, записываются первыми среди атрибутов и отделяются от остальных чертой (атрибуты Фамилия, Имя, Отчество и атрибут ID-студента на рис. 5.19).

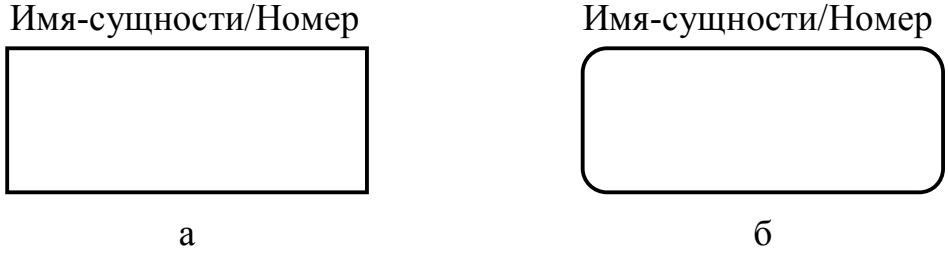


Рис. 5.18. Представление сущности: а – независимой; б – зависимой

а б

Рис. 5.19. Графическое представление сущности Студент:
а – с использованием составного идентификатора;
б – с использованием идентификационного номера

2. Текстовый способ

При текстовом способе представления сущность описывается с помощью указания ее имени, ее номера в модели (если он определен) и заключенного в круглые скобки списка атрибутов. На первом месте в списке атрибутов записываются привилегированные идентификаторы, которые некоторым образом выделяются (например подчеркиваются).

Например, сущность, представленная на рис. 5.19, а, при текстовом способе представления будет описана следующим образом:

Студент/1 (Фамилия, Имя, Отчество, Домашний-адрес, Специальность, Группа, Дата-рождения).

Текстовый способ представления сущностей удобно использовать при описании информационной модели, например, в документации.

3. Табличный способ

При табличном способе представления сущность в информационной модели интерпретируется как таблица. Каждый экземпляр сущности представляет собой строку в таблице. Строка заполняется значениями атрибутов, соответствующими данному экземпляру.

Например, сущность Студент (см. рис. 5.19, б) при табличном способе представления интерпретируется так, как показано на рис. 5.20. На данном рисунке название таблицы представляет собой имя сущности и ее номер, первая строка таблицы содержит имена атрибутов сущности, остальные строки – значения атрибутов для конкретных экземпляров сущности.

Следует отметить, что при физическом представлении базы данных каждая сущность представляется в виде реальной таблицы.

Студент/1

ID-студента	Фамилия	Имя	Отчество	Домашний адрес	Специальность	Группа	Дата-рождения
1	Иванов	Иван	Иванович	Бровки, 1-9	ПОИТ	951005	12.01.83
2	Сидоров	Петр	Власович	Скорины, 8-16	ПОИТ	651003	17.08.87
…	…	…	…	…	…	…	…

Рис. 5.20. Интерпретация сущности в виде таблицы

Резюме

Существует три основных способа представления сущностей с атрибутами: графический, текстовый, табличный. В методологии IDEF1X используется графический способ. При физическом представлении базы данных каждая сущность представляется в виде таблицы. Каждый экземпляр сущности представляет собой строку в таблице.

5.4.5. Правила атрибутов

Информационное моделирование основано на *реляционной модели данных* – представлении данных в виде отношений между ними. Поэтому сущности и их атрибуты в информационной модели должны удовлетворять требованиям к реляционной модели данных. Одним из основных требований является нормализация данных.

Нормализация – процесс уточнения и перегруппировки атрибутов в сущностях в соответствии с нормальными формами. Нормализация позволяет устранить аномалии в организации данных и сократить объем памяти для их хранения. Известны шесть нормальных форм. На практике чаще всего ограничиваются приведением модели данных к третьей нормальной форме [2].

Первая нормальная форма (First Normal Form, 1NF) – сущность находится в 1NF тогда и только тогда, когда все ее атрибуты содержат только элементарные значения.

Вторая нормальная форма (Second Normal Form, 2NF) – сущность находится в 2NF тогда и только тогда, когда она находится в 1NF и каждый ее неключевой атрибут зависит от всего первичного ключа, а не от его части.

Третья нормальная форма (Third Normal Form, 3NF) – сущность находится в 3NF тогда и только тогда, когда она находится в 2NF и каждый ее неключевой атрибут не зависит от другого неключевого атрибута.

С учетом приведенных нормальных форм в информационной модели должны соблюдаться следующие *правила атрибутов* [35].

Первое правило. Один экземпляр сущности имеет одно единственное значение для каждого атрибута в любой момент времени. Данное правило вытекает из 1NF.

В табличной интерпретации сущности это означает, что должен существовать один и только один элемент данных в каждом пересечении столбца со строкой. Например, если у экземпляра сущности Служащий имеется два телефона, то нельзя одновременно присвоить их номера атрибуту Номер-телефона.

Второе правило. Атрибут не должен содержать никакой внутренней структуры. Данное правило также вытекает из 1NF.

Например, если определен атрибут Дата-рождения, то он считается одной характеристикой и его нельзя разделить на независимые атрибуты Число, Месяц, Год.

Третье правило. Если сущность имеет идентификатор, состоящий из нескольких атрибутов, то каждый атрибут, не являющийся частью идентификатора, представляет собой характеристику всей сущности, а не части его идентификатора. Данное правило вытекает из 2NF.

Например, для сущности

Перемещение-жидкости (ID-источника, ID-приемника, Объем-жидкости)

атрибут Перемещение-жидкости.Объем-жидкости обозначает объем перемещаемой жидкости, а не объем источника или приемника жидкости.

Четвертое правило. Каждый атрибут, не являющийся частью идентификатора, представляет собой характеристику экземпляра, указанного идентификатором, а не характеристику другого атрибута-неидентификатора. Данное правило вытекает из 3NF.

Например, для сущности

Порция (ID-порции, ID-рецепта, Вес, Время-приготовления)

атрибут Порция.Время-приготовления определяет фактическое время приготовления порции, а не время, определяемое рецептом.

Резюме

Информационное моделирование основано на реляционной модели данных. Поэтому сущности и их атрибуты должны удовлетворять требованию нормализация данных. Обычно ограничиваются приведением модели данных к третьей нормальной форме. В информационной модели должны соблюдаться правила атрибутов, вытекающие из нормальных форм.

5.4.6. Связи

Между различными видами существующих в предметной области объектов имеются некоторые отношения и взаимосвязи. В общем случае ***связь*** – это

представление набора отношений, которые систематически возникают между различными видами реальных или абстрактных объектов (людей, предметов, мест, идей, событий, их комбинаций) в предметной области.

Применительно к IDEF1X-модели под ***связью*** понимается отношение между двумя сущностями или между экземплярами одной и той же сущности.

Соединительной связью называется связь между родительской и дочерней сущностью. Таким образом, если две сущности связаны соединительной связью, то одна из них является родительской, а вторая – дочерней.

Сущность называется ***дочерней***, если ее экземпляры могут быть связаны с нулем или одним экземпляром другой сущности (родительской). Сущность называется ***родительской***, если ее экземпляры могут быть связаны с любым количеством экземпляров другой сущности (дочерней). При этом каждый экземпляр соединительной связи связывает конкретные экземпляры родительской и дочерней сущностей. Поэтому соединительная связь полностью называется ***специфической*** (конкретной, определенной, specific) ***соединительной связью***.

Например, если две сущности Отец и Ребенок связаны соединительной связью, то сущность Отец является родительской (отец может иметь любое количество детей), а сущность Ребенок – дочерней (ребенок имеет одного отца). При этом конкретный экземпляр сущности Отец (например Сидоров Константин) связан с конкретными экземплярами сущности Ребенок (например Иван и Николай).

Очевидно, что зависимая сущность всегда является дочерней. Независимая сущность по отношению к данной соединительной связи может являться как родительской, так и дочерней.

Графически соединительная связь представляется линией от родительской к дочерней сущности с точкой со стороны дочерней сущности (рис. 5.21).

Каждой связи в модели присваивается уникальный номер вида R/1, R/2, …, R/i (**R** – Relationship – связь).

Рис. 5.21. Графическое представление соединительной связи

Каждой связи присваивается имя, образованное на основе глагола. Имя связи называется ***меткой***. В пределах модели уникальность имени связи не является обязательной. Однако для связей, существующих между двумя конкретными сущностями, имена должны быть уникальны.

Имя связи должно быть образовано в направлении от родительской сущности к дочерней таким образом, чтобы можно было составить осмысленное предложение, в котором участвуют имя родительской сущности, имя связи и имя дочерней сущности. Например, связь между родительской сущностью Владелец-собаки и дочерней сущностью Собака может быть описана следующим образом (имя связи подчеркнуто):

Владелец-собаки владеет Собака

Метка связи может также содержать пару имен – имя связи с точки зрения родительской сущности и имя связи с точки зрения дочерней сущности. В этом случае метка образуется следующим образом: вначале записывается имя связи с точки зрения родительской сущности, затем после символа «/» – имя связи с точки зрения дочерней сущности. Связь, именуемая с точки зрения дочерней сущности, называется ***обратной (реверсной) связью***.

Для предыдущего примера связь может быть поименована следующим образом:

Владелец-собаки владеет/принадлежит Собака

Резюме

Связь определяет отношение между двумя сущностями или между экземплярами одной и той же сущности. Соединительной связью называется связь между родительской и дочерней сущностью. Сущность называется дочерней, если ее экземпляры могут быть связаны с нулем или одним экземпляром другой сущности (родительской). Сущность называется родительской, если ее экземпляры могут быть связаны с любым количеством экземпляров другой сущности (дочерней). Графически соединительная связь представляется линией от родительской к дочерней сущности с точкой со стороны дочерней сущности.

5.4.7. Безусловные и условные связи и их мощность

В теории информационного моделирования существуют ***две базовые формы связей*** – безусловные и условные.

Если в связи участвуют все экземпляры обеих сущностей, то связь называется ***безусловной***.

Существует ***три вида безусловных связей***:

1) один-к-одному (1 : 1);
2) один-ко-многим (1 : M);
3) многие-ко-многим (M : M).

Данные виды связей называются ***фундаментальными***, поскольку на их основе могут быть построены другие виды связей.

Связь один-к-одному (1 : 1) существует, когда один экземпляр родительской сущности связан с единственным экземпляром дочерней сущности и каждый экземпляр дочерней сущности связан строго с одним экземпляром родительской сущности.

Например, муж женат на одной жене, жена замужем за одним мужем.

Связь один-ко-многим (1 : M) существует, когда один экземпляр родительской сущности связан с одним или более экземпляром дочерней сущности, и каждый экземпляр дочерней сущности связан строго с одним экземпляром родительской сущности.

Например, каждый владелец собаки владеет одной или несколькими собаками, каждая собака принадлежит только одному владельцу.

Связь многие-ко-многим (M : M) существует, когда один экземпляр некоторой сущности связан с одним или более количеством экземпляров другой сущности и каждый экземпляр второй сущности связан с одним или более количеством экземпляров первой.

Максимальное количество экземпляров сущности, связанных с каждым экземпляром другой сущности, называется ***мощностью связи***.

В ***условной связи*** могут существовать экземпляры одной из сущностей, которые не принимают участия в связи. Связь, условная с обеих сторон, называется ***биусловной***. В этом случае могут существовать экземпляры обеих сущностей, которые не участвуют в связи.

С учетом безусловных, условных и биусловных связей существует ***десять форм связей*** между сущностями (рис. 5.22). В соответствии с положениями теории информационного моделирования на данном рисунке буквой «у» на конце связи обозначена условность данной связи.

Связь многие-ко-многим в IDEF1X-моделировании называется ***неспецифической связью*** [2]. Данный вид связи рассматривается в п. 5.4.11.

Связи один-к-одному и один-ко-многим являются соединительными связями между родительской и дочерней сущностью. Правила их графического представления в стандарте IDEF1X приведены в п. 5.4.8.

Резюме

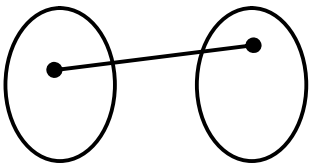
Существует три вида безусловных связей: один-к-одному, один-ко-многим, многие-ко-многим. Мощность связи определяет максимальное количество экземпляров сущности, связанных с каждым экземпляром другой сущности. С учетом безусловных, условных и биусловных связей существует десять форм связей между сущностями.

5.4.8. Графическое представление мощности соединительных связей в IDEF1X-моделировании

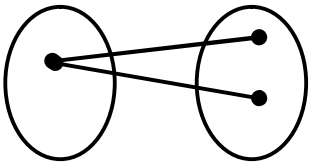
На рис. 5.23 представлено графическое представление в IDEF1X мощности соединительной связи с позиции родительской сущности [2]. Как видно из данного рисунка, с каждым экземпляром родительской сущности в общем случае может быть связано различное количество экземпляров дочерней сущности.

Максимальное количество экземпляров дочерней сущности, связанных с каждым экземпляром родительской сущности, называется ***дочерней мощностью связи***. Дочерняя мощность связи записывается рядом с точкой, находящейся на конце связи со стороны дочерней сущности.

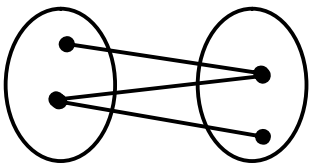
Безусловные формы



1 : 1

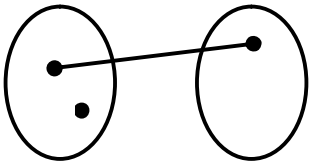


1 : M

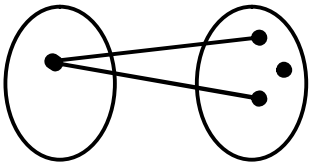


M : M

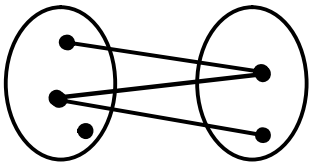
Условные формы



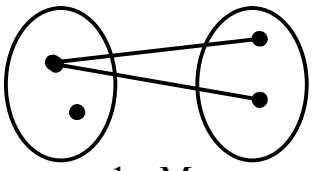
1 : 1y



1y : M

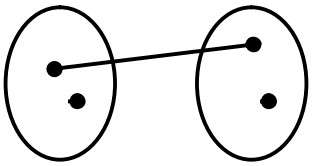


M : My

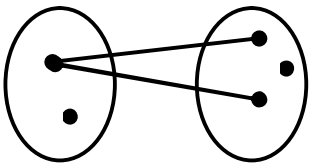


1 : My

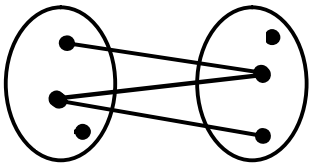
Биусловные формы



1y : 1y



1y : My



My : My

Рис. 5.22. Десять форм связи

По умолчанию значение дочерней мощности равно ноль-один-или-много. Это означает, что в каждом экземпляре связи участвует ноль, один или более экземпляров дочерней сущности. При этом дочерняя мощность связи рядом с точкой не отмечается (см. рис. 5.23, а).

Например, между сущностями Семья и Ребенок существует связь один-к-нулю-одному-или-многим, поскольку каждый экземпляр сущности Семья может не иметь ни одного ребенка, иметь одного ребенка или иметь большее количество детей (рис. 5.24).

Если рядом с точкой записан символ **P** (см. рис. 5.23, б), то это значит, что в экземпляре связи участвует один или много экземпляров дочерней сущности. Например, каждый экземпляр сущности Владелец-собаки владеет одним или несколькими экземплярами сущности Собака (рис. 5.25).

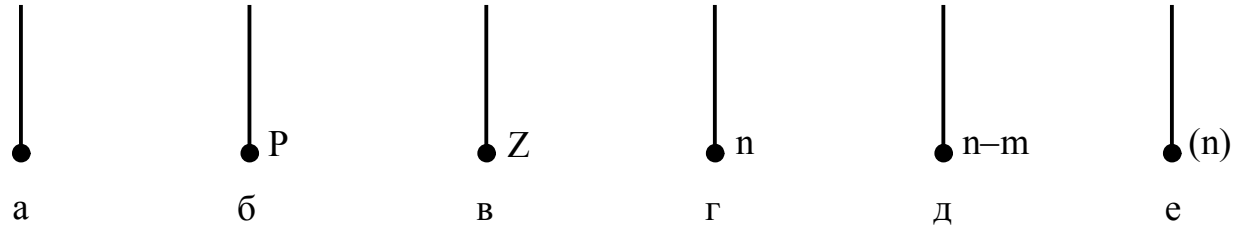


Рис. 5.23. Графическое представление
различных видов дочерней мощности соединительных связей в IDEF1X:
а – ноль-один-или-много; б – один-или-много; в – ноль-или-один;
г – точно-n; д – от n до m;
е – мощность определяется условием, записанным в скобках

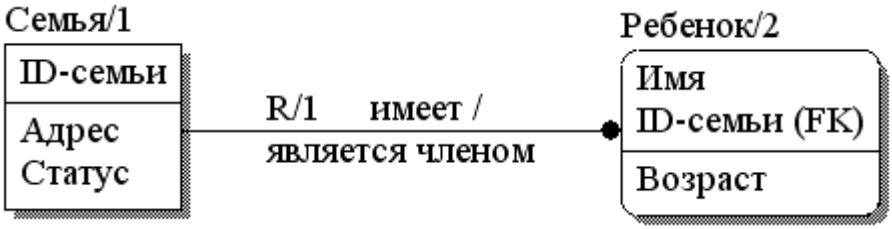


Рис. 5.24. Графическое представление в IDEF1X
связи мощностью один-к-нулю-одному-или-многим

Это соответствует мощности связи один-к-одному-или-многим.

Символ **Z**, помещенный рядом с точкой, означает, что ноль или один экземпляр дочерней сущности участвует в экземпляре связи (см. рис. 5.23, в). Например, каждый экземпляр сущности Мужчина неженат (женат на нуле женщин) или женат на одной женщине (рис. 5.26).

Это соответствует мощности связи один-к-нулю-или-одному.

Конкретное число, помещенное рядом с точкой (см. рис. 5.23, г), определяет точную дочернюю мощность связи (точное количество экземпляров до-

черней сущности, участвующих в связи с экземпляром родительской сущности). Например, каждый экземпляр сущности Год связан точно с двенадцатью экземплярами сущности Месяц (в году 12 месяцев, рис. 5.27).

Это соответствует мощности связи один-к-точно-12.

Рис. 5.25. Графическое представление в IDEF1X
связи мощностью один-к-одному-или-многим

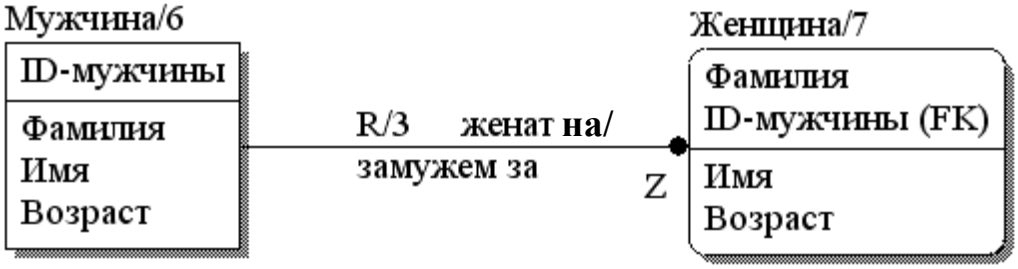


Рис. 5.26. Графическое представление в IDEF1X
связи мощностью один-к-нулю-или-одному

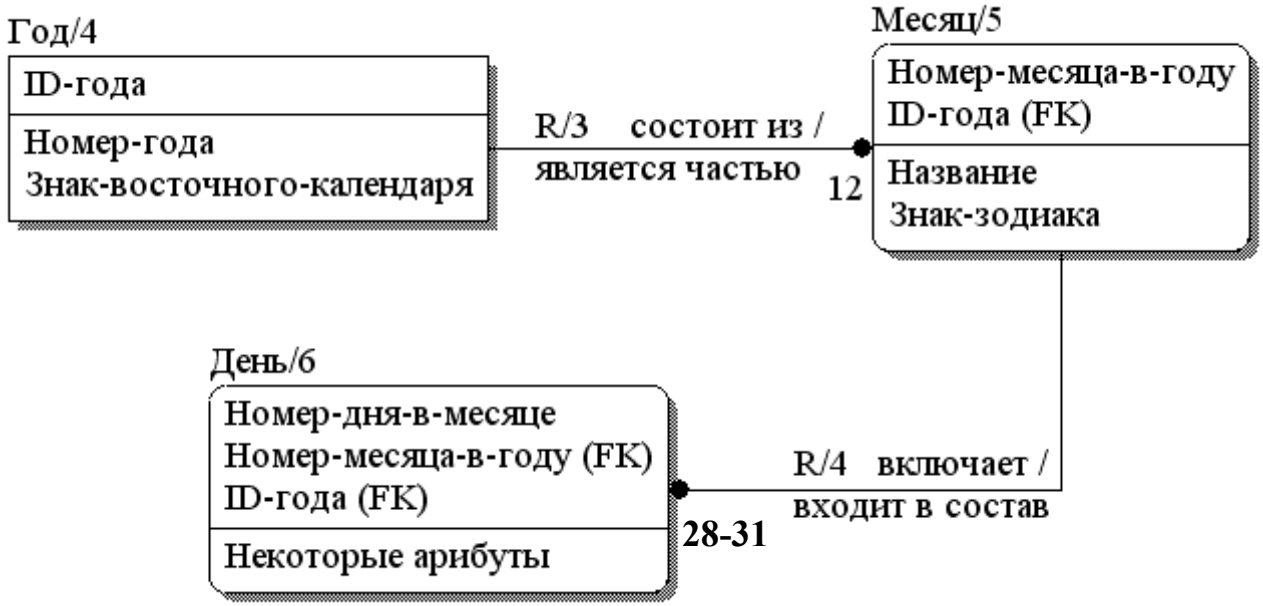


Рис. 5.27. Графическое представление в IDEF1X
связей мощностью один-к-точно-n и один-к-диапазону-n-m

Если мощность дочерней связи определена в диапазоне от некоторого значения n до некоторого значения m, то данный диапазон записывается рядом с точкой в виде **n – m** (см. рис. 5.23, д). Например, каждый экземпляр сущности Месяц связан с 28 – 31 экземплярами дочерней сущности День (см. рис. 5.27). Это соответствует мощности связи один-к-диапазону-28-31.

Другие условия, характеризующие дочернюю мощность связи, определяются в круглых скобках (см. рис. 5.23, е). Например, книга может содержать не менее двух глав. В этом случае рядом с точкой на связи должно быть записано условие (>= 2), определяющее дочернюю мощность связи (рис. 5.28). Это соответствует мощности связи один-к-(>= 2).

Следует отметить, что два последних вида мощности (один-к-диапазону--n-m и один-к-(условие)) современные CASE-средства зачастую не поддерживают, поскольку такой вид связей в предметных областях встречается достаточно редко и может быть реализован с помощью других видов мощности. Эти виды мощностей не поддерживает, например, и одно из самых распространенных в Беларуси CASE-средств для разработки баз данных CA ERwin Data Modeler, более известное под названием Erwin (см. подразд. 7.6).

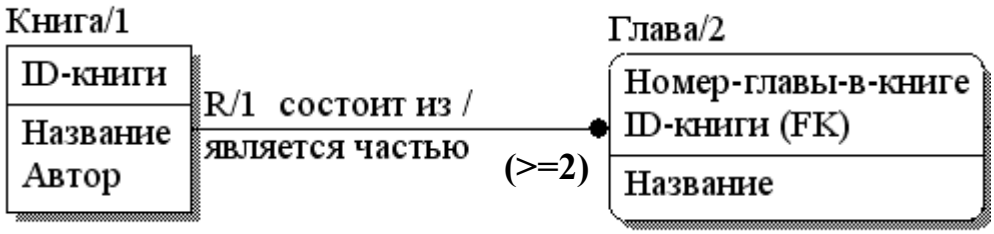


Рис. 5.28. Графическое представление в IDEF1X
связи мощностью один-к-(условие)

С учетом дочерней мощности связь между родительской сущностью и дочерней сущностью должна описываться в прямом направлении следующим образом:

<Имя-родительской-сущности> <Имя-связи> <Дочерняя-мощность-связи> <Имя-дочерней-сущности>

и в обратном направлении как

<Дочерняя-мощность-связи> <Имя-дочерней-сущности> <Имя-связи> <Имя-родительской-сущности> .

Например, прямая и обратная связь между родительской сущностью Владелец-собаки и дочерней сущностью Собака, представленная на рис. 5.25, описывается следующим образом:

Владелец-собаки/3 R/2 владеет один-или-много Собака/4 ,

Один-или-много Собака/4 R/2 принадлежит Владелец-Собаки/3 .

Как следует из рис. 5.23, виды дочерней мощности связи, согласно стандарту IDEF1X, представляют как условные, так и безусловные связи (см. п. 5.4.10).

Резюме

В стандарте IDEF1X определены следующие дочерние мощности соединительных связей: ноль-один-или-много; один-или-много; ноль-или-один; точно-n; от n до m; мощность, определяемая условием. Каждая дочерняя мощность имеет свое графическое обозначение. Не все виды дочерней мощности поддерживаются CASE-средствами.

5.4.9. Формализация соединительных связей

Целью связи является установление соотношения конкретного экземпляра одной сущности с конкретным экземпляром другой. В соединительных связях данная цель достигается размещением вспомогательных атрибутов (внешних ключей) в дочерней сущности. Связь, определенная с помощью вспомогательных атрибутов, называется ***связью, формализованной в данных***.

В качестве вспомогательных атрибутов дочерней сущности используются идентифицирующие атрибуты родительской сущности. Вспомогательные атрибуты помечаются аббревиатурой FK (Foreign Key) в скобках (см. рис. 5.24 – 5.28).

Существует *два вида связей, формализованных в данных*: идентифицирующая связь и неидентифицирующая связь.

Если конкретные экземпляры дочерней сущности определяются через связь с родительской сущностью, то такая связь называется ***идентифицирующей***. При данном виде связи каждый экземпляр дочерней сущности должен быть связан точно с одним экземпляром родительской сущности.

Графически идентифицирующие связи изображаются *сплошной линией* с точкой со стороны дочерней сущности. При идентифицирующей связи вспомогательные атрибуты включаются в состав идентификатора дочерней сущности, а сама дочерняя сущность является *зависимой* и представляется прямоугольником с закругленными углами.

Таким образом, связи, приведенные на рис. 5.24 – 5.28, являются идентифицирующими.

Например, экземпляр сущности Ребенок (см. рис. 5.24) не может быть однозначно определен по своему идентифицирующему атрибуту Имя, поскольку в данной сущности могут быть экземпляры с одинаковыми именами. Поэтому только совокупность имени ребенка и номера (идентификатора) семьи однозначно определяют конкретного ребенка.

Рис. 5.25 отражает ситуацию, при которой для каждого экземпляра сущности Владелец-собаки экземпляры сущности Собака нумеруются (Номер-

собаки), начиная с единицы. Поэтому очевидно, что в сущности Собака существует много экземпляров с одинаковыми номерами и их нельзя идентифицировать без связи с конкретными экземплярами родительской сущности.

Родительская сущность в идентифицирующей связи является независимой. Однако если данная сущность является дочерней сущностью в другой идентифицирующей связи, то в рассматриваемой связи обе сущности (и родительская, и дочерняя) будут зависимыми. На рис. 5.27 родительская сущность Месяц связи R/4 одновременно является дочерней сущностью идентифицирующей связи R/3. Поэтому в связи R/4 как родительская, так и дочерняя сущности являются зависимыми. Идентификатор родительской сущности Год (ID-года) по связи R/3 мигрирует в дочернюю сущность Месяц. Отсюда идентифицирующие атрибуты (ID-года и Номер-месяца-в-году) по связи R/4 мигрируют в дочернюю сущность День.

Если каждый экземпляр дочерней сущности может быть уникально идентифицирован без связи с родительской сущностью, то такая связь называется ***неидентифицирующей***.

Графически неидентифицирующие связи изображаются *штриховой линией* с точкой на конце дочерней сущности. При неидентифицирующей связи вспомогательные атрибуты не включаются в состав идентификатора дочерней сущности, а дочерняя сущность является *независимой*. Однако если данная сущность является дочерней сущностью другой идентифицирующей связи, то она будет зависимой.

На рис. 5.29 – 5.31 представлены примеры неидентифицирующих связей между родительской и дочерней сущностями. Данные примеры даны в привязке к примерам, приведенным на рис. 5.24 – 5.28.

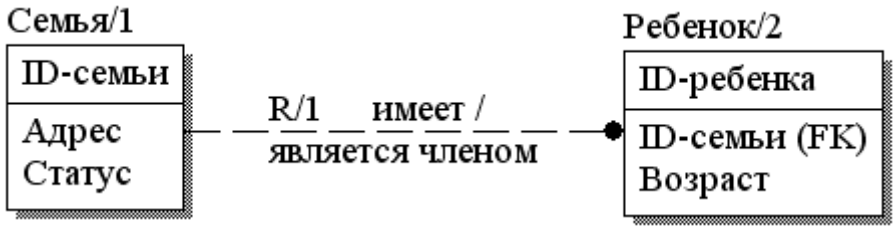


Рис. 5.29. Графическое представление в IDEF1X
неидентифицирующей связи
мощностью один-к-нулю-одному-или-многим

Организация неидентифицирующей связи между сущностями Семья и Ребенок (см. рис. 5.29) стала возможной, потому что в качестве идентификатора дочерней сущности Ребенок выбран идентификационный номер экземпляра сущности Ребенок (ID-ребенка), уникально определяющий каждый ее экземпляр (сравните с неуникальным идентификатором Имя на рис. 5.24).

Аналогично для определения экземпляров дочерних сущностей Собака и Женщина (см. рис. 5.30, 5.31) также используются идентификационные номера

ID-собаки и ID-женщины, что позволило отказаться от применения в данных моделях идентифицирующих соединительных связей (сравните с рис. 5.25, 5.26).

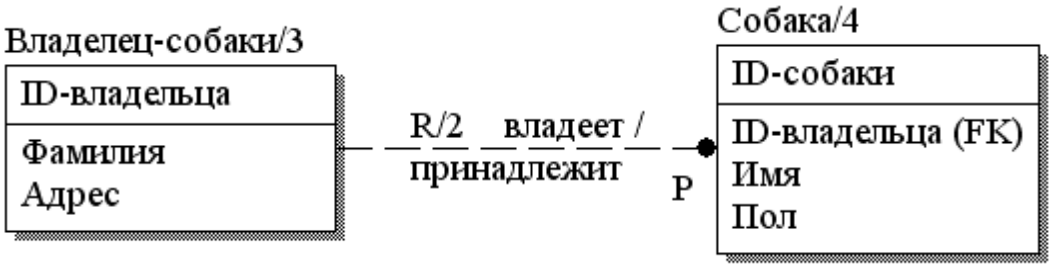


Рис. 5.30. Графическое представление в IDEF1X
неидентифицирующей связи
мощностью один-к-одному-или-многим

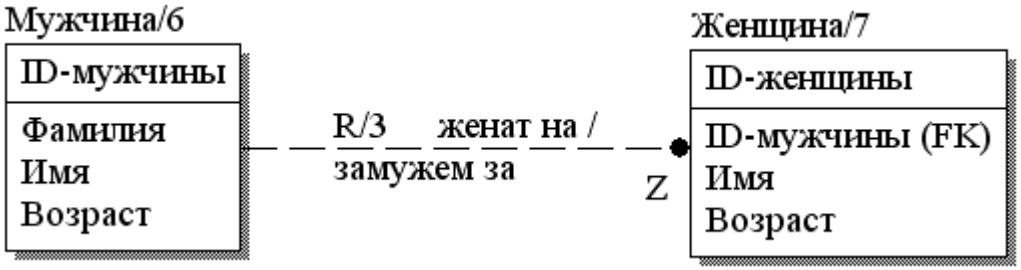


Рис. 5.31. Графическое представление в IDEF1X
неидентифицирующей связи
мощностью один-к-нулю-или-одному

Как показывают рассмотренные выше примеры (см. рис. 5.24 – 5.31), одну и ту же информационную модель в общем случае можно разработать, используя идентифицирующие или неидентифицирующие связи. Следует обратить внимание, что применение идентифицирующих связей приводит к появлению составных идентификаторов в дочерних сущностях (см. например рис. 5.27). Составные идентификаторы при работе с реальными базами данных обрабатывать сложнее, чем простые. Их использование приводит к снижению эффективности работы с базами данных. Поэтому при разработке информационных моделей предметной области предпочтение по возможности следует отдавать неидентифицирующим связям перед идентифицирующими.

Резюме

Связь, определенная с помощью вспомогательных атрибутов, называется связью, формализованной в данных. Существуют идентифицирующие и неидентифицирующие связи, формализованные в данных. При идентифицирующей связи вспомогательные атрибуты включаются в состав идентификатора

дочерней сущности, при неидентифицирующей – не включаются. Предпочтение по возможности следует отдавать неидентифицирующим связям перед идентифицирующими.

5.4.10. Реализация безусловных и условных связей в IDEF1X-моделировании

Анализ рис. 5.23 показывает, что совокупность видов дочерней мощности связи, определенных в стандарте IDEF1X, определяет собой как условные, так и безусловные связи.

Виды дочерней мощности связи, не включающие в себя ноль (см. рис. 5.23) определяют, что конкретный экземпляр родительской сущности может быть соединен не менее чем с одним экземпляром дочерней сущности. К таким видам дочерней мощности относятся мощности один-или-много, точно-n, диапазон-n-m, мощность, определяемая условием (два последних вида, если диапазон и условие не включают ноль).

Например, сущность Владелец-собаки владеет не менее чем одной собакой (см. рис. 5.25, 5.30), сущность Год состоит из двенадцати месяцев, сущность Месяц включает 28 – 31 день (см. рис. 5.27).

Таким образом, в этом случае отдельные экземпляры родительской сущности всегда участвуют в соединительной связи. И, следовательно, связи с дочерней мощностью, не включающей в себя ноль, определяют ***безусловность связи со стороны родительской сущности***.

Значения дочерней мощности связи, включающие в себя ноль (мощности ноль-один-или-много, ноль-или-один, диапазон-n-m, мощность, определяемая условием, если диапазон и условие включают ноль), определяют, что конкретный экземпляр родительской сущности может быть соединен с нулем экземпляров дочерней сущности.

Например, некоторые экземпляры сущности Семья могут не иметь ни одного ребенка (см. рис. 5.24, 5.29), некоторые экземпляры сущности Мужчина могут не иметь жены (см. рис. 5.26, 5.31).

Таким образом, отдельные экземпляры родительской сущности могут не участвовать в соединительной связи.

Следовательно, ***условность связи со стороны родительской сущности*** определяется дочерней мощностью связи, включающей в себя ноль.

Если связь является идентифицирующей, то со стороны дочерней сущности она всегда является безусловной, поскольку в данном случае выделение конкретного экземпляра дочерней сущности по определению невозможно без связи с родительской сущностью. Поэтому условность связи со стороны дочерней сущности возможна, только если связь является неидентифицирующей, причем условность или безусловность данной связи определяется ее видом.

Существует два вида неидентифицирующих связей: обязательные и необязательные.

Неидентифицирующая связь называется ***обязательной (mandatory)***, если каждый экземпляр дочерней сущности связан точно с одним экземпляром родительской сущности. Таким образом, каждый экземпляр дочерней сущности участвует в связи.

Следовательно, обязательная неидентифицирующая связь определяет ***безусловность связи со стороны дочерней сущности***.

Например, сущности Муж и Жена на рис. 5.32 связаны обязательной неидентифицирующей связью, поскольку каждая жена обязательно имеет одного мужа. Дочерняя мощность связи равна точно-1. Поэтому связь между сущностями Муж и Жена является безусловной с обеих сторон и имеет мощность один-к-одному.

Неидентифицирующая связь называется ***необязательной (optional)***, если каждый экземпляр дочерней сущности может быть связан с нулем или одним экземпляром родительской сущности. В данном случае конкретные экземпляры дочерних сущностей могут существовать без связи с конкретными экземплярами родительской сущности и существуют экземпляры дочерней сущности, не участвующие в связи.

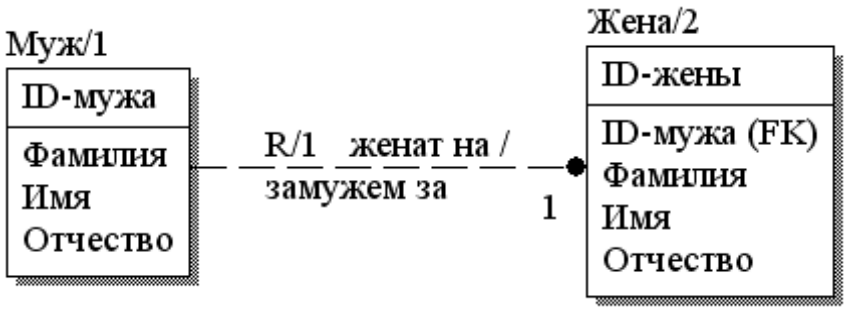


Рис. 5.32. Графическое представление в IDEF1X
обязательной неидентифицирующей связи мощностью один-к-одному

Таким образом, необязательная неидентифицирующая связь определяет ***условность связи со стороны дочерней сущности***.

Например, в рассмотренном в п. 5.4.9 примере (см. рис. 5.31) была определена условность связи со стороны родительской сущности Мужчина за счет дочерней мощности связи ноль-или-один (конкретный мужчина женат на нуле или одной женщине). При этом не был учтен факт, что конкретная женщина также может не иметь мужа. Следовательно, конкретные экземпляры сущности Женщина могут не участвовать в связи. Поэтому между сущностями мужчина и Женщина следует организовать необязательную неидентифицирующую связь.

Графически необязательная неидентифицирующая связь изображается *пунктирной линией с ромбом* со стороны родительской сущности.

На рис. 5.33 учтена условность связи со стороны дочерней сущности Женщина за счет использования необязательной неидентифицирующей связи. Результирующая мощность связи на данном рисунке равна ноль-или-один-к-

нулю-или-одному. Таким образом, со стороны обеих сущностей в мощности связи присутствует ноль. Следовательно, связь между сущностями Мужчина и Женщина является *биусловной*.

Таким образом, обязательность или необязательность неидентифицирующей связи определяет ***родительскую мощность*** данной связи (количество экземпляров родительской сущности, связанных с каждым экземпляром дочерней сущности). Если связь идентифицирующая или обязательная неидентифицирующая, то родительская мощность связи равна единице. Для необязательных неидентифицирующих связей родительская мощность равна ноль-или-один.

Родительская мощность соединительной связи, равная единице, графически не отображается. Признаком родительской мощности, равной ноль-или-один, является наличие ромба у родительского конца связи.

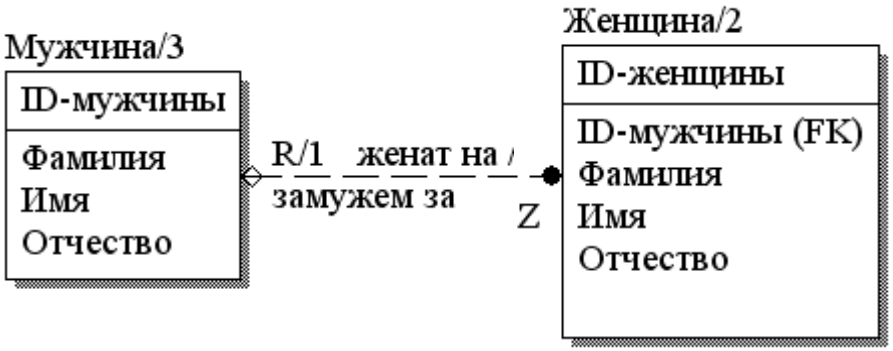


Рис. 5.33. Графическое представление в IDEF1X необязательной неидентифицирующей связи мощностью ноль-или-один-к-нулю-или-одному

Резюме

Безусловность связи со стороны родительской сущности определяется дочерней мощностью, не включающей в себя ноль. Условность связи со стороны родительской сущности определяется дочерней мощностью связи, включающей в себя ноль. Существуют обязательные и необязательные неидентифицирующие связи. Обязательная неидентифицирующая связь определяет безусловность связи со стороны дочерней сущности. Необязательная неидентифицирующая связь определяет условность связи со стороны дочерней сущности.

5.4.11. Неспецифические связи

Неспецифической связью (неконкретной, неопределенной, non-specific) называется связь, при которой экземпляр каждой сущности может быть связан с некоторым количеством экземпляров другой сущности. Неспецифическая связь называется также ***связью многие-ко-многим*** (см. п. 5.4.7). Мощность неспецифической связи в общем случае равна ноль-один-или-много-к-нулю-

одному-или-многим. Графически неспецифическая связь изображается сплошной линией с точками на обоих концах.

На рис. 5.34 приведен пример неспецифической связи между сущностями Владелец-квартиры и Квартира. Между данными сущностями существует связь многие-ко-многим, поскольку каждый экземпляр сущности Владелец-квартиры может владеть несколькими квартирами, и каждый экземпляр сущности Квартира может являться собственностью нескольких владельцев.

Неспецифические связи между сущностями могут быть полезны в начале работы над информационной моделью, когда сложно определить конкретные связи между экземплярами сущностей. На более поздних этапах разработки модели каждую неспецифическую связь следует заменять на пару специфических соединительных связей.

С этой целью создается третья сущность, представляющая собой общую дочернюю сущность, связанную соединительными связями с двумя исходными сущностями.

Сущности, вводимые для устранения неспецифических связей, называются ***ассоциативными.*** Ассоциативные сущности содержат идентификаторы каждого из участвующих в связи экземпляров исходных сущностей, что позволяет формализовать в данных связи многие-ко-многим, устранить избыточность информационной модели, а следовательно, и соответствующей базы данных.

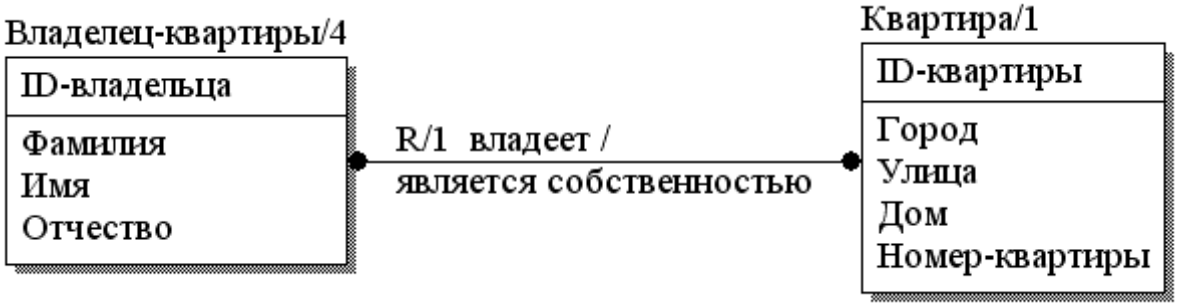


Рис. 5.34. Графическое представление в IDEF1X неспецифической связи

На рис. 5.35 создана ассоциативная сущность Владение, содержащая вспомогательные атрибуты, в качестве которых используются идентификаторы сущностей Квартира и Владелец-квартиры.

После введения ассоциативной сущности следует определить мощность каждой из связанных с ней соединительных связей и присвоить данным связям соответствующие имена. В рассматриваемом примере каждый экземпляр сущности Владелец-квартиры владеет одним или несколькими владениями (мощность связи один-к-одному-или-многим, что отражено символом **P** на конце связи, см. рис. 5.35). Каждая квартира (с учетом того, что она может быть, например, не продана) не является владением, является одним или несколькими

владениями. Это определяет мощность связи один-к-нулю-одному-или-многим).

Следует отметить, что в ряде CASE-средств (например ERwin, см. подразд. 7.6) формализация связи многие-ко-многим посредством ассоциативной сущности реализована автоматически при переходе от логического уровня представления модели к физическому [21, 28].

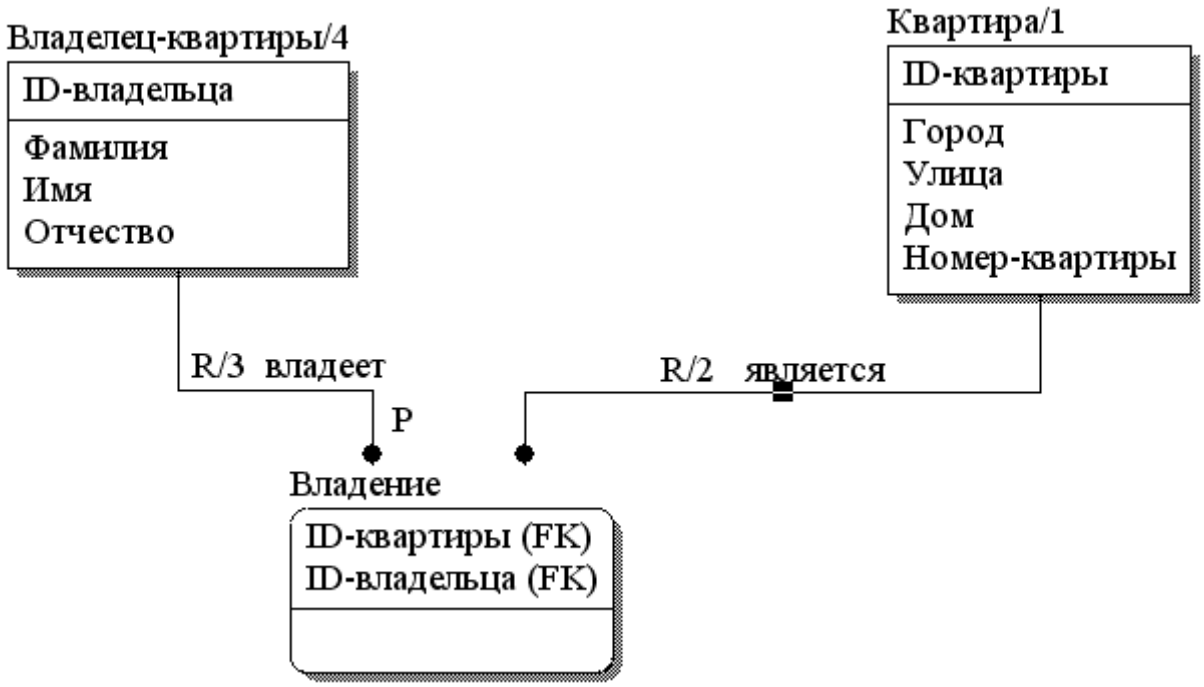


Рис. 5.35. Формализация связи многие-ко-многим посредством ассоциативной сущности

Резюме

При неспецифической связи (связи многие-ко-многим) экземпляр каждой сущности может быть связан с некоторым количеством экземпляров другой сущности. Каждую неспецифическую связь следует заменять на две специфических соединительных связи. Для этого вводится ассоциативная сущность, содержащая идентификаторы каждого из участвующих в экземпляре связи экземпляров исходных сущностей.

5.4.12. Организация рекурсивных связей в IDEF1X

Неидентифицирующие соединительные связи и неспецифические связи могут быть ***рекурсивными***, то есть могут связывать экземпляры сущности с другими экземплярами той же сущности.

На рис. 5.36 приведен пример организации рекурсивной связи между экземплярами сущности с использованием *неидентифицирующей соединитель-*

ной связи. Каждый экземпляр сущности Сотрудник связан с другими экземплярами рекурсивной связью R13 (руководит / подчиняется). Каждый сотрудник имеет неидентифицирующий атрибут Должность. Если должность руководящая, то данный сотрудник руководит одним или несколькими подчиненными. Если должность подчиненная, то данный сотрудник может не руководить другими сотрудниками, но подчиняется некоторому руководителю. Один и тот же сотрудник может руководить нулем, одним или несколькими подчиненными и в то же время подчиняется руководителю более высокого ранга. Некоторые сотрудники (например руководитель самого высокого ранга) руководителя не имеют, то есть в обратной связи (подчиняется) не участвуют. Поэтому общая мощность связи между экземплярами сущности Сотрудник равна ноль-или-один-к-нулю-одному-или-многим. Таким образом, данная связь является биусловной.

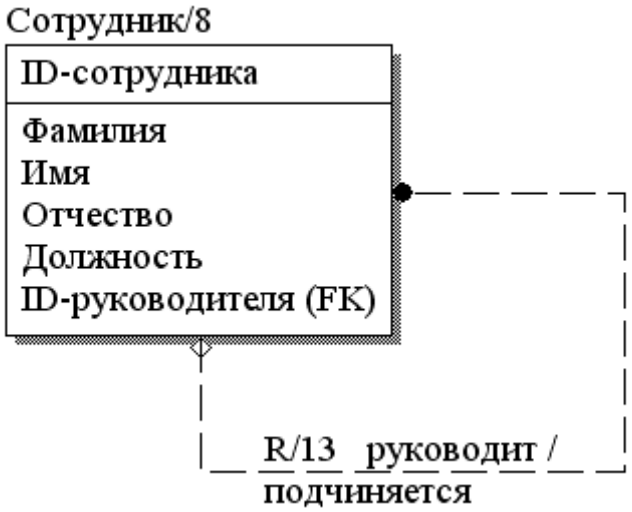


Рис. 5.36. Организация рекурсивной связи
мощностью ноль-или-один-к-нулю-одному-или-многим
(использование имени роли)

В сущности Сотрудник на рис. 5.36 появился вспомогательный атрибут (внешний ключ) ID-руководителя. Его имя отличается от идентификатора ID-сотрудника, мигрирующего по связи руководит из родительского в дочерний экземпляр сущности. В данном случае внешнему ключу присвоено имя роли. ***Имя роли*** – это имя, альтернативное базовому имени мигрирующего из родительской сущности внешнего ключа, отражающее роль данного ключа в дочерней сущности. Имя роли называется еще ***функциональным именем*** внешнего ключа.

Имя роли обычно используется, если нужно отразить назначение внешнего ключа в дочерней сущности. Например, на рис. 5.31 вместо базового имени вспомогательного атрибута ID-мужчины в сущности Женщина можно использовать имя роли ID-мужа.

Иногда применение имен ролей является обязательным. Это касается тех случаев, когда два или несколько атрибутов одной сущности определены в одной и той области значений, но имеют разный смысл.

В рассмотренном выше примере (см. рис. 5.36) такими атрибутами являются ID-сотрудника и ID-руководителя. В данном случае применение имени роли для внешнего ключа является обязательным, поскольку по определению имена атрибутов в пределах сущности должны быть уникальны.

В общем случае в качестве имени внешнего ключа может использоваться базовое имя, имя роли, а также полное имя

<Имя-роли>.<Базовое-имя>,

включающее имя роли и базовое имя вспомогательного атрибута (рис. 5.37).

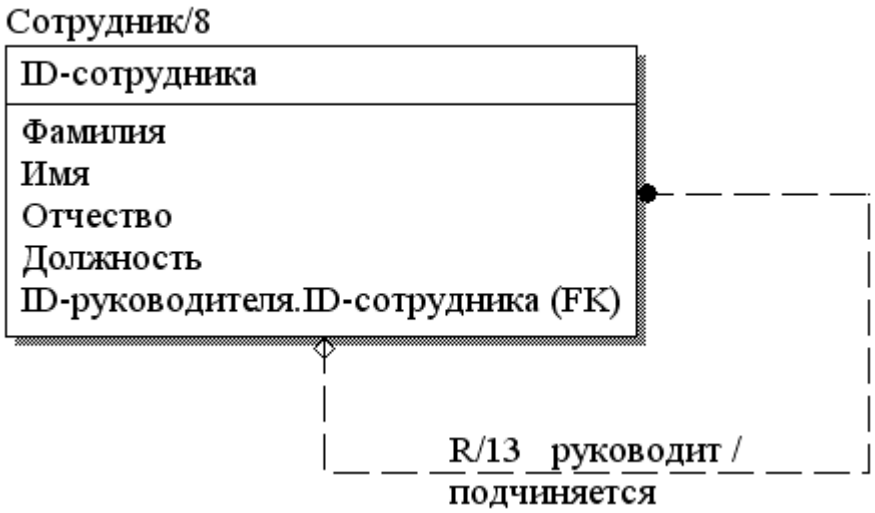


Рис. 5.37. Организация рекурсивной связи
мощностью ноль-или-один-к-нулю-одному-или-многим
(использование полного имени внешнего ключа)

Как следует из рассмотренного примера, использование неидентифицирующей соединительной связи в общем случае позволяет организовать иерархическую рекурсию между экземплярами сущности (рис. 5.38).

На рис. 5.39 приведен пример организации рекурсивной связи между экземплярами сущности с использованием *неспецифической связи*.

В данном примере некоторый экземпляр сущности Программный-модуль может вызывать ноль, один или много других экземпляров данной сущности и в то же время вызываться нулем, одним или многими другими экземплярами этой же сущности.

Рекурсивная связь многие-ко-многим устраняется аналогично рассмотренному в п. 5.4.11 введением ассоциативной сущности. На рис. 5.40 введена ассоциативная сущность Модуль. Связь многие-ко-многим заменена двумя соединительными связями между сущностями Программный-модуль и Модуль,

каждая из которых имеет мощность один-к-нулю-одному-или-многим (программный модуль может вызывать ноль, один или много других модулей и в то же время вызываться нулем, одним или многими другими модулями).

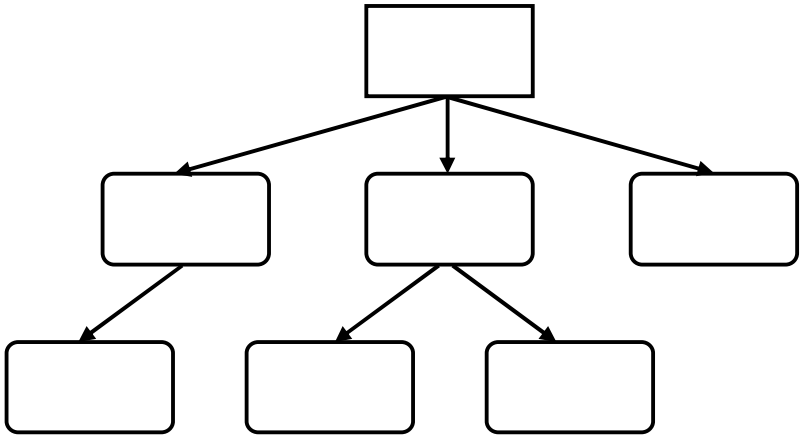


Рис. 5.38. Иерархическая рекурсия между экземплярами сущности

Рис. 5.39. Организация рекурсивной связи
мощностью многие-ко-многим

В ассоциативной дочерней сущности Модуль имеются два внешних ключа, определенных в одной и той области значений, но имеющих разный смысл. В этом случае обязательно использование имен ролей. По связи R12 (вызывает) экземпляр сущности Программный модуль связан с вызываемым модулем, по связи R11 (вызывается) – с вызывающим модулем. Поэтому в качестве имен внешних ключей в сущности Модуль используются составные имена, включающие имена ролей ID-вызываемого-модуля и ID-вызывающего-модуля.

Из рассмотренного примера видно, что использование связи многие-ко-многим в общем случае позволяет организовать сетевую рекурсию между эк-

земплярами сущности (рис. 5.41), при которой возможны связи между любыми экземплярами сущностей.

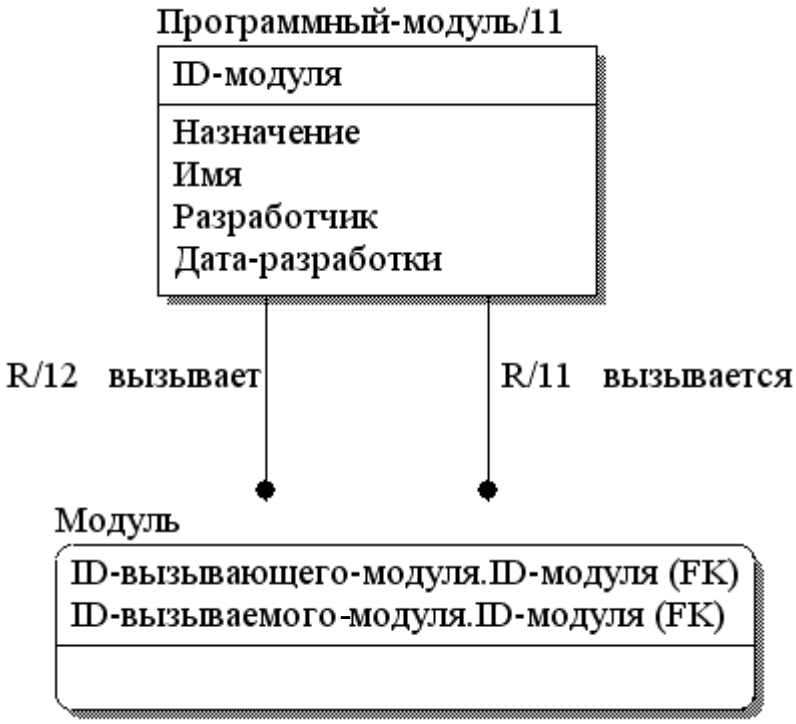


Рис. 5.40. Формализация рекурсивной связи мощностью многие-ко-многим посредством ассоциативной сущности

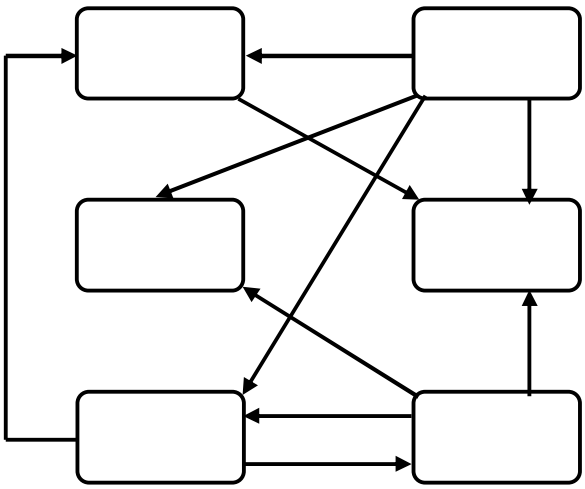


Рис. 5.41. Сетевая рекурсия между экземплярами сущности

Резюме

Возможна организация рекурсивной связи между экземплярами одной и той же сущности. Для этого может использоваться неидентифицирующая со-

единительная связь или неспецифическая связь. Неидентифицирующая соединительная связь позволяет организовать иерархическую рекурсию. Неспецифическая связь позволяет организовать сетевую рекурсию. В качестве имени внешнего ключа может использоваться базовое имя, имя роли и составное имя. Имя роли отражает роль внешнего ключа в дочерней сущности.

5.4.13. Связи категоризации в IDEF1X

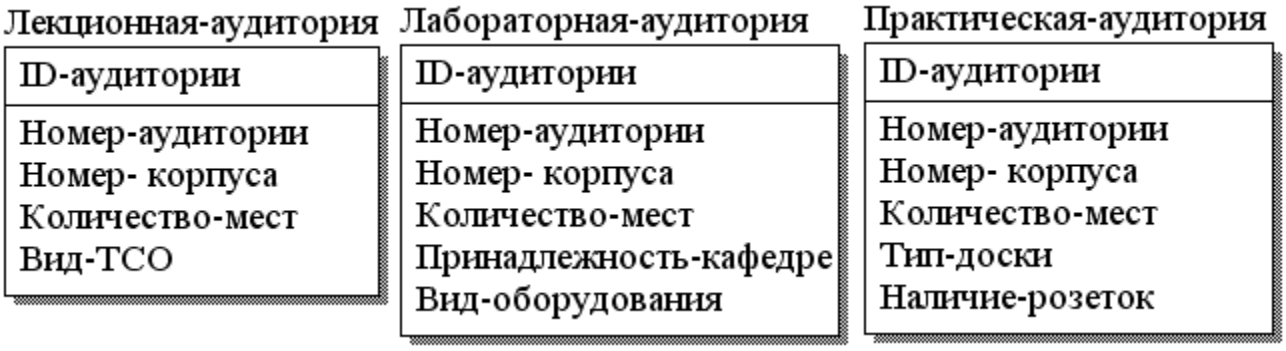
При разработке информационной модели предметной области могут возникнуть ситуации, когда несколько сущностей имеют общие атрибуты.

Рис. 5.42 иллюстрирует такую ситуацию. В аудиторном фонде университета есть аудитории различных типов (например, лекционные, лабораторные, практические). Сущности, представляющие данные аудитории, имеют ряд одинаковых атрибутов (Номер-аудитории, Номер-корпуса, Количество-мест).

Для устранения избыточности информационной модели в таких случаях рекомендуется создать общую сущность для представления характеристик, совместно принадлежащих специализированным сущностям. Общая сущность называется ***групповой сущностью*** (родовой сущностью, generic entity), специализированная сущность – ***сущностью-категорией*** (category entity). Такие сущности связаны между собой связью категоризации (рис. 5.43).

Связь категоризации (связь супертип-подтип) – это связь между групповой сущностью и сущностью-категорией, при которой один реальный или виртуальный объект предметной области представляется комбинацией экземпляра групповой сущности и экземпляра сущности-категории.

Групповая сущность представляет собой полный набор объектов, сущности-категории – подтипы этих объектов. Сущности-категории всегда являются зависимыми.



ТСО – технические средства обучения

Рис. 5.42. Сущности с общими атрибутами
для предметной области «Аудиторный фонд университета»

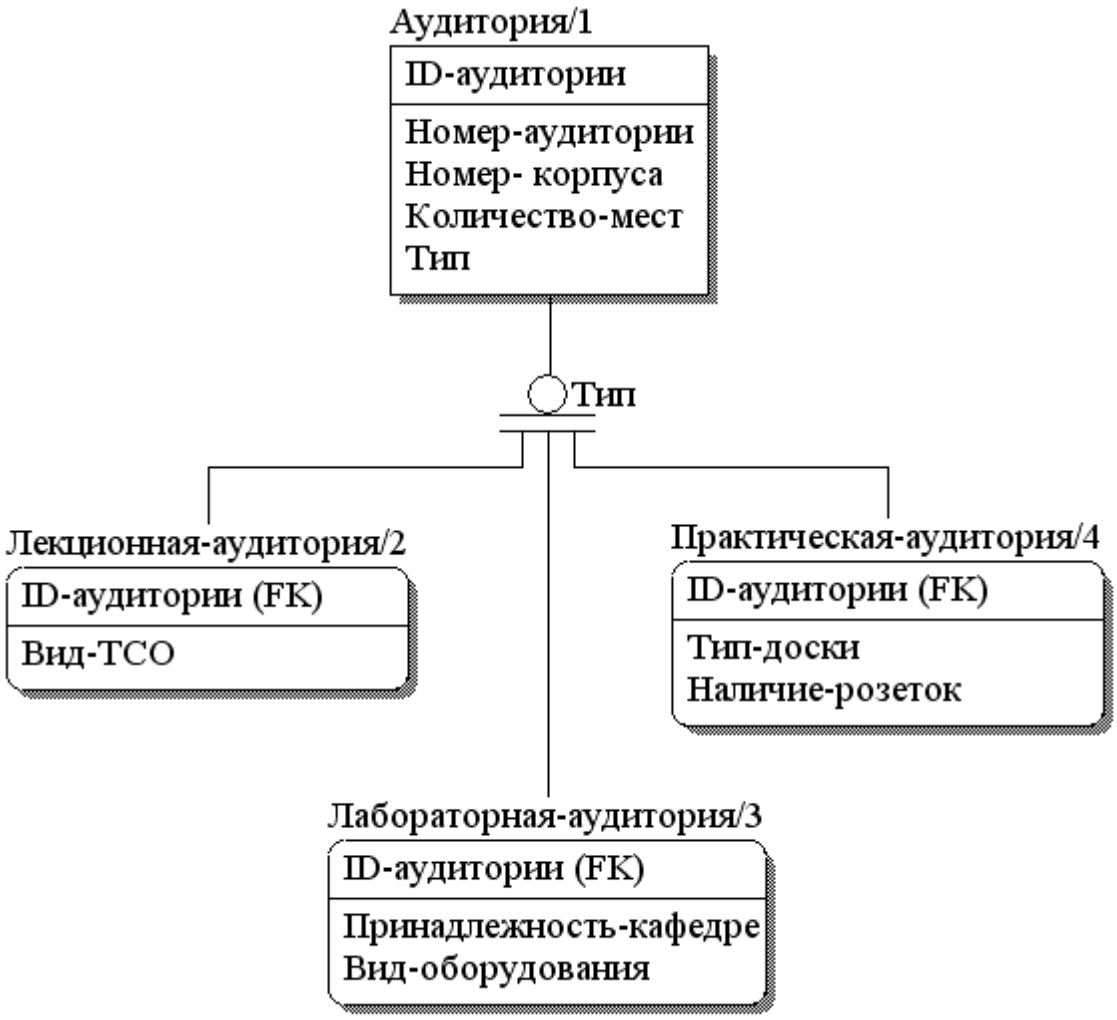


Рис. 5.43. Связь категоризации. Полная группа категорий

Таким образом, связи категоризации используются для представления структур, в которых некоторая сущность является подтипом (категорией) другой сущности. Поэтому групповая сущность называется также ***сущностью-супертипом***, а сущность-категория – ***сущностью-подтипом***.

На рис. 5.43 групповой сущностью является сущность Аудитория, сущностями-категориями – сущности Лекционная-аудитория, Практическая-аудитория, Лабораторная-аудитория.

Термин ***группа категорий*** (category cluster) определяет набор из одной или более связей категоризации и связанных с ними сущностей-категорий. Каждый экземпляр сущности-категории связан строго с одним экземпляром групповой сущности, и каждый экземпляр групповой сущности может быть связан с экземпляром только одной сущности-категории из входящих в группу. Поэтому сущности-категории в группе являются взаимно исключающими. Например, сущности Лекционная-аудитория, Практическая-аудитория, Лабораторная-

аудитория входят в состав одной группы категорий, поэтому аудитория не может быть лекционной и лабораторной одновременно (см. рис. 5.43).

Однако сущность может быть групповой для нескольких групп категорий. В этом случае групповая сущность может быть связана одновременно с одной из сущностей-категорий в каждой группе. В то же время сущность-категория может входить в состав только одной группы категорий и не может быть дочерней сущностью в идентифицирующей соединительной связи с некоторой родительской сущностью.

Если в группе категорий представлены все возможные сущности-категории, то такая группа называется ***полной группой категорий*** [2]. В этом случае каждый экземпляр групповой сущности связан с экземпляром сущности-категории.

Графически полная группа категорий представляется линией, выходящей из групповой сущности, с кругом, который подчеркивается двойной линией (см. рис. 5.43). От подчеркивания отходят отдельные линии к каждой сущности-категории, входящей в данную группу. Каждая пара линий (от групповой сущности к кругу и от круга к сущности-категории) представляет собой одну из связей категоризации, входящих в группу.

Если в группе категорий представлены не все возможные сущности-категории, то такая группа называется ***неполной группой категорий*** [2]. В этом случае экземпляр групповой сущности может существовать без связи с экземпляром сущности-категории.

При графическом представлении неполной группы категорий круг, объединяющий собой группу связей категоризации, подчеркивается одинарной линией (рис. 5.44).

Мощность связей категоризации обычно графически не изображается, поскольку она всегда равна один-к-одному для полной группы категорий и один-к-нулю-или-одному для неполной группы категорий. При необходимости рядом со связью на стороне сущности-категории может быть поставлен символ 1 или **Z**, явно отражающий данную мощность.

Один из атрибутов групповой сущности или ее предка может быть определен как дискриминатор для группы сущностей-категорий.

Дискриминатор – это атрибут групповой сущности, значение которого определяет категорию его экземпляра. Имя дискриминатора записывается рядом с кругом, объединяющим группу связей категоризации. Для примеров, приведенных на рис. 5.43, 5.44, дискриминатором является атрибут Тип групповой сущности.

Имена дискриминаторов всех групп в модели должны быть уникальны.

Связям категоризации явные имена не присваиваются. Каждая связь групповой сущности с сущностью-категорией для неполных групп категорий может быть прочитана как «*может быть*». Например, Аудитория может быть Лекционной-аудиторией или Лабораторной-аудиторией (см. рис. 5.44).

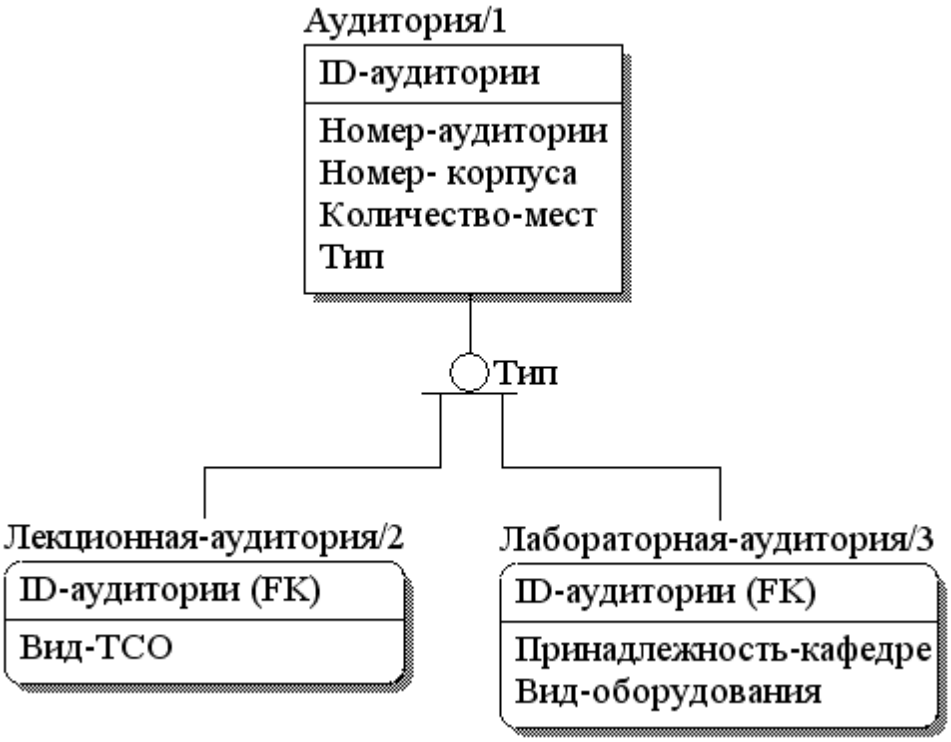


Рис. 5.44. Связь категоризации. Неполная группа категорий

Каждая связь групповой сущности с сущностью-категорией для полных групп категорий может быть прочитана как «*должен быть*». Например, Аудитория должна быть Лекционной-аудиторией, Лабораторной-аудиторией или Практической-аудиторией (см. рис. 5.43).

В обратном направлении связь читается как «*есть*». Например, Лекционная-аудитория есть Аудитория.

Групповая сущность и каждая сущность-категория должны иметь одинаковые идентификаторы (см. рис. 5.43). Идентификатор сущности-категории называется ***унаследованным идентификатором***. Он может быть передан из групповой сущности (см. рис. 5.43) или ее предка (рис. 5.45) во вложенных связях категоризации.

В примере, приведенном на рис. 5.45, введена групповая сущность Помещение. Данная сущность связана связями категоризации с сущностями-категориями Служебное-помещение и Учебная-аудитория. Последние образуют неполную группу категорий, дискриминатором в которой является атрибут Назначение сущности Помещение. В названные сущности-категории передан унаследованный из групповой сущности идентификатор ID-помещения.

Сущность Учебная-аудитория в свою очередь является групповой сущностью для полной группы категорий, в состав которой входят сущности-категории Лекционная-аудитория, Лабораторная-аудитория, Практическая-аудитория. Дискриминатором в данной группе категорий является атрибут Тип сущности Учебная-аудитория.

Рис. 5.45. Вложенность связей категоризации.
Использование имени предка в качестве имени
унаследованного идентификатора

В данную группу сущностей-категорий передается идентификатор ID-помещения, унаследованный из сущности-предка Помещение.

Таким образом, групповые сущности, связанные с сущностями-категориями связями категоризации, образуют иерархии наследования.

В качестве имен унаследованных идентификаторов в сущностях-категориях возможно использование имен ролей. На рис. 5.46 в сущности-категории Учебная-аудитория используется идентификатор

ID-аудитории.ID-помещения,

состоящий из имени роли и базового имени унаследованного идентификатора.

Сущности-категории Лекционная-аудитория и Практическая-аудитория в качестве имени своих идентификаторов унаследовали имя роли ID-аудитории своей групповой сущности Учебная-аудитория.

В сущности-категории Лабораторная-аудитория в качестве унаследованного идентификатора используется имя роли ID-лаборатории. При этом в качестве базового имени идентификатора используется имя групповой сущности (см. рис. 5.45) или имя ее роли, если оно есть (см. рис. 5.46).

Резюме

При использовании связи категоризации один реальный или виртуальный объект предметной области представляется комбинацией экземпляра групповой сущности и экземпляра сущности-категории. В групповую сущность включаются атрибуты, совместно принадлежащие сущностям-категориям. В группу категорий входит набор из одной или более связей категоризации и связанных с ними сущностей-категорий. Возможна организация полных и неполных групп категорий. Значение дискриминатора определяет категорию экземпляра групповой сущности. В качестве имен унаследованных идентификаторов в сущностях-категориях могут использоваться имена ролей.

Рис. 5.46. Вложенность связей категоризации.
Использование имени роли в качестве имени
унаследованного идентификатора

5.4.14. Рабочие продукты информационного моделирования

Для информационной модели разрабатываются *три рабочих продукта*.

1. *Диаграмма информационной структуры* – графическое представление информационной модели.

В IDEF1X-моделировании существуют три концептуальных уровня представления диаграмм [2]:

- диаграммы «Сущность – Связь», называемые *ER-диаграммами* (Entity–Relationship); на данных диаграммах отображаются сущности и связи между ними, а атрибуты не отображаются; представляют информационную модель верхнего уровня; используются в начале работы над моделью данных;
- диаграммы, основанные на ключах, называемые *KB-диаграммами* (Key–Based); на данных диаграммах отображаются сущности с первичными ключами и связи между сущностями; являются вторым уровнем детализации информационной модели;
- полные диаграммы с атрибутами, называемые *FA-диаграммами* (Fully–Attributed); на данных диаграммах отображаются сущности с первичными и вторичными ключами, а также связи между сущностями; являются наиболее детальным отображением модели данных; представляют данные в третьей нормальной форме.

Все рассмотренные ранее в подразд. 5.4 примеры были даны в виде FA-диаграмм. Например, на рис. 5.45 содержится информационная модель, представленная в виде диаграммы FA-уровня. Та же модель, представленная ER-диаграммой, приведена на рис. 5.47. На рис. 5.48 представлен уровень KB-диаграммы для той же модели.

2. *Описание сущностей и атрибутов* – списки всех сущностей модели, всех атрибутов (вместе с их доменами) и их описание.

3. *Описание связей* – перечни связей вместе с их описаниями.

Резюме

Для информационной модели предметной области разрабатываются диаграмма информационной структуры, описание сущностей и атрибутов и описание связей. В IDEF1X-моделировании существует три концептуальных уровня представления диаграмм: ER-диаграммы, KB-диаграммы, FA-диаграммы.

Рис. 5.47. Уровень ER-диаграммы

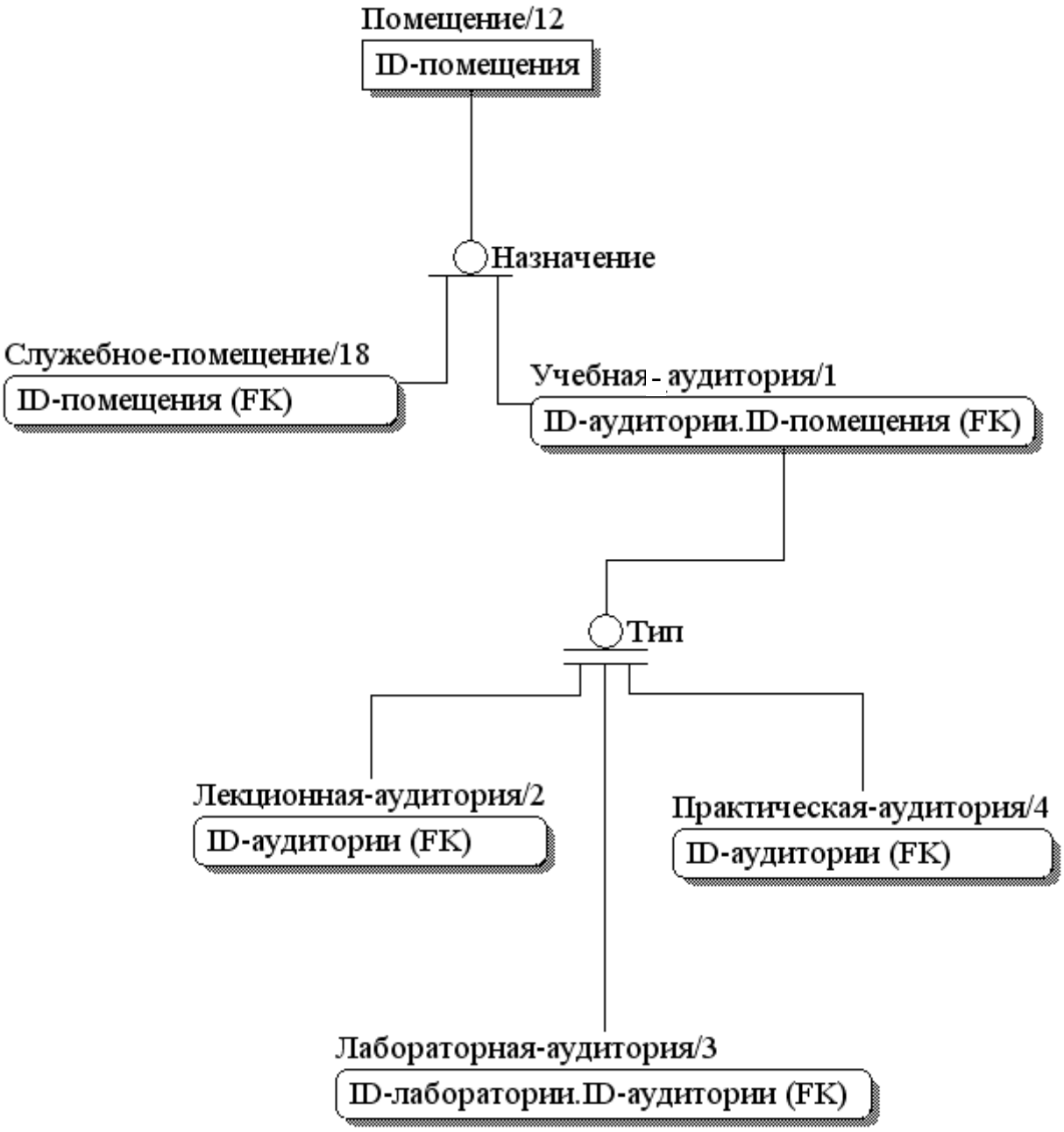


Рис. 5.48. Уровень КВ-диаграммы

5.5. Методологии, ориентированные на данные

В методологиях, ориентированных на данные, структура программного средства определяется, исходя из структур входных и выходных данных. Основными методологиями в данной группе являются методологии, базирующиеся на применении метода JSD Джексона и диаграмм Варнье–Орра.

5.5.1. Метод JSD Джексона

В подразд. 4.6 рассмотрен классический метод разработки ПС, ориентированный на данные – метод структурного программирования JSP, разработанный М. Джексоном. В настоящее время некоторыми CASE-средствами поддерживается развитие данного метода – метод JSD (Jackson System Development), также разработанный Майклом А. Джексоном [38]. Метод JSD зачастую называют первым объектно-ориентированным методом проектирования компьютерных систем [39].

Метод JSD предназначен для анализа требований и проектирования систем, в которых важное значение имеет фактор времени. Такие системы могут быть описаны с использованием последовательности событий. Метод JSD базируется на положении о том, что проектирование систем является расширением проектирования ПС.

Существует два основных различия между методом JSD и другими методами разработки систем. Первое заключается в том, что начальной областью интересов метода JSD является область ПО. Вторая особенность – метод JSD базируется на последовательности событий.

В основе применения метода JSD лежат *три принципа*:

- разработка системы должны начинаться с описания и моделирования предметной области, и только затем должно выполняться определение и структурирование функций, выполняемых системой;
- адекватная модель ориентированной на время предметной области сама должна быть время – ориентированной;
- реализация системы должна быть основана на преобразовании спецификации в эффективный набор процессов.

Метод JSD состоит из *шести шагов*, объединенных в *три стадии*:

- 1-я стадия – стадия анализа, называемая стадией моделирования (Modelling Stage); состоит из двух шагов:
  - сущность/действие (Entity/Action Step);
  - структуры сущностей (Entity Structures Step);
- 2-я стадия – стадия проектирования, называемая стадией сети (Network Stage); состоит из трех шагов:
  - начальная модель (Initial Model Step);
  - функции (Function Step);
  - согласование системы по времени (System Timing Step);
- 3-я стадия – стадия реализации (Implementation Stage); состоит из одного шага:
  - реализация (Implementation Step).

В методе JSD используются *три вида диаграмм*:

- диаграммы структур сущностей (Entity Structure Diagram, ED); предназначены для описания действий, выполняемых системой в хронологическом порядке;

• диаграммы описания системы (System Specification Diagram, SSD), называемые также сетевыми диаграммами (Network Diagram, ND);предназначены для описания взаимодействий между процессами системы;

• диаграммы реализации системы (System Implementation Diagram, SID).

На *стадии моделирования* выделяются сущности системы, выполняемые ими действия, устанавливается последовательность действий в жизненном цикле сущностей, определяются атрибуты сущностей и действий.

На данной стадии создается совокупность ESD-диаграмм. Целью данных диаграмм является описание всех аспектов моделируемой системы. При разработке ESD-диаграмм используется нотация структурных диаграмм, применяемых в методе JSP Джексона. Данная нотация подробно рассмотрена в подразд. 4.6. Напомним, что основными конструкциями данной нотации являются конструкции последовательности, выбора и повторения.

Сущность представляет собой некоторый объект моделируемой системы. Сущности характеризуются следующими особенностями:

• сущности выполняют действия или подвергаются воздействиям во времени;

• сущности должны существовать в реальной предметной области;

• сущности должны рассматриваться как отдельные объекты моделируемой системы.

С сущностями связана совокупность выполняемых ими действий. Действия характеризуются следующими особенностями:

• действия имеют место в определенный момент времени; процесс не является действием, так как он выполняются в течение определенного периода времени;

• действия должны иметь место в реальной предметной области;

• действия не могут декомпозироваться на поддействия.

Стадия моделирования реализуется двумя шагами.

На шаге сущность/действие создается список сущностей и действий предметной области. Каждому действию ставится в соответствие сущность и набор атрибутов.

На шаге структуры сущностей действия располагаются в порядке их выполнения. На базе полученной информации создаются ESD-диаграммы моделируемой системы. Пример общего вида ESD-диаграммы приведен на рис. 5.49.

ESD-диаграмма представляет собой иерархию действий, выполняемых некоторой сущностью во времени. Сущности и действия на ESD-диаграмме представляются в виде прямоугольников. На каждой ESD-диаграмме имеется только одна сущность. Она помещается в корне дерева ESD-диаграммы [38].

Сущности связаны с действиями, действия связаны между собой с помощью конструкций последовательности (связи сущности с действиями 1, 2, 3, 4 на рис. 5.49), выбора (связи действия 2 с выбираемыми действиями 5, 6, 7) или повторения (связи действия 4 с действием 8 и действия 6 с действием 9).

Если в конструкции выбора при некотором условии выполнять действие не нужно, используется пустой компонент (Null), обозначаемый прямоугольником с горизонтальной чертой (см. рис. 5.49).

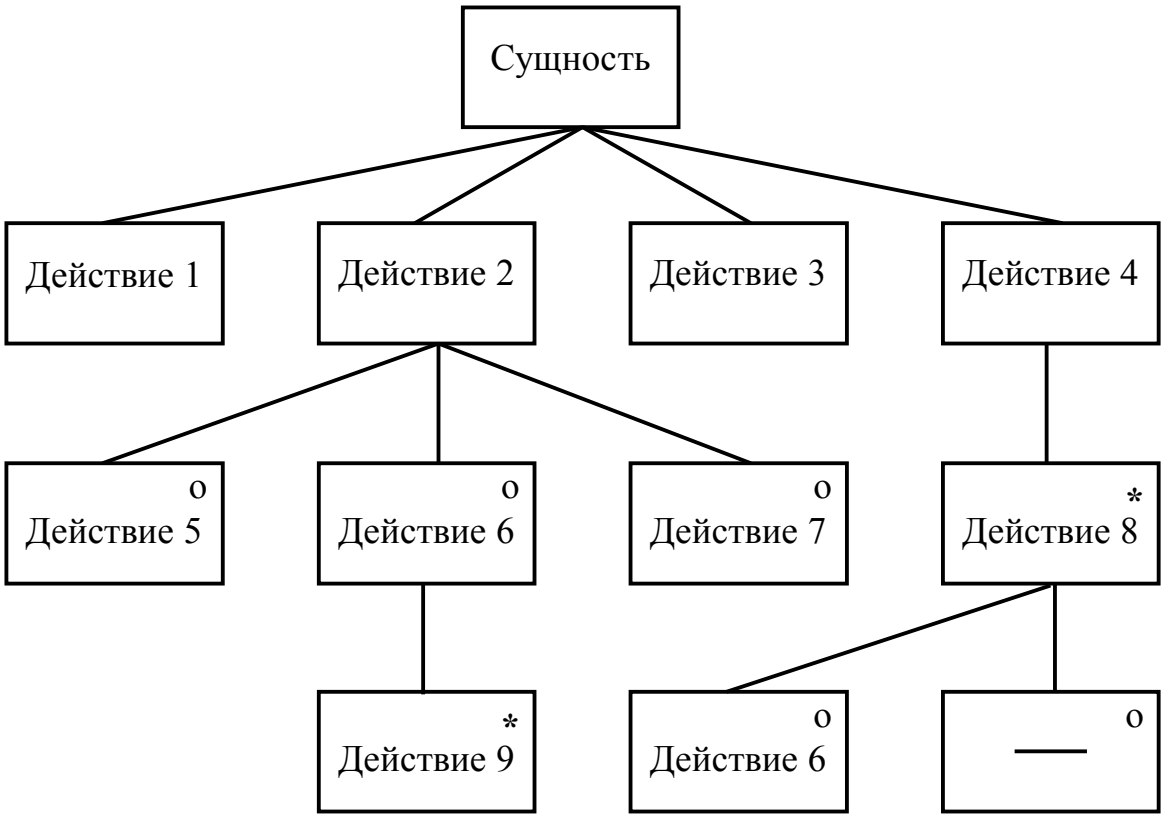


Рис. 5.49. Общий вид ESD-диаграммы

На *стадии сети* разрабатывается модель системы, представляемая в виде SSD-диаграммы (ND-диаграммы). ND-диаграммы отражают процессы моделируемой системы и их связи друг с другом. На данных диаграммах процессы изображаются прямоугольниками, связи между ними определяются посредством векторов состояний (State Vector, **SV**) или потоков данных (DataStream, **D**) [38]. Пример ND-диаграммы приведен на рис. 5.50.

Потоки данных определяют информацию, которой процессы обмениваются. Поток данных изображается кругом на связи между процессами (на рис. 5.50 процессы 4 и 5 связаны потоком данных D1).

Векторы состояния представляют собой альтернативные пути соединения процессов. Они определяют характеристики или состояние сущностей, используемых процессами, то есть условия перехода от одного процесса к другому. Вектор состояния изображается ромбом на связи между процессами (на рис. 5.50 представлены альтернативные связи между процессом 1 и процессами 2 и 4, а также между процессом 2 и процессами 3 и 5).

На этой стадии определяется функциональность системы. Каждая сущность ESD-диаграмм становится программой на сетевой ND-диаграмме. Ре-

зультатом стадии является представление моделируемой системы в виде набора сетевых диаграмм. Стадия завершается описанием данных и связей между процессами и программами.

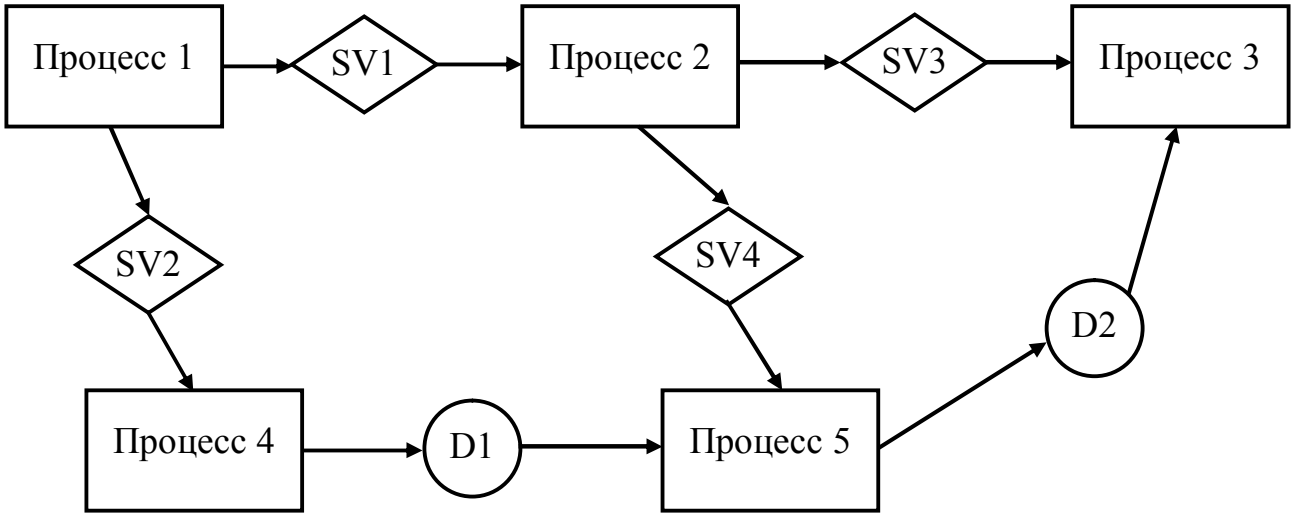


Рис. 5.50. Общий вид ND-диаграммы

Стадия сети реализуется тремя шагами.

На шаге начальной модели определяется укрупненная модель предметной области. На шаге функций добавляются модели выполняемых операций и процессы, необходимые для генерирования выхода системы. На шаге согласования системы по времени выполняется синхронизация процессов, определяются необходимые ограничения.

На *стадии реализации* сетевая модель преобразуется в физическую модель, представленную в виде SID-диаграммы. Данная диаграмма представляет систему в виде диспетчера процессов, который вызывает модули, реализующие процессы. Потоки данных на SID-диаграмме изображаются в виде вызовов замкнутых процессов.

Основным назначением шага реализации является оптимизация системы с целью сокращения количества процессов. С этой целью выполняется объединение некоторых из них.

В настоящее время за рубежом существует ряд CASE-средств, поддерживающих метод JSD. К ним относится, например, SmartDraw – CASE-средство, предназначенное для моделирования предметных областей и представления моделей с помощью различных диаграмм [43]. SmartDraw поддерживает большое количество методологий анализа и проектирования, в том числе и методологию, базирующуюся на применении метода JSD.

Следует отметить, что среди методологий структурного анализа и проектирования ПС методология Джексона широкого применения не получила.

Резюме

Метод разработки систем JSD Джексона является развитием его метода структурного программирования JSP. Метод JSD состоит из шести шагов, объединенных в три стадии: стадию анализа, стадию проектирования и стадию реализации. В методе JSD используются диаграммы структур сущностей ESD, диаграммы описания системы SSD и диаграммы реализации системы SID.

5.5.2. Диаграммы Варнье–Орра

Методология, основанная на применении диаграмм Варнье–Орра, была разработана одновременно с методологией Джексона. Данные методологии имеют ряд общих черт [22, 26].

Основным принципом методологии Варнье–Орра, как и методологии Джексона, является зависимость структуры проектируемой программы от структур данных. Структуры данных и структура программы могут быть представлены единым набором основных конструкций.

Однако если в методологии Джексона структура программы определяется слиянием структур входных и выходных данных, то в методологии Варнье–Орра структура программы зависит только от структуры выходных данных. Считается, что структуру входных данных в общем случае можно привести в соответствие структуре выходных данных и структуре программы.

Вторым отличием методологии Варнье–Орра является то, что декомпозиция структуры данных на диаграммах выполняется не сверху вниз, как в методологии Джексона, а слева направо.

В диаграммах Варнье–Орра используются *четыре базовые конструкции*: иерархия, последовательность, выбор, повторение. Три последних конструкции соответствуют по смыслу аналогичным конструкциям методологии Джексона. Представление конструкций Варнье–Орра в графических нотациях, применяемых в различных CASE-средствах, может отличаться формой скобки (например фигурная или квадратная) и использованием различных специальных символов.

1. Конструкция иерархии данных

Конструкция иерархии данных отражает вложенность некоторых конструкций данных в другие компоненты данных. Графически данные объединяются в конструкцию иерархии с помощью скобки. Вложенность скобок определяет уровень иерархии соответствующих конструкций данных. Пример представления конструкции иерархии данных «Отчет» приведен на рис. 5.51.

На данном рисунке на втором уровне иерархии структуры данных «Отчет» находится конструкция иерархии данных, состоящая из компонентов 1, 2, 3. Компонент 1 представляет собой конструкцию данных, включающую компоненты 4, 5, компонент 3 – конструкцию, содержащую компоненты 6, 7. Компоненты 4 – 7 находятся на третьем уровне иерархии. В состав компонента 5 входят компоненты 8, 9 четвертого уровня иерархии.

Данные в каждой из конкретных конструкций иерархии могут быть представлены конструкциями последовательности, выбора или повторения.

2. Конструкция последовательности данных

Эта конструкция возникает, когда два или более компонента данных помещаются вместе, строго последовательным образом, и образуют единый компонент данных. Графически последовательные компоненты данных в конструкции последовательности изображаются сверху вниз. На рис. 5.52 приведена структура конструкции данных «Дата». Данная конструкция представляет собой последовательность данных «Число N», «Месяц M», «Год Y».

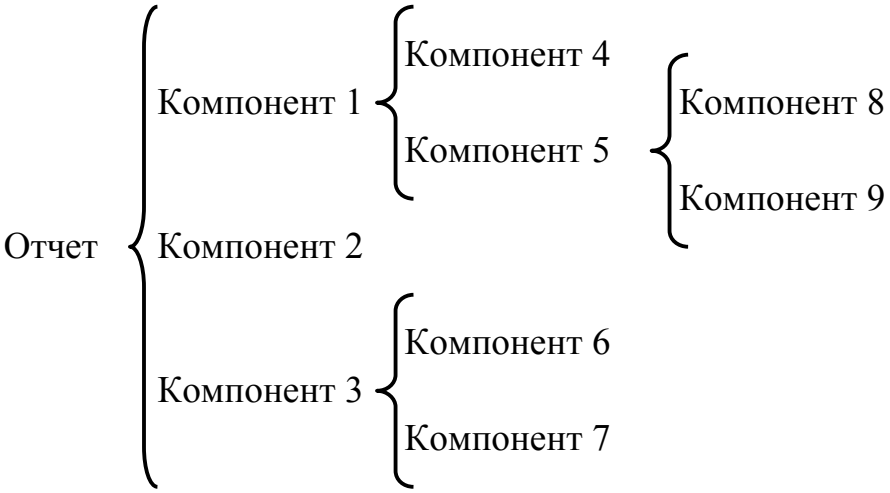


Рис. 5.51. Конструкция иерархии данных

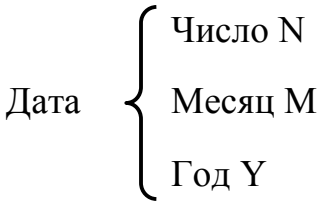


Рис. 5.52. Конструкция последовательности данных

Все конструкции, входящие в состав иерархической конструкции «Отчет» (см. рис. 5.51), представляют собой конструкции последовательности данных.

3. Конструкция выбора данных

Конструкция выбора данных – это конструкция сведения результирующего компонента данных к одному из двух или более выбираемых подкомпонентов. В зависимости от конкретной графической нотации между выбираемыми подкомпонентами помещается один из следующих символов: **@**, **OR**, ⊕.

Пример конструкции выбора данных приведен на рис. 5.53. На данном рисунке конструкция «Сезон» представляет собой конструкцию выбора из аль-

тернативных подкомпонентов «Зима W», «Весна P», «Лето S», «Осень A» (сезон представляет собой зиму, весну, лето или осень).

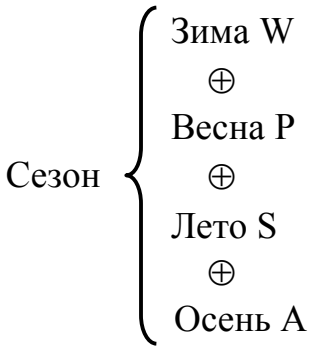


Рис. 5.53. Конструкция выбора данных

4. Конструкция повторения данных

Данная конструкция применяется тогда, когда конкретный подкомпонент данных может повторяться некоторое число раз. У конструкции повторения только один подкомпонент. Рядом с данным подкомпонентом в скобках отмечается нижняя и верхняя границы количества его повторений или количество повторений, если оно постоянно.

Представление повторения данных на диаграммах Варнье–Орра иллюстрирует рис. 5.54. На данном рисунке компонент «Файл» состоит из повторяющихся подкомпонентов «Запись», подкомпонент «Запись» может повторяться от одного до **n** раз. Компонент «Неделя» состоит из повторяющихся семь раз подкомпонентов «День».

а б

Рис. 5.54. Конструкция повторения данных:
а – повторение от одного до n раз; б – повторение ровно семь раз

Рис. 5.55 иллюстрирует представление иерархической структуры данных, приведенной на рис. 4.37 в нотации Джексона, в виде диаграммы Варнье–Орра.

Расширением диаграмм Варнье–Орра является применение *двух дополнительных конструкций* – конструкции параллелизма и конструкции рекурсии.

5. Конструкция параллелизма

Данная конструкция используется, если подкомпоненты некоторого компонента могут выполняться в любом порядке, в том числе и параллельно. При графическом представлении в данной конструкции между подкомпонентами

помещается символ +. На рис. 5.56 приведена конструкция параллелизма «Выполнение задания». Подкомпонентами данной конструкции являются «Выполнение задания 1», «Выполнение задания 2», «Выполнение задания 3». Из рис. 5.56 следует, что задания 1, 2 и 3 могут быть выполнены в любом порядке.

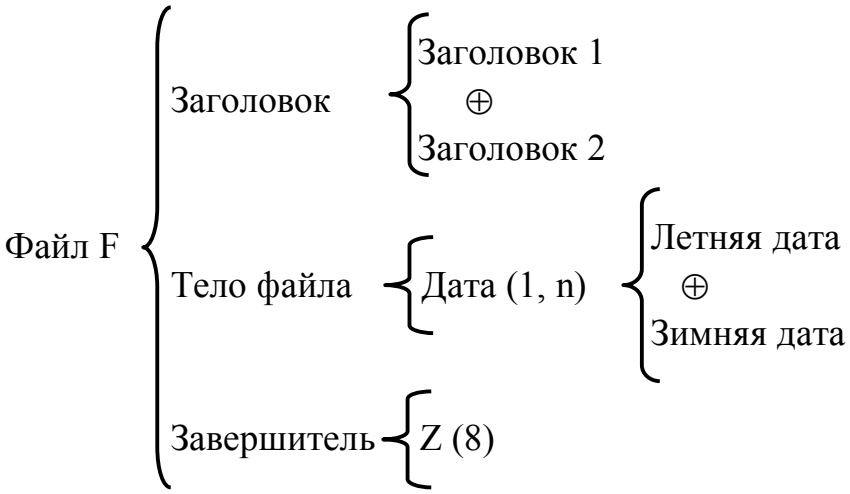


Рис. 5.55. Пример иерархической структуры данных

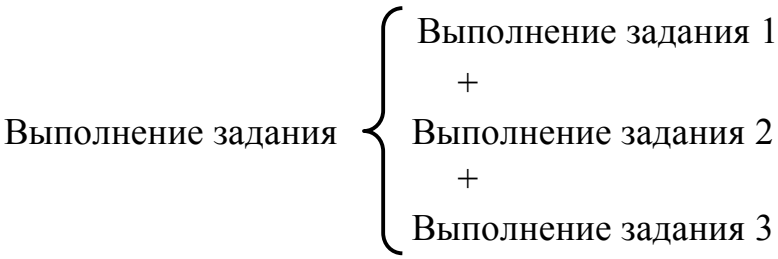


Рис. 5.56. Конструкция параллелизма

6. Конструкция рекурсии

Конструкция рекурсии используется, если в состав некоторого компонента в качестве подкомпонента входит сам компонент. Графически рекурсия обозначается двойной скобкой. Пример конструкции рекурсии приведен на рис. 5.57. На данном рисунке компонент «Агрегат» состоит из **n** узлов. Каждый узел в свою очередь может содержать от 0 до **к** узлов.

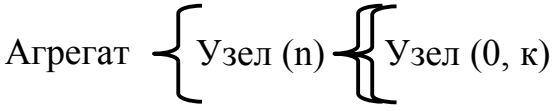


Рис. 5.57. Конструкция рекурсии

Построение диаграмм Варнье – Орра поддерживается некоторыми из современных CASE-средств, в частности, CASE-средством SmartDraw [43].

Однако в целом методологии проектирования ПС, основанные на применении диаграмм Варнье–Орра, широкого применения не получили.

В настоящее время техника диаграмм Варнье–Орра используется для структуризации представления сложной информации в различных предметных областях.

Резюме

В методологии Варнье–Орра структура программы зависит только от структуры выходных данных. Декомпозиция структуры данных выполняется слева направо. В диаграммах Варнье–Орра используются четыре базовые конструкции: иерархия, последовательность, выбор, повторение. Дополнительными конструкциями являются конструкции параллелизма и рекурсии.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Перечислите наиболее известные методы структурного анализа и проектирования.
2. Что означает термин CASE?
3. Определите понятие CASE-технологии.
4. Перечислите методологии семейства IDEF, назовите их назначение.
5. Назовите документы, регламентирующие синтаксис методологий IDEF0 и IDEFIX.
6. Определите назначение методологии SADT.
7. Перечислите достоинства методологии SADT.
8. Определите назначение методологии IDEF0.
9. Дайте формальное определение IDEF0-модели.
10. Определите понятия субъекта и объекта моделирования, границ, цели и точки зрения модели в IDEF0.
11. Объясните правила изображения функциональных блоков в IDEF0.
12. Назовите типы дуг в IDEF0-моделировании, поясните их назначение и правила изображения.
13. Что такое доминирование в IDEF0-моделировании?
14. Сколько блоков должна содержать диаграмма IDEF0?
15. Назовите типы взаимосвязей между блоками в методологии IDEF0.
16. Назовите правила изображения дуг в методологии IDEF0.
17. Что называется меткой в методологии IDEF0?
18. Определите назначение словаря данных в методологии IDEF0.
19. Опишите назначение полей в стандартном IDEF0-бланке.
20. Поясните назначение и правила образования C-номеров в методологии IDEF0.
21. Что называется декомпозицией в методологии IDEF0?
22. Что такое родительский блок и родительская диаграмма в методологии IDEF0?
23. Что называется контекстной диаграммой IDEF0-модели?
24. Опишите, чем отличается контекстная диаграмма от других диаграмм IDEF0-модели по назначению и синтаксису.
25. Поясните назначение и правила образования номера узла в методологии IDEF0?
26. Назовите способы организации связей между диаграммами, используемые в методологии IDEF0?
27. Какие дуги называются внешними и граничными в методологии IDEF0?
28. Что обозначает схема кодирования ICOM в методологии IDEF0?
29. Поясните правила стыковки внешних дуг с граничными дугами.
30. Что называется «вхождением дуги в тоннель» в IDEF0?
31. Поясните правила изображения тоннельных дуг на IDEF0-диаграммах.

32. Нарисуйте IDEF0-модель для предметной области «Процесс разработки программных средств» (модель должна содержать не менее двух уровней иерархии).
33. Какая диаграмма называется диаграммой дерева узлов?
34. Поясните смысл метода декомпозиции ограниченного субъекта, используемого при IDEF0-моделировании.
35. Перечислите стратегии декомпозиции, используемые при IDEF0-моделировании.
36. Опишите на уровне IDEF0-диаграммы основные этапы процесса моделирования в IDEF0.
37. Какие роли выделяются для участников проектов при IDEF0-моделировании?
38. Какие вы знаете CASE-средства, поддерживающие IDEF0-моделирование?
39. Определите назначение методологии DFD.
40. Определите основные понятия DFD-модели.
41. Что отражает DFD-диаграмма?
42. Какие компоненты может содержать DFD-диаграмма?
43. Опишите назначение и правила изображения различных видов блоков на DFD-диаграммах.
44. Назовите типы дуг в DFD-моделировании, поясните их назначение и правила изображения.
45. Нарисуйте DFD-модель для предметной области «Процесс разработки программных средств» (модель должна содержать не менее двух уровней иерархии).
46. Назовите различия между контекстными диаграммами при IDEF0- и DFD-моделировании?
47. Какие вы знаете CASE-средства, поддерживающие DFD-моделирование?
48. Определите назначение методологии IDEF1X.
49. Перечислите компоненты IDEF1X-моделей.
50. Перечислите виды связей в IDEF1X-моделях.
51. Что называется сущностью в методологии IDEF1X?
52. Какие сущности называются независимыми и зависимыми?
53. Назовите основные категории сущностей.
54. Что называется атрибутом в методологии IDEF1X?
55. Что называется доменом в методологии IDEF1X?
56. Как классифицируются атрибуты в методологии IDEF1X?
57. Что называется первичным ключом в методологии IDEF1X?
58. Что называется привилегированным идентификатором в методологии IDEF1X?
59. Что называется идентификационным номером экземпляра сущности в методологии IDEF1X?
60. Определите нормальные формы, обычно используемые при нормализации информационных моделей.

61. Назовите правила атрибутов, которые должны соблюдаться в информационной модели.
62. Перечислите и поясните способы представления сущностей с атрибутами при информационном моделировании.
63. Что называется связью в методологии IDEF1X?
64. Какая связь соединяет родительскую и дочернюю сущности?
65. Что называется родительской и дочерней сущностями?
66. К какой сущности направлена точка на конце соединительной связи?
67. Какая связь называется реверсной?
68. Какие связи называются условными и безусловными?
69. Назовите фундаментальные виды безусловных связей.
70. Чем определяется мощность связи?
71. Перечислите десять форм связей между сущностями.
72. Что называется дочерней мощностью связи?
73. Чему равно значение дочерней мощности связи по умолчанию?
74. Какие виды мощности соединительных связей определены в методологии IDEF1X?
75. Как графически представляются различные виды дочерней мощности соединительных связей в IDEF1X?
76. Приведите примеры различных видов мощности соединительных связей между сущностями.
77. Поясните правила именования сущностей, атрибутов и связей в методологии IDEF1X.
78. Необходимо ли обеспечить в пределах IDEF1X-модели уникальность имен для сущностей? Для атрибутов? Для связей?
79. Как описывается связь между родительской и дочерней сущностями в прямом и обратном направлениях?
80. Какая связь называется связью, формализованной в данных?
81. Какая связь называется идентифицирующей связью и как она графически изображается?
82. Какая связь называется неидентифицирующей связью и как она графически изображается?
83. Приведите примеры идентифицирующих и неидентифицирующих связей между сущностями.
84. Каким связям (идентифицирующим или неидентифицирующим) следует отдавать предпочтение при разработке информационных моделей предметной области и почему?
85. Какие связи в IDEF1X определяют безусловность и условность связи со стороны родительской сущности и почему?
86. Приведите примеры безусловных и условных связей со стороны родительской сущности.
87. Какие связи в IDEF1X определяют безусловность и условность связи со стороны дочерней сущности и почему?

88. Приведите примеры безусловных и условных связей со стороны дочерней сущности.
89. Какие связи называются обязательными и необязательными и как они изображаются графически?
90. Приведите примеры обязательных и необязательных связей между сущностями.
91. Что называется родительской мощностью связи?
92. Какая связь называется неспецифической?
93. Приведите примеры неспецифических связей между сущностями .
94. Как формализуются неспецифические связи?
95. Какие связи называются рекурсивными?
96. Приведите примеры рекурсивных связей.
97. С помощью каких видов связей организуется иерархическая рекурсия?
98. С помощью каких видов связей можно организовать сетевую рекурсию?
99. Что называется именем роли и в каких случаях оно используется?
100. Какие связи называются связями категоризации и как они графически представляются?
101. Приведите примеры связей категоризации между сущностями.
102. Какая сущность называется сущностью-супертипом и сущностью-подтипом?
103. Что обозначает термин «группа категорий»?
104. Что называется полной и неполной группой категорий?
105. Что называется дискриминатором?
106. Что называется унаследованным идентификатором?
107. Перечислите рабочие продукты информационного моделирования.
108. Назовите концептуальные уровни представления диаграмм в IDEF1X-моделировании.
109. Приведите пример представления диаграммы на различных концептуальных уровнях.
110. Для проектирования каких систем предназначен метод JSD, разработанный М. Джексоном?
111. Назовите принципы, являющиеся основой применения метода JSD.
112. Назовите стадии и шаги, из которых состоит метод JSD, и поясните их содержание.
113. Поясните назначение и перечислите компоненты диаграмм, используемых в методе JSD.
114. Перечислите особенности сущностей, выделяемых в методе JSD, и выполняемых ими действий.
115. Какой принцип положен в основу методологии Варнье–Орра?
116. Перечислите различия методологий Варнье–Орра и Джексона.
117. Назовите базовые конструкции диаграмм Варнье–Орра.
118. Приведите пример структуры данных, представленной с помощью диаграмм Варнье–Орра.

РАЗДЕЛ 6. МЕТОДОЛОГИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО АНАЛИЗА И ПРОЕКТИРОВАНИЯ СЛОЖНЫХ СИСТЕМ

6.1. Основы объектно-ориентированного анализа и проектирования

6.1.1. Математические основы объектно-ориентированного анализа и проектирования

В основе многих концепций моделирования сложных систем лежат понятия теории множеств, теории графов и семантических сетей [27]. Кратко рассмотрим основные из данных понятий.

Под ***множеством*** **A** понимается некоторая совокупность различимых объектов $\mathbf{a_i}$. Данные объекты называются ***элементами*** множества. Конечное множество обозначается следующим образом:

$$\mathbf{A = \{a_1, a_2, a_3, \ldots, a_n\}},$$

где **n** – количество элементов, входящих в множество, называемое *мощностью* множества.

Подмножеством является любая часть совокупности объектов, входящих в множество.

Отношение множеств (связь, соотношение множеств) обозначает любое подмножество кортежей, построенных из элементов исходных множеств. Под ***кортежем*** понимается набор упорядоченных элементов исходных множеств. Например, кортеж из трех элементов может обозначаться следующим образом:

$$\mathbf{<a_1, a_2, a_3>}.$$

Упорядоченность набора входящих в кортеж элементов обозначает, что каждый элемент кортежа имеет строго фиксированное место (например, первый элемент всегда идет первым, второй – вторым и т.д.). Отдельные элементы

кортежа могут принадлежать как одному, так и нескольким множествам.

Последовательность элементов в кортежах фиксирована и определяется конкретной задачей. Отношение множеств характеризует способ выбора элементов из одного или нескольких множеств для создания упорядоченных кортежей.

Граф можно интерпретировать как графическое представление бинарного отношения двух множеств. Бинарное отношение состоит из кортежей, содержащих два элемента некоторого множества.

Неориентированный граф G задается двумя множествами – множеством вершин **V** и множеством ребер **E**:

$$\mathbf{G} = (\mathbf{V}, \mathbf{E});$$
$$\mathbf{V} = \{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\};$$
$$\mathbf{E} = \{\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_m}\},$$

где **n** – количество вершин графа; **m** – количество ребер графа.

Неориентированному графу ставится в соответствие бинарное отношение $\mathbf{P_G}$, состоящее из таких кортежей $<\mathbf{v_i}, \mathbf{v_j}>$, для которых вершины $\mathbf{v_i}$ и $\mathbf{v_j}$ соединяются в графе **G** некоторым ребром $\mathbf{e_к}$.

Маршрутом в неориентированном графе называется упорядоченная последовательность ребер, в которой два соседних ребра имеют общую вершину.

Ориентированный граф задается двумя множествами – множеством вершин **V** и множеством дуг **E**. Каждая дуга $\mathbf{e_к}$ обозначается стрелкой и имеет свое начало в некоторой вершине v_i и конец в вершине $\mathbf{v_j}$. Ориентированному графу ставится в соответствие бинарное отношение $\mathbf{P_G}$, состоящее из таких кортежей $<\mathbf{v_i}, \mathbf{v_j}>$, для которых вершины $\mathbf{v_i}$ и $\mathbf{v_j}$ соединяются в графе **G** некоторой дугой $\mathbf{e_к}$ с началом в вершине $\mathbf{v_i}$ и концом в вершине $\mathbf{v_j}$.

Ориентированным маршрутом называется упорядоченная последовательность дуг, в которой две соседние дуги имеют общую вершину, являющуюся концом предыдущей и началом следующей дуги.

Частным случаем графа является дерево – граф, между любыми двумя вершинами которого существует маршрут, с неповторяющими ребрами (или дугами). Дерево может быть ориентированным и неориентированным.

Семантическая сеть – это некоторый граф $\mathbf{G_s} = (\mathbf{V_s}, \mathbf{E_s})$, в котором множество вершин $\mathbf{V_s}$ и множество ребер $\mathbf{E_s}$ разделены на отдельные типы, характерные для конкретной предметной области. Вершины соответствуют сущностям предметной области и именуются соответствующими смысловыми именами. Типы ребер соответствуют видам связей между сущностями.

Для семантических сетей характерно наличие различных графических обозначений для представления отдельных типов вершин и ребер.

К различным видам семантических сетей относятся, например, структуры данных Джексона (см. рис. 4.37, 4.49, 4.50), информационная модель, представленная на рис. 5.47, и большинство других, рассмотренных в предыдущих разделах структур данных и моделей предметной области.

Резюме

В основе методологии объектно-ориентированного анализа и проектирования лежат понятия теории множеств, теории графов и семантических сетей.

6.1.2. Исторический обзор развития методологии объектно-ориентированного анализа и проектирования

В 80-х гг. XX ст. появилось большое количество методологий и графических нотаций структурного анализа и проектирования. К ним относятся, например, методы JSP и JSD Джексона (см. подразд. 4.6 и п. 5.5.1), методологии семейства IDEF (см. подразд. 5.1), методология структурного анализа потоков данных DFD (см. подразд. 5.3) и ряд других. Многие из этих методологий реализованы в CASE-средствах.

Методологии структурного анализа и проектирования базируются на функциональной декомпозиции структуры предметной области или проектируемой системы.

Одновременно с развитием методологий структурного анализа и проектирования начали появляться отдельные языки объектно-ориентированного анализа и проектирования. Данные яыки ориентированы на объектную декомпозицию структуры предметной области или проектируемой системы.

На дальнейшее развитие объектно-ориентированных языков моделирования оказали непосредственное влияние многие идеи структурного анализа и проектирования. В этой связи особо следует выделить методологии IDEF0, IDEF1, DFD, базовые принципы которых положены в основу ряда объектно-ориентированных диаграмм моделирования. Данные методологии рассмотрены подробно в подразд. 5.2 – 5.4.

К середине 90-х гг. XX ст. наибольшую известность из методов объектно-ориентированного анализа и проектирования (ООАП) приобрели методы Гради Буча, Джеймса Румбаха и Айвара Джекобсона, ориентированные на поддержку различных этапов ООАП. Данные методы послужили основой *Унифицированного языка моделирования UML (Unified Modeling Language)*.

Первое описание языка UML появилось в 1996 г. В 1998 г. компания Rational Software Corporation разработала одно из первых CASE-средств Rational Rose 98, в котором был реализован язык UML.

В настоящее время существует большое количество CASE-средств, базирующихся на применении языка UML. В данном пособии рассмотрена линейка CASE-средств компании Telelogic, ориентированных на UML (см. подразд. 7.5).

Разработаны среды визуального программирования на основе UML (Visual C++, Java, Delphi, Power Builder, Ada и др.).

Резюме

На развитие объектно-ориентированных языков моделирования оказали непосредственное влияние методологии структурного анализа и проектирования IDEF0, IDEF1, DFD. Их базовые принципы положены в основу объектно-ориентированных диаграмм моделирования. Основой Унифицированного языка моделирования UML являются методы Гради Буча, Джеймса Румбаха и Айвара Джекобсона, ориентированные на поддержку различных этапов объектно-ориентированного анализа и проектирования.

6.1.3. Основы языка UML

Язык UML – это язык визуального моделирования, позволяющий разрабатывать концептуальные, логические и физические модели сложных систем. Он предназначен для визуализации, анализа, спецификации, проектирования и документирования предметных областей, сложных систем вообще и ПС в частности.

Язык UML основан на следующих *принципах объектно-ориентированного анализа и проектирования* [27, 30]:

- *принцип абстрагирования* – предписывает включать в модель только те аспекты предметной области, которые имеют непосредственное отношение к выполнению проектируемой системой своих функций; абстрагирование сводится к формированию абстракций, определяющих основные характеристики внешнего представления объектов;
- *принцип инкапсуляции* – предписывает разделять элементы абстракции на секции с различной видимостью, что позволяет отделить интерфейс абстракции от его реализации; обычно скрываются структура объектов и реализация их методов;
- *принцип модульности* – определяет возможность декомпозиции проектируемой системы на совокупность сильно связанных и слабо сцепленных модулей (см. подразд. 4.7); определение модулей выполняется при физической разработке системы, определение классов и объектов – при логической разработке;
- *принцип иерархии* – означает формирование иерархической структуры абстракций; принцип предписывает выполнять иерархическое построение моделей сложных систем на различных уровнях детализации;
- *принцип многомодельности* – обозначает, что при моделировании предметной области необходимо разрабатывать различные модели проектируемой сложной системы, отражающие различные аспекты ее поведения или структуры.

Конкретным представлением абстракции является ***объект*** – элемент предметной области, существующий во времени и пространстве. Понятие объекта аналогично понятию экземпляра класса. Каждый объект характеризуется индивидуальностью, состоянием и поведением [30].

Индивидуальность – характеристики объекта, отличающие его от других объектов. *Состояние* – перечень свойств объекта, имеющих текущие значения. *Поведение* – воздействие объекта на другие объекты или его реакция на воздействия других объектов, выраженная в терминах изменения его состояний и передачи сообщений.

Между объектами существуют два основных вида отношений: связи и агрегация.

Связь – это равноправное (клиент-серверное) отношение между объектами, обозначающее физическое или логическое соединение между ними. С помощью связей перемещаются данные между объектами и вызываются операции объектов.

Агрегация – это иерархическое отношение объектов вида «целое – часть».

Класс – это описание множества объектов, имеющих общие свойства, операции, отношения и семантику. Каждый объект представляет собой экземпляр класса.

Между классами существует четыре основных вида отношений: ассоциация, зависимость, обобщение – специализация, целое – часть [30].

Отношение ассоциации определяет связи между экземплярами классов. Ассоциативное отношение характеризуется ***мощностью ассоциации***. Существует три типа мощности ассоциации:

- один-к-одному;
- один-ко-многим;
- многие-ко-многим.

Понятие мощности ассоциативной связи и типы мощности в объектно-ориентированном анализе и проектировании заимствованы из теории информационного моделирования и подробно рассмотрены в п. 5.4.7.

Отношение зависимости определяет влияние одного независимого класса на другой зависимый и обычно представляется в форме *отношения использования* (отношения между клиентом, запрашивающим услугу, и сервером, предоставляющим услугу).

Отношение обобщения-специализации подразделяется на две разновидности отношений – наследование и конкретизацию. *Наследование* – это отношение, при котором класс разделяет структуру и поведение, определенные в другом или других классах. *Конкретизация* – это отношение между родовым классом (шаблоном) и другими классами, наполняющими параметры шаблона.

Отношение целое-часть (отношение агрегации, реализации, включения) обеспечивается агрегацией между экземплярами классов.

Резюме

Язык UML основан на принципах абстрагирования, инкапсуляции, модульности, иерархии, многомодельности. Основными понятиями языка UML являются понятия объекта, связи, агрегации, класса. Между классами существует четыре основных вида отношений: ассоциация, зависимость, обобщение-специализация, целое-часть.

6.2. Диаграммы моделирования в языке UML

Для моделирования различных аспектов предметной области или проектируемой системы в языке UML предусмотрены следующие виды диаграмм [27, 40, 41]:

- диаграмма вариантов использования (Use Case Diagram);
- диаграмма классов (Class Diagram);
- диаграммы поведения (Behavior Diagram), в том числе:
  - диаграмма состояний (Statechart Diagram);
  - диаграмма деятельности (Activity Diagram);
  - диаграммы взаимодействия (Interaction Diagram), в том числе:
    - диаграмма последовательности (Sequence Diagram);
    - диаграмма кооперации (Collaboration Diagram);
- диаграммы реализации (Implementation Diagram), в том числе:
  - диаграмма компонентов (Component Diagram);
  - диаграмма развертывания (Deployment Diagram).

Модели языка UML подразделяются на *два вида*:

- *структурные модели (статические модели)* описывают структуру сущностей предметной области или компонентов моделируемой системы, включая их классы, атрибуты, связи, интерфейсы; к данному виду моделей относятся диаграммы вариантов использования, классов, компонентов, развертывания;
- *модели поведения (динамические модели)* описывают функционирование сущностей предметной области или компонентов системы во времени, включая их методы, взаимодействия между ними, изменение состояний отдельных сущностей, компонентов и системы в целом; к данному виду моделей относятся диаграмма состояний, диаграмма деятельности, диаграмма последовательности, диаграмма кооперации.

Все модели языка UML подразделяются на *три уровня*:

- *концептуальные модели* представляют собой верхний, наиболее общий и абстрактный уровень описания моделируемой системы; к данному уровню относится диаграмма вариантов использования; с уровня концептуальной модели должно начинаться моделирование предметной области или проектируемой системы (программного средства);
- *логические модели* представляют собой второй уровень описания моделируемой системы; элементы моделей данного уровня не имеют физического воплощения и отражают логические аспекты структуры и поведения реальной предметной области или системы; логические модели должны строиться после концептуальных моделей; к данному уровню относятся диаграммы классов, состояний, деятельности, последовательности, кооперации;
- *физические модели* представляют собой нижний уровень описания моделируемой системы; элементы моделей данного уровня представляют собой

конкретные материальные сущности физической системы; данные модели рекомендуется строить в последнюю очередь; к данному уровню относятся диаграммы компонентов и развертывания.

Процесс объектно-ориентированного анализа и проектирования, базирующийся на построении различных типов диаграмм UML, получил название *рационального унифицированного процесса RUP* (Rational Unified Process). Основы данного процесса разработаны одним из авторов языка UML Джекобсоном.

Диаграмма вариантов использования описывает функциональное назначение моделируемой предметной области или системы.

Диаграмма классов является основной для создания кода приложения. Она описывает внутреннюю структуру программного средства, наследование и взаимное положение классов.

Диаграмма состояний описывает возможные последовательности состояний и переходов, выполняемых в ответ на некоторые события. Данная диаграмма характеризует поведение элементов модели в течение их жизненного цикла.

Диаграмма деятельности моделирует алгоритмическую и логическую реализации выполнения операций в системе и является аналогом схем алгоритмов, предназначенным для использования в объектно-ориентированных приложениях.

Диаграмма последовательности отображает синхронные процессы, описывающие взаимодействие объектов модели во времени. Время в данной модели присутствует в явном виде.

Диаграмма кооперации описывает структурные связи между взаимодействующими объектами модели.

Диаграмма компонентов описывает физическое представление проектируемой системы и позволяет определить ее архитектуру в терминах модулей, исходных и исполняемых кодов, файлов.

Диаграмма развертывания (диаграмма размещения) отображает общую конфигурацию и топологию распределенной системы. Данная диаграмма представляет распределение программных компонентов по отдельным узлам системы и маршруты передачи информации между ними.

Диаграмма вариантов использования является первой из диаграмм, разрабатываемых при моделировании предметной области, системы или программного средства. Она является базой при разработке спецификации функциональных требований и имеет основополагающее значение с точки зрения полноты и корректности дальнейшего моделирования проектируемой системы (программного средства). С учетом этого в следующем подразделе детально рассмотрены правила построения диаграмм вариантов использования.

Резюме

В языке UML предусмотрены восемь видов диаграмм: диагаммы вариантов использования, классов, состояний, деятельности, последовательности,

кооперации, компонентов, развертывания. Все модели языка UML подразделяются на два вида: структурные модели и модели поведения. Модели языка UML подразделяются на три уровня: концептуальные модели, логические модели, физические модели.

6.3. Диаграмма вариантов использования

Диаграмма вариантов использования (Use Case Diagram) определяет функциональное назначение моделируемой системы или предметной области [27, 40, 41].

Данная диаграмма отображает множество актеров, взаимодействующих с проектируемой системой (программным средством) с помощью вариантов использования. Таким образом, основными элементами диаграммы вариантов использования являются актер и вариант использования.

Актер – это внешняя по отношению к моделируемой системе сущность, взаимодействующая с системой для решения некоторых задач. В качестве актера может использоваться человек, другая система, устройство или программное средство. Графическое изображение актера представлено на рис. 6.1, а. Имя актера основано на использовании существительного.

а б в

Рис. 6.1. Графическое представление основных элементов диаграммы вариантов использования:
а – актер; б – вариант использования; в – примечание

Вариант использования определяет некоторый набор действий (операций), которые должны быть выполнены моделируемой системой или программным средством при взаимодействии с актером. Графическое изображение варианта использования представлено на рис. 6.1, б. Название варианта использования базируется на неопределенной форме глагола.

В качестве примера рассмотрим предметную область автоматизации выполнения студентами лабораторных работ. Примеры построения функциональных моделей для данной предметной области с использованием методологий IDEF0 и DFD были рассмотрены в подразд. 5.2 и 5.3 (см. рис. 5.2 и 5.15).

Очевидно, что внешними сущностями, взаимодействующими с моделируемой системой, являются актеры «Преподаватель», «Студент» и «Лаборант».

На рис. 6.2 приведен пример диаграммы вариантов использования системы автоматизации выполнения лабораторных работ для актеров «Преподаватель» и «Лаборант». Рис. 6.3 содержит тот же пример для актера «Студент».

Из рис. 6.2 следует, что при взаимодействии с актером «Преподаватель» система должна обеспечить возможность выполнения следующих основных функций:

- выбор темы работы;
- выдача индивидуального задания;
- контроль хода выполнения индивидуального задания;
- проверка результатов выполнения индивидуального задания;
- проверка отчета;
- прием лабораторной работы.

Рис. 6.2. Диаграмма вариантов использования системы автоматизации выполнения лабораторных работ для актеров «Преподаватель» и «Лаборант»

Рис. 6.3. Диаграмма вариантов использования
системы автоматизации выполнения лабораторных работ
для актера «Студент»

При взаимодействии с актером «Лаборант» система должна обеспечить возможность выполнения следующих основных функций:

- включение системы;
- проверка системы;
- выключение системы.

При взаимодействии с актером «Студент» (см. рис. 6.3) система должна позволять выполнять следующие функции:

- изучение теории;
- ответы на контрольные вопросы по изученному материалу;
- выполнение индивидуального задания;
- оформление отчета по лабораторной работе;
- сдача лабораторной работы преподавателю.

Помимо актеров и вариантов использования диаграмма вариантов использования может содержать примечания – элементы, служащие для размещения на диаграмме поясняющей текстовой информации. Графическое изображение примечания представлено на рис. 6.1, в. Примечание может относиться к любому элементу диаграммы и соединяется с данным элементом штриховой линией (см. например рис. 6.2).

На диаграммах вариантов использования определены следующие *виды отношений* между отдельными элементами:

- отношение ассоциации;
- отношение включения;
- отношение расширения;
- отношение обобщения.

Графическое представление различных видов отношений приведено на рис. 6.4.

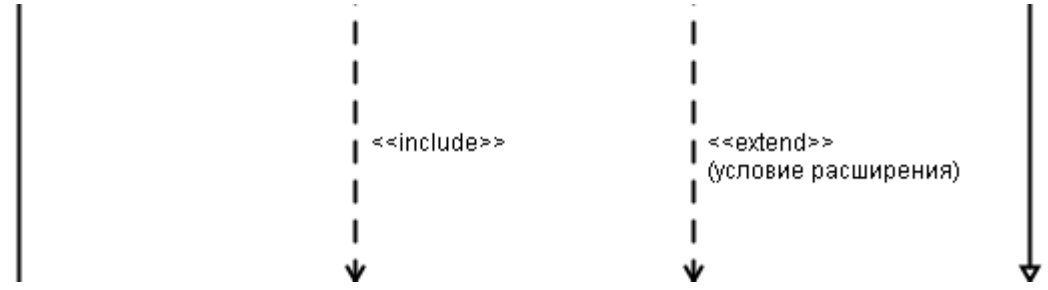


Рис. 6.4. Графическое представление отношений
на диаграмме вариантов использования:
а – отношение ассоциации; б – отношение включения;
в – отношение расширения; г – отношение обобщения

Отношение ассоциации (association relationship) определено для всех видов диаграмм языка UML. На диаграммах вариантов использования данное

отношение связывает актеров с вариантами использования и определяет конкретную роль актера при взаимодействии с вариантом использования. Графически отношение ассоциации изображается сплошной линией между актером и вариантом использования (см. рис. 6.4, а).

Например, на рис. 6.2 между актерами «Преподаватель» и «Лаборант» и соответствующими вариантами использования существуют отношения ассоциации.

Как отмечалось в п. 6.1.3, отношения ассоциации характеризуются мощностью ассоциации.

Мощность (кратность, multiplicity) ассоциации определяет количество экземпляров обеих сущностей, которое может участвовать в данной ассоциации. Графически значение мощности отмечается возле линии отношения ассоциации на стороне соответствующей сущности.

В диаграммах вариантов использования определено три типа мощности ассоциации: один-к-одному; один-ко-многим; многие-ко-многим. Возможна организация условной и биусловной ассоциации, при которой отдельные экземпляры одной или обеих сущностей могут не участвовать в ассоциации (см. п. 5.4.7). Например, могут быть организованы мощности ассоциации ноль-или-один-к-нулю-или-одному, ноль-или-один-к-нулю-или-многим; ноль-или-много-к-нулю-или-многим.

Множественность ассоциации обозначается * на стороне соответствующей сущности. Условность ассоциации обозначается введением нуля в обозначение мощности. Например, запись 0..1 обозначает мощность ноль-или-один, 0..* – мощность ноль-или-много.

Если мощность ассоциативной связи не указана, то по умолчанию она принимается равной один-к-одному.

Например, на рис. 6.2 между актером «Преподаватель» и вариантом использования «Выбрать тему работы» существует ассоциативная связь мощностью многие-ко-многим, поскольку один преподаватель может выбрать разные темы работы, при этом одну и ту же работу могут выбрать разные преподаватели. Между актером «Преподаватель» и вариантом использования «Проверить отчет» существует связь мощностью один-ко-многим, так как один преподаватель может принимать много отчетов, но конкретный отчет принимается только одним преподавателем. На рис. 6.3 между актером «Студент» и вариантом использования «Оформить отчет» существует связь один-к-одному, что отражает тот факт, что один студент оформляет один отчет по лабораторной работе и каждый отчет оформляется одним студентом.

Отношение включения (отношение целое-часть, include relationship) может иметь место между двумя вариантами использования. Данное отношение определяет, что последовательность действий одного варианта использования (включаемого) включается в качестве составной части в последовательность действий другого варианта использования (базового). Графически отношение включения изображается штриховой стрелкой между вариантами использова-

ния, помеченной служебным словом «include» (включает, см. рис. 6.4, б). Стрелка направлена от базового варианта использования к включаемому варианту (базовый вариант «включает» действия включаемого варианта).

Например, на рис. 6.3 базовый вариант использования «Оформить отчет» связан отношениями включения с вариантами использования «Сформулировать цель», «Написать условие», «Описать метод», «Описать реализацию» и «Привести результаты». По сути данные варианты использования определяют, что должно быть включено в содержание отчета по лабораторной работе.

Отношение расширения (extend relationship) может существовать между двумя вариантами использования. Данное отношение определяет, что некоторый вариант использования является расширением для другого варианта использования (базового). Это означает, что последовательность действий базового варианта использования может быть дополнена последовательностью действий варианта-расширения. Графически отношение расширения изображается штриховой стрелкой между вариантами использования, помеченной служебным словом «extend» (расширяет, см. рис. 6.4, в). Стрелка направлена от варианта использования, являющегося расширением, к базовому варианту (вариант-расширение «Расширяет» базовый вариант).

Например, на рис. 6.3 базовый вариант использования «Ответить на вопросы» связан отношением расширения с вариантом использования «Изучить теорию». Это означает, что при ответах студентом на контрольные вопросы может появиться необходимость в изучении теории. Таким образом, вариант использования «Изучить теорию», с одной стороны, связан отношением ассоциации с актером «Студент», то есть является независимым вариантом (изучение теории является этапом выполнения лабораторной работы). С другой стороны, данный вариант является расширением для варианта использования «Ответить на вопросы». Это отражает тот факт, что, во-первых, при ответах на контрольные вопросы студенту может не хватить полученных при изучении теории знаний и потребуется повторное обращение к изучению теории, и, во-вторых, то, что студент, обладающий достаточными знаниями, может без предварительного изучения теории отвечать на контрольные вопросы и только при необходимости обращаться к ее изучению.

Отношение расширения включает в себя условие, при котором вариант-расширение подключается к базовому варианту, и ссылки на точки расширения (extension points). Ссылки на точки расширения указывают на те места в базовом варианте использования, куда должно быть подключено расширение, если условие выполняется. На рис. 6.3 условием расширения варианта использования «Ответить на вопросы» является наличие запроса студента на изучение теории. Точками расширения базового варианта являются те точки, в которых при ответах на вопросы возник такой запрос.

Отношение обобщения (generalization relationship) может существовать как между актерами, так и между вариантами использования. Данное отношение определяет, что некоторая сущность **A** является специализацией сущности **B**.

В этом случае сущность **A** называется потомком сущности **B** (дочерней сущностью), а сущность **B** – предком сущности **A** (родительской сущностью). При этом дочерние варианты использования наследуют свойства и поведение вариантов-предков и могут наделяться новыми свойствами и поведением. Актеры-потомки наследуют все роли актеров-предков. Графически отношение обобщения изображается сплошной линией с треугольной стрелкой (см. рис. 6.4, г). Стрелка направлена от сущности-потомка к сущности-предку.

Например, на рис. 6.2 актеры «Преподаватель» и «Лаборант» являются специализациями актера «Сотрудник», что определяется наличием соответствующих отношений обобщения между ними.

Следует отметить, что диаграмма вариантов использования представляет функции моделируемой системы или программного средства, но не отражает последовательности выполнения этих функций и связей между ними (в отличие, например, от методологии функционального моделирования IDEF0, см. подразд. 5.2). Таким образом, для всестороннего анализа предметной области необходима разработка других диаграмм языка UML.

Резюме

Диаграмма вариантов использования определяет функциональное назначение моделируемой системы или предметной области. Основными элементами диаграммы вариантов использования являются актер и вариант использования. Между отдельными элементами на диаграммах вариантов использования определены отношение ассоциации; отношение включения; отношение расширения; отношение обобщения. Отношения ассоциации характеризуются мощностью ассоциации.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Назовите математические основы объектно-ориентированного анализа и проектирования.
2. Дайте определения множества, элемента множества, подмножества, отношения множеств, кортежа.
3. Определите понятия графа, неориентированного графа, маршрута, ориентированного графа, ориентированного маршрута, семантической сети.
4. Кратко расскажите историю развития методов объектно-ориентированного анализа и проектирования.
5. Поясните назначение языка UML.
6. Перечислите основные принципы объектно-ориентированного анализа и проектирования, положенные в основу языка UML.
7. Назовите и охарактеризуйте виды отношений между объектами.
8. Назовите и охарактеризуйте виды отношений между классами.
9. Что называется мощностью ассоциации?
10. Перечислите типы мощности ассоциации.
11. Перечислите виды диаграмм языка UML и кратко опишите их назначение.
12. Перечислите и охарактеризуйте виды моделей языка UML.
13. Перечислите и охарактеризуйте уровни моделей языка UML.
14. Назовите основные элементы диаграммы вариантов использования.
15. Что в диаграммах вариантов использования называется актером? Как графически изображается актер?
16. Что в диаграммах вариантов использования называется вариантом использования? Как графически изображается вариант использования?
17. Какие виды отношений между отдельными элементами определены на диаграммах вариантов использования?
18. Приведите графическое представление различных видов отношений на диаграммах вариантов использования.
19. Поясните применение отношения ассоциации на диаграммах вариантов использования.
20. Как графически обозначается мощность ассоциации на диаграммах вариантов использования?
21. Поясните применение отношения включения на диаграммах вариантов использования.
22. Поясните применение отношения расширения на диаграммах вариантов использования.
23. Поясните применение отношения обобщения на диаграммах вариантов использования.
24. Приведите конкретный пример диаграммы вариантов использования, в которой применяются различные виды отношений.

РАЗДЕЛ 7. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

7.1. История развития CASE-средств

Очевидно, что большие размеры и высокая сложность разрабатываемых ПС при ограничениях на бюджетные и временные затраты проекта могут привести к низкому качеству конечных программных продуктов и системы в целом. В этой связи в последнее время все большее внимание уделяется современным технологиям и инструментальным средствам, обеспечивающим автоматизацию процессов ЖЦ ПС (CASE-средствам). Использование таких инструментальных средств позволяет существенно сократить длительность и стоимость разработки систем и ПС при одновременном повышении качества процесса разработки и, как следствие, качества разработанных ПС.

В истории развития CASE-средств обычно выделяется ***шесть периодов***. Данные периоды различаются применяемой техникой и методами разработки ПС. Эти периоды используют в качестве инструментальных средств следующие средства [24].

Период 1. Ассемблеры, анализаторы.

Период 2. Компиляторы, интерпретаторы, трассировщики.

Период 3. Символические отладчики, пакеты программ.

Период 4. Системы анализа и управления исходными текстами.

Период 5. Первое поколение CASE (CASE-I). Это CASE-средства, позволяющие выполнять поддержку начальных работ процесса разработки ПС и систем (анализ требований к системе, проектирование архитектуры системы, анализ требований к программным средствам, проектирование программной архитектуры, логическое проектирование баз данных). Адресованы непосредственно системным аналитикам, проектировщикам, специалистам в предметной области. Поддерживают графические модели, экранные редакторы, словари данных. Не предназначены для поддержки полного ЖЦ ПС.

Период 6. Второе поколение CASE (CASE-II). Представляют собой, как правило, набор (линейку) инструментальных средств, каждое их которых предназначено для поддержки отдельных этапов процесса разработки или других процессов ЖЦ ПС. В совокупности обычно поддерживают практически полный ЖЦ ПС. Используют средства моделирования предметной области, графиче-

ского представления требований, поддержки автоматической кодогенерации ПС. Содержат средства контроля и управления разработкой, интеграции системной информации, оценки качества результатов разработки. Поддерживают моделирование и прототипирование системы, тестирование, верификацию, анализ сгенерированных программ, генерацию документации по проекту.

Ко второму поколению CASE-средств относятся, например, линейки Telelogic и AllFusion, обзор которых приведен в подразд. 7.5, 7.6.

CASE-технологии предлагают новый, основанный на автоматизации подход к концепции ЖЦ ПС. Современные варианты CASE-моделей ЖЦ, называемые обычно RAD-моделями, рассмотрены в подразд. 2.3.

Наибольшие изменения в ЖЦ ПС при использовании CASE-технологий касаются первых этапов ЖЦ, связанных с анализом требований и проектированием. CASE-средства позволяют использовать визуальные среды разработки, средства моделирования и быстрого прототипирования разрабатываемой системы или ПС. Это позволяет на ранних этапах разработки оценить, насколько будущая система или программное средство устраивает заказчика и насколько она дружественна будущему пользователю.

Табл. 7.1 содержит усредненные оценки трудозатрат по основным этапам разработки ПС при различных подходах к процессу разработки [24]. Номерам строк в данной таблице соответствуют: 1 – традиционная разработка с использованием классических технологий; 2 – разработка с использованием современных структурных методологий проектирования; 3 – разработка с использованием CASE–технологий.

Таблица 7.1

Сравнительная оценка трудозатрат по этапам процесса разработки программных средств

№ подхода	Анализ, %	Проектирование, %	Кодирование, %	Тестирование, %
1	20	15	20	45
2	30	30	15	25
3	40	40	5	15

Из таблицы видно, что при традиционной разработке ПС основные усилия направлены на кодирование и тестирование, а при использовании CASE-технологий – на анализ и проектирование, поскольку CASE предполагают автоматическую кодогенерацию, автоматизированное тестирование и автоматический контроль проекта. Сопровождение кодов ПС заменяется сопровождением спецификаций проектирования. В результате данных факторов цена ошибок,

вносимых в проект при разработке и сопровождении ПС и систем, существенно снижается.

Резюме

В истории развития CASE-средств обычно выделяется шесть периодов. При традиционной разработке ПС основные усилия направлены на кодирование и тестирование, при использовании CASE-технологий – на анализ и проектирование.

7.2. Базовые принципы построения CASE-средств

Большинство CASE-средств основано на ***парадигме метод – нотация – средство*** [24].

Парадигма – это система изменяющихся форм некоторого понятия. В данном случае метод реализуется с помощью нотаций. Метод и нотации поддерживаются инструментальными средствами.

Метод – это систематическая процедура или техника генерации описаний компонент ПС. Примерами являются метод JSP Джексона, методология SADT (см. подразд. 4.6, п. 5.2.1).

Нотация – это система обозначений, предназначенная для описания структуры системы, элементов данных, этапов обработки; может включать графы, диаграммы, таблицы, схемы алгоритмов, формальные и естественные языки. Например, метод JSP реализуется с помощью нотации, базирующейся на применении четырех базовых конструкций данных. Современной нотацией методологии SADT является IDEF0.

Средства – это инструментарий для поддержки методов, помогающий пользователям при создании и редактировании графического проекта в интерактивном режиме, способствующий организации проекта в виде иерархии уровней абстракции, выполняющий проверки соответствия компонентов. Например, средством, поддерживающим метод JSP, являеся SmartDraw. IDEF0 поддерживается средством BPwin.

Фактически ***CASE-средство*** – это совокупность графически ориентированных инструментальных средств, поддерживающих процессы или отдельные этапы процессов ЖЦ ПС и систем.

К ***CASE-средствам*** может быть отнесено любое программное средство, обеспечивающее автоматическую помощь при разработке ПС, их сопровождении или управлении проектом, базирующееся на следующих ***основополагающих принципах***:

1. *Графическая ориентация*. В CASE-средствах используется мощная графика для описания и документирования систем или ПС и для улучшения интерфейса с пользователем.

2. *Интеграция*. CASE-средство обеспечивает легкость передачи данных между своими компонентами и другими средствами, входящими в состав линейки CASE-средств. Это позволяет поддерживать совокупность процессов ЖЦ ПС.

3. *Локализация всей проектной информации в репозитории* (компьютерном хранилище данных). Исполнителям проекта доступны соответствующие разделы репозитория в соответствии с их уровнем доступа. Это обеспечивает поддержку принципа коллективной работы. Информация из репозитория может использоваться для работ над текущим проектом, в том числе для автоматической кодогенерации ПС или систем, разработки следующих проектов, сбора статистики по выполненным ранее проектам организации.

Помимо данных принципов в основе концептуального построения CASE-средств лежат следующие ***положения*** [24].

1. *Человеческий фактор*. Его учет позволяет привести процессы ЖЦ ПС и систем к легкой, удобной и экономичной форме.

2. *Использование базовых программных средств*, применяющихся в других приложениях (СУБД, компиляторы с различных языков программирования, отладчики, языки четвертого поколения 4GL и др.).

3. *Автоматизированная или автоматическая кодогенерация*. При автоматизированной кодогенерации выполняется частичная генерация кодов программного средства, остальные участки программируются вручную. При автоматической кодогенерации выполняется полная генерация кодов программного средства. Возможны различные виды генерации (например, генерация проектной документации, базы данных по информационной модели, кодов из разработанных спецификаций программного средства; автоматическая сборка модулей, хранящихся в репозитории).

4. *Ограничение сложности*. Такое ограничение позволяет поддерживать сложность компонентов разрабатываемого программного средства или системы на уровне, доступном для понимания, использования и модификации.

5. *Доступность для различных категорий пользователей*, в том числе заказчиков, специалистов в предметной области, системных аналитиков, проектировщиков, программистов, тестировщиков, инженеров по качеству, менеджеров проектов. CASE-средства содержат инструменты различного функционального назначения, поддерживающие различные этапы основных, вспомогательных и организационных процессов ЖЦ ПС и систем (см. подразд. 1.2, 7.5).

6. *Рентабельность,* обеспечивающая быструю окупаемость денежных средств, вложенных в приобретение CASE-средства, за счет сокращения сроков и стоимости проектов.

7. *Сопровождаемость*. CASE-средства обладают способностью адаптации к изменяющимся требованиям и целям проекта.

Резюме

CASE-средства представляют собой совокупность графически ориентированных инструментальных средств, поддерживающих ЖЦ ПС и систем.

CASE-средства базируются на принципах графической ориентации, интеграции и локализации всей проектной информации в репозитории. В основе построения CASE-средств лежат человеческий фактор, использование базовых ПС, автоматизированная или автоматическая кодогенерация, ограничение сложности, доступность для разных категорий пользователей, рентабельность, сопровождаемость.

7.3. Основные функциональные возможности CASE-средств

В состав CASE-средств входят ***четыре основных компонента*** [24]:

1. *Средства централизованного хранения всей информации о проекте (репозиторий).* Предназначены для хранения информации о разрабатываемом программном средстве или системе в течение всего ЖЦ разработки.

2. *Средства ввода*. Служат для ввода данных в репозиторий, организации взаимодействия участников проекта с CASE-средством. Должны поддерживать различные методологии анализа, проектирования, тестирования, контроля. Предназначены для использования в течение ЖЦ программного средства или системы различными категориями участников проекта (системными аналитиками, проектировщиками, программистами, тестировщиками, менеджерами, специалистами по качеству и т.д.).

3. *Средства анализа и разработки*. Предназначены для анализа различных видов графических и текстовых описаний и их преобразований в процессе разработки.

4. *Средства вывода*. Служат для кодогенерации, создания различного вида документов, управления проектом.

Все компоненты CASE-средств в совокупности обладают следующими ***функциональными возможностями*** [24]:

- поддержка графических моделей;
- контроль ошибок;
- поддержка репозитория;
- поддержка основных, вспомогательных и организационных процессов ЖЦ ПС.

Поддержка графических моделей

В CASE-средствах разрабатываемые ПС представляются схематически. На разных уровнях проектирования могут использоваться различные виды и нотации графического представления ПС. Обычно применяются диаграммы различных типов, в том числе иерархии требований, диаграммы функционального моделирования (например IDEF0, DFD, см. подразд. 5.2, 5.3), диаграммы информационного моделирования (например IDEF1X, см. подразд. 5.4), струк-

турограммы (см. п. 4.1.3), диаграммы Джексона (см. подразд. 4.6, п. 5.5.1), диаграммы Варнье – Орра (см. п. 5.5.2), UML-диаграммы и т.п.

Разработка диаграмм осуществляется с помощью специальных графических редакторов, основными *функциями* которых являются создание и редактирование иерархически связанных диаграмм, их объектов и связей между объектами, а также автоматический контроль ошибок.

Контроль ошибок

В CASE-средствах, как правило, реализуются следующие ***типы контроля*** [24]:

1. *Контроль синтаксиса диаграмм и типов их элементов*. Например, при IDEF0-моделировании контролируется максимальное и минимальное количество функциональных блоков на диаграммах, наличие дуги управления и выходной дуги для любого функционального блока и т.п.

2. *Контроль полноты и корректности диаграмм*. При данном типе контроля выполняется проверка наличия имен у всех элементов диаграмм, проверка наличия необходимых описаний в репозитории и др.

3. *Контроль декомпозиции функций*. При данном типе контроля выполняется оценка декомпозиции на основе различных метрик. Например, может быть оценена эффективность и корректность декомпозиции с точки зрения связности и сцепления модулей (см. подразд. 4.7).

4. *Сквозной контроль диаграмм* одного или различных типов на предмет их взаимной корректности. Например, при IDEF0-моделировании контролируется соответствие граничных дуг родительского блока с внешними дугами дочерней диаграммы. При разработке IDEF0- и IDEF1X-моделей предметной области выполняется контроль их взаимной корректности и непротиворечивости.

Поддержка репозитория

Основными функциями репозитория являются хранение, обновление, анализ, визуализация всей информации по проекту и организация доступа к ней. Репозиторий обычно хранит более 100 типов объектов (например, диаграммы, определения экранов и меню, проекты отчетов, описания данных, модели данных, модели обработки, исходные коды, элементы данных) [24].

Каждый информационный объект, хранящийся в репозитории, описывается совокупностью своих свойств, например, идентификатор, тип, текстовое описание, компоненты, область значений, связи с другими объектами, времена создания и последнего обновления объекта, автор и т.п.

Репозиторий является базой для автоматической генерации документации по проекту. Основными ***типами отчетов*** являются:

• *отчеты по содержимому* – включают информацию по потокам данных и их компонентов; списки функциональных блоков диаграмм и их входных и выходных потоков; списки всех информационных объектов и их атрибутов; ис-

торию изменений объектов; описания модулей и интерфейсов между ними; планы тестирования модулей и т.п.;

• *отчеты по перекрестным ссылкам* – содержат информацию по связям всех вызывающих и вызываемых модулей; списки объектов репозитория, к которым имеет доступ конкретный исполнитель проекта; информацию по связям между диаграммами и конкретными данными; маршруты движения данных от входа к выходу;

• *отчеты по результатам анализа* – включают данные по взаимной корректности диаграмм, списки неопределенных информационных объектов, списки неполных диаграмм, данные по результатам анализа структуры проекта и т.п.;

• *отчеты по декомпозиции объектов* – включают совокупности объектов, входящих в каждый объект, а также объекты, в состав которых входит каждый объект.

Поддержка процессов жизненного цикла программных средств и систем

Основой поддержки процесса разработки являются следующие свойства современных CASE-средств [24].

1. *Покрытие всего жизненного цикла систем или программных средств.* Как отмечалось в подразд. 7.1, современные CASE-средства поддерживают практически полный ЖЦ ПС. Однако первоочередное внимание уделяется начальным работам процесса разработки – анализу требований к системе, проектированию системной архитектуры, анализу требований к программным средствам и проектированию программной архитектуры (см. подразд. 1.2). Грамотная разработка требований к системе и ПС является основой всего проекта, их полнота и корректность определяют уровень соответствия результатов разработки требованиям заказчика.

2. *Поддержка прототипирования.* Большинство моделей ЖЦ, предназначенных для разработки сложных или критичных продуктов, базируются на применении прототипирования. Это касается в первую очередь моделей, поддерживающих инкрементную и эволюционную стратегии разработки (см. пп. 2.1.3, 2.1.4, 2.3.4, 2.5.3 – 2.5.6). Прототипирование применяется на ранних этапах ЖЦ и позволяет уточнять требования к системе или программному средству, а также прогнозировать поведение разрабатываемого продукта.

3. *Поддержка современных методологий разработки систем или программных средств.* Современные линейки CASE-средств поддерживают, как правило, различные методологии, предназначенные для использования на различных этапах процесса разработки. При этом выполняется графическая поддержка построения диаграмм различных типов, контроль корректности использования шагов проектирования и подготовка документации.

4. *Автоматическая кодогенерация.* Кодогенерация позволяет построить автоматически до 90 % исходных кодов на языках высокого уровня. Различными CASE-средствами поддерживаются практически все известные языки программирования.

Средства кодогенерации можно подразделить на два вида:

- средства генерации управляющей структуры продукта; данные средства выполняют автоматическое построение логической структуры программного средства, кодов для базы данных, файлов, экранов, отчетов. Остальные фрагменты программного средства кодируются вручную;
- средства генерации полного продукта; данные средства позволяют на основе разработанных спецификаций или моделей генерировать полные коды программного средства, пользовательскую и программную документацию к нему.

Резюме

В состав CASE-средств входят средства централизованного хранения информации о проекте (репозиторий), средства ввода, средства анализа и разработки, средства вывода. Все компоненты CASE-средств в совокупности поддерживают графические модели, репозиторий, процесс разработки и ряд вспомогательных и организационных процессов, выполняют контроль ошибок.

7.4. Классификация CASE-средств

Все CASE-средства подразделяются на типы, категории и уровни [24].

7.4.1. Классификация по типам

Данная классификация отражает *функциональное назначение* CASE-средства в ЖЦ ПС и систем.

1. Анализ и проектирование

Средства этого типа используются для поддержки начальных этапов процесса разработки: анализа предметной области, разработки требований к системе, проектирования системной архитектуры, разработки требований к программным средствам, проектирования программной архитектуры, технического проектирования программных средств (см. подразд. 1.2). Средства данного типа поддерживают известные методологии анализа и проектирования. На выходе генерируются спецификации системы, ее компонентов и интерфейсов, связывающих эти компоненты, архитектура системы, архитектура программного средства, технический проект программного средства, включая алгоритмы и определения структур данных. Таким образом, поддерживаются работы 2 – 6 процеса разработки ПС и систем.

К средствам данного типа можно отнести, например, AllFusion Process Modeler (BPwin), CASE.Аналитик, Design/IDEF, Telelogic DOORS, Telelogic

Modeler, Telelogic TAU, Telelogic Rhapsody, Telelogic Statemate (см. подразд. 7.5, 7.6).

2. Проектирование баз данных и файлов

Средства этого типа обеспечивают логическое моделирование данных, автоматическое преобразование моделей данных в третью нормальную форму, автоматическую генерацию схем баз данных и описаний форматов файлов на уровне программного кода. К средствам этого типа можно отнести, например, AllFusion Data Modeler (ERwin), CA ERwin Data Model Validator (ранее ERwin Examiner), S-Designor, Silverrun, Designer2000, Telelogic TAU, Telelogic Rhapsody (см. подразд. 7.5, 7.6).

3. Программирование и тестирование

Средства этого типа поддерживают седьмую работу процесса разработки (программирование и тестирование, см. подразд. 1.2). Данные средства выполняют автоматическую кодогенерацию ПС на основе спецификаций или моделей. Содержат графические редакторы, средства поддержки работы с репозиторием, генераторы и анализаторы кодов, генераторы тестов, анализаторы покрытия тестами, отладчики.

К средствам данного типа можно отнести, например, TAU/Developer, TAU/Tester, Logiscope Audit, Logiscope RuleChecker, Logiscope TestChecker, Logiscope Reviewer, Rhapsody Developer (см. подразд. 7.5).

4. Сопровождение и реинженерия

Общей целью средств этого типа является поддержка корректировки, изменения, преобразования, реинженерия существующей системы, поддержка документации по проекту. К данным средствам относятся средства документирования, анализаторы программ, средства управления изменениями и конфигурацией ПС и систем, средства реструктурирования и реинженерии (реинженерия, реинженеринг – reverse engineering – обратное проектирование, например, построение спецификаций или моделей по исходным текстам программ), средства обеспечения мобильности, позволяющие перенести разработанную систему или программные средства в новое операционное или аппаратное окружение.

Средства реинженерии включают:

- *статические анализаторы* для генерирования схем программного средства из его кодов и оценки влияния модификаций;
- *динамические анализаторы*, включающие трансляторы со встроенными отладочными возможностями;
- *документаторы*, автоматически обновляющие документацию при изменении кода программного средства;
- *редакторы кодов*, автоматически изменяющие при редактировании кодов предшествующие ему структуры, в том числе и спецификации требований;
- *средства доступа к спецификациям*, позволяющие выполнять их модификацию и генерацию модифицированного кода;

- *средства реверсной инженерии*, транслирующие коды в спецификации или модели.

К средствам данного типа можно отнести, например, Telelogic DocExpress, Telelogic Synergy, Telelogic Change, средства линейки AllFusion Change Management Suite (см. подразд. 7.5, 7.6).

Следует отметить, что ряд CASE-средств других типов содержат в своем составе средства реинженерии. Это касается, например, CASE-средств AllFusion Data Modeler, Telelogic Rhapsody.

5. Окружение

К средствам данного типа относятся средства поддержки интеграции CASE-средств и данных. К данному типу можно отнести, например, Telelogic Rhapsody Gateway, Telelogic Rhapsody Interface Pack, AllFusion Data Profiler, AllFusion Model Manager, AllFusion Model Navigator (см. подразд. 7.5, 7.6).

6. Управление проектом

К средствам данного типа относятся средства поддержки процесса управления ЖЦ ПС и систем. Их функциями являются планирование, контроль, руководство, организация взаимодействия и т.п. К средствам данного типа можно отнести, например, Telelogic Focal Point, Telelogic Dashboard, AllFusion Process Management Suite, ADvisor (см. подразд. 7.5, 7.6).

Резюме

Классификация CASE-средств по типам отражает функциональное назначение CASE-средства в ЖЦ ПС. Выделяют типы CASE-средств, ориентированные на следующие этапы процесса разработки и другие процессы ЖЦ: анализ и проектирование, проектирование баз данных и файлов, программирование и тестирование, сопровождение и реинженерия, окружение, управление проектом.

7.4.2. Классификация по категориям

Данная классификация отражает *уровень интегрированности* CASE-средств по выполняемым функциям.

1. Категория Tool (tool – рабочий инструмент)

Включает средства самого низкого уровня интегрированности. В данную категорию средств входят инструментальные средства, решающие небольшую автономную задачу при разработке программного средства или системы. Обычно средства данной категории являются компонентами CASE-средств более высокого уровня интегрированности.

2. Категория ToolKit (toolkit – набор инструментов, пакет разработчика)

Включает CASE-средства среднего уровня интегрированности. Средства данной категории используют репозиторий для всей информации о проекте и ориентированы обычно на поддержку одного этапа или одной работы процесса

разработки или на поддержку одного из вспомогательных или организационных процессов ЖЦ ПС или систем. CASE-средства данной категории представляют собой интегрированную совокупность инструментальных средств, имеющих как правило общую функциональную ориентацию.

К CASE-средствам данной категории может быть отнесено, например, большинство CASE-средств из линеек Telelogic и AllFusion при их изолированном использовании (см. подразд. 7.5, 7.6).

3. Категория Workbench (workbench – рабочее место).

CASE-средства данной категории обладают самой высокой степенью интеграции. Они представляют собой интегрированную совокупность инструментальных средств, поддерживающих практически весь процесс разработки и ряд вспомогательных и организационных процессов ЖЦ ПС и систем. Используют репозиторий для хранения информации по проекту, поддерживают организацию коллективной работы над проектом.

Обычно к категории Workbench относятся линейки CASE-средств при их интегральном использовании. Примерами являются линейки Telelogic и AllFusion (см. подразд. 7.5, 7.6). Данные линейки CASE-средств поддерживает практически полностью процесс разработки ПС и систем, процессы сопровождения, документирования, управления конфигурацией, частично процессы обеспечения качества, верификации, аттестации. Таким образом, линейки Telelogic и AllFusion поддерживают практически весь ЖЦ ПС и систем.

Резюме

Классификация по категориям отражает уровень интегрированности CASE-средств по выполняемым функциям. Различают категории Tool, ToolKit, Workbench.

7.4.3. Классификация по уровням

Данная классификация связана с *областью действия* CASE-средств в ЖЦ ПС, систем и организаций.

1. Верхние (Upper) CASE-средства

CASE-средства данного уровня называют средствами компьютерного планирования. Их основной целью является помощь руководителям организаций, предприятий и конкретных проектов в определении политики организации и создании планов проекта. CASE-средства данного уровня позволяют строить модель предметной области, проводить анализ различных сценариев (существующего, наилучших, наихудших), накапливать информацию для принятия оптимальных решений. Таким образом, применительно к ЖЦ ПС и систем данные средства поддерживают процесс заказа и первую работу процесса разработки (подготовка процесса разработки). Графические средства данного уровня используются как формализованный язык общения между заказчиком (пользователем) и разработчиком требований (см. п. 5.2.6).

К средствам данного уровня можно отнести, например, Telelogic System Architect, Telelogic Focal Point, Telelogic Dashboard, средства линейки AllFusion Modeling Suite (см. подразд. 7.5, 7.6).

2. Средние (Middle) CASE-средства

CASE-средства данного уровня поддерживают начальные этапы процесса разработки (анализ предметной области, разработка требований к системе, проектирование системной архитектуры, разработка требований к программным средствам, проектирование программной архитектуры). Таким образом, фактически поддерживаются работы 2 – 6 процесса разработки (см. подразд. 1.2). При этом встроенные графические средства используются как формализованный язык общения между заказчиком (пользователем) и разработчиком спецификаций требований (см. п. 5.2.6).

Обычно данные средства обладают возможностями накопления и хранения информации по проекту. Это позволяет использовать накопленные данные как в текущем, так и в других проектах. Например, с помощью накопленной информации могут оцениваться продукты текущего проекта. При этом аналогичная информация предыдущих проектов используется в качестве базовой для оценки.

CASE-средства данного уровня зачастую поддерживают прототипирование и автоматическое документирование.

К CASE-средствам данного уровня можно отнести, например, линейку AllFusion Modeling Suite, средства Telelogic DOORS, Telelogic Modeler, Telelogic Tau, Telelogic Rhapsody, Telelogic Statemate, Telelogic DocExpress (см. подразд. 7.5, 7.6).

3. Нижние (Lower) CASE-средства

CASE-средства данного уровня поддерживают вторую половину работ процесса разработки ПС (как правило, начиная с шестой работы, см. п. 5.2.6). Содержат графические средства, исключающие необходимость разработки физических мини – спецификаций для программных модулей. Спецификации представляются обычно в виде моделей, которые непосредственно преобразуются в программные коды разрабатываемого программного средства или системы. Автоматически генерируется до 90 % кодов. Входной информацией для кодогенераторов являются спецификации, разработанные как в CASE-средствах данного уровня, так и в CASE-средствах среднего уровня.

CASE-средства нижнего уровня, как правило, поддерживают также прототипирование, тестирование, управление конфигурацией, генерацию документации, облегчают модификацию и сопровождение ПС или систем.

К CASE-средствам данного уровня можно отнести AllFusion Data Modeler, Telelogic Rhapsody, Telelogic Tau, Telelogic Statemate, Telelogic TAU Logiscope, Telelogic Change, Telelogic Synergy, Telelogic DocExpress (см. подразд. 7.5, 7.6).

Следует отметить, что в состав CASE-средств среднего и высокого уровней интегрированности обычно входят инструментальные средства, относя-

щиеся к нескольким уровням. Линейки CASE-средств, предназначенные для поддержки всего ЖЦ ПС и систем, включают в свой состав средства всех трех уровней.

Резюме

Классификация по уровням связана с областью действия CASE-средств в ЖЦ ПС и систем. Различают верхние, средние и нижние CASE-средства. Линейки CASE-средств включают в свой состав средства всех трех уровней.

7.5. Инструментальные средства Telelogic, предназначенные для автоматизации жизненного цикла организаций, систем и программных средств

К современным инструментальным средствам, обеспечивающим эффективную поддержку ЖЦ организаций в целом, систем и ПС, относятся интегрированные CASE-средства ***Telelogic*** [31]. В их состав входят средства поддержки оптимальных методов разработки структуры организации и увязывания ее бизнес-процессов с используемыми IT-технологиями и IT-системами; средства принятия решений, средства управления организацией, проектами, требованиями и изменениями; средства проектирования и моделирования систем и ПС на основе языков UML (Unified Modeling Language), SysML (Systems Modeling Language), SDL (Specification and Description Language). Все средства имеют репозиторий и поддерживают автоматическое создание документации.

Использование данных средств значительно сокращает общие затраты на разработку и сопровождение ПС и систем, снижает продолжительность разработки, повышает качество процессов ЖЦ, а следовательно, и качество промежуточных и конечного продуктов разработки.

Основными среди инструментальных средств Telelogic являются следующие:

- семейство Telelogic System Architect – поддерживает построение архитектуры предприятия. Предоставляет возможность создания и взаимной увязки интегрированного набора моделей и документов для четырех ключевых областей архитектуры предприятия: бизнеса, информации, IT-систем, технологий. Обеспечивает поддержку всех областей моделирования, в том числе моделирование бизнес-процессов, компонентное и объектное моделирование с помощью UML, моделирование данных, структурный анализ и проектирование. CASE-средства данного семейства могут использоваться заказчиком в процессе заказа ЖЦ ПС и систем (см. подразд. 1.2). В состав семейства Telelogic System Architect входят:

- Telelogic System Architect – включает все инструменты, необходимые для успешного создания архитектуры предприятия;
- Telelogic System Architect для DoDAF – поддерживает архитектуру предприятий на основе стандарта DoDAF;
- Telelogic System Architect/Publisher – поддерживает создание WEB-сайта, содержащего полный набор всех моделей архитектуры предприятия, автоматически создает и публикует разносторонние отчеты о моделях;
- Telelogic System Architect для FEA – поддерживает архитектуру предприятий для правительственных учреждений;
- Telelogic System Architect/XT – представляет собой WEB-решение для архитектуры предприятия;
- Telelogic System Architect/ERP – позволяет в исследовательских целях извлекать метаданные из приложений, используемых на предприятии, просматривать сложные внутренние взаимосвязи системы планирования ресурсов предприятия;

• Telelogic Focal Point – WEB-система, поддерживающая принятие решений при управлении требованиями, проектами, рисками, разработке продуктов, планировании сценариев, выборе проектов и продуктов, выполнении любой другой деятельности, связанной со сбором и анализом информации в масштабах производства. Может использоваться в процессах заказа, поставки, разработки, управления ЖЦ ПС и систем (см. подразд. 1.2);

• Telelogic Dashboard – поддерживает контроль за разработкой и принятие решений менеджерами проектов; поддерживает оценку состояния, рисков и тенденций проекта; позволяет автоматизировать сбор метрической информации из Telelogic Change и Telelogic DOORS, проводить ее анализ и получать отчетность по важным характеристикам; предоставляет метрики оценки проектов, в том числе оценки требований, изменений, конфигурации. Может использоваться в процессах обеспечения качества и управления ЖЦ ПС и систем (см. подразд. 1.2);

• Семейство Telelogic DOORS – семейство, предназначенное для управления требованиями к ПС или системам. Поддерживает сбор требований различного уровня, автоматическую трассировку требований, управление внесением изменений в требования, анализ влияния изменений на другие требования [14]. Может использоваться в процессах разработки и сопровождения ЖЦ ПС и систем (см. подразд. 1.2). В состав семейства Telelogic DOORS входят:

- Telelogic DOORS – поддерживает управление требованиями при создании инженерных систем или разработке ПО; обеспечивает комплексную поддержку записи, регистрации, структурирования, управления и анализа требований и их трассировки; предоставляет возможность контроля за изменением требований;
- Telelogic DOORS/XT – предназначено для управления требованиями в компаниях глобального уровня;

- Telelogic DOORS/Analyst – используется для моделирования требований на основе UML в рамках процесса управления требованиями; позволяет создавать модели, рисунки и диаграммы требований непосредственно в DOORS;
- Telelogic DOORS/Net – обеспечивает WEB-доступ к управлению требованиями для удаленных пользователей;
- Telelogic DOORS Fastrak – предназначено для управления требованиями в проектах с короткими сроками реализации; поддерживает WEB-доступ;

• Telelogic Modeler – средство проектирования ПО на основе стандартного графического языка (UML), позволяющего визуализировать проектирование систем и ПС. Распространяется бесплатно. Может использоваться в процессах разработки и сопровождения ЖЦ ПС и систем (см. подразд. 1.2);

• Семейство Telelogic Tau – семейство, предназначенное для моделирования требований к ПС или системам, проектирования моделей их архитектуры, создания тестов и тестирования моделей, автоматической кодогенерации на основе проверенных моделей. Поддерживает отраслевые стандарты визуального моделирования UML, SysML, SDL. Обеспечивает системное и интеграционное тестирования на основе стандартов TTCN-2 и TTCN-3. Позволяет унифицировать язык общения между системными аналитиками, проектировщиками, программистами и другими разработчиками. Может использоваться в процессах разработки и сопровождения ЖЦ ПС и систем (см. подразд. 1.2). В состав семейства Telelogic Tau входят:

- Telelogic Tau – служит для разработки систем и ПС с использованием языков моделирования UML и SysML; поддерживает этапы ЖЦ, начиная от разработки архитектуры системы или программного средства и заканчивая кодогенерацией и автоматическим созданием тестов; поддерживает промышленные стандарты визуализации разработки и тестированиия систем и ПС (UML 2.1, SysML, MDA, DoDAF, SOA, UML Testing Profile). Содержит следующие инструменты:
  - TAU/Model Author – используется авторами моделей предметной области; позволяет рисовать диаграммы моделей;
  - TAU/Architect – используется системными инженерами; поддерживает возможности TAU/Model Author и верификацию диаграмм;
  - TAU/Developer – используется разработчиками ПС; поддерживает возможности TAU/Architect и генерацию кода;
  - TAU/Tester – используется тестировщиками ПС;
- Telelogic SDL Suite – поддерживает разработку ПО систем реального времени; предоставляет возможности создания ПО для сложных, событийно-управляемых коммуникационных систем, описываемых с использованием стандарта SDL; обеспечивает визуализацию проектирования и автоматическую кодогенерацию;

- Telelogic TTCN Suite – поддерживает автоматизацию тестирования систем реального времени и телекоммуникационных систем на основе скриптового языка сценариев тестирования TTCN-2 (Tree and Tabular Combined Notation);
- Telelogic Tau/DoDAF – поддерживает проектирование архитектуры систем, совместимых со стандартом DoDAF (Department of Defense Architecture Framework) – структурой архитектуры Министерства обороны США; поддерживает моделирование на UML и SysML, атоматическую кодогенерацию, автоматическое обнаружение ошибок;
- Telelogic Tester – поддерживает тестирование ПС на основе стандарта TTCN-3 (Testing and Test Control Notation); предоставляет возможности автоматизиции системного и интеграционного тестирования; поддерживает весь ЖЦ тестирования ПС (постановку задачи тестирования, разработку сценариев, анализ, исполнение, отладку);

• семейство Telelogic Rhapsody – представляет среду разработки на основе моделей (Model-Driven Development). Поддерживает языки моделирования UML и SysML. Предоставляет возможность создания языка предметной области DSL (Domain Specific Language), позволяющего создавать новые диаграммы и элементы диаграмм, приближенные к конкретной предметной области. Обеспечивает проектирование, кодогенерацию и тестирование встраиваемых систем и ПО реального времени. При проектировании систем использует функционально-ориентированный подход, при проектировании ПО – объектно-ориентированный подход. Содержит средства реинженерии. Может использоваться в процессах разработки и сопровождения ЖЦ ПС и систем (см. подразд. 1.2). Семейство Telelogic Rhapsody состоит из следующего набора модулей и пакетов:

- Rhapsody System Architect – поддерживает визуализацию, разработку спецификаций и документирование систем;
- Rhapsody Systems Designer – поддерживает визуализацию, разработку спецификаций, аттестацию и документирование систем;
- Rhapsody Architect – поддерживает визуализацию, разработку спецификаций и документирование ПО;
- Rhapsody Developer – поддерживает визуализацию, разработку спецификаций, аттестацию, кодогенерацию и документирование ПО;
- Пакет AUTOSAR – поддерживает проектирование модулей систем на базе стандарта AUTOSAR;
- Пакет DoDAF – поддерживает проектирование и моделирование спецификаций систем на базе стандарта DoDAF;
- Gateway – обеспечивает интеграцию с DOORS и другими инструментами управления требованиями;
- Test Conductor – поддерживает функциональное тестирование на основе требований;
- Automatic Test Generation – поддерживает автоматическое создание тестов на основе модели;

- Interface Pack – обеспечивает интеграцию с инструментами управления конфигурацией, интеграцию с Simulink, импорт моделей Rational Rose, поддержку XMI;
- Tools & Utilities Pack – поддерживает быстрое прототипирование и настраиваемое создание документов на основе WEB-интерфейса пользователя;
- Value Pack – сочетает возможности Tools & Utilities Pack и Interfaces Pack;
- Gateway Value Pack – сочетает возможности Gateway и Value Pack;
- Teamcenter System Engineering Interface – поддерживает управление жизненным циклом продукта; представляет собой результат интеграции Teamcenter и Telelogic Rhapsody;

• Telelogic Statemate – предназначено для разработки прототипов. Поддерживает моделирование, графическое представление системных требований, исполняемых спецификаций и проектирования, автоматическую генерацию кодов и тестов прототипов для быстрой разработки сложных встраиваемых систем. Использует стандартные диаграммы проектирования и языка UML. Позволяет обнаруживать и исправлять ошибки на ранних этапах разработки систем. Может использоваться в процессах разработки и сопровождения ЖЦ ПС и систем (см. подразд. 1.2);

• семейство Telelogic DocExpress – поддерживает автоматизацию документирования ПО, упрощает управление отчетностью. Предоставляет возможности сбора из различных инструментов Telelogic, форматирования и публикации проектной и технической документации в интерактивном режиме. Поддерживает исходные данные для документирования и документацию в актуальном состоянии. Упрощает сопровождение документов. Может использоваться в процессе документирования ЖЦ ПС и систем (см. подразд. 1.2). В состав семейства Telelogic DocExpress входят:

- Telelogic DocExpress Word – поддерживает представление документации в Microsoft Word; обеспечивает легкое форматирование различных типов данных и их включение в один документ; позволяет экспортировать в Word проектную информацию, в том числе структуры данных и модели;
- Telelogic DocExpress Factory – обеспечивает автоматизированное составление отчетности в автономном режиме для выбора, сбора и публикации проектной документации в различных форматах (например Adobe FrameMaker, Microsoft Word, HTML);

• Telelogic Logiscope – поддерживает оценку и обеспечение качества ПС; помогает увеличить тестовое покрытие, автоматизирует анализ кода, идентификацию модулей, которые могут содержать ошибки. Может использоваться в процессах разработки, сопровождения, обеспечения качества, верификации, аттестации ЖЦ ПС и систем (см. подразд. 1.2). Данное семейство состоит из следующего набора инструментов:

- Logiscope Audit – поддерживает оценку качества и графический анализ исходных программных кодов;
- Logiscope RuleChecker – позволяет выполнять проверку исходного кода на соответствие принятым правилам;
- Logiscope TestChecker – выполняет проверку степени покрытия исходного кода тестовыми наборами;
- Logiscope Reviewer – сочетает возможности Logiscope Audit и Logiscope RuleChecker;

• Семейство Telelogic Synergy – поддерживает управление изменениями и конфигурацией ПС. Предоставляет возможность оценки и авторизации запросов на изменения. Позволяет управлять версиями ПС. Может использоваться в процессах разработки, сопровождения, управления конфигурацией ЖЦ ПС и систем (см. подразд. 1.2). В состав семейства Telelogic Synergy входят:

- Telelogic Change – обеспечивает WEB-доступ к управлению изменениями; позволяет отслеживать статус запросов на изменения на протяжении их ЖЦ; содержит централизованный репозиторий; позволяет составлять отчеты по изменениям требуемого формата;
- Telelogic Synergy – поддерживает процесс управления конфигурацией ЖЦ ПС (см. подразд. 1.2), компонентно-ориентированную разработку (см. п. 2.5.6), управление выпуском версий; содержит мощный распределенный репозиторий; поддерживает коллективную разработку продукта.

Резюме

CASE-средства Telelogic интегрируются друг с другом. Их совместное использование поддерживает практически весь ЖЦ ПС и систем, регламентированный стандартом *СТБ ИСО/МЭК 12207–2003*.

7.6. Инструментальные средства Computer Associates, предназначенные для автоматизации жизненного цикла организаций, систем и программных средств

К современным инструментальным средствам, обеспечивающим эффективную поддержку ЖЦ организаций или предприятий, систем и ПС, относится линейка AllFusion компании Computer Associates. Данная линейка представляет собой набор интегрированных средств для разработки, внедрения и управления информационными системами в организации. Все средства линейки имеют репозиторий и поддерживают автоматическое создание документации. Совокупное использование средств линейки AllFusion покрывает практически весь ЖЦ

ПС и систем, регламентированный стандартом *СТБ ИСО/МЭК 12207–2003,* позволяет сократить затраты на их разработку и сопровождение, снижает продолжительность и стоимость разработки, повышает качество процессов ЖЦ, качество промежуточных и конечных продуктов разработки.

По функциональному назначению основные компоненты линейки AllFusion можно подразделить на следующие типы [32]:

- средства моделирования баз данных, бизнес-процессов и компонентов ПО;
- средства управления изменениями и конфигурациями;
- средства интегрированного управления проектами и процессами.

Средства моделирования баз данных, бизнес-процессов и компонентов ПО представлены линейкой ***CA ERwin Modeling Suite*** **(*AllFusion Modeling Suite*)**. Данная линейка предназначена для анализа и моделирования различных аспектов деятельности организации и проектирования информационных систем. Средства данной линейки позволяют моделировать предметные области, базы данных, компоненты ПО, деятельность и структуру организаций. CASE-средства линейки CA ERwin Modeling Suite поддерживают структурные методы анализа и проектирования, базирующиеся на использовании методологий моделирования IDEF0, DFD, IDEF3 и IDEF1X. CASE-средства данной линейки поддерживают процессы разработки, сопровождения, верификации, аттестации и документирования ЖЦ ПС и систем.

Основными в данной линейке являются следующие средства:

- CA ERwin Process Modeler (AllFusion Process Modeler, называемый ранее BPwin) – средство функционального моделирования предметной области. Поддерживает три методологии моделирования – IDEF0 (функциональное моделирование), DFD (моделирование потоков данных) и IDEF3 (моделирование потоков работ), что позволяет комплексно описывать предметную область. Поддерживает методы функционально-стоимостного анализа хозяйственной деятельности организаций. Содержит генератор отчётов, имеет широкий набор средств документирования моделей и проектов. Методологии IDEF0 и DFD подробно описаны в подразд. 5.2, 5.3;
- CA ERwin Data Modeler (AllFusion Data Modeler, ранее ERwin) – средство проектирования, документирования и сопровождения баз данных и хранилищ данных. Поддерживает методологии информационного моделирования IDEF1X, IE, Dimensional. Содержит средства автоматической кодогенерации баз данных, средства их сопровождения и реинженерии; поддерживает различные типы СУБД. Содержит генератор отчётов, имеет широкий набор средств документирования моделей и проектов. Методология IDEF1X подробно описана в подразд. 5.4;
- CA ERwin Data Profiler (AllFusion Data Profiler) – средство для анализа и профилирования исходной информации, поступающей из множественных источников данных. Предоставляет возможности профилирования данных для ряда платформ, включая реляционные базы данных, электронные таблицы и на-

следуемые неструктурированные файлы. Позволяет улучшить повторное использование данных, предоставляет средства миграции и интеграции данных. Позволяет генерировать метрики и статистику качества данных;

- CA ERwin Data Model Validator (ранее ERwin Examiner) – средство для проверки структуры баз данных и качества моделей CA ERwin Data Modeler;
- CA ERwin Model Manager (ранее ModelMart) – среда для совместного моделирования в CA ERwin Data Modeler и/или CA ERwin Process Modeler;
- CA ERwin Saphir Option – средство обнаружения метаданных. Предназначено для извлечения, анализа и публикации метаданных из пакетированных корпоративных приложений. Преобразует специфические для пакета метаданные в модели данных CA ERwin Data Modeler;
- CA ERwin Model Navigator – инструмент для просмотра моделей, созданных в CA ERwin Data Modeler и CA ERwin Process Modeler. Предоставляет доступ ко всему набору отчетов из CA ERwin Data Modeler и CA ERwin Process Modeler для анализа и разработки приложений.

Средства управления изменениями и конфигурациями представлены линейкой ***AllFusion Change Management Suite.*** Данная линейка предназначена для управления изменениями на различных стадиях ЖЦ разработки приложений и выполнения интегрированных задач по настройке ПО всей организации, включая разработку многоплатформенного ПО и WEB-приложений. Средства данной линейки поддерживают процессы разработки, сопровождения, управления конфигурацией и документирования ЖЦ ПС и систем.

Основными в данной линейке являются следующие средства:

- AllFusion Harvest Change Manager (CCC/Harvest) – средство для управления изменениями, конфигурациями и версиями при разработке сложных корпоративных систем на основе общего репозитория; помогает синхронизировать деятельность разработчиков на различных платформах в течение всего жизненного цикла;
- AllFusion Change Manager Enterprise Workbench – средство автоматизации процесса разработки ПО на различных платформах; поддерживает интеграцию функций управления изменениями в приложениях в масштабе организации;
- AllFusion Endevor Change Manager – средство управления изменениями, версиями, конфигурациями для мэйнфреймов; поддерживает разработку приложений в среде OS/390, Windows и UNIX;
- AllFusion CA-Librarian – средство управления библиотеками при разработке на мэйнфреймах; обеспечивает безопасность и осуществляет контроль программных ресурсов организации;
- AllFusion CA-Panvalet – средство управления кодом разрабатываемых приложений (для мэйнфреймов); предоставляет возможности централизованного хранения исходного кода приложений.

Средства интегрированного управления проектами и процессами предназначены для управления готовыми продуктами, проектами, ресурсами и передаваемыми файлами на предприятии. Средства данного типа поддерживают в первую очередь процесс управления ЖЦ ПС и систем. В состав средств данного типа входят:

- AllFusion Process Management Suite (ранее Process Continuum) – представляет собой полный набор инструментальных средств для эффективного управления процессом разработки ПО, в том числе методиками, проектами, ресурсами и конечными результатами процесса разработки;
- ADvisor – WEB-среда для разработки приложений, управления информацией и работами.

Резюме

Инструментальные средства линейки AllFusion компании Computer Associates подразделяются на средства моделирования баз данных, бизнес-процессов и компонентов ПО; средства управления изменениями и конфигурациями; средства интегрированного управления проектами и процессами. Их совместное использование поддерживает практически весь ЖЦ ПС и систем, регламентированный стандартом *СТБ ИСО/МЭК 12207–2003*.

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Перечислите периоды развития CASE-средств.
2. Дайте сравнительную оценку трудозатрат по этапам разработки при различных подходах к процессу разработки ПС.
3. Поясните суть парадигмы метод – нотация – средство.
4. Какое программное средство называется CASE-средством?
5. Перечислите основополагающие принципы, на которых базируются CASE-средства.
6. Какие положения лежат в основе концептуального построения CASE-средств?
7. Перечислите и охарактеризуйте основные компоненты CASE-средств.
8. Какие типы контроля реализуются обычно в CASE-средствах?
9. Перечислите основные типы отчетов, реализуемые при автоматической генерации документации по проекту в CASE-средствах.
10. Перечислите свойства современных CASE-средств, обеспечивающие поддержку процесса разработки программных продуктов.
11. По каким критериям подразделяются средства кодогенерации?
12. Что отражает классификация CASE-средств по типам?
13. Перечислите и охарактеризуйте типы CASE-средств.
14. Что отражает классификация CASE-средств по категориям?
15. Перечислите и охарактеризуйте категории CASE-средств.
16. Что отражает классификация CASE-средств по уровням?
17. Перечислите и охарактеризуйте уровни CASE-средств.
18. Перечислите и охарактеризуйте основные CASE-средства линейки Telelogic.
19. Перечислите типы инструментальных средств, входящих в линейку AllFusion компании Computer Associates.
20. Перечислите и охарактеризуйте основные CASE-средства линейки AllFusion компании Computer Associates.

ЛИТЕРАТУРА

1. IEEE Std. 1320.1–1998. Стандарт IEEE по языку функционального моделирования – Синтаксис и семантика IDEF0. – Введ. 1998-09-25. – Нью-Йорк : IEEE, 1998.

2. IEEE Std. 1320.2–1998. Стандарт IEEE по синтаксису и семантике языка концептуального моделирования IDEFIX97 (IDEF Object). – Введ. 1998-06-25. – Нью-Йорк : IEEE, 1998.

3. ISO/IEC 12207:1995. Информационная технология – Процессы жизненного цикла программных средств. – Введ. 1995-08-01. – Женева : ISO/IEC, 1995.

4. ISO/IEC 12207:2008. Системная и программная инженерия – Процессы жизненного цикла программных средств. – Введ. 2008-02-01. – Нью-Йорк : ISO/IEC-IEEE, 2008.

5. ISO/IEC14598–1:1999. Информационная технология – Оценка программного продукта – Ч. 1 : Общий обзор.

6. ISO/IEC 9126–1:2001. Программная инженерия – Качество продукта – Ч. 1: Модель качества. – Введ. 2001 – 06 – 15. – Женева : ISO/IEC, 2001.

7. ГОСТ Р ИСО/МЭК ТО 12182 – 2002. Информационная технология. Классификация программных средств. – Введ. 2002 – 06 – 11. – М. : Изд-во стандартов, 2002.

8. ГОСТ Р ИСО/МЭК ТО 15271 – 2002. Информационная технология. Руководство по применению ГОСТ Р ИСО/МЭК 12207 (Процессы жизненного цикла программных средств). – Введ. 2002 – 06 – 05. – М. : Изд-во стандартов, 2002.

9. СТБ ИСО/МЭК 12207 – 2003. Информационные технологии. Процессы жизненного цикла программных средств. – Введ. 2003 – 03 – 19. – Минск : Госстандарт Респ. Беларусь, 2003.

10. СТБ ИСО/МЭК 9126 – 2003. Информационные технологии. Оценка программной продукции. Характеристики качества и руководства по их применению. – Введ. 2003 – 03 – 19. – Минск : Госстандарт Республики Беларусь, 2003.

11. Р 50.1.028 – 2001. Информационные технологии поддержки жизненного цикла продукции. Методология функционального моделирования. Рекомендации по стандартизации. – Введ. 2001 – 07 – 02. – М. : Изд-во стандартов, 2001.

12. РД IDEF0 – 2000. Методология функционального моделирования IDEF0. Руководящий документ. – М. : Изд-во стандартов, 2000.

13. Бахтизин, В. В. Методология функционального проектирования IDEF0 : учеб. пособие по курсу «Технология разработки программного обеспечения» для студ. спец. 40 01 01 «Программное обеспечение информационных технологий» / В. В. Бахтизин, Л. А. Глухова. – Минск : БГУИР, 2003.

14. Бахтизин, В. В. Работа с требованиями в среде DOORS : учеб.-метод. пособие / В. В. Бахтизин, Л. А. Глухова, С. Н. Неборский. – Минск : БГУИР, 2007.

15. Бахтизин, В. В. Стандартизация и сертификация программного обеспечения : учеб. пособие / В. В. Бахтизин, Л. А. Глухова. – Минск : БГУИР, 2006.

16. Бахтизин, В. В. Структурный анализ и моделирование в среде CASE-средства BPwin : учеб. пособие по курсу «Технология проектирования программ» для студ. спец. 40 01 01 «Программное обеспечение информационных технологий» / В. В. Бахтизин, Л. А. Глухова. – Минск : БГУИР, 2002.

17. Брауде, Э. Технология разработки программного обеспечения / Э. Брауде. – СПб. : Питер, 2004.

18. Волохов, А. Telelogic DOORS. Requirements Management [Электронный ресурс] / А. Волохов. – 2005. – Режим доступа: http://www.telelogic.ru

19. Глухова, Л. А. Методология структурного анализа и проектирования SADT : учеб. пособие по курсу «Технология проектирования программ» для студ. спец. Т.10.02.00 / Л. А. Глухова, В. В. Бахтизин. – Минск : БГУИР, 2001.

20. Глухова, Л. А. Основы алгоритмизации и структурного проектирования программ : учеб. пособие по курсам «Основы алгоритмизации и программирования» и «Технология разработки программного обеспечения» для студ. спец. 40 01 01 «Программное обеспечение информационных технологий» дневн. формы обуч. / Л. А. Глухова, В. В. Бахтизин. – Минск : БГУИР, 2003.

21. Глухова, Л. А. Информационное моделирование с помощью CASE-средства ERwin 3.0 : учеб. пособие по курсу «Технология проектирования программ» для студ. спец. Т.10.02.00 «Программное обеспечение информационных технологий» / Л. А. Глухова, В. В. Бахтизин. – Минск : БГУИР, 1999.

22. Зиглер, К. Методы проектирования программных средств / К. Зиглер. – М. : Мир, 1985.

23. Йодан, Э. Структурное проектирование и конструирование программ / Э. Йодан. – М. : Мир, 1979.

24. Калянов, Г. Н. CASE-технологии. Консалтинг при автоматизации бизнес-процессов / Г. Н. Калянов. – М. : Горячая линия-Телеком, 2000.

25. Каменнова, М. С. Системный подход к проектированию сложных систем / М. С. Каменнова // Журнал д-ра Добба. – 1993. – №1.

26. Кинг, Д. Создание эффективного программного обеспечения / Д. Кинг. – М. : Мир, 1991.

27. Леоненков, А. В. Самоучитель UML 2 / А. В. Леоненков. – СПб : BHV-СПб, 2007.

28. Маклаков, С. В. Создание информационных систем с AllFusion Modeling Suite / С. В. Маклаков. – М. : ДИАЛОГ-МИФИ, 2003.

29. Марка, Д. Методология структурного анализа и проектирования SADT / Д. Марка, К. МакГоуэн. – М. : МетаТехнология, 2003.

30.Орлов, С. А. Технологии разработки программного обеспечения / С. А. Орлов. – СПб. : Питер, 2002.

31. Продукция [Электронный ресурс]. – 2008. – Режим доступа: http://www.telelogic.ru/products/.

32. Средства моделирования (CASE) и поддержки всех стадий разработки ПО [Электронный ресурс]. – 2010. – Режим доступа: http://www.interface.ru/home.asp?artId=99.

33. Фатрелл, Р. Управление программными проектами: достижение оптимального качества при минимуме затрат / Р. Фатрелл, Д. Шафер, Л. Шафер. – М. : Вильямс, 2003.

34. Черемных, С. В. Моделирование и анализ систем. IDEF-технологии : практикум / С. В. Черемных, И. О. Семенов, В. С. Ручкин. – М. : Финансы и статистика, 2002.

35. Шлеер, С. Объектно-ориентированный анализ: моделирование мира в состояниях / С. Шлеер, С. Меллор. – Киев : Диалектика, 1993.

36. A Guide to the Project Management Body of Knowledge (PMBOK). – Upper Darby : PMI Standards Committee. – 1996.

37. CHAOS Surveys and Reports [Электронный ресурс]. – 2003. – Режим доступа : www.standishgroup.com/CHAOS Surveys and Reports/

38. Jackson, M. A. A system development method / M. A. Jackson [Электронный ресурс]. – 2008. – Режим доступа: http://mcs.open.ac.uk/mj665/Nice1981.pdf

39. Jackson System Development [Электронный ресурс]. – 2008. – Режим доступа : http://en.wikipedia.org/wiki/Jackson_System_Development

40. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2, Beta 1 [Электронный ресурс]. – 2008. – Режим доступа: http://www.omg.org/docs/formal/09-02-02.pdf

41. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.2, Beta 1 [Электронный ресурс]. – 2008. – Режим доступа : http://www.omg.org/docs/formal/09-02-04.pdf.

42. Software Development Methodology Today. Software Development Strategies and Life-Cycle Models [Электронный ресурс]. – 2008. – Режим доступа: http://www.informit.com/articles/article.aspx?p=605374&seqNum=2.

43. Examples [Электронный ресурс]. – 2008. – Режим доступа: http://www.smartdraw.com/examples/preview/index.aspx?example=Jackson_ _Structured_Programming_Diagram.

Учебное издание

Бахтизин Вячеслав Вениаминович
Глухова Лилия Александровна

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

УЧЕБНОЕ ПОСОБИЕ

Редактор *Т. П. Андрейченко*
Корректор *Л. А. Шичко*
Компьютерная верстка *М. В. Чечетко*

Подписано в печать 28.09.2010. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 15,69 Уч.-изд. л. 16,0 Тираж 400 экз. Заказ 485.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6