Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Н.В. Зариковская

Анализ и разработка моделей информационных процессов и структур

Учебное пособие

Томск, 2018

Зариковская Н.В. Анализ и разработка моделей информационных процессов и структур. Учебное пособие - Томск: Изд-во ТУСУР, 2018. - 189 с.

Рассмотрены вопросы применение современных языков и инструментов для моделирования предметной области автоматизации. Приведены современные парадигмы и инструменты моделирования, возможности различных инструментов по описанию предметной области автоматизации на различных этапах создания информационных систем. Особое внимание уделено объектно-ориентированному анализу и проектированию на базе инструмента Enterprise Architect и методологии структурного анализа и проектирования на базе AllFusion Modeling Suite.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Принципиальными факторами, влияющими на структуру информационных систем и технологические процессы обработки информации, являются функциональные бизнес-процессы, потоки динамически изменяющихся данных, организация данных в сложные структуры. Такой подход в анализе и проектировании систем обусловил наибольшее распространение определенных видов CASE-технологий, отраженных в стандартах UML, IDEF0, DFD, IDEF3 и IDEF1X, а также соответствующих программных инструментах почти всех ведущих фирм – производителей программного обеспечения.

Одним из факторов, от которых зависит успех проекта, является наличие строгого стандарта языка моделирования, включающего элементы модели – фундаментальные концепции моделирования и их семантику, нотацию – визуальное представление элементов моделирования, и руководство по использованию языка – правила применения его элементов в рамках построения тех или иных типов моделей программного обеспечения.

Данное учебное пособие посвящено характеристике современных языков моделирования бизнес-процессов и описанию их роли и места в общем процессе проектирования информационных систем, представлению наиболее известных стандартов описания бизнес-процессов, моделированию бизнес-процессов на основе BPMN-диаграмм. Приводятся основы методологии разработки информационных систем на базе моделей предметной области. А также содержит сведения об объектно-ориентированном подходе к разработке программного обеспечения. Представлена характеристика UML-2 с примерами на базе Enterprise Architect. Дана развернутая характеристика унифицированного процесса разработки программного обеспечения.

Помимо вышесказанного, показан процесс анализа и управления требованиями, способы их описания, документирования и трассировки.

ГЛАВА 1. ХАРАКТЕРИСТИКА СОВРЕМЕННЫХ ЯЗЫКОВ МОДЕЛИРОВАНИЯ БИЗНЕС-ПРОЦЕССОВ

Процесс выполнения тех или иных видов работ по управлению и обработке информационных ресурсов (документооборот, библиотеки, архивы данных и т. д.) представляет собой регламентированный набор действий, который требуется выполнить для достижения необходимого результата. При этом в процессе подготовки входных и выходных артефактов каждого этапа потока работ исполнители используют обширный набор инструментальных программных продуктов для частичной автоматизации своего участка работ. Такая частичная автоматизация «ручной деятельности», конечно, имеет ряд положительных моментов, но задача упрощения координации процесса по обработке информационных ресурсов в целом данным подходом не решается. Следующие ресурсоемкие задачи не могут быть решены путем простой автоматизации деятельности сотрудников на местах:

автоматизированная подготовка входных и выходных артефактов каждого этапа процесса;

координация потока управления и потока данных;

полная автоматизация отдельных участков потока работ (способность взаимодействия с «программными» исполнителями заданий в рамках некоторого унифицированного интерфейса);

возможность быстрого, с минимальными трудозатратами, создания нового описания регламента с возможной модульностью (декомпозицией на подпроцессы) для повторного использования описаний, с поддержкой быстрой и безболезненной для участников процесса модификацией имеющихся регламентов;

эффективная реакция потока работ на возникновение непредвиденных обстоятельств на пути его выполнения (например, недоступность в данный момент тех или иных ресурсов), в том числе четко регламентированные действия по устранению последствий некорректно выполненного этапа процесса;

хорошая управляемость процессом с доступом к данным любого активного этапа;

четкое ролевое разделение участников процесса (в том числе поддержка динамической взаимозаменяемости исполнителей в случае недоступности нужных ресурсов);

возможность сбора статистики выполнения процесса для последующей оптимизации.

Для эффективного решения перечисленных выше задач большая часть усилий разработчиков программного обеспечения на текущий момент сконцентрирована вокруг теории автоматизированных потоков работ (Workflow) и систем, способных эффективно решать задачи их

исполнения и координации (Workflow Management Systems). Количество подобных информационных систем (интеграционных платформ и серверов), в основу которых на формальном уровне заложена базовая концепция интеграции распределенных ресурсов (как программных систем, так и человеческих ресурсов) для выполнения некоторой общей задачи, увеличивается очень быстрыми темпами. При этом новые решения приводят к появлению новых задач, адресованных системам исполнения потоков работ.

К основным из них можно отнести:

строгую формализацию описаний потоков работ на некотором языке при построении автономных систем исполнения потоков работ (вне контекста конкретного варианта их использования);

предоставление системой разработки потоков работ (являющейся частью интеграционного сервера) развитых средств по созданию и модификации описаний потоков работ, поскольку основной контингент пользователей подобных систем составляют исполнители на местах – аналитики, менеджеры (т. е. люди, неискушенные в программировании маршрута потока работ с использованием некоторых формальных языков). При этом должен использоваться наиболее интуитивно понятный обычному пользователю способ формирования потоков работ – визуальные диаграммы (так как практически каждый управляющий процессом человек привык применять для таких целей некоторое прикладное средство, такое как, например, диаграммы активности деятельности UML). Таким образом, помимо эффективного «языка программирования» потоков работ пользователям должна быть представлена удобная графическая нотация для их описания с достаточным уровнем абстракции;

предоставление интеграционным сервером создателям описаний потоков работ полного набора средств для проверки их работоспособности до начала опытной эксплуатации. Такие средства должны включать как простые статические верификаторы корректности созданных описаний потоков работ, так и динамические отладчики;

интерактивные средства взаимодействия с пользователями, которые являются важной частью интеграционного сервера, поскольку именно пользователи управляют ходом выполнения потока работ;

задачи обеспечения безопасности данных и поддержки транзакционности автоматизированных потоков работ, которые имеют большую важность.

По данным направлениям организациями – законодателями теоретических основ систем исполнения потоков работ выполняется большая работа по стандартизации универсальных решений и технологий. При этом для создания гибкой расширяемой архитектуры интеграционного сервера необходима четкая декомпозиция процесса разработки и

исполнения потока работ на отдельные этапы и анализ задач, возникающих на каждом из них. Ниже рассмотрен подход к созданию интеграционного сервера, автоматизирующего весь жизненный цикл потока работ от создания до отладки и исполнения.

1.1. Формальное представление потока работ

В основе каждого автоматизируемого потока работ заложено понятие так называемой модели потока работ, которая представляет собой формализованное описание потоков работ, отражающее реально существующую или предполагаемую деятельность в рамках некоторого реального производственного процесса.

Модель потока работ должна давать ответы на вопросы:

какие процедуры (функции, работы) необходимо выполнить для получения заданного конечного результата;

в какой последовательности выполняются эти процедуры;

какие механизмы контроля и управления существуют в рамках рассматриваемого потока работ;

кто выполняет процедуры процесса;

какие входящие документы/информацию использует каждая процедура процесса;

какие исходящие документы/информацию генерирует процедура процесса;

какие ресурсы необходимы для выполнения каждой процедуры процесса;

какая документация/условия регламентирует выполнение процедуры;

какие параметры характеризуют выполнение процедур и процесса в целом.

В настоящее время разрабатываются многочисленные стандарты, целью которых является интеграция существующих методов и языков моделирования потоков работ и создание единого методического и технологического базиса моделирования автоматизированных потоков работ.

Метод SADT (Structured Analysis and Design Technique) создан Дугласом Россом (SoftTech, Inc.) еще в 1969 г. и поддерживается Министерством обороны США, которое было инициатором разработки семейства стандартов IDEF (Integrated DEFinition Methods). Метод SADT реализован в одном из стандартов этого семейства – IDEF0, утвержденным в качестве федерального стандарта США в 1993 г. Это процессный метод управления, т. е. система рассматривается именно как набор потоков работ и не строится организационно-штатная структура. Эта модель описывается как на естественных языках, так и при помощи диаграмм. В описании

данной методологии указано, что все диаграммы модели SADT взаимосвязаны и организованы в иерархию.

Вершина иерархии описывает систему в целом, это самое общее описание, а в подразделах – описания детализированные. После описания системы в целом проводится разбиение ее на крупные фрагменты. Этот процесс называется функциональной декомпозицией, а диаграммы, которые описывают каждый фрагмент и взаимодействие фрагментов, – диаграммами декомпозиции. После декомпозиции контекстной диаграммы проводится декомпозиция каждого большого фрагмента системы на более мелкие и так далее до достижения нужного уровня подробности описания.

В конце 1980-х гг. был разработан метод моделирования IDEF3, являющийся частью семейства стандартов IDEF. Предполагалось использование метода для моделирования работы ВВС США. С помощью этого метода стало возможным моделировать последовательность действий в рамках некоторого процесса. Основой модели IDEF3 служит так называемый сценарий процесса, который выделяет последовательность действий и подпроцессов анализируемой системы. Главной единицей модели IDEF3 является диаграмма. Другой важный компонент модели – действие, или в терминах IDEF3 «единица работы» (Unit of Work). Каждое действие имеет идентификатор. Существуют связи между действиями (однонаправленные стрелки) и три типа таких связей: временное предшествование, объектный поток, нечеткое отношение (вид взаимодействия каждый раз оговаривается отдельно).

Также для описания потоков работ используется моделирование потоков данных. Диаграммы потоков данных (Data Flow Diagrams – DFD) представляют собой иерархию процессов, которые связаны между собой этими потоками. Диаграммы показывают, как обрабатывает информацию каждый процесс, как процессы связаны друг с другом, а также как работает сама система, каким образом она обрабатывает поступающие данные.

Качественно новым шагом в моделировании потоков работ стало появление нотации Process Modeling Notation (BPMN), представленной консорциумом Business Process Management Initiative (BPMI) в 2003 г. Целью этого проекта является создание общей нотации для различных категорий специалистов: от аналитиков и экспертов до разработчиков ПО. BPMN-модель визуализируются с помощью диаграммы под названием Business Process Diagram (BPD).

1.2. Стандарты описания бизнес-процессов

Одной из основных целей разработки стандартов (особенно стандартов языков описаний является обеспечение переносимости разработки между различными системами (табл. 1.2, рис. 1.1). Именно такую цель заявляли разработчики BPEL (табл. 1.1). Эти стандарты нашли

свою реализацию в программных продуктах фирм, специализирующихся в разработке инструментов, автоматизирующих процесс создания программного обеспечения (табл. 1.2).

Таблица 1.1. Список популярных стандартов, связанных с описанием бизнес-процессов

Стандарт	Авторы/Текущие разработчики	Краткое описание
Business Process Modeling Notation (BPMN)	Создан *Business Modeling & Integration.* В настоящее время передан *Object Management Group (OMG)*	Система графических обозначений для наглядного визуального представления схемы бизнес-процесса. На сегодня это, пожалуй, наиболее популярная нотация для визуализации схем БП
Unified Modeling Language (UML)	Был разработан в *Rational Software*. Передан *Object Management Group (OMG)*	Для описания бизнес-процессов используются диаграммы активностей, которые несколько схожи с нотацией BPMN. После передачи BPMN в OMG, он, вероятно, целиком или частично войдет в UML
Business Process Executable Language (BPEL)	Разрабатывается *Organization for the Advancement of Structured Information Standards (OASIS)*	Представляет собой xml-нотацию для описания бизнес-процессов. Рассматривает бизнес-процесс как связанную последовательность веб-сервисов. Совместная разработка IBM, BEA, Microsoft, SAP, Siebel. Первоначально назывался BPEL4WS (Business Process Execution Language for Web Services), сейчас полное название – WS-BPEL (Web Services Business Process Execution Language). Основной недостаток – ориентация только на автоматические процессы
XML Process Definition Language (XPDL)	Разрабатывается *Workflow Management Coalition* http://www.wfmc.org/	Конкурентный с BPEL формат (XML-формат для обмена информацией между средствами анализа бизнес-процессов и BPM-системами). В отличие от BPEL, в XPDL нет жесткой привязки к веб-сервисам, а используется абстрактное понятие внешнего приложения, кроме того, имеется явное определение пользователей и ролей

Диаграмма BPD имеет два основных достоинства.

Во-первых, она проста в использовании и понимании. Применяя ее на начальном уровне сложности, можно найти общий язык с нетехническим персоналом. Во-вторых, поднимаясь на более высокий уровень сложности описания, можно постепенно подойти к естественному

отображению на языке исполнения потоков работ. Определяя достаточный уровень абстракции, нотация BPMN позволяет наглядным образом описывать модели потоков работ безотносительно среды их функционирования. На базе такой «скелетной» реализации процесса можно получать различные варианты «исполняемого кода». Помимо этого, графическая нотация BPMN 1.0 определяет ряд дополнительных функциональных возможностей, делающих ее наиболее выгодным средством для перевода реальных потоков работ в формальную объектную модель:

широкий набор элементов декларации потока работ, сочетающий простоту визуализации и богатую выразительность;

возможность экспорта в распространенных форматах описания потоков работ, таких как диаграммы активности UML 1.0 и 2.0 ввиду схожей базовой метамодели.

Таблица 1.2. Инструментальная поддержка нотаций моделирования и языков программирования

Инструмент	Нотация (Язык моделирования)	Язык программирования
Power Designer	ebXML BPSS v 1.01,1.04; BPEL4WS 1.1; WSBPEL 2.0; Analisis; BPMN 1.0; DFD; Servise Oriented Architecture; Sybase WorkSpace Business Architecture Process	Analisis; C#; C#2; C++; Eclipse Modeling Framework; IDL-CORBA; Java: Java 1.x; PowerBuilder; Visual Basic.NET 2005; XML-DTD; XML-Schema
Enterprise Architect	UML-2; ArcGIS; ArchMate; BPMN 1.0; 1.1; 2.0; DFD; CodeEngineering; ErikssonPenker; SOMF; SPEM; Strategic Modeling; User Interface; Web Modeling	Action Script; C; C#; C++; Delphi; Java; PHP; Pyton; VBNet; Visual Basic

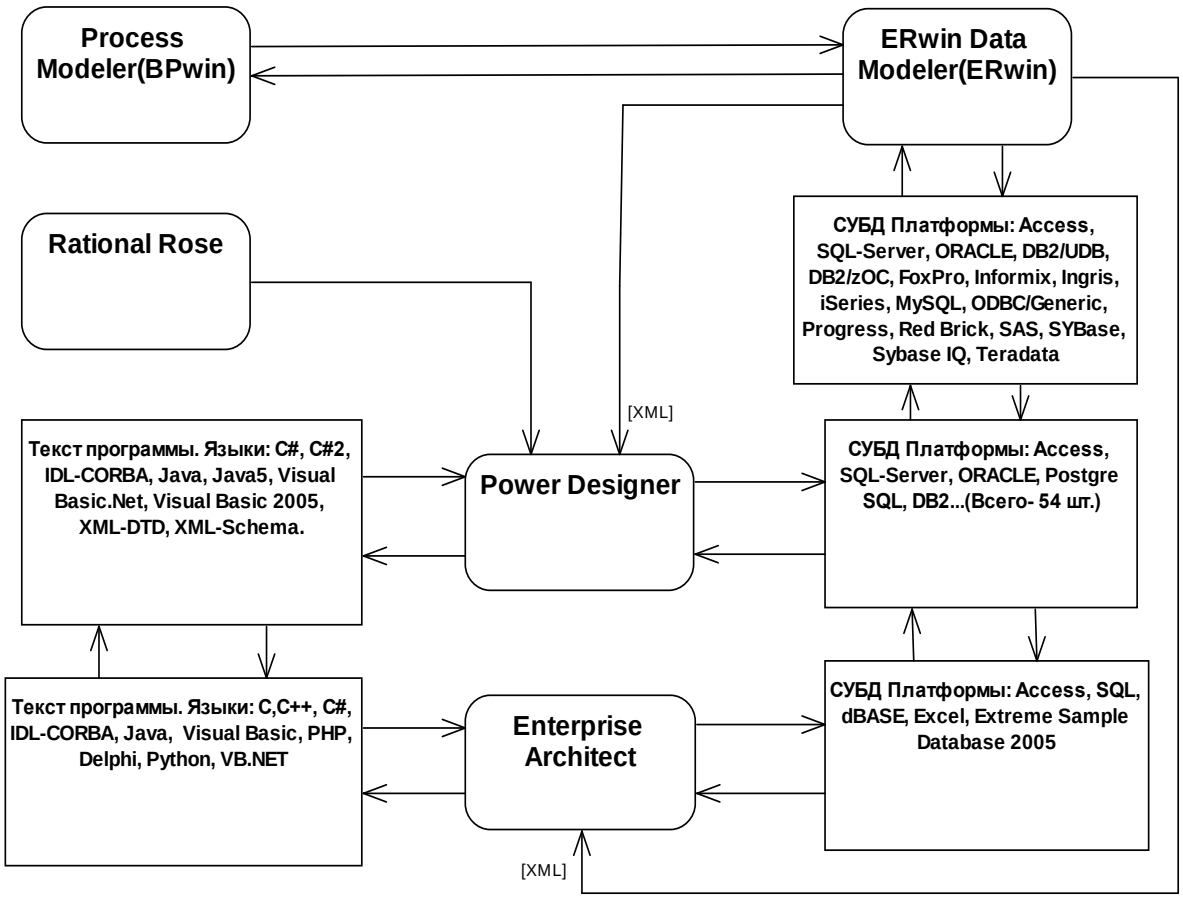


Рис. 1.1. Взаимодействие основных инструментальных средств при разработке приложений

1.3. Моделирование бизнес-процессов на основе BPMN-диаграмм

BPMN – это нотация, основанная на построении блок-схем, для моделирования бизнес-процессов, она принята как стандарт OMG в феврале 2006 г. Нотация предназначена для описания действий или активностей (activities), выполняемых организациями для управления и, если необходимо, для улучшения своих бизнес-процессов. Корпорация Oracle предлагает ряд инструментов для поддержки жизненного цикла BPM.

Проектирование новых или описание существующих бизнес-процессов. Oracle Business Process Architect, компонент Oracle Business Process Analysis Suite, может использоваться для этой задачи. Он предоставляет компании широкие возможности моделирования и проектирования бизнес-процессов, IT-систем, ландшафтов и организационных структур. Бизнес-аналитики могут моделировать бизнес-процесс, используя BPMN.

*Моделирование или имитирование бизнес-процессов***.** С использованием комплекта Oracle Business Process Analysis suite бизнес-процессы могут быть имитированы при помощи соответствующего компонента (simulation component). Имитации выполняются на основе различных сценариев.

Исполнение процесса. Корпорация Oracle как часть Oracle SOA Suite предлагает движок BPEL Process Manager и Oracle JDeveloper для создания BPEL-приложений.

*Улучшение процесса***.** Результат мониторинга процесса является входом для улучшений этого процесса, которые могут быть спроектированы, сымитированы и выполнены с использованием перечисленных выше средств.

Направление BPM возникло в результате конвергенции четырех сегментов рынка корпоративных систем:

Workflow Automation – автоматизация процессов, выполняемых людьми;

Enterprise Application Integration (*EAI*) – интеграция приложений и информационный обмен между гетерогенными системами;

Business Process Modeling and Analysis – моделирование и анализ бизнес-процессов;

Business Activity Monitoring (*BAM*) – мониторинг и анализ эффективности работы предприятия в целом и выполняемых им операций.

Большой прогресс был достигнут с появлением «движков» (engine), способных исполнять бизнес-процессы, описанные с помощью средств моделирования. По сути, это интерпретаторы языков моделирования, а очнее, их подмножества (так называемых языков исполняемых бизнес-процессов), которые функционируют на том или ином сервере приложений. Наиболее популярным из них сегодня является язык BPEL. Такой движок не выполняет задания, входящие в состав бизнес-процесса, он лишь следит за логикой маршрутизации заданий и в зависимости от соблюдения в точках ветвления заранее заданных условий передает эти задания на исполнение компьютерной программе или человеку.

BPM и стандартизация

Средства BPM призваны решить чрезвычайно непростую задачу: заставить работать множество разнородных прикладных систем, установленных на всевозможных программно-аппаратных платформах, как единое суперприложение со своими механизмами фиксации транзакций, идентификации пользователей, поддержки ролей и т. д. Сделать это без открытых, поддерживаемых всем ИТ-сообществом стандартов было бы просто невозможно. В настоящее время приняты базовые спецификации, определяющие как правила построения и выполнения самих бизнес-процессов, так и способы их взаимодействия с гетерогенной программной

средой. Но процесс стандартизации еще далеко не завершен. Ниже перечислены основные спецификации и указаны решаемые ими задачи.

Business Process Execution Language (*BPEL*) – язык, базирующийся на математической модели Pi calculus и описывающий исполнение распределенных транзакционных бизнес-процессов. Включает в себя средства для организации согласованной работы нескольких приложений (orcherstration) и обмена сообщениями между ними. Его разработка контролируется консорциумом OASIS.

Business Process Modeling Language (*BPML*) – аналогичен по своему назначению языку BPEL. Был создан под патронатом организации Business Process Management Initiative (BPMI.org) раньше, чем BPEL. После того как BPMI.org присоединилась к разработке BPEL в рамках консорциума OASIS, дальнейшее развитие BPML было приостановлено.

Business Process Modeling Notation (*BPMN*) – первый стандарт графической нотации бизнес-процесса. Разработан BPMI.org. Допускает автоматическую трансляцию в исполняемый язык BPEL и обеспечивает возможность обмена моделями, созданными разными средствами проектирования бизнес-процессов.

Business Process Query Language (*BPQL*) – язык запросов, позволяющий получать информацию о состоянии и характеристиках активных экземпляров бизнес-процессов в реальном масштабе времени (BPMI.org).

Business Activity Monitoring Language (*BAML*) – дает возможность определять метрики процессов, инструменты и фильтры их мониторинга, ключевые показатели эффективности (KPI), а также задавать способы их визуального отображения.

Business Process Audit Trail Schema (*BPATS*) – определяет стандартную структуру данных XML Schema, описывающую сериализацию экземпляров бизнес-процессов.

Business Transaction Protocol (*BTP*) – служит для координации запросов к распределенным разнородным приложениям и ответов на них в рамках комплексных бизнес-транзакций.

Главное достоинство современных BPM-систем многие видят в том, что описания процессов на языке BPEL интерпретируются «движком» без какой-либо предварительной подготовки, а это означает, что после внесения менеджером изменения в бизнес-процесс он немедленно может быть исполнен. В идеале все должно происходить без участия программиста, но не следует забывать, что BPEL – это XML-подобный язык, а потому читать и писать BPEL-описания обычному менеджеру было бы явно не по силам. На помощь здесь приходят инструменты визуального моделирования и обработки бизнес-правил, упоминавшиеся выше. Возможность быстро модифицировать бизнес-процессы и вводить их в строй таит в себе одну опасность. Дело в том, что к моменту внесения

изменений в какой-то процесс тысячи его экземпляров могут еще находиться в незавершенном состоянии и во избежание логических конфликтов нужно будет дождаться их завершения и только потом вводить в действие новую версию бизнес-процесса, а это крайне неудобно. Некоторые BPM-системы способны контролировать версии процессов и корректно выполнять одновременно разные их редакции, позволяя тем самым сосуществовать «новым» и «старым» процессам без каких-либо конфликтов.

Предшественником BPM-движков можно считать подсистемы workflow, входящие в состав многих ERP-продуктов. Но они управляют потоками работ только в рамках «своего» продукта. Если же на предприятии разные контуры управления автоматизированы с помощью разнородных приложений, такие «локальные» workflow-инструменты либо неприменимы, либо нуждаются в дополнительной интеграции друг с другом.

Нотация моделирования бизнес процессов (Business Process Modeling Notation, BPMN) была разработана Business Process Management Initiative (BPMI) и поддерживается Object Management Group после слияния организаций в 2005 г. Текущая версия BPMN – 2.0. Пример моделирования бизнес процесса в нотации BPMN 1.1 демонстрирует возможности нотации при описании оказания услуг связи (рис.1.2).

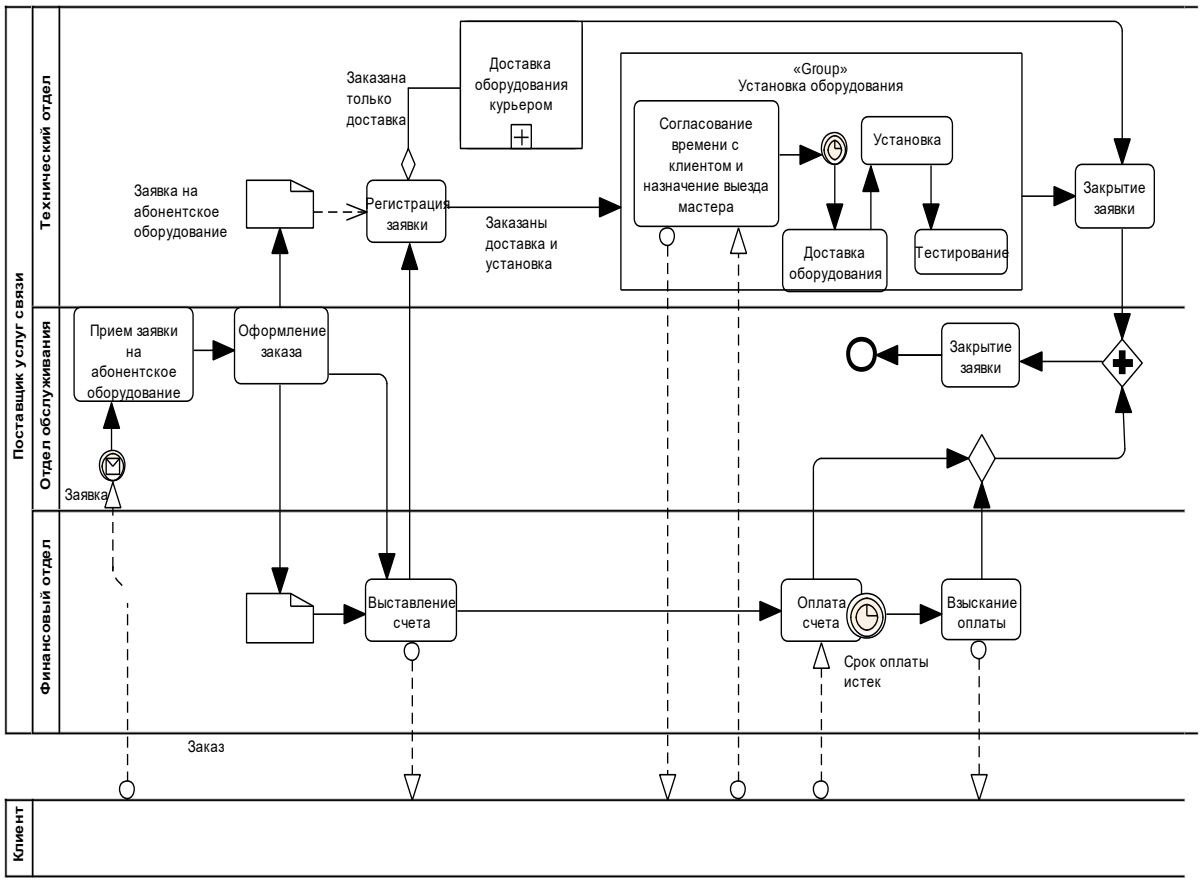


Рис. 1.2. Пример моделирования бизнес - процесса в нотации BPMN 1.1 (поставка услуг связи)

Спецификация BPMN описывает графическую нотацию в виде диаграмм бизнес - процессов (ДБП). BPMN ориентирована как на технических специалистов, так и на бизнес-пользователей. Для этого язык использует базовый набор интуитивно понятных элементов, которые позволяют определять сложные семантические конструкции. Кроме того, спецификация BPMN определяет, как диаграммы, описывающие бизнес - процесс, могут быть трансформированы в исполняемые модели на языке BPEL.

Основная цель BPMN – создание стандартной нотации, понятной всем бизнес пользователям. Бизнес-пользователи включают в себя бизнес-аналитиков, создающих и улучшающих процессы, технических разработчиков, ответственных за реализацию процессов, и менеджеров, следящих за процессами и управляющих ими. Следовательно, BPMN призвана служить связующим звеном между фазой дизайна бизнес-процесса и фазой его реализации.

В настоящий момент существует несколько конкурирующих стандартов для моделирования бизнес-процессов. Распространение BPMN поможет унифицировать способы представления базовых концепций бизнес-процессов (например, открытые и частные бизнес-процессы,

хореографии), а также более сложные концепции (например, обработка исключительных ситуаций, компенсация транзакций).

BPMN поддерживает лишь набор концепций, необходимых для моделирования бизнес-процессов. Моделирование иных аспектов находится вне зоны внимания BPMN (например, не описываются модель данных; организационная структура.)

Несмотря на то, что BPMN позволяет моделировать потоки данных и потоки сообщений, а также ассоциировать данные с действиями, она не является схемой информационных потоков.

Элементы. Моделирование в BPMN осуществляется посредством диаграмм с небольшим числом графических элементов. Это помогает пользователям быстро понимать логику процесса. Выделяют четыре основные категории элементов:

объекты потока управления: события, действия и логические операторы;

соединяющие объекты: поток управления, поток сообщений и ассоциации;

роли: пулы и дорожки;

артефакты: данные, группы и текстовые аннотации.

Элементы этих четырех категорий позволяют строить простейшие ДБП. Для повышения выразительности модели спецификация разрешает создавать новые типы объектов потока управления и артефактов.

Объекты потока управления. Объекты потока управления разделяются на три основных типа: события (events), действия (activities) и логические операторы (gateways).

События изображаются окружностью и означают какое-либо происшествие в мире; инициируют действия или являются их результатами (рис. 1.3).

Событие - нечто, что "случается" во время выполнения процесса и влияет на его ход. Событие либо начинает выполнение процесса, либо происходит во время его выполнения, либо заканчивает процесс. На рисунке слева показаны элементы для начального, промежуточного и конечного событий соответственно. В элемент события может быть вложена пиктограмма, показывающая причину события (получение сообщения, таймер, исключение и т.п.).

Рис. 1.3. Типы событий в BPMN 1.1

Согласно расположению в процессе события могут быть классифицированы на начальные (start), промежуточные (intermediate) и завершающие (end). Начиная с BPMN 1.1, различают события обработки и генерации. Ниже представлена категоризация событий по типам.

Простые события (plain events) – это нетипизированные события, использующиеся чаще всего для того, чтобы показать начало или окончание процесса.

События-сообщения (message events) показывают получение и отправку сообщений в ходе выполнения процесса.

События-таймеры (timer events) моделируют события, регулярно происходящие во времени. Также позволяют моделировать моменты времени, периоды и таймауты.

События-ошибки (error events) позволяют смоделировать генерацию и обработку ошибок в процессе. Ошибки могут иметь различные типы.

События-отмены (cancel events) инициируют или реагируют на отмену транзакции.

События-компенсации (compensation events) инициируют компенсацию или выполняют действия по компенсации.

События-условия (conditional events) позволяют интегрировать бизнес-правила в процесс.

События-сигналы (signal events) рассылают и принимают сигналы между несколькими процессами. Один сигнал может обрабатываться несколькими получателями. Таким образом, события-сигналы позволяют реализовать широковещательную рассылку сообщений.

Составные события (multiple events) моделируют генерацию одного события из множества.

События-ссылки (link events) используются как межстраничные

соединения. Пара соответствующих ссылок эквивалентна потоку управления.

События-остановы (terminate events) приводят к немедленному завершению всего бизнес-процесса (во всей диаграмме).

Действия изображаются прямоугольниками со скругленными углами (рис. 1.4). Среди действий различают задания и подпроцессы. Графическое изображение свернутого подпроцесса снабжено знаком «плюс» у нижней границы прямоугольника.

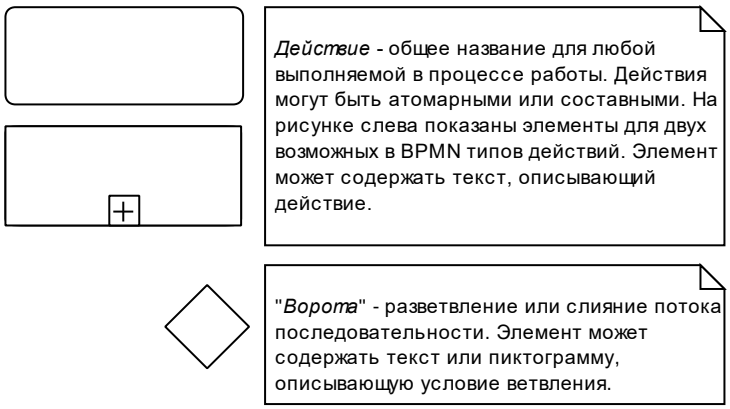


Рис. 1.4. Типы действий в BPMN 1.1

Задание (task) – это единица работы, элементарное действие в процессе.

Множественные экземпляры (multiple instances) действия показывают, что одно действие выполняется многократно, по одному разу для каждого объекта. Например, для каждого объекта в заказе клиента выполняется один экземпляр действия. Экземпляры действия могут выполняться параллельно или последовательно.

Циклическое действие (loop activity) выполняется, пока условие цикла верно. Условие цикла может проверяться до или после выполнения действия.

Свернутый подпроцесс (collapsed subprocess) является сложным действием и содержит внутри себя правильную ДБП.

Развернутый подпроцесс (expanded subprocess) также является составным действием, но скрывает детали реализации процесса.

Ad-hoc подпроцесс (ad-hoc subprocess) содержит задания. Задания выполняются до тех пор, пока не выполнено условие завершения подпроцесса.

Логические операторы изображаются ромбами и представляют собой точки принятия решений в процессе (рис. 1.5). С помощью логических операторов организуется ветвление и синхронизация потоков управления в модели процесса.

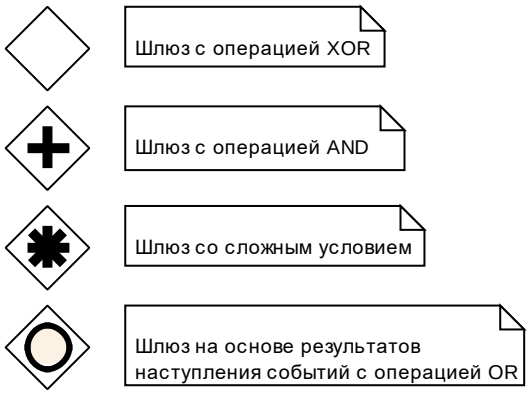


Рис. 1.5. Типы логических операторов в BPMN 1.1

Оператор исключающего ИЛИ управляемый данными (data-based exclusive gateway). Если оператор используется для ветвления, то поток управления направляется лишь по одной исходящей ветви. Если оператор используется для синхронизации, то он ожидает завершения выполнения одной входящей ветви и активирует выходной поток.

Оператор исключающего ИЛИ управляемый событиями (event-based exclusive gateway) направляет поток управления лишь по первой исходящей ветви, где произошло событие. После оператора данного типа могут следовать только события или действия – обработчики сообщений.

Оператор И (parallel gateway), использующийся для ветвления, разделяет один поток управления на несколько параллельных. При этом все исходящие ветви активируются одновременно. Если оператор используется для синхронизации, то он ожидает завершения выполнения всех входящих ветвей и лишь затем активирует выходной поток.

Оператор включающего ИЛИ (inclusive gateway) активирует одну или более исходящих ветвей в случае, когда осуществляется ветвление. Если оператор используется для синхронизации, то он ожидает завершения выполнения одной входящей ветви и активирует выходной поток.

Сложный оператор (complex gateway) имеет несколько условий, в зависимости от выполнения которых активируются исходящие ветви. Оператор затрудняет понимание диаграммы, так как условия, определяющие семантику оператора, графически не выражены на диаграмме. Вследствие этого использование оператора нежелательно.

Соединяющие объекты. Объекты потока управления связаны друг с другом соединяющими объектами. Существует три вида соединяющих объектов: потоки последовательности, потоки сообщений и ассоциации (рис. 1.6).

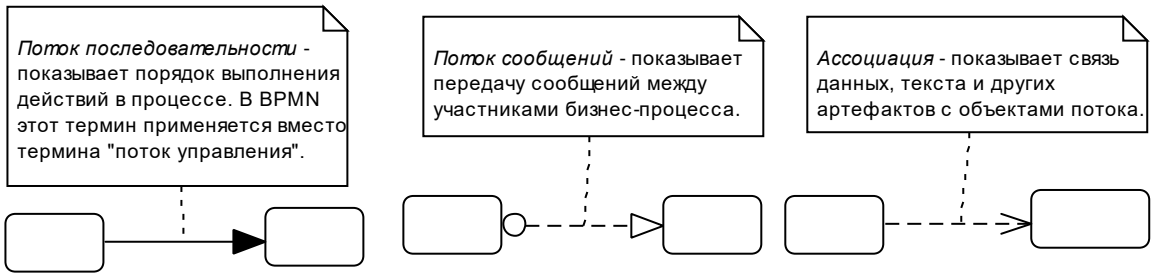


Рис. 1.6. Типы потоков управления в BPMN 1.1

Поток последовательности изображается сплошной линией, оканчивающейся закрашенной стрелкой. он задает порядок выполнения действий. Если линия потока перечеркнута диагональной чертой со стороны узла, из которого она исходит, то она обозначает поток, выполняемый по умолчанию.

Поток сообщений изображается штриховой линией, оканчивающейся открытой стрелкой. Поток сообщений показывает, какими сообщениями обмениваются участники.

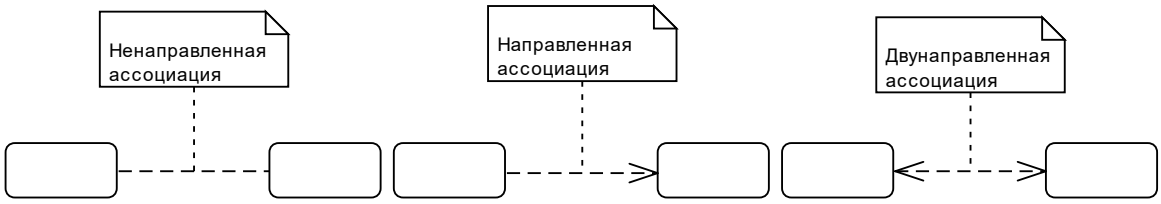


Рис. 1.7. Типы ассоциаций в BPMN 1.1

Ассоциации изображаются пунктирной линией, заканчивающейся стрелкой (рис. 1.7). Используются для ассоциирования артефактов, данных или текстовых аннотаций с объектами потока управления.

Роли. Это визуальный механизм организации различных действий в категории со сходной функциональностью. Существует два типа ролей: «бассейн» и «дорожка» (рис.1.8).

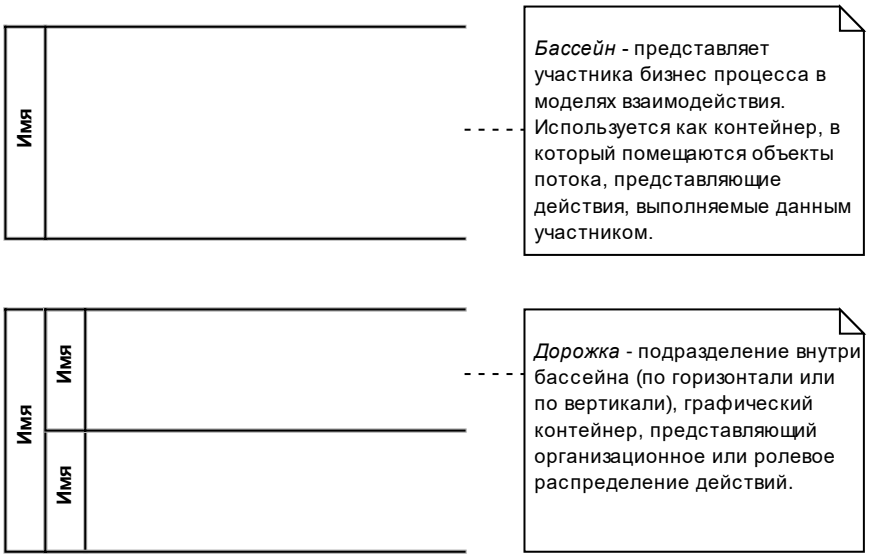


Рис. 1.8. Типы ролей в BPMN 1.1

Пулы изображаются прямоугольником, который содержит несколько объектов потока управления, соединяющих объекты и артефакты.

Дорожки представляют собой часть пула. Дорожки позволяют организовать объекты потока управления, связывающие объекты и артефакты.

Артефакты позволяют разработчикам отображать дополнительную информацию в диаграмме (рис. 1.9). Это делает диаграмму более читабельной и насыщенной информацией. Существует три предопределённых вида артефактов:

1. Данные. Показывают читателю, какие из них необходимы действиям для выполнения и какие действия производят.

2. Группа. Изображается прямоугольником с закругленными углами, граница которого – штриховая линия. Группа позволяет объединять различные действия, но не влияет на поток управления в диаграмме.

3. Текстовые аннотации. Используются для уточнения значения элементов диаграммы и повышения ее информативности.

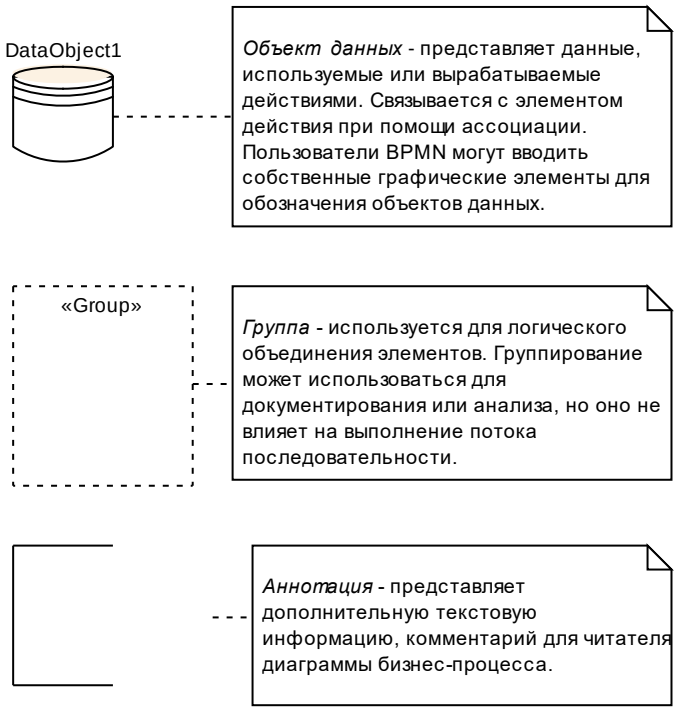


Рис. 1.9. Артефакты

Использование BPMN

Моделирование бизнес-процессов используется для донесения широкого спектра информации до различных категорий пользователей. Диаграммы позволяют описывать сквозные бизнес-процессы, но в то же время помогают читателям быстро понимать процесс и легко ориентироваться в его логике. В сквозной BPMN модели можно выделить три типа подмоделей:

частные (внутренние) бизнес-процессы;

абстрактные (открытые) бизнес-процессы;

процессы взаимодействия (глобальные).

Частные (внутренние) бизнес - процессы

Описывают внутреннюю деятельность организации. Они представляют бизнес-процессы в общепринятом понимании (business processes или workflows). При использовании ролей частный бизнес-процесс помещается в отдельный пул, поэтому поток управления находится внутри одного пула и не может пересекать его границ. Поток сообщений, напротив, пересекает границы пулов для отображения взаимодействия между различными частными бизнес-процессами.

Абстрактные (открытые) бизнес - процессы

Абстрактный процесс показывает окружающим последовательность событий, с помощью которой можно взаимодействовать с данным бизнес-процессом. Абстрактные процессы помещаются в пулы и могут моделироваться как отдельно, так и внутри большей ДБП для отображения

потока сообщений между действиями абстрактного процесса с другими элементами.

Процессы взаимодействия (глобальные)

Отображают взаимодействия между двумя и более сущностями. Эти взаимодействия определяются последовательностью действий, обрабатывающих сообщения между участниками. Процессы взаимодействия могут помещаться в пул. Эти процессы могут моделироваться как отдельно, так и внутри большей ДБП для отображения ассоциаций между действиями и другими сущностями.

Пример бизнес-процесса «Регистрация на рейс». Данный пример рассмотрен с целью показать использование конструкций нотации BPMN.

Словесное описание бизнес-процесса

Когда пассажир прибывает в аэропорт, его приоритетной задачей является регистрация на рейс. Сотрудник на стойке регистрации приветствует клиента и берет у него документы: билет на рейс и паспорт. Если документы клиента не в порядке (например, истек срок действия паспорта), он не может быть зарегистрирован на рейс и процесс завершается. При этом клиент получает документы обратно.

Если паспорт и билет в порядке, то сотрудник авиакомпании регистрирует клиента на рейс и распечатывает посадочный талон. При этом он взаимодействует с информационной системой авиакомпании. Сотрудник отдает пассажиру посадочный талон и паспорт, после чего уточняет, нет ли в багаже пассажира запрещенных грузов (например, воспламеняющихся веществ). Если таковые есть, то они изымаются из багажа. Сотрудник авиакомпании забирает багаж и ручную кладь пассажира и проводит регистрацию. При этом сотрудник снова взаимодействует с информационной системой авиакомпании. Если выясняется, что есть перевес, то сотрудник уведомляет об этом пассажира и сообщает, сколько необходимо заплатить. После получения денег от пассажира сотрудник регистрирует оплату в системе.

В итоге пассажир получает багажную квитанцию. Сотрудник желает пассажиру приятного полёта, и процесс завершается.

Модель бизнес-процесса в BPMN. Модель бизнес-процесса «Регистрация на рейс» показана на рис. 1.10.

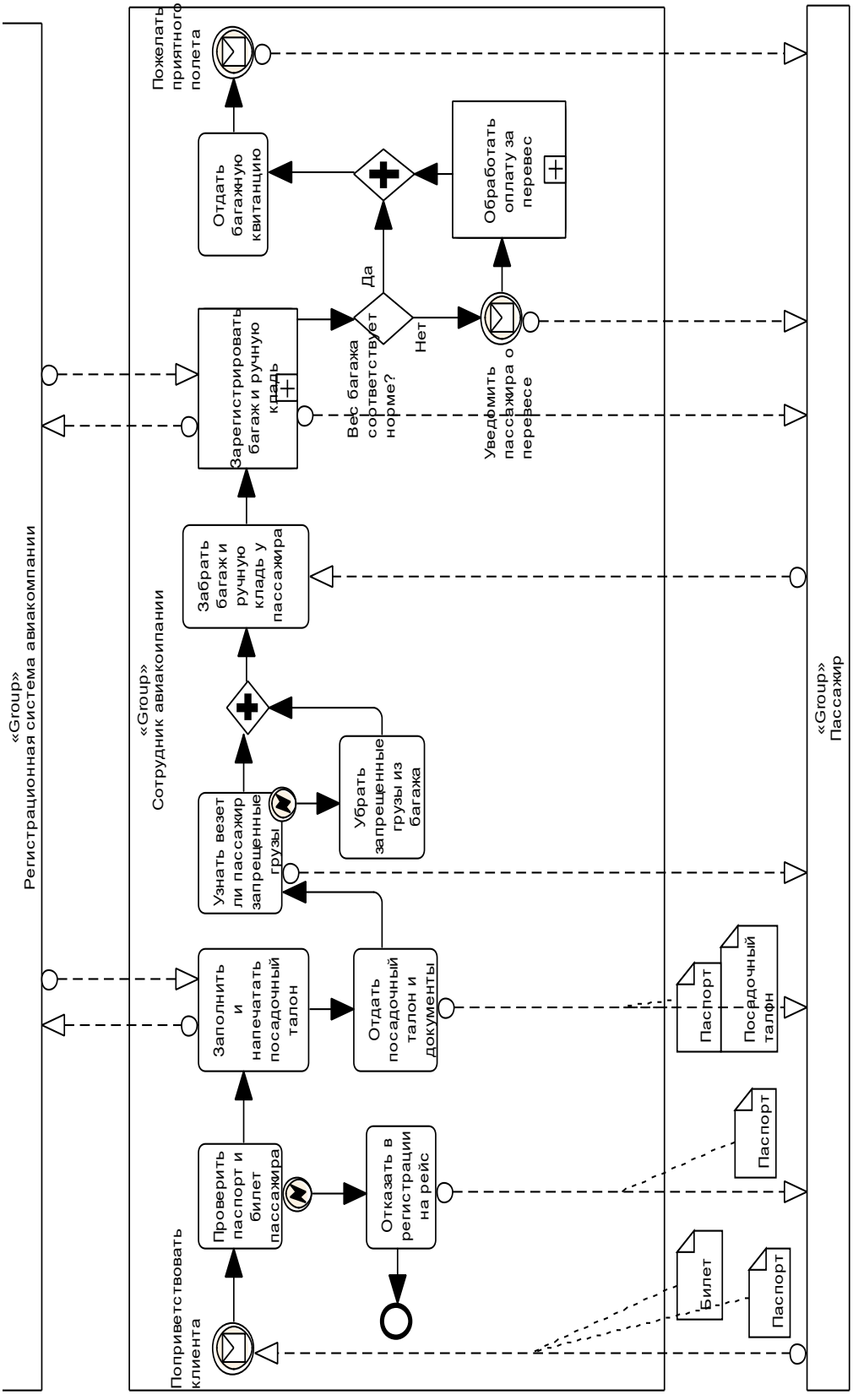


Рис. 1.10. Пример моделирования бизнес - процесса в BPMN 1.1

1.4. Стандарт для описания метамоделей MOF и стандарт обмена моделями и метамоделями XMI

Спецификация XMI (XML Metadata Interchange) [68–72], как и неразлучный с ней стандарт MOF (Meta Object Facility) [60], были созданы OMG в 2000 г. MOF сейчас существует в версии 2.0 и представляет собой набор сущностей, таких как классы, пакеты, ассоциации, типы данных и предназначен для описания метамоделей так же, как UML (Unified Modeling Language) предназначен для описания моделей. Сам язык UML является метамоделью и может быть записан на MOF, а диаграмма, сделанная на UML, является моделью, на основе которой уже создаются объекты. Это может быть проиллюстрировано с помощью классической четырехуровневой архитектуры метаданных (рис. 1.11).

Рис. 1.11. Четырехуровневая архитектура метаданных

Спецификация XMI предназначена для обмена моделями и метамоделями в рамках MOF (рис. 1.12).

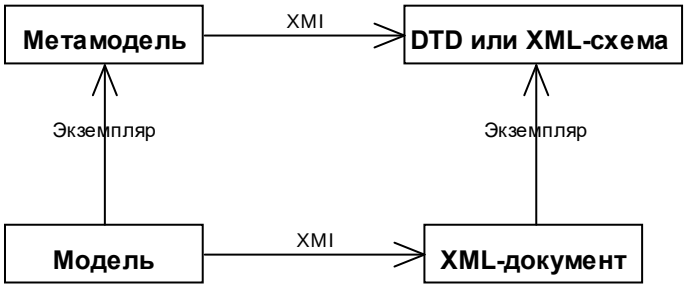


Рис. 1.12. XMI-отображения

В литературе назначения стандартов MOF и XMI описываются так: «Стандарт MOF позволяет описывать метамодели и метаданные, а стандарт XMI дает возможность представлять их единообразно в формате

XML. Возможность оперирования и обмена метаданными в едином формате позволяет программам взаимодействовать и обмениваться данными, даже если они не имеют информации друг о друге, т. е. о форматах данных, используемых другой программой. Обмен данными между разнородными источниками может управляться формальными описаниями метаданных о типах, преобразованиях и типизированных отображениях».

Отметим схожесть языков MOF и UML: у них достаточно много общих конструкций (класс, пакет, ассоциация, композиция и др.), и, скорее всего, в будущем OMG произведет слияние этих двух стандартов. В настоящее время UML предоставляет графическую нотацию для MOF. Для изображения MOF-метамоделей можно пользоваться UML-редакторами.

Язык BPEL содержит набор базовых управляющих конструкций и конструкцию для работы с данными для описания исполняемого потока работ, участниками которого являются WEB-сервисы. Для него не определена строгая графическая нотация. Поэтому задача разработки описания потоков работ на языке BPEL может быть эффективно решена с использованием декомпозиции общего процесса разработки:

разработка графического представления потока работ и его модели с помощью нотации BPMN (выполняется аналитиками);

генерация исполняемого кода на языке BPEL по полученной модели потока работ (автоматизированный этап разработки, так как спецификация BPMN 1.0 описывает способ отображения BPD диаграмм в конструкции языка BPEL);

доработка и отладка полученного BPEL-описания (выполняется техническим специалистом).

Интеграция процессов: моделирование и хореография

Компонентная модель бизнес-процесса: UML, BPMN, BPEL

Очевидно, что наборы бизнес-функций, выполняемых в составе различных бизнес-процессов, во многом должны пересекаться. Также очевидно, что при изменяющихся требованиях и внешних условиях может изменяться состав и порядок выполнения функций этого набора, а отдельные бизнес-функции этого набора могут изменяться независимо от других. Отсюда вытекает необходимость выявления компонентной структуры бизнес-процесса и создания модели бизнес-процесса как набора входящих в него компонентов и порядка их выполнения. Компоненты в такой модели будут повторно используемыми, что позволит оперативно изменять бизнес-процессы в соответствии с изменением спецификаций и конструировать новые бизнес-процессы из готовых компонентов.

Если описание компонентной модели бизнес-процесса будет формализованным, то такое моделирование, помимо вышеназванных, может обеспечивать следующие преимущества:

компонентная модель процесса может использоваться для документирования определенных этапов разработки информационной системы; исследования поведения методами эмуляции при различных характеристиках его компонентов, с различными входными данными и условиями и для оптимизации бизнес-процесса на основании этих исследований;

описание компонентов модели бизнес-процесса может служить спецификацией для разработчиков интерфейсов программных сервисов, реализующих компоненты модели, и для разработчиков программных объектов, реализующих эти сервисы;

описание взаимодействия компонентов модели может служить программой для некоторого «движка», автоматически обеспечивающего последовательность (линейную или нелинейную) выполнения компонентов как целостную единицу работы, так называемую хореографию процессов.

Полный потенциал Web-сервисов как интеграционных платформ может быть реализован только тогда, когда приложения и бизнес-процессы смогут интегрировать свои сложные взаимодействия при помощи стандартной модели интеграции.

Поскольку компонентная модель бизнес-процесса является ключевым элементом в создании эффективных и гибких информационных систем, то естественно, что были разработаны средства для создания таких моделей. Наиболее популярным из таких средств является унифицированный язык моделирования UML (Unified Modelling Language), спецификации которого поддерживаются Object Management Group – OMG. Спецификация UML (http://www.omg.org/technology/documents/profile_catalog.htm) определяет большое число диаграмм, которые можно разбить на три категории:

описания статической структуры приложения;

динамического поведения;

управления и организации программного решения.

Отметим некоторые особенности UML, которые не позволяют ему стать единственным средством моделирования бизнес-процессов.

Во-первых, UML предназначен прежде всего для архитекторов и разработчиков программного обеспечения, т. е. для специалистов в области информационных технологий. Средства UML настолько хороши для описания структуры объектов, что создают возможность автоматической генерации программного кода. UML предлагает объектно-ориентированный подход к моделированию, т. е. большинство методик применения UML требует сначала определить объекты, используя описания статической структуры, а лишь затем определять их поведение в динамике. Но такие подходы чужды для большинства бизнес-аналитиков, которым более привычно определять процесс как последовательность выполняемых действий.

Во-вторых, описание динамического поведения в UML – диаграммы деятельности (activity) и вариантов применения (use case) – не обеспечивают такой метамодели выполнения, которая могла бы использоваться для автоматического управления хореографией бизнес-процессов.

Для преодоления указанных недостатков UML был создан ряд дополнительных средств моделирования, из которых наиболее распространенными представляются нотация для моделирования бизнес-процессов (BPMN – Business Process Modelling Notation) и язык выполнения бизнес-процессов (BPEL – Business Process Execution Language).

BPMN предназначена для описания бизнес-процесса в графической форме, ориентированной на последовательность выполнения бизнес-процесса, и предназначена для бизнес-аналитиков. BPEL описывает выполнение бизнес-процесса в формализованном виде, удобном для автоматической интерпретации, и является как инструментом системных архитекторов и разработчиков сервисов-компонентов, так и средством хореографии бизнес-процесса.

UML, BPMN и BPEL не заменяют, а взаимно дополняют друг друга и используются разными специалистами, участвующими в создании информационной системы, как показано на рис. 1.13.

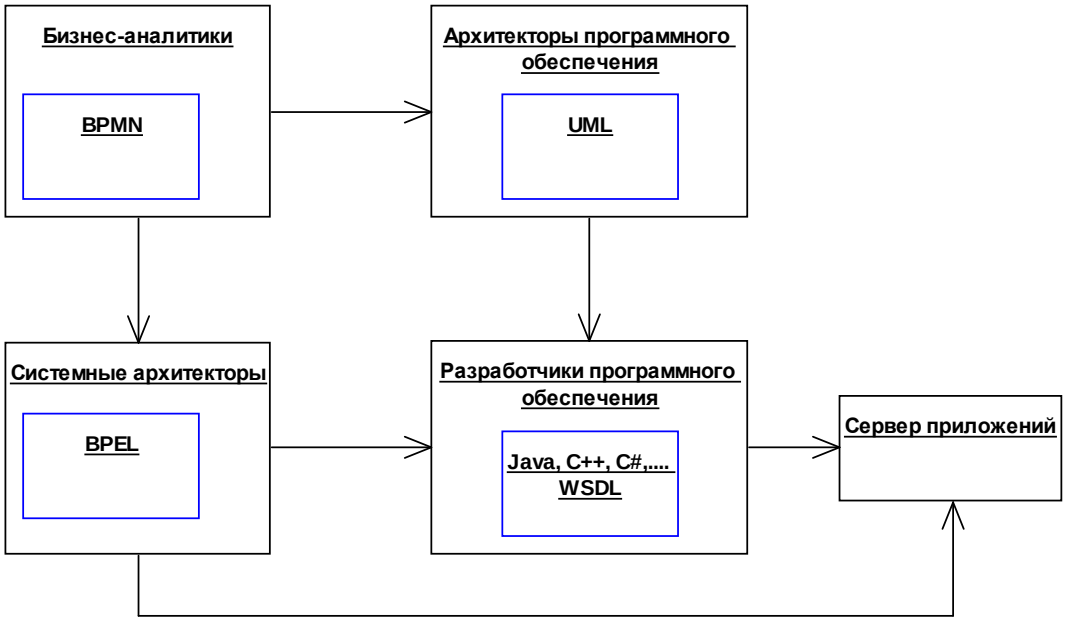


Рис. 1.13. Взаимодействие UML, BPMN и BPEL

Модель, созданная в BPMN, может быть импортирована как в UML для детализации объектной структуры и последующей разработки кодов компонентов, так и в BPEL для последующей разработки интерфейсов сервисов и поддержки выполнения бизнес-процесса на сервере приложений.

Процессы-приложения и бизнес-протоколы

Приложения, создаваемые при помощи BPEL, называются *процессами-приложениями*. Они состоят из двух уровней: верхний уровень – бизнес-процесс, написанный на BPEL и представляющий логику потока выполнения приложения, нижний уровень – Web-сервисы, представляющие функциональную логику.

Эта структура имеет целый ряд преимуществ. Во-первых, как бизнес-процесс, так и вызываемые им Web-сервисы могут быть изменены без какого-либо влияния на другие Web-сервисы в приложении или на Web-сервисы, представляемые самим процессом. Кроме того, бизнес-процесс и отдельные Web-сервисы могут разрабатываться и тестироваться раздельно.

Во-вторых, готовые процессы-приложения могут настраиваться под определенную среду выполнения без каких-либо изменений в самом приложении. Это достигается за счет отделения определения бизнес-партнеров, с которыми взаимодействует процесс, от характеристик действительных партнеров. В BPEL определяются только типы портов и операции, обеспечивать их могут различные реальные партнеры. Механизм привязки ссылки на конечную точку к реальному партнеру остается за рамками спецификации BPEL. Типичным подходом является обеспечение привязки ссылок на конечные точки во время установки (развертывания) бизнес-процесса в конкретной среде. Такая привязка обычно имеет форму дескриптора развертывания. В простейшем случае дескриптор развертывания может содержать фиксированную информацию о характеристиках реального партнера. В более сложных случаях дескриптор может содержать указание на некоторый механизм (например, UDDI), который используется для получения характеристик реального партнера.

Процессы-приложения являются переносимыми между любыми средами, поддерживающими BPEL и Web-сервисы.

С другой стороны, BPEL может использоваться также для определения *бизнес-протоколов*. Бизнес-протокол описывает последовательность обмена сообщениями между партнерами, необходимую для достижения некоторой цели. Другими словами, бизнес-протокол описывает порядок, в котором определенный партнер в определенном контексте посылает сообщения и ожидает сообщения от других партнеров. Примером механизма для определения бизнес-протокола может быть спецификация ebXML.

Обычно обмен сообщениями происходит в результате каких-то действий внутри бизнес-процесса. Бизнес-протокол может рассматриваться как внешнее представление бизнес-процесса; из этого представления исключаются внутренние детали, такие как обращения к базе данных, полная структура сообщений, составляющих контекст,

сложные операции манипулирования данными (превосходящие возможности выражений, используемых в действиях присваивания BPEL), бизнес-правила, определяющие ветвления и т. д.

В BPEL язык для определения бизнес-протоколов является подмножеством языка для определения процессов-приложений. Это позволяет определять внутренний выполняемый процесс вместе с его представлением на одном и том же языке. Этим поддерживается разработка как «снаружи внутрь» – начиная от внешнего представления и расширяя его до процесса-приложения, так и «изнутри наружу» – проецирование процесса-приложения в его внешнее представление.

В общем случае бизнес-протокол не является выполняемым. Это получается из-за того, что в нем намеренно скрываются детали и сложность бизнес-процесса. Например, сообщения, составляющие контекст протокола, могут быть упрощенной проекцией реальных сообщений, используемых в процессе-приложении; конструкция и способ передачи сообщений могут быть заданы не полностью, условия ветвления – неточно определены в терминах данных, видимых в контексте бизнес-протокола и т. п.

Поскольку бизнес-протоколы могут никогда не выполняться, в BPEL их называют *абстрактными процессами*. Они абстрагируются от сложных деталей реализации процесса. Абстрактный процесс может быть представлен как упрощенный, удобный для понимания процесс. Это дает возможность вести разработку, начиная с простейшего варианта процесса и итеративно усложняя его до получения выполняемого сложного бизнес-процесса

Пример. Формирование ресурсов электронной библиотеки. В рамках проекта по программе информатизации РАН была сформулирована задача макетирования электронной библиотеки научного наследия. При этом одной из основных задач в процессе сопровождения библиотеки является процесс подготовки электронных документов.

В качестве основы для представления цифровых копий удобно использовать формат Adobe PDF:

PDF позволяет сохранять файл после распознавания в режиме «текст под изображением», а значит, полностью исключить процедуру ручного исправления ошибок распознавания;

средствами PDF достаточно легко можно организовать полнотекстовый поиск. После распознавания текста отсканированных бумажных изданий каждая электронная копия содержит «невидимый» слой распознанного текста, по которому организовывается полнотекстовый поиск;

возможности сжатия файлов в PDF достаточны для размещения на одном CD-R необходимого количества отсканированных страниц.

Весь технологический процесс получения электронных версий бумажных изданий может быть представлен в виде ряда элементарных операций, при этом работа может быть организована по принципу конвейера: каждый участник процесса выполняет поставленную ему задачу. В целом технологический процесс можно представить в виде следующей последовательности действий:

просмотр и подготовка бумажного издания;

определение оборудования, на котором будет проходить сканирование и параметров сканирования;

сканирование (или оцифровка) бумажных изданий возможно с параллельным приведением к другим физическим формам, например, микрофишированием;

контроль сканирования, исправление ошибок (повторное сканирование «бракованных» или пропущенных страниц);

автоматическая постраничная разрезка отсканированных разворотов;

контроль автоматической разрезки и исправление ошибок;

постраничная обработка (удаление дефектов сканирования и восстановление истинных размеров страницы) и размещение электронных версий в промежуточном хранилище;

распознавание в Adobe Fine Reader целых или частей некоторых материалов;

дополнительное сжатие PDF файлов;

размещение электронной версии в постоянном хранилище, создание резервных копий;

сопровождение полученных электронных версий дополнительными метаданными (автор, аннотация, оглавление электронной публикации и т. д.);

классификация электронной версии документа (привязка к рубрикаторам и разделам электронной библиотеки);

определение параметров безопасности и шифрование PDF файла;

размещение в оперативном хранилище для online публикации;

подготовка к online публикации (индексирование, кэширование);

размещение полученной электронной версии в открытом доступе.

Крупноблочное описание данного процесса в нотации BPMN представлено на рис. 1.14.

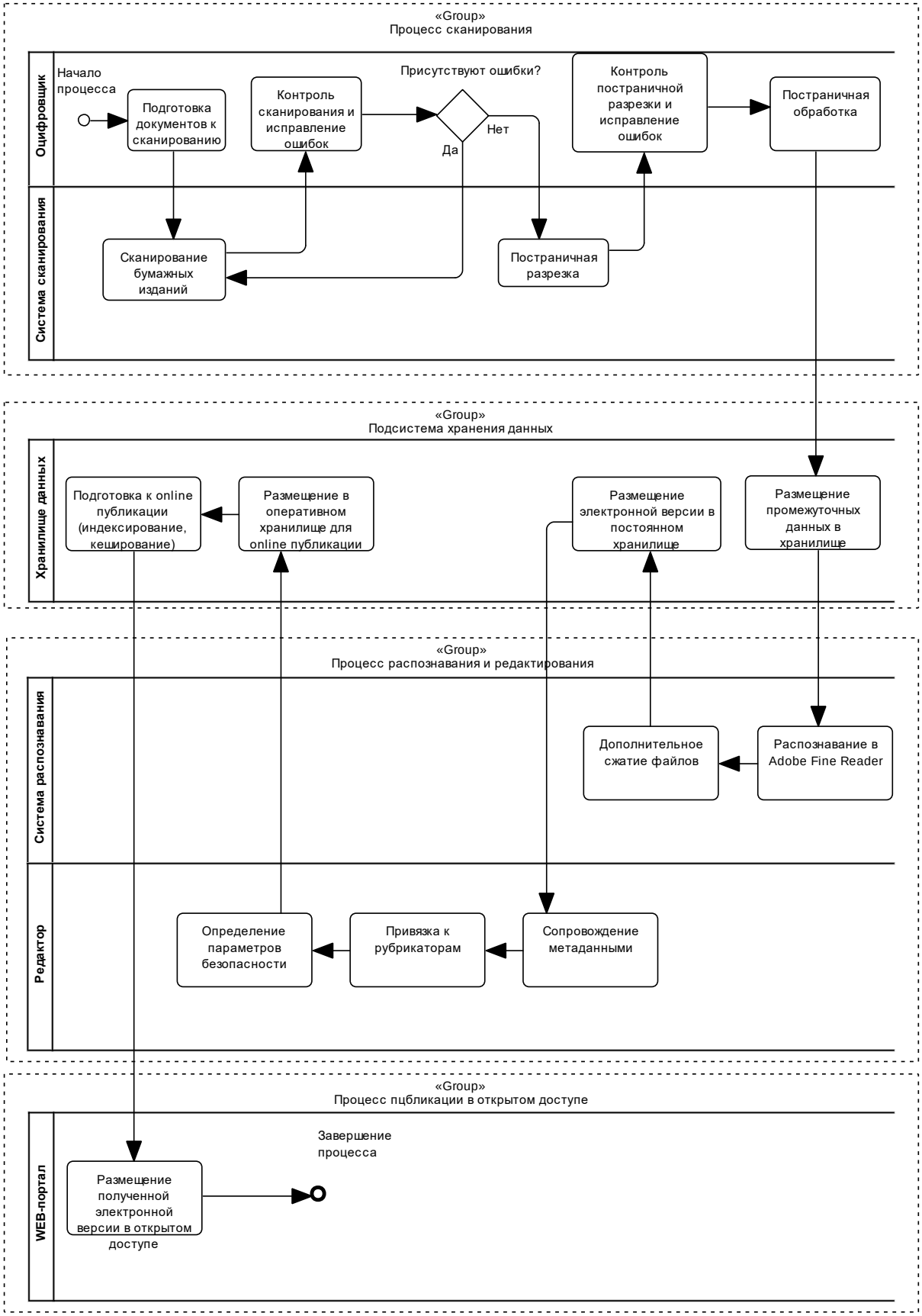


Рис. 1.14. BPD-диаграмма процесса создания электронных версий бумажных изданий

В данном процессе условно выделено четыре типа подпроцессов: сканирование бумажных носителей; операции по работе с хранилищем данных; распознавание и редактирование электронных изданий; публикация электронных изданий в открытом доступе.

Выделено несколько ролей участников.

«Оцифровщик» – технический специалист, ответственный за подготовку бумажных изданий к сканированию, управление в полученных электронных версиях.

«Система сканирования» – автоматизированная информационная система, управляющая сканирующим оборудованием и выполняющая дополнительные прикладные задачи (например, разбивка на страницы). Имеет внешний интерфейс (Web-сервис).

«Подсистема хранения данных» – автоматизированная информационная система, представленная хранилищем данных и внешним интерфейсом (Web-сервисом) для доступа к операциям извлечения и модификации информации.

«Система распознавания» – автоматизированная информационная система, выполняющая распознавание текста полученных на предыдущих этапах электронных документов и выполняющая дополнительные прикладные задачи (сжатие, шифрование файлов и т. д.). Имеет внешний интерфейс (Web-сервис).

«Редактор» – технический специалист, ответственный за управление процедурой распознавания текста, сопровождение электронных документов дополнительной метаинформацией и размещение их в открытом доступе.

«Web-портал» – Web-приложение, предоставляющее внешним пользователям доступ к ресурсам электронной библиотеки, а также ряд дополнительных сервисов.

Указанные процедуры допускают автоматизацию в виде потока работ как с полностью, так и с частично автоматизированными этапами.

Сама спецификация BPMN не определяет формата файла, в котором можно сохранять описание и которым можно обмениваться (в том числе, и с системой), однако уже есть как минимум одна спецификация, описывающая этот формат – это XPDL. В спецификации XPDL v.2.00 явно указано, что одно из ее назначений – служить описанием формата файла для нотации BPMN, т. е. XPDL позволяет хранить не только логику процесса, но и его графическое BPMN-представление. Таким образом, формат файла BPMN уже существует, что позволяет «понимать друг друга» различным инструментальным средствам моделирования.

Спецификация BPMN – это книга размером в триста страниц, которая просто заполнена графическими иллюстрациями ее применения с подробными комментариями: всего около 130 рисунков! Кроме того, спецификация содержит подробный пример описания реального

завершенного процесса – процесса разрешения разногласий с помощью голосования по электронной почте (рис. 1.15). Это процесс, который реально работал в организации BPMI при разработке самой спецификации.

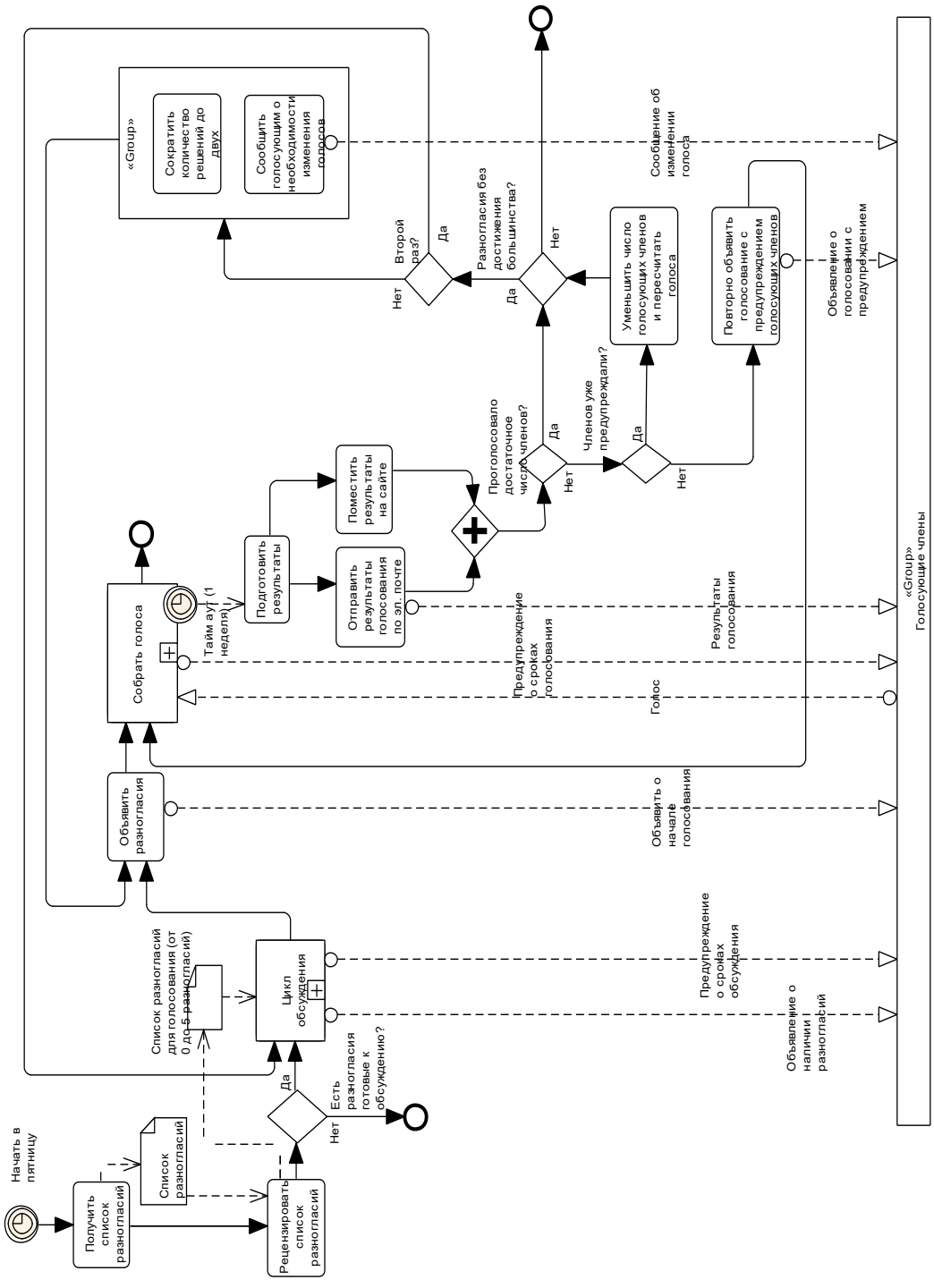


Рис. 1.15. Верхний уровень описания процесса разрешения разногласий с помощью голосования по электронной почте в нотации BPMN – реального процесса, использовавшегося при коллективной разработке самой спецификации BPMN

OMG о графической нотации моделированиябизнес-процессов

В своем выпуске 2006 г. OMG описывает видение сервисно-ориентированной архитектуры (SOA) и описаний бизнес-процессов как части архитектуры, управляемой моделями (MDA). При этом указано соответствие стандартов OMG (в т. ч. BPMN и UML) уровням моделей (уровням окружения) SOA.

BPMN, согласно OMG, применяется на самом верхнем уровне – уровне бизнес-процессов, а UML – на уровне компонентов программного обеспечения для описания интерфейсов между компонентами программного обеспечения и бизнес-сервисами.

В итоге сферы применения BPMN и UML однозначно разделены самим разработчиком обеих спецификаций. Следовательно, можно уверенно использовать спецификацию BPMN для моделирования бизнес-процессов, не боясь ее замены или вытеснения диаграммой деятельности UML.

OMG планирует использовать нотацию BPMN в метамодели (сейчас она называется «OMG-спецификации бизнес-процессов метамодели»), оперирующую понятиями уровня бизнес-процессов, а не уровня программного обеспечения. И только там, где есть пересечения с уже существующей метамоделью UML 2, предполагается использовать элементы метамодели UML 2 в метамодели BPDM. В этом, пожалуй, и состоит очень существенное отличие диаграмм BPMN от диаграмм деятельности UML: разная семантика элементов моделей. И именно поэтому правильно моделировать бизнес-процессы, используя диаграммы, обладающие соответствующим смыслом (семантикой), т. е. BPMN в данном случае.

Нарисовать «похожие» диаграммы можно и с помощью диаграмм деятельности UML, после чего эти модели могут быть экспортированы в BPEL (и в BPMN), но это уже будет моделированием «снизу вверх».

Пример. Модель функционирования торгового предприятия

1. Введение. Повышение покупательской способности населения, являясь положительным признаком развития рыночной экономики, стимулирует рост сферы розничной торговли. Розничная торговля – наиболее динамичный, быстро растущий вид торговли.

Существует недостаток в моделях анализа функционирования торговли во времени, позволяющих обосновать решения по организации работы предприятия, управлению заготовительным и сбытовым процессами с учетом особенностей покупательского спроса, тех или иных предпочтений клиентов.

2. Постановка задачи. Необходимо разработать дискретно-событийную модель функционирования розничного торгового предприятия, позволяющую обосновать стратегию управления запасами продукции и меры по организации работы торгового зала. Принципы

максимального удовлетворения потребностей покупателей и достижения высокого уровня качества обслуживания обуславливают необходимость моделирования потока покупателей и их поведения в торговом зале. Поведенческие аспекты отражаются процессной моделью поведения покупателей в торговом зале в нотации методологии BPMN, а их математическая формализация заключается в определении событийных автоматов.

3. Результаты. Структурные модели логистических систем описывают фактическую схему осуществляемых процессов и «оптимальную», т. е. ту, которая позволит повысить эффективность данной системы. Они используются в качестве основы для подготовки и принятия управленческих решений, обеспечивая общее видение и единое понимание на предприятии логистических процессов, выявление «проблемных мест», определение приоритетных задач и стоимость процессов. Структурные (процессные) модели отражают состав агентов, их функции, последовательность операций, события и используемые ресурсы. Следует добавить, что для отображения перемещений сущностей (клиентов, продукции, персонала) в торговом зале необходимо использовать инфологическую карту.

Процессная модель (BPM.1) поведения покупателя в торговом зале с закрепленными к прилавкам продавцами учитывает следующие концептуальные аспекты:

задается блок генерации покупателей на основе статических данных, полученных в ходе мониторинга торгового зала;

каждый покупатель обслуживается продавцом-консультантом, при этом оплата может производиться покупателем в кассе или через продавца-консультанта;

указываются приемлемые и неприемлемые для клиента размеры очереди. Если размер очереди имеет критическое для клиента значение, то он покидает торговый зал;

при формировании покупателем корзины допускается согласие либо отказ от приобретения товара в меньшем количестве.

Следовательно, ключевыми показателями оценки эффективности работы торгового предприятия являются:

количество упущенных клиентов вследствие возникновения критического размера очереди;

количество клиентов, чей запрос удовлетворен не в полном объеме, и соответствующий финансовый результат в виде упущенного дохода и прибыли от продажи;

количество необслуженных клиентов из-за отсутствия товара в продаже и соответствующая величина упущенного дохода и прибыли;

маркетинговая оценка потери клиентов в результате возникновения случаев неудовлетворенного спроса.

Диаграмма модели BPM.1 в нотации методологии BPMN (Business Process Modeling Notation), разработанная при помощи инструмента Enterprise Architect, представлена на рис. 1.16. Пример модели банковского процесса кредитования изображен на рис. 1.17.

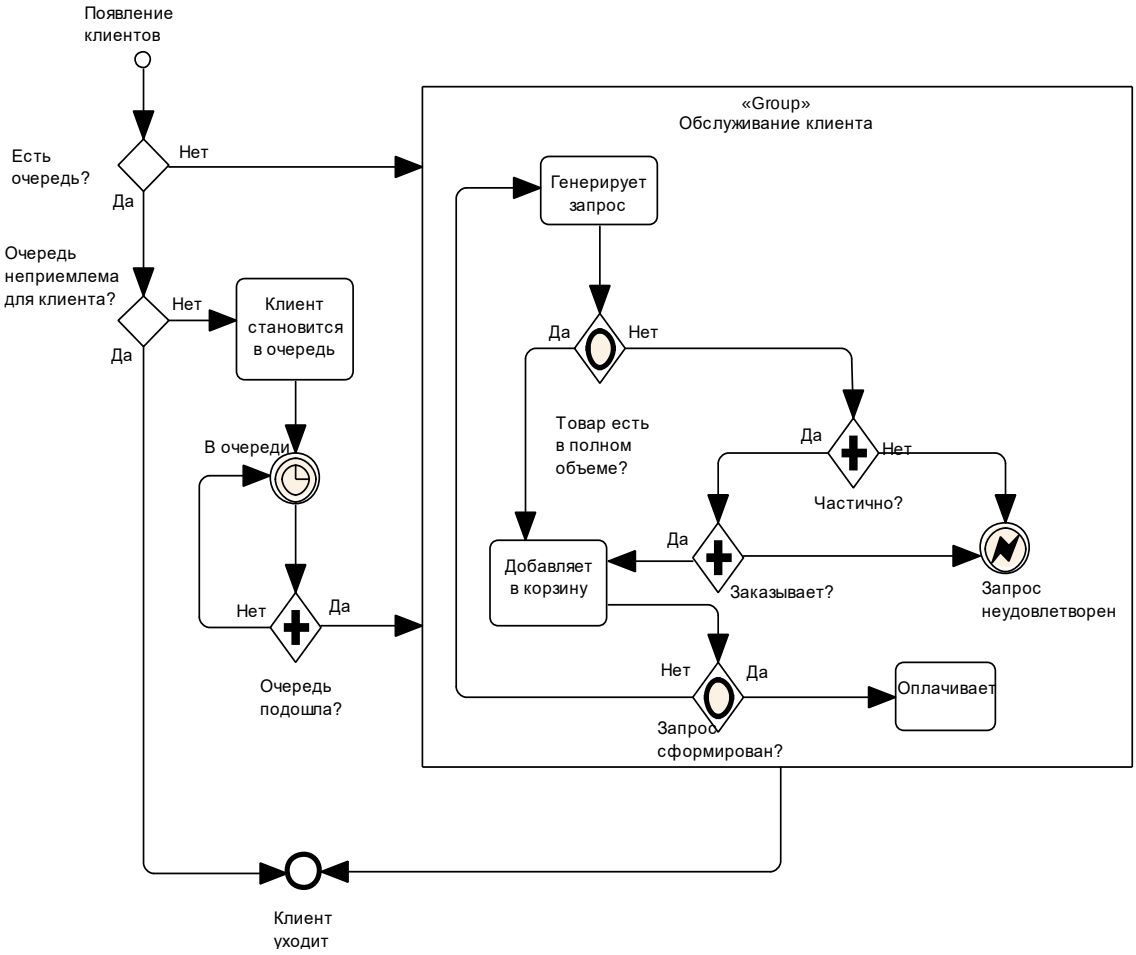


Рис. 1.16. Диаграмма процессной модели поведения покупателя в торговом зале с закрепленными к прилавкам продавцами в нотации BPMN

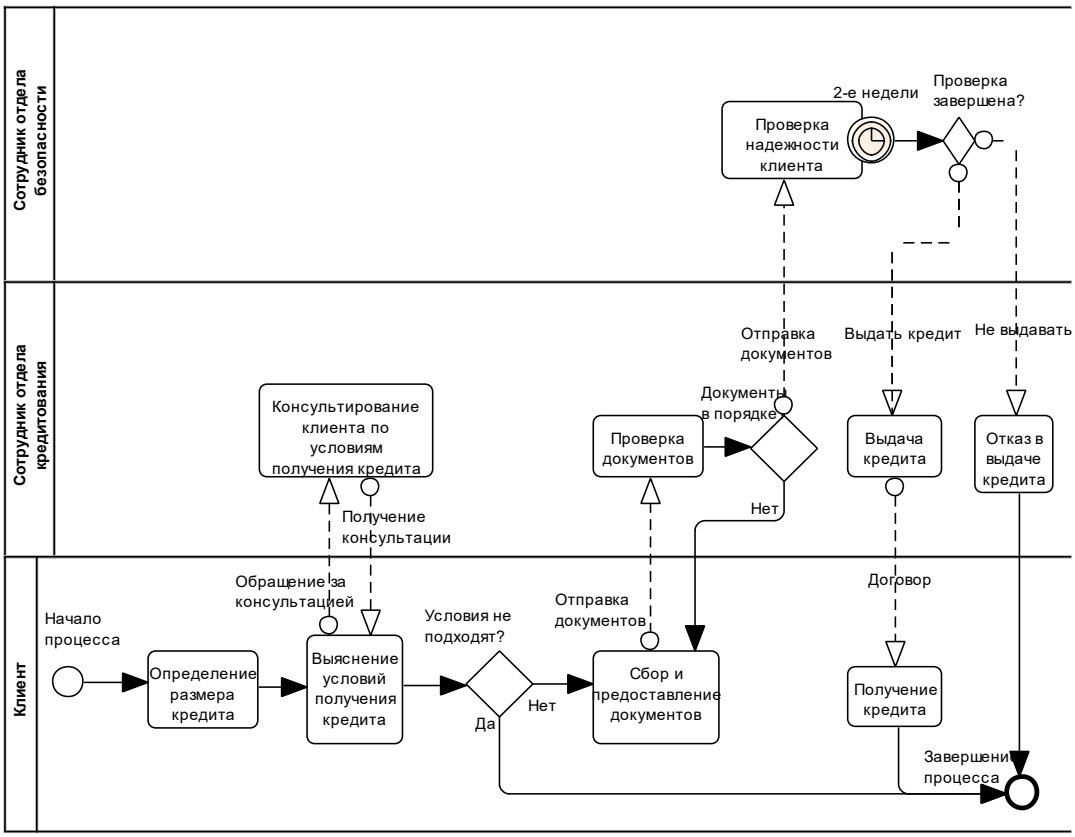


Рис. 1.17. Диаграмма модели бизнес-процесса по выдаче кредита

Формализованное описание бизнес-процессов позволяет проводить их оптимизацию, проектировать новые процессы, оптимизировать оргструктуру, совершенствовать систему управления бизнесом. Оптимизация бизнес-процессов обычно включает в себя:

изменение бизнес-логики процесса (добавление, удаление, реструктуризация процедур);

переработку форм документов, содержания нормативных документов, входов-выходов процесса;

перераспределение ответственности и исполнителей.

Приведенные примеры показывают перспективность формализации бизнеса с помощью средств визуального графоаналитического моделирования. При этом подтверждается актуальность развития оригинальной технологии моделирования в терминах «Узел – Функция – Объект» с использованием собственного инструментального средства «UFO-toolkit», так как данная технология обладает рядом полезных свойств (например, возможность автоматизации построения моделей), отсутствующих у других технологий. Кроме того, практически полезным и наукоемким в настоящее время является разработка визуальных графоаналитических моделей с помощью спецификации BPMN, так как данная спецификация предусматривает возможность имитации исполнения бизнес-процессов.

ГЛАВА 2. ОПИСАНИЕ БИЗНЕС-ПРОЦЕССОВ КАК ОДНОГО ИЗ ЭТАПОВ АВТОМАТИЗАЦИИ

За прошедшие годы индустрия информационных технологий не только разработала и выпустила в виде спецификаций новые методы описания бизнес-процессов (и соответствующие диаграммы), но и реализовала возможность автоматизированных систем *исполнять бизнес-процессы*. Сегодня существуют не только коммерческие «движки исполнения бизнес-процессов», но и аналогичные продукты. Эти перемены позволяют приблизить людей бизнеса к автоматизированным системам, сократить время и затраты на автоматизацию и т. п. Для нас в рамках данной главы важно то, что «формирование модели (описания) бизнес-процессов» – это не конечная цель (проекта, клиента ...), а лишь один из шагов к системе, исполняющей (полностью или частично) данные бизнес-процессы.

2.1. Способы описания бизнес-процессов

Текстовое описание

Описание бизнес-процессов в текстовой форме является самым неудобным способом, потому что сплошной текст не позволяет проверять правильность описания бизнес-процессов и затрудняет его анализ. Лучше всего использовать вместо текстового графическое или описание в табличной форме.

Табличная форма

Эта форма является удобной для небольших компаний, потому что в этом случае нет необходимости в закупке инструментальных средств описания бизнес-процессов. Если в таблице четко определить поля: название функции, исполнитель бизнес-процесса, входящие документы, исходящие документы, время исполнения, логика маршрутизации, то описание процесса может быть приемлемым для использования.

Простая графическая форма

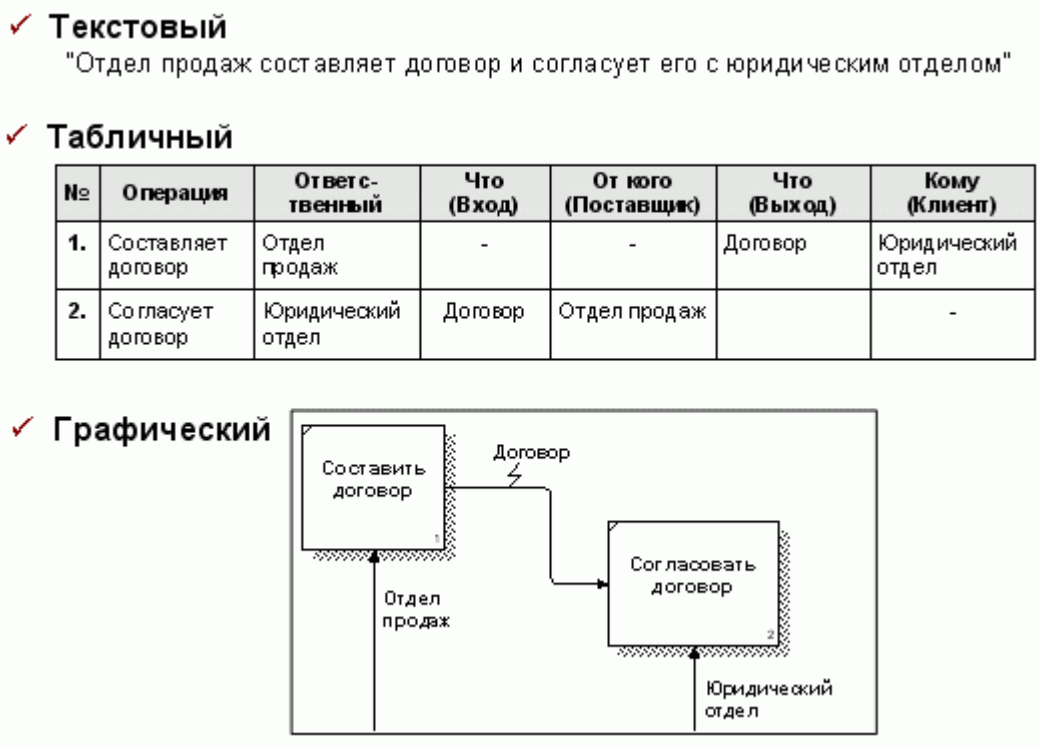
Графическая форма описания бизнес-процессов считается наиболее эффективной. Простая графическая форма может быть использована в продуктах MS Power Point и MS Visio. В крупном проекте описания деятельности может быть создано более 1 000 моделей, что потребует единого хранилища и специализированных средств анализа бизнес-процессов.

Специализированная графическая форма

Для крупных проектов по описанию бизнес-процессов могут быть использованы специализированные инструментальные средства, такие как All Fusion Modeling Suit, Power Designer, ARIS, Casewise, Telelogic, Oracle

BPA Suite. Однако высокая стоимость данных продуктов требует серьезного бюджета, что окупается их богатой функциональностью.

Основные подходы к горизонтальному описанию бизнес-процессов представлены на рис. 2.1.



№	Операция	Ответс-твенный	Что (Вход)	От кого (Поставщик)	Что (Выход)	Кому (Клиент)
1.	Составляет договор	Отдел продаж	-	-	Договор	Юридический отдел
2.	Согласует договор	Юридический отдел	Договор	Отдел продаж		-

Рис. 2.1. Способы описания бизнес-процессов

2.2. Классическая методология описания бизнес-процессов

После описания окружения бизнес-процесса наступает очередь описания его внутренней структуры. При вертикальном описании видно, из каких работ состоит бизнес-процесс. На этапе горизонтального описания описываются взаимодействия между работами, включая материальные и информационные потоки. В настоящее время существует несколько десятков подходов или стандартов описания бизнес-процессов – *ARIS*, *IDEF*0 и др.

Кажущаяся на первый взгляд сложность описания бизнес-процессов является преувеличенной. Классическая технология описания бизнес-процессов, которая была разработана на заре рождения процессных технологий управления, достаточно проста и состоит всего лишь из двух стандартов – *DFD* и *WFD*. Большинство остальных современных стандартов, несмотря на другие названия, представляют собой небольшие разновидности и дополнения этих двух классических подходов. Согласно классическому подходу стандарт *DFD* (*Data Flow Diagram)* – это диаграмма потоков данных, которая используется для описания бизнес-

процессов верхнего уровня. В свою очередь стандарт *WFD* расшифровывается как *Work Flow Diagram* и представляет собой диаграмму потоков работ, которая используется для описания бизнес-процессов нижнего уровня. У диаграммы потоков работ имеются и другое название – диаграмма алгоритмов. Рассмотрим два этих стандарта, составляющих классическую методологию описания бизнес-процессов.

Декомпозиция бизнес-процесса

При построении *DFD*-схемы бизнес-процесса необходимо использовать правило «7», согласно которому нужно выбрать такой уровень абстрагирования и детализации, при котором схема будет состоять в среднем из семи работ.

Если для достижения целей оптимизации бизнес-процесса потребуется большая его детализация, ее нужно будет сделать посредством проведения декомпозиции работ, составляющих процесс. Для этого каждую или некоторые работы процесса рассматривают как подпроцесс и описывают в виде отдельной схемы бизнес-процесса второго уровня (рис. 2.2).

При классическом подходе описания бизнес-процессов для разработанной схемы второго уровня может использоваться как *DFD*-, так и *WFD*-формат описания в зависимости от уровня и глобальности работы. Если работа глобальна и ее невозможно представить в виде временной последовательности более мелких работ, то используют *DFD*-стандарт ее описания. В противном случае работу целесообразно описать посредством *WFD*-модели.

В случае необходимости работы на схеме процесса второго уровня могут быть декомпозированы на схемы бизнес-процессов третьего уровня и т. д. В данном случае удобно использовать понятия «вложенный процесс» или «подпроцесс», что показано на рис. 2.2: процессная схема работы 3 является вложенным процессом или подпроцессом процесса верхнего уровня. Аналогичным образом процессные схемы работ 3.1 и 3.4 являются вложенными процессами или подпроцессами процесса второго уровня.

В итоге описание бизнес-процесса представляет собой иерархически упорядоченный набор *DFD* и *WFD*-схем, в котором схемы верхнего уровня ссылаются на схемы нижнего уровня. При этом схемы *DFD*, используемые на более высоких уровнях декомпозируются или ссылаются на схемы *DFD* и *WFD*. Схемы *WFD*, используемые на более низких уровнях, декомпозируются или ссылаются только на схемы *WFD*.

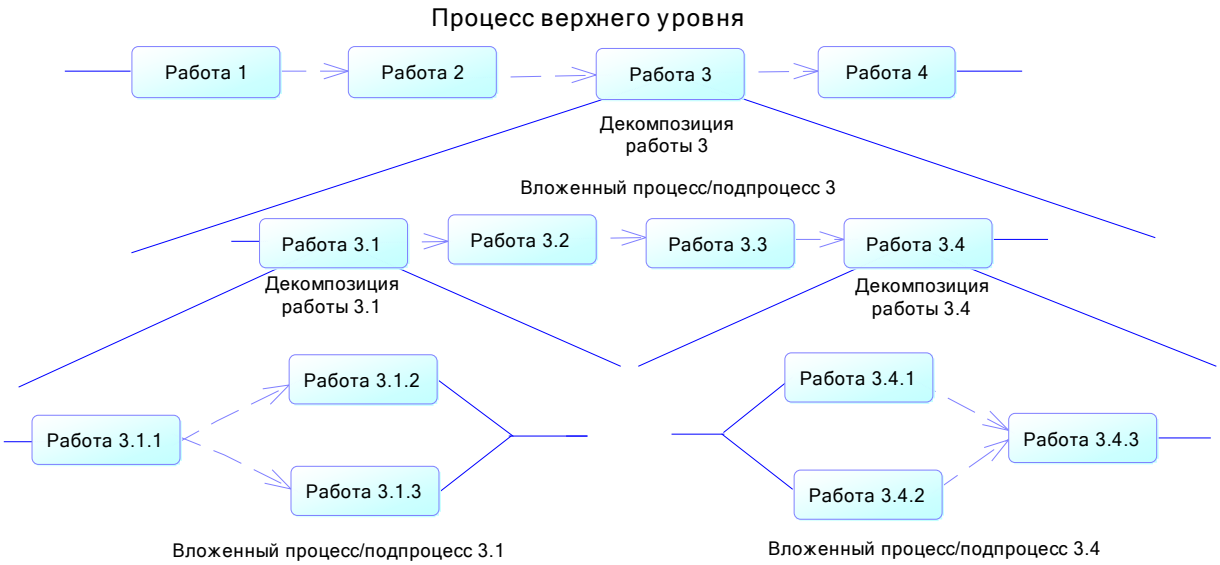


Рис. 2.2. Декомпозиция бизнес-процесса

2.3. Современные методологии описания бизнес-процессов

Особенности построения многоуровневых информационных систем на основе структурной и объектно-ориентированной декомпозиций

Информационные системы, призванные осуществлять процессы поддержки принятия и исполнения решений на всех уровнях управления хозяйственным субъектом и ориентированные на одновременное использование несколькими пользователями, в литературе получили название многоуровневые ИС.

Построение многоуровневых ИС характеризуется наличием двух основных подходов. Первый подход называют функционально-модульным или структурным, второй – объектно-ориентированным (рис. 2.3).

В основу структурного подхода проектирования ИС положен принцип функциональной декомпозиции, в соответствии с которым производится разделение функций системы на модули по функциональной принадлежности, где каждый модуль выполняет определенную последовательность действий в общем процессе. Рассматриваемая ИС делится на подсистемы, подсистемы – на комплексы задач и т. п.

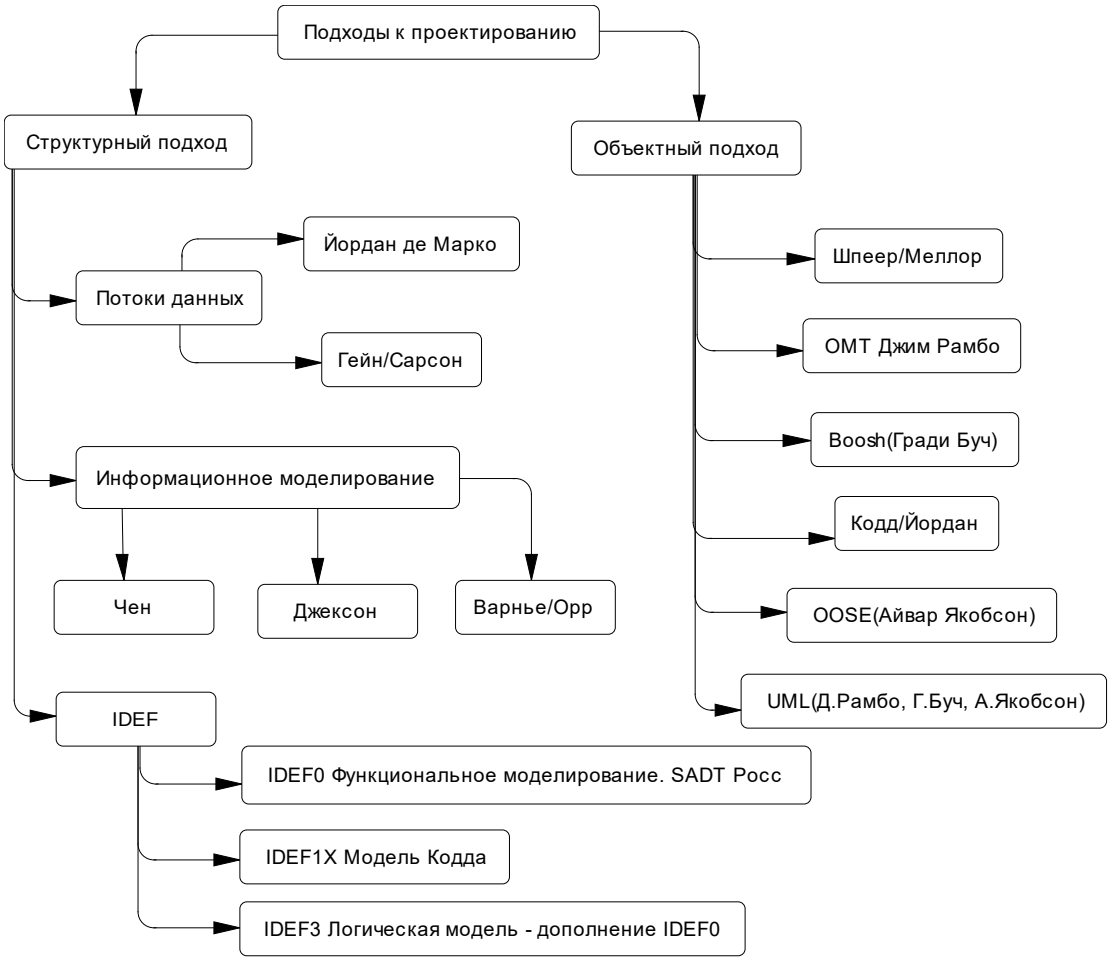


Рис. 2.3. Подходы к проектированию информационных систем

Среди методологий, ориентированных на функционально-модульный подход, наиболее распространены: IDEF; ориентированные на потоки данных (Гейн-Сарсон, Йодан); информационного моделирования, основанные на моделях Джексона, Чена и Варнье-Орра.

IDEF (ICAM DEFininition) была разработана в середине 70-х гг. в рамках программы ICAM (Integrated Computer Aided Manufacturing) для военнокосмических сил США, а затем была принята как стандарт Министерства обороны США. Она представляет собой семейство независимых, но дополняющих друг друга методологий, основанных на графическом представлении информации, и включает механизмы построения логических и семантических моделей данных, состоящих из методологий: IDEF0 – метод функционального моделирования, в основе которого лежит методология структурного анализа и проектирования SADT (*Structured Analysis and Design Technique),* предложенная Дугласом Россом в 1973 году; IDEF1 – метод информационного моделирования, основанный на ER (EntityRelationship) – моделях Чена; IDEF1X – расширение IDEF1 на основе дополнения методологиями Т. Кодда,

П. Чена; IDEF2 – метод создания динамической модели системы, основанный на цветных сетях Петри (CPN – Colored Petri Nets) и др.

Модель многоуровневой ИС на базе SADT основывается либо на функциях (активностная модель), либо на сущностях предметной области (модель данных). Для полного описания сложной системы необходимо построить множество активностных моделей и моделей данных. Наиболее часто используются активностные модели. Результатом анализа является диаграмма, которая состоит из элементов двух типов: блоки, изображающие активности ИС; дуги, связывающие блоки и обозначающие взаимосвязи между ними.

Модель SADT объединяет диаграммы в иерархические древовидные структуры. Каждый функциональный блок может быть представлен в виде композиции других взаимосвязанных блоков. Некоторые из этих блоков могут быть представлены более детально, т. е. чем выше уровень диаграммы, тем меньше уровень детализации. Основным достоинством SADT является простота и удобство в использовании.

В основе IDEF0 лежит принцип декомпозиции процессов и данных. Функциональная модель отражает архитектуру многоуровневой ИС на основе взаимосвязанных модулей и состоит из диаграмм, фрагментов текста и глоссария, имеющих ссылки друг на друга.

Метод IDEF0 характеризуется постепенным введением все больших уровней детализации по мере создания диаграмм, отображающих модель. Таким образом обеспечивается представление информации, и пользователь располагает хорошо очерченным предметом изучения с приемлемым объемом новой информации, представленной на каждой следующей диаграмме.

Модель проектируемой ИС начинается с представления всей системы в виде простейшей компоненты блока и дуг, изображающих интерфейсы с функциями вне системы. Затем данный блок. представляющий многоуровневую ИС в качестве единого модуля, детализируется на другой диаграмме с помощью нескольких блоков, соединенных интерфейсными дугами. Эти блоки представляют собой основные подфункции (подмодули) единого исходного модуля. Каждый из этих подмодулей декомпозирован подобным образом для более детального рассмотрения. Таким образом рассматриваются все аспекты функционирования ИС.

IDEF1X – это расширенная версия методологии моделирования IDEF1, получившей название «концептуальная схема». Целью данного метода является выработка непротиворечивого интегрированного определения семантических характеристик данных. С помощью IDEF1X осуществляется концептуальное проектирование баз данных в форме одной модели или нескольких локальных моделей функционального взгляда, которые относительно легко могут быть отображены в любую

систему управления базами данных. IDEF1X использует подход «сущность – связь» (сущностей отношений) к семантическому моделированию данных и представляет комбинацию реляционной теории Т. Кодда, методологии «EntityRelationship» и диаграммы «сущности отношения» П. Ченна.

Метод IDEF1X содержит набор правил и процедур для построения информационной модели. Моделирование IDEF1X представляет собой процесс, состоящий из ряда стадий, каждая из которых завершается поддающимися оценке результатами, конкретными проектными материалами. Такой порядок моделирования в отличие от других подходов обеспечивает модульность как процесса, так и его результатов, что позволяет избежать некорректности, неполноты, противоречивости и неточности.

Информационная модель содержит две основные компоненты:

диаграммы, отображающие структурные характеристики модели, обеспечивая выразительное представление информации;

словари, содержащие смысловое значение каждого элемента модели, выраженное с помощью краткого текста, и обозначения (указателей), которые в совокупности точно определяют информацию, отображенную в модели.

Результатом моделирования IDEF1X является структурно проработанная информационная модель. Она характеризуется взаимосвязью сущностей (объектов), отношениями (связями между объектами) и атрибутами (характеристиками объектов). Каждая сущность в моделях полностью определена и идентифицирована через свои атрибуты. Модель характеризуется простыми отношениями, и каждая сущность проявляется в модели только один раз. Совокупность функциональной и информационной моделей обеспечивает реальное отображение процесса функционирования несложной ИС.

В методологиях, ориентированных на потоки данных, центральное место занимают диаграммы потоков данных – ДПД (DataFlow Diagrams – DFD). Метод потоков данных получил широкое распространение как в структурных методологиях (методология Гейна – Сарсона, основанная на ДПД и методология Йордана – Де Марко, где ДПД занимает центральное место), так и в объектно-ориентированных (методология объектно-ориентированного анализа Шлеер – Меллора, где используется расширенный вариант ДПД, и методология OMT (Object Modeling Technique – метод объектного моделирования).

Модель потоков данных строится при помощи небольшого набора логических абстракций внешних сущностей, моделирующих источники и приемники данных; процессов, преобразующих данные; накопителей, хранящих данные; информационных потоков, связывающих внешние сущности, процессы и накопители.

Отдельно взятый процесс может быть представлен более детализированной диаграммой потоков данных. Подобная декомпозиция продолжается до тех пор, пока не будет достигнут такой уровень, при котором каждый процесс становится элементарным, непригодным для дальнейшей детализации.

Использование модели потоков данных как центральной в анализе ИС непригодно для крупных ИС, поскольку при изменении требований или ошибках, совершенных при создании диаграмм верхнего уровня, но обнаруженных при рассмотрении нижнеуровневых диаграмм, приходится по инерции переделывать целые поддеревья со всеми их подуровнями в иерархии ДПД.

Методологии информационного моделирования предназначены для проектирования схем баз данных и структур данных. Наиболее широкое распространение получили метод ER (Entity Relationship) и методологии, построенные на его основе (методологии, ориентированные на концептуальное моделирование).

При проектировании сложных многоуровневых ИС целесообразно использовать *объектно-ориентированный подход*. Особенность данного подхода заключается в описании взаимодействующих объектов многоуровневой ИС. При этом каждый объект ИС обладает собственным поведением, моделирующим поведение реального объекта. Многоуровневая ИС рассматривается как совокупность объектов, взаимодействующих друг с другом путем посылки сообщений. Разделение ИС на слабосвязанные части позволяет разрабатывать их практически независимо друг от друга. Применение объектно-ориентированной методологии создает большие удобства в планировании и управлении разработкой проекта ИС. К основным понятиям объектно-ориентированного подхода следует отнести объект, экземпляр объекта, класс.

Объект – это такая абстракция множества предметов реального мира, при которой все предметы этого множества (экземпляры объекта) имеют одинаковые характеристики и правила поведения. Объект представляет собой типичный и неопределенный экземпляр некоторого множества предметов реального мира. Иногда вместо термина «экземпляр объекта» употребляют слово «объект», и значение этого понятия можно определить из контекста.

Класс – это множество объектов, связанных общностью структуры и поведения (например, класс расчетно-денежных документов). Таким образом, объект – это типичный представитель класса, а экземпляр объекта – конкретный элемент класса. С точки зрения анализа информационной модели понятия «описание класса» и «описание объекта» эквивалентны, так как для описания множества схожих элементов (т. е. класса) достаточно описать его типичного представителя (т. е. объект). Тем не

менее в описанных в литературе методологиях понятия «класс», «объект», «экземпляр объекта» иногда смешиваются, и смысл того или иного понятия выявляется на основании анализа контекста.

Объекту присущи три основные свойства: инкапсуляция; наследование; полиморфизм. *Инкапсуляция* – объединение идей абстрагирования данных и алгоритмов для работы с ними. Объекты наделяются некоторой структурой и обладают определенным набором операций (методов), т. е. поведением. Внутренняя структура объекта скрыта от пользователя; манипуляция объектом, изменение его состояния возможны лишь посредством его методов. Таким образом, благодаря инкапсуляции объекты можно рассматривать как самостоятельные сущности, отделенные от внешнего мира. Для того чтобы объект произвел некоторое действие, ему необходимо извне послать сообщение, которое инициирует выполнение нужного метода. *Наследование* – построение новых классов на основе существующих с наследованием данных и методов и с возможностью добавления новых. *Полиморфизм* – возможность единообразного обращения к объектам при сохранении уникальности поведения каждого из них. Различные объекты могут получать одинаковые сообщения, но реагировать на них по-разному, в соответствии с тем, как реализованы у них методы, реагирующие на эти сообщения. Например, объект класса «линия» отреагирует на сообщение «нарисовать» рисованием линии, тогда как объект класса «окружность» – рисованием окружности.

Применение объектно-ориентированной методологии охватывает все этапы жизненного цикла многоуровневой ИС. В настоящий момент, как правило, используются объектно-ориентированные методологии Шлеер – Меллора, Рамбо (OMT Object Modeling Technique), Буча, Кода – Йордана и Якобсона (OOSE).

Методология OMT представляет собой методологию анализа и проектирования. В ней выделяют три основных этапа: анализ, системное и объектное проектирование. Анализ характеризуется *моделью объектов,* включающей диаграмму классов (описание классов и отношений между ними), диаграмму объектов (описание связей между экземплярами объектов) и словарь данных (словесное описание предметной области в виде обыкновенного толкового словаря); *моделью поведения*, включающей диаграммы перехода состояний (описание жизненных циклов объектов всех классов), диаграммы взаимодействия объектов (диаграмма потоков событий); *функциональной моделью*, включающей диаграммы потоков данных и ограничения функциональной модели.

Системное проектирование характеризуется разбиением системы на подсистемы; проектированием архитектуры программной системы; распределением множества подсистем на множество задач операционной системы; выбором стратегии хранения данных в терминах структур

данных, файлов и баз данных (проектирование схемы базы данных); определением глобальных ресурсов и механизма управления доступом к ним.

Объектное проектирование включает детальное описание классов, отношений между классами, событий, состояний, действий, данных, алгоритмов. Это описание используется на этапе кодирования ИС. Осуществляется комплектование программных модулей, содержащих реализации классов и отношений между ними.

В отличие от OMT в методологии Шлеер – Меллора существенное внимание уделяется таким вопросам, как анализ динамики связей между объектами; взаимодействие объектов и их параллельное функционирование; выделение отдельных доменов; разбиение доменов на подсистемы; преобразование моделей объектно-ориентированного анализа в модели объектно-ориентированного проектирования; создание не зависящей от языка нотации объектно-ориентированного проектирования.

Появление за последнее десятилетие множества программных продуктов, поддерживающих различные методологии проведения объектного анализа, характеризовались тем, что используемые в них методологии разрабатывались разрозненно и не обеспечивали целостного описания предметной области как некоего результата анализа. Однако пользователи желали получить такой инструмент объектного моделирования, который бы отвечал их потребностям в той же мере, что и методология SADT в структурном анализе. Так, середина 90-х ознаменовалась конкретными шагами в этом направлении при участии таких известных специалистов в области объектного анализа, как Гради Буч, Джим Рамбо и Айвар Якобсон. Втроем они возглавили разработку специального языка объектного моделирования UML (Unified Modeling Language) в фирме Rational Software.

Появление в свет первой версии этого языка 13.01.1997 года стало логическим продолжением развития методологий анализа и проектирования. За время своего существования язык UML стал стандартом де-факто. UML представляет собой набор диаграмм, описывающих предметную область как в статике (объектные диаграммы), так и в динамике (диаграммы жизненных циклов объектов), а также он описывает программную среду с помощью диаграмм модулей. С помощью этого языка достаточно просто переложить результаты анализа на конкретную реализацию с применением понятия модулей или компонент, которые достаточно часто прямо отражают домены предметной области.

Распределенная ИС в моделях UML представляется в виде некой совокупности взаимосвязанных диаграмм, описывающих наиболее важные аспекты системы с точки зрения разработчика. Это дает возможность наиболее полно охватить систему, увидеть ее несовершенства, внести необходимые изменения, избежать многих ошибок на ранней стадии

создания программного обеспечения. В UML используются все те же диаграммы, что и в других средствах анализа, однако собранные воедино, они дают возможность наиболее полно рассмотреть предметную область и подвергнуть ее более глубокому анализу. Так, из методологии OOSE Айвара Якобсона были взяты диаграммы прецедентов (Use Case Diagram) для анализа предметной области. Для создания иерархической структуры объектов в методе UML применяются диаграммы классов (Class Diagram). Они отображают взаимосвязи между объектами, классами системы, их атрибуты и поведение. По своей значимости диаграммы классов, пожалуй, являются наиболее весомыми в объектном анализе, так как от качества классификации объектов зависит жизнеспособность будущей системы. Диаграмма поведения системы (Interaction Diagram), диаграммы последовательности (Sequence Diagram) и взаимодействия (Collaboration Diagram) вместе дают трехмерную модель взаимодействия объектов системы относительно времени. С помощью диаграмм состояний (State Diagram) описывается структура переходов объектов из одного состояния в другое при возбуждении определенных событий. Также используются диаграммы активности (Activity Diagram) для описания алгоритмов выполнения определенных операций. Для того чтобы описать и спроектировать архитектуру системы, а также обозначить процессы, которые будут выполняться на определенных узлах распределенной системы, используются диаграммы размещения (Deployment Diagram). Для разбиения системы на отдельные компоненты, выявления возможности повторного использования уже существующих компонент используются диаграммы компонент (Component Diagram).

Таким образом, объектно-ориентированная декомпозиция на базе UML поддерживает все стадии жизненного цикла проектов – от анализа требований к системе до проекта на конкретном языке программирования. Средства визуального проектирования, поддерживающие язык UML (такие как, например, CASE-средство Rational Rose), обеспечивают генерацию кода и обратное проектирование для множества языков программирования и описания интерфейсов (например, PowerBuilder, CORBA IDL и др.), а также имеют поддержку моделей ERwin и языка описания данных DDL для большинства СУБД.

2.4. Основы методологии разработки информационных систем на базе моделей предметной области

Тенденция развития информационных систем приводит к возрастанию их сложности. Современные крупные проекты характеризуются следующими особенностями:

сложность описания, требующая тщательного моделирования и анализа данных и процессов;

наличие тесно взаимодействующих компонентов;

отсутствие прямых аналогов ограничивает возможность использования тепловых решений;

необходимость интеграций существующих и разрабатываемых приложений;

функционирование в неоднородной среде различных аппаратных программных платформ;

разнородность отдельных групп разработчиков по уровню квалификации и традициям использования тех или иных инструментальных средств;

существенная временная протяженность проекта, которая обусловлена, во-первых, ограниченными возможностями коллектива разработчиков, во-вторых, степенью готовности подразделений и организаций к внедрению информационной системы.

До недавнего времени проектирование информационных систем выполнялось на интуитивном уровне с использованием неформализованных методов, т. е. методов, основанных на практическом опыте, экспертных оценках и дорогостоящих экспериментальных проектах.

Ключом решения проблем проектирования информационных систем является грамотная организация процесса создания программного обеспечения, реализация технологических принципов промышленного конструирования информационных систем. Эти же проблемы способствовали появлению программных технологических средств социального класса, так называемых case-средств (Computer Aided Software Engineering).

Case-средства реализует case-технология создания и сопровождения информационных систем.

Под термином «case-средства» понимают программные средства, поддерживающие этапы анализа и формулировки требований к системе, проектирование прикладного программного обеспечения и баз данных, автоматическую генерацию кода, тестирование, документирование, управление конфигурацией информационной системы и управление проектом.

Case-технология представляет собой методологию проектирования информационных систем, а также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область. Она также позволяет анализировать эту модель на всех этапах разработки и разрабатывать приложения в соответствии с потребностями пользователей.

Большинство существующих case-средств основано на методологиях структурного анализа и проектирования, которые используют спецификации в виде диаграмм для описания внешних требований, связи

между моделями, динамики поведения системы и архитектуры программных средств.

Использование case-средств дает разработчику следующие преимущества:

улучшает качество программного обеспечения за счет средств автоматического контроля проекта;

за короткое время можно получить прототип создаваемой системы. Это позволяет на ранних этапах проектирования оценить ожидаемый результат;

освобождает разработчика от рутинной работы;

поддерживает сопровождение программного обеспечения.

Структурный анализ и проектирование

Определение структурного анализа

На этапе анализа требований к системе формализуются, документируются и уточняются требования заказчика. Именно на этом этапе закладывается успех всего проекта. Из-за неполноты и нечеткости требований разработка проекта может закончиться неудачей. Поэтому список требований к разрабатываемой системе должен включать совокупность условий, при которых будет эксплуатироваться программная система, в том числе аппаратные и программные ресурсы и требования квалификации сотрудников; описание выполняемых системой функций; ограничения на процессы разработки, к которым можно отнести сроки завершения работ и мероприятия по защите информации.

На этапе анализа определяется архитектура системы, ее функции, распределение функций между аппаратурой и программным обеспечением.

На этапе проектирования исследуется структура системы и взаимосвязи ее компонентов. Проектирование часто определяют как процесс получения логической модели системы. Этап проектирования обычно разделяется на два подэтапа: проектирование структуры и проектирование интерфейса для отдельных компонентов программной системы, согласование функций и технических требований компонентов системы; детальное проектирование, включающее разработку спецификаций для каждого компонента.

Особенностью разработки программного обеспечения является то, что наиболее сложные работы выполняются на начальных этапах жизненного цикла, т. е. на этапах анализа и проектирования. Ошибки, допущенные на этапах анализа и проектирования, порождают на следующих этапах трудные, а иногда неразрешимые проблемы.

Метод структурного анализа состоит в том, что исследование предметной области начинается с общего обзора, а затем выполняется более детальное исследование, результаты которого приобретают иерархическую структуру.

Для метода структурного анализа характерно разбиение описания системы на уровне абстракции. Причем количество элементов на каждом уровне ограничено и обычно составляет от трех до девяти. На каждом уровне учитываются только существенные для данного уровня детали, определяется правило и формальное описание компонентов.

Методология структурного анализа базируется на ряде общих принципов. Эти принципы регламентируют организацию работ на начальных этапах жизненного цикла, а также используются для выработки рекомендаций по организации работ на последующих этапах. Перечислим основные принципы:

решение трудных задач выполняется путем их разбиения на множество меньших независимых задач;

каждая отдельная задача существенна для понимания всей системы в целом. Этот принцип называется также принципом иерархического упорядочивания и означает, что система может быть представлена по уровню, но каждый уровень должен добавлять в систему новые существенные детали.

Соблюдение перечисленных принципов необходимо при организации работ на начальных этапах жизненного цикла независимо от типа разрабатываемой системы.

Использование принципов структурного анализа позволяет на более ранних стадиях разработки получить представления о создаваемой системе, обнаружить промахи и недоработки. Это облегчает работу на последующих этапах жизненного цикла и снижает стоимость разработки.

Основные этапы подхода Мартина

IE-методология Мартина предоставляет общую стратегию разработки информационных систем, фокусирующую внимание на стратегическом планировании и бизнес-процессах. В то же время она является и инженерным подходом к разработке ПО, так как обеспечивает нисходящую пошаговую процедуру построения информационной системы (позволяя при этом работать с неиерархическими структурами данных). Подход Мартина базируется на двух концепциях: послойного целостного подхода к разработке интегрированных приложений, базирующегося на стратегическом плане развития информационных систем; первоначальной направленности на моделирование данных, а затем на функциональное моделирование. Основные этапы подхода Мартина приведены на рис. 2.4.

Рис. 2.4. Основные этапы подхода Мартина

Этап *стратегического информационного планирования* начинается с построения стратегического плана для бизнес-системы, включающего цели и стратегии их достижения. Далее строится модель предметной области, отражающая существующую специфику и определяющая основные бизнес-процессы и организационную структуру бизнес-системы, а также определяется порядок разработки информационной системы. При моделировании используются диаграммы декомпозиции (иерархические древовидные структурные диаграммы) и диаграммы «сущность–связь» для представления основных бизнес-процессов и структур данных, соответственно.

На этапе *анализа* основные бизнес-процессы, разработанные на этапе 1, используются для разбиения общей задачи на частные, при этом основное внимание уделяется определению информационной и функциональной моделей для частных задач. При этом диаграммы «сущность–связь» трансформируются в нормализованную модель данных, а диаграммы декомпозиции распределяются по подзадачам. Для представления процессов служат DFD, диаграммы зависимости данных (диалект DFD) и диаграммы декомпозиции, а для соотнесения данных и процессов, в которых эти данные используются, применяются матрицы «сущность–процесс».

На этапе *логического проектирования* базой являются процессы, разработанные на этапе анализа. С помощью методик нисходящей функциональной декомпозиции проектируются спецификации обработки в процессах и их логические структуры данных. При этом используются

диаграммы структуры данных, определяющие типы сущностей, их атрибуты и связи, диаграммы декомпозиции и диаграммы деятельности, детализирующие логику процессов. Для согласования требований пользователя создаются прототипы пользовательских интерфейсов с помощью схем экранов/отчетов.

На этапе *физического проектирования* и реализации производится преобразование логической модели ИС в физическую и ее реализация.

Средства структурного анализа

Существует три группы средств структурного анализа, иллюстрирующие: функции, которые система должна выполнить; описание отношений между данными; поведение системы, зависящей от времени.

В методологии структурного анализа наиболее популярными являются следующие средства:

диаграмма потоков данных (Data Flow Diagram) для иллюстрации функций, которые система должна выполнить, и информационных потоков, связывающих процессы в системе. DFD используются совместно со спецификациями процессов и словарями данных;

диаграмма «сущность–связь» (Entity Relationship Diagram). Эти диаграммы используются для иллюстрации отношений между данными;

диаграммы перехода в состояние (State Transition Diagram).

Перечисленные средства содержат графические и текстовые инструменты моделирования. Графические инструменты предназначены для демонстрации основных компонентов системы. Текстовые инструменты позволяют точно определить компоненты системы и связи между моделями.

Логические диаграммы потоков данных показывают внешние по отношению к системе источники данных и адресаты, а также идентифицируют логические функции или процессы.

Группы элементов данных, связывающие одну логическую функцию с другими, называются потоками данных. Кроме того, на DFD идентифицируются хранилища или накопители, к которым осуществляется доступ.

Структура потоков данных и описывающие их компоненты хранятся в словаре данных. Каждый процесс на диаграмме потоков данных может быть детализирован с помощью DFD нижележащего уровня. Когда степень детализации процессов становится достаточной, переходят к определению логики процессов с помощью спецификации процессов. Кроме того, в словаре данных описывается также содержимое каждого хранилища.

Моделирование потоков данных

В основе методологии моделирования потоков данных лежит построение модели – анализируемые информационные системы. Модель системы определяется как иерархия диаграмм потоков данных,

описывающих асинхронный процесс при образовании иерархии от ее ввода в систему до выдачи пользователю.

Диаграмма верхнего уровня иерархии определяет основные процессы внешними входами и выходами. Они детализируются при помощи диаграмм более низкого уровня. Декомпозиция продолжается до тех пор, пока не будет достигнут такой уровень детализации, на котором процесс становится элементарным. Результатом декомпозиции является многоуровневая иерархия диаграмм. Источники информации, которыми часто являются внешние сущности, порождают информационные потоки, которые переносят информацию к основным процессам или подсистемам. Процессы в свою очередь преобразуют информацию и порождают новые потоки, которые переносят информацию к другим процессам, к накопителям данных или к внешним сущностям потребителя информации.

Таким образом, структура информационной системы представляется в виде сети или графов, узлами которых являются процессы, а связями – потоки данных.

Основными элементами диаграмм потоков данных являются внешние сущности, процессы (подсистемы), хранилища (накопители информации) и потоки данных (рис. 2.5). Для изображения диаграмм используются две нотации: Йордана – Де Марко и Гейна – Сарсона.

Элемент	Описание	Нотация Йордона-Де Марко	Нотация Гейна-Сарсона
Функция	Работа	Имя функции Номер	Имя функции Номер
Поток данных	Объект, над которым выполняется работа. Может быть логическим или управляющим. (Управляющие потоки обозначаются пунктирной линией со стрелкой)	Имя объекта	Имя объекта (Понятие управляющего потока отсутствует)
Хранилище данных	Структура для хранения информационных объектов	Имя объекта	Имя объекта
Внешняя сущность	Внешний по отношению к системе объект, обменивающийся с нею потоками	Имя внешнего объекта	Имя внешнего объекта

Рис. 2.5. Основные элементы диаграмм потоков данных

Поток данных используется для моделирования, передачи информации от источника к получателю. Ориентация стрелки указывает направление движения информации. В некоторых случаях информация

передается в одном направлении, обрабатывается и возвращается к источнику. Такая ситуация может моделироваться двумя встречными потоками или одной двунаправленной стрелкой.

Процесс выполняет преобразование входных потоков данных в выходные в соответствии с действием, которое определяется именем процесса. Имя процесса должно содержать глагол в неопределенной форме или отглагольное существительное и возможно дополнения, например: выдать информацию о текущих расходах, проверить кредитоспособность и т. д. Использование таких глаголов, как обработать, отредактировать, организовать и т. д. означает, что данный процесс требует дальнейшего анализа. Физически процесс может быть реализован программой, аппаратными средствами, некоторым подразделением организаций, выполняющих обработку информации и т. д. Каждый процесс на диаграмме имеет уникальный номер для ссылки на него внутри диаграммы. Номер процесса совместно с номером диаграммы образуют уникальный индекс процесса во всей модели.

Хранилище (накопитель данных) определяет данные, которые сохраняются между процессами. Накопитель данных представляет собой абстрактное устройство для хранения информации. Информацию можно в любой момент поместить в накопитель и через некоторое время извлечь, причем в любом порядке. Физически, накопитель может быть реализован в виде картотеки, массива в оперативной памяти, файла на диске, базы данных и т. д. В общем случае накопитель является прообразом базы данных информационной системы и служит для описания данных и их увязывания с информационной моделью. Имя хранилища идентифицирует его содержимое и должно быть существительным. Если поток данных входит и выходит из хранилища и его структура соответствует структуре хранилища, то поток данных должен иметь то же самое имя, что и хранилище.

Внешняя сущность – это объект области, не входящей в контекст предметной области информационной системы и являющейся источником или получателем данных, например, заказчики, поставщики, клиенты. Определение объекта предметной области в качестве внешней сущности указывает на то, что этот объект находятся за пределами границ информационной системы и в обработке не участвует. Имя внешней сущности должно быть существительным. В процессе анализа некоторые внешние сущности могут быть перенесены внутрь информационной системы или, наоборот, часть процессов информационной системы может быть вынесена за пределы диаграммы и представлена внешними сущностями.

Контекстная диаграмма. При построении модели сложной системы информационная система может быть представлена в самом общем виде на контекстной диаграмме. Контекстная диаграмма моделирует систему

наиболее общим образом. Обычно в центре контекстной диаграммы находится главный процесс, соединенный с источниками и получателями информации, которым соответствуют внешние сущности. В общем случае каждый проект должен иметь только одну контекстную диаграмму, и ее единственный процесс не нумеруется. Однако для сложной информационной системы ограничиться единственной контекстной диаграммой трудно, так как она будет содержать слишком большое количество внешних сущностей, которые будет сложно расположить на диаграмме.

Для сложных информационных систем строятся иерархии контекстных диаграмм, при этом контекстная диаграмма верхнего уровня содержит не единственный процесс, а набор подсистем, соединенных потоками данных. Иерархия контекстных диаграмм определяет взаимодействие основных функциональных подсистем, проектируемые информационные системы как между собой, так и с внешними входными и выходными потоками данных, и с внешними сущностями. Для каждой подсистемы выполняется декомпозиция контекстной диаграммы при помощи диаграмм потоков данных.

Каждый процесс на DFD в свою очередь может быть детализирован при помощи DFD более низкого уровня или описан при помощи миниспецификаций.

Построение иерархии диаграмм потоков данных. Главная цель построения иерархии диаграмм потоков данных состоит в том, чтобы сделать ясными и понятными требования к проектируемой системе на каждом уровне ее детализации.

В процессе построения модели следует придерживаться следующих правил:

правило балансировки. Означает, что при детализации процесса детализирующая диаграмма может содержать только те компоненты информационных потоков, которые определены в детализируемой подсистеме;

на каждой диаграмме может быть расположено от двух до девяти процессов;

несущественные детали на данном уровне использоваться не должны;

декомпозиция потоков данных проводится одновременно с декомпозицией процессов; имена процессов и потоков данных должны отражать их суть;

функционально идентичные процессы нужно определять однократно на самом верхнем уровне, где процесс необходим, а затем на нижних уровнях на этот процесс ссылаться; следует разделять управляющие и входные потоки.

2.5. Методология функционального моделирования SADT (Structured Analysis and Design Technique)

Основоположником SADT является Дуглас Росс. Методология SADT [10, 11, 32] является основой методологии ICAM definition (ICAM – Integration Computer and Manufacture). Методология IDEF0 является основной частью программной интеграции компьютерных и промышленных технологий. Методология SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели какой-либо предметной области. Функциональная модель SADT отражает функциональную структуру объекта, т. е. производимое им действие и связи между этими действиями.

Методология SADT основана на следующих концепциях:

1. Графическое представление блочного моделирования. Каждая функция изображается в виде блока, а интерфейс входа и выхода представляется входными и выходными дугами. Взаимодействие блоков друг с другом описывается посредством интерфейса дуг.

2. Строгость и точность. Как правило, SADT требует точности исполнения, но не накладывает чрезмерных ограничений на действия аналитиков.

Правила SADT:

ограниченное количество блоков на каждом уровне декомпозиции (обычно от 3 до 6);

связность диаграмм посредством нумерации блоков;

уникальность меток и наименований;

синтаксические правила для блоков и дуг;

разделение входных и управляющих дуг;

исключение влияния организационной структуры на функциональную модель.

Методология SADT может использоваться для моделирования широкого класса систем, для определения требования к системе и для реализации систем, которые удовлетворяют этим требованиям.

Методология IDEF3

Стандарт IDEF0, который был рассмотрен ранее, является развитием классического DFD-подхода и предназначен для описания бизнес-процессов верхнего уровня. Для описания временной последовательности и алгоритмов выполнения работ стандарт IDEF0 не подходит. Для решения этой задачи стандарт IDEF0 получил дальнейшее развитие, в результате чего был разработан стандарт IDEF3, который входит в семейство стандартов IDEF. Стандарт IDEF3, в свою очередь, является развитием WFD-подхода и предназначен для описания бизнес-процессов нижнего уровня. Он содержит объекты – логические операторы, с помощью

которых показывают альтернативы и места принятия решений в бизнес-процессе, а также объекты – стрелки, которые показывают временную последовательность работ в бизнес-процессе (рис. 2.6).

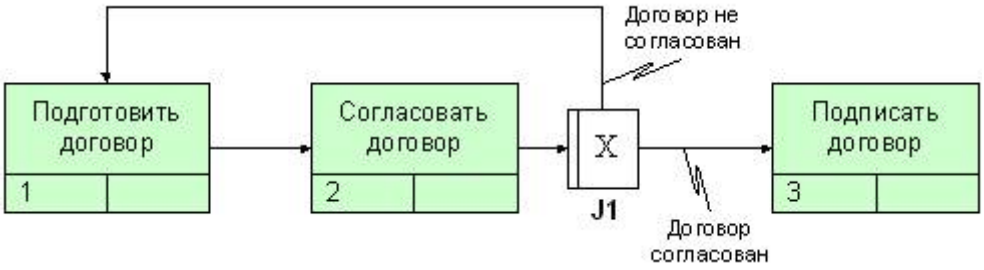
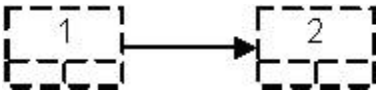
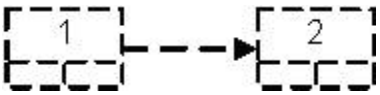


Рис. 2.6. Схема бизнес-процесса в стандарте IDEF***3***

В отличие от классической методологии WFD, в стандарте IDEF3 связи между работами делятся на три типа, обозначения, названия и смысл которых приведены в табл. 2.1.

Таблица 2.1. Типы связей между работами в стандарте IDEF***3***

Название связи	Вид связи	Смысл связи
Предшествования		Обозначает, что вторая работа начинает выполняться после завершения первой
Отношения		Обозначает, что вторая работа может начаться и даже закончиться до того момента, когда закончится выполнение первой
Потоков объектов		Данный тип связи обозначает одновременно как временную последовательность работ, так и сам материальный, либо информационный поток. В данном примере вторая работа начинает выполняться после завершения первой. При этом выходом первой работы является объект, название которого надписано над стрелкой (в данном примере документ). Эта связь также обозначает, что объект, порождаемый первой работой, используется в последующих работах

Помимо наличия нескольких типов связей между работами в стандарте IDEF3 логические операторы, которые в данном случае называются перекрестками, также делятся на несколько типов. «Исключающий ИЛИ», «И» и «ИЛИ».

Перекресток «Исключающий ИЛИ» обозначает, что после завершения работы «А», начинает выполняться только одна из трех расположенных параллельно работ B, C или D в зависимости от условий 1, 2 и 3 (рис. 2.7). Перекресток «И» обозначает, что после завершения работы «А», начинают выполняться одновременно три параллельно расположенные работы B, C и D. Перекресток «ИЛИ» обозначает, что после завершения работы «А», может запуститься любая комбинация трех параллельно расположенных работ B, C и D. Например, может запуститься только одна из них, три работы, а также двойные комбинации B и C, либо C и D, либо B и D. Перекресток «Исключающий ИЛИ» является самым неопределенным, так как предполагает несколько возможных сценариев реализации бизнес-процесса и применяется для описания слабо формализованных ситуаций.

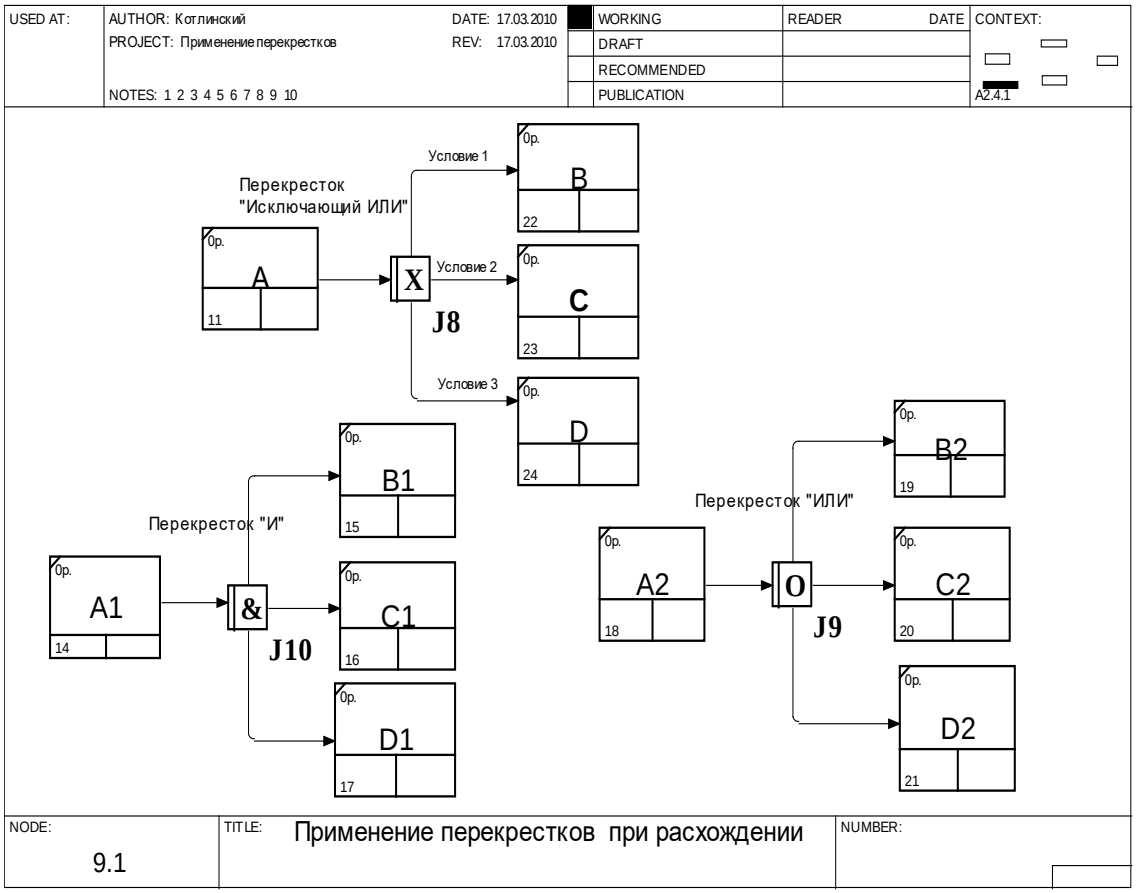


Рис. 2.7. Применение перекрестков «Исключающий ИЛИ», «И» и «ИЛИ» в случае схемы расхождения

Перекрестки «И» и «ИЛИ» подразделяются еще на два подтипа – синхронные и асинхронные. Перекрестки синхронного типа обозначают, что работы B, C и D запускаются одновременно после завершения работы

А. Перекрестки асинхронного типа требований к одновременности не предъявляют. Приведенные на рис. 2.7 схемы взаимосвязи работ и перекрестков называются схемами расхождения, так как от перекрестков расходятся несколько работ. Существуют и другие схемы взаимосвязи перекрестков и работ – это так называемые схемы схождения, когда к перекрестку подходит несколько работ (рис. 2.8).

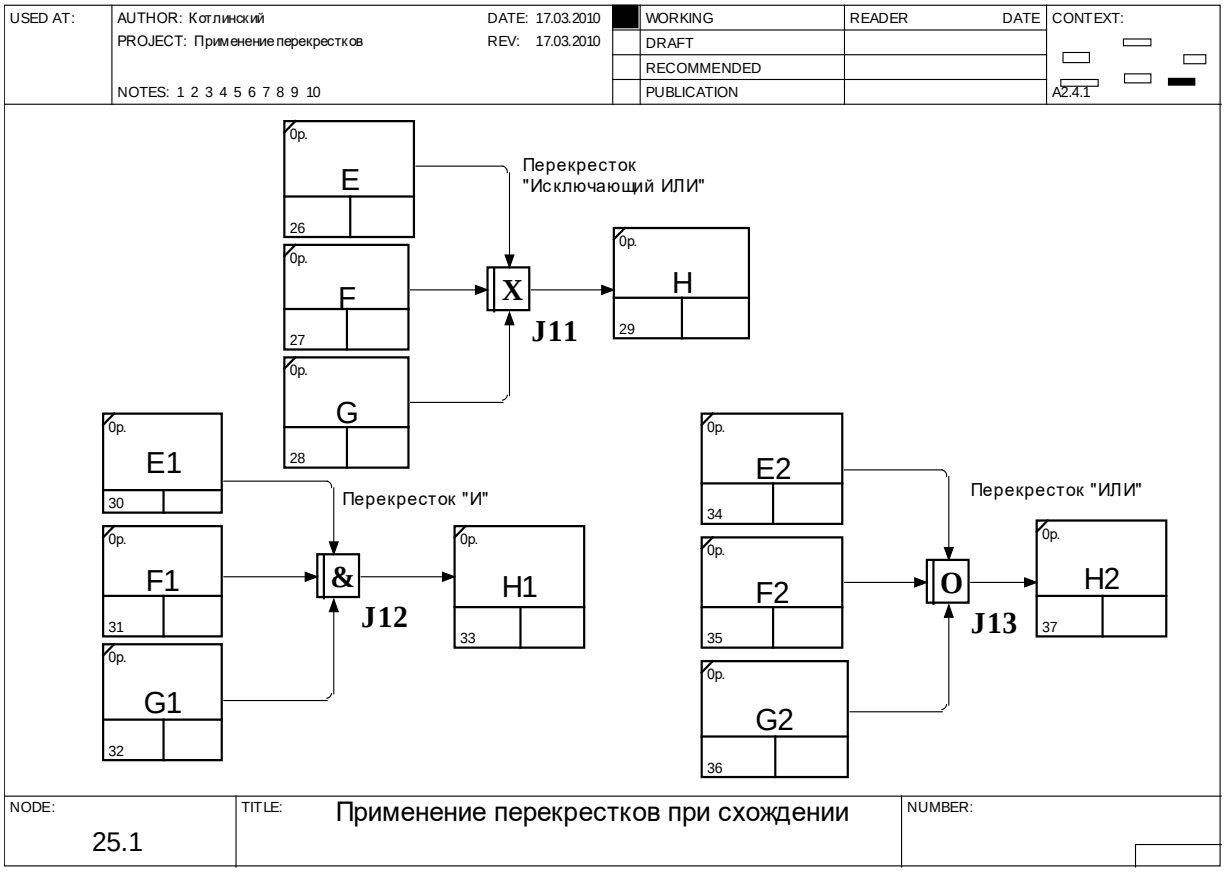
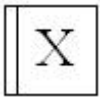
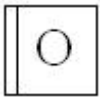
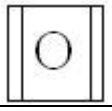


Рис. 2.8. Применение перекрестков «Исключающий ИЛИ», «И» и «ИЛИ» в случае схемы схождения

Характеристики всех типов перекрестков, как в схемах схождения, так и в схемах расхождения приведены в табл. 2.2.

Таблица 2.2. Обозначения, названия и смысл типов перекрестков в схемах схождения и расхождения

Название перекрестков		Обозначение перекрестков	Смысл перекрестков	
			Схема расхождения	Схема схождения
«Исключающий ИЛИ»			Только одна последующая работа запускается	Только одна предшествующая работа должна быть завершена
«И»	Асинхронный		Все последующие работы запускаются	Все предшествующие работы должны быть завершены
	Синхронный		Все последующие работы запускаются одновременно	Все предшествующие работы должны быть завершены одновременно
«ИЛИ»	Асинхронный		Одна или несколько последующих работ запускаются	Одна или несколько предшествующих работ должны быть завершены
	Синхронный		Работы запускаются одновременно	Работы завершаются одновременно

Последним отличием стандарта IDEF3 от классической методологии WFD является использование на схеме бизнес-процесса такого элемента, как «объект ссылки», с помощью которого показывается прочая важная информация, которую целесообразно зафиксировать при описании бизнес-процесса.

2.6. Построение информационной модели системы. Проектирование баз данных

Диаграммы «сущность–связь» (entity – relationship ERD)

ERD предназначены для разработки информационных моделей системы, т. е. моделей данных; обеспечивает стандартный способ определения данных и отношений между ними. С помощью ERD осуществляется детализация хранилищ данных функциональной модели системы.

Такой подход был предложен Ченом и получил дальнейшее развитие в работах Баркера.

Нотация Баркера. Модель «сущность – связь» в нотации Баркера. Методология IDEF1X

В нотации Баркера используется только один тип диаграмм – диаграмма «сущность – связь» ERD. Она основывается на важной семантической информации о реальном мире. Может использоваться в

качестве основы для унификации различных представлений данных на основе иерархической, сетевой и реляционной моделей. Диаграмма включает в себя два графических элемента:

1. Сущность (entity) – это «предмет», который может быть идентифицирован некоторым способом, отличающим его от других «предметов» (определение Питера Чена). Каждая сущность обладает набором атрибутов (рис. 2.9). Атрибут – отдельная характеристика сущности. Сущность состоит из экземпляров, каждый из которых должен отличаться от другого. Пример: сущность – сотрудник, экземпляр сущности – сотрудник Петров Виктор Сергеевич.

2. Связь (relationship) – это логическая ассоциация, устанавливаемая между сущностями; представляет собой бизнес-правило или ограничение. Сущность отображается прямоугольником, сверху которого представлено название сущности. Прямоугольник делится горизонтальной линией: атрибуты, представленные выше линии, являются атрибутами первичного ключа; ниже линии представлены неключевые атрибуты.

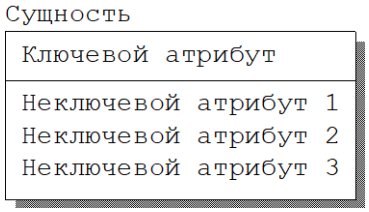


Рис. 2.9. Отображение сущности

Атрибут первичного ключа – атрибут, значение которого позволяет уникально идентифицировать экземпляр сущности. Атрибут является частью первичного ключа, который необходим для того, чтобы отличать экземпляры сущности друг от друга. Первичный ключ может быть простым – состоящим из одного атрибута и составным – состоящим из нескольких атрибутов.

Связи отображаются как линии между сущностями (рис. 2.10). В зависимости от роли в связи сущность может быть родительской или дочерней. В методике IDEF1X у дочерней сущности на связи присутствует точка. Каждая связь имеет глагольную фразу, которая ее характеризует. Глагольная фраза читается по следующей формуле: название родительской сущности + глагольная фраза + название дочерней сущности. Иногда для большей ясности разработчики используют вторую глагольную фразу, которая читается по формуле: название дочерней сущности + глагольная фраза + название родительской сущности.

Существует два типа сущностей:

зависимая сущность. Для определения экземпляра такой сущности необходимо сослаться на экземпляр независимой сущности, с которой

связана зависимая сущность. Пример: сущности – заказ и позиция заказа. Для идентификации позиции заказа нужно сослаться на заказ, в который входит данная позиция;

независимая сущность. Для определения экземпляра сущности нет необходимости ссылаться на другие сущности. Пример: сущности – заказ и позиция заказа. Для определения заказа (сущности) нет необходимости ссылаться на позиции этого заказа (другие сущности).

Существует несколько видов связей.

Идентифицирующая связь указывает на то, что дочерняя сущность в связи является зависимой от родительской, т. е. экземпляр зависимой сущности может быть однозначно определен только в том случае, если в этом экземпляре есть ссылка на экземпляр независимой сущности. Идентифицирующая связь отображается сплошной линией, причем дочерняя сущность является зависимой и поэтому отображается прямоугольником со скругленными углами (рис. 2.10).

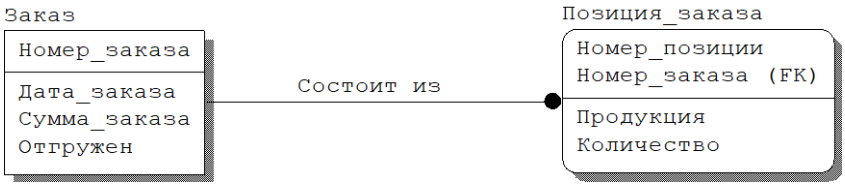


Рис. 2.10. Идентифицирующая связь

Неидентифицирующая связь показывает на зависимость между родительской и дочерней сущностями, при этом экземпляр дочерней сущности может быть однозначно идентифицирован без ссылки на экземпляр родительской сущности. Неидентифицирующая связь отображается штриховой линией (рис. 2.11).

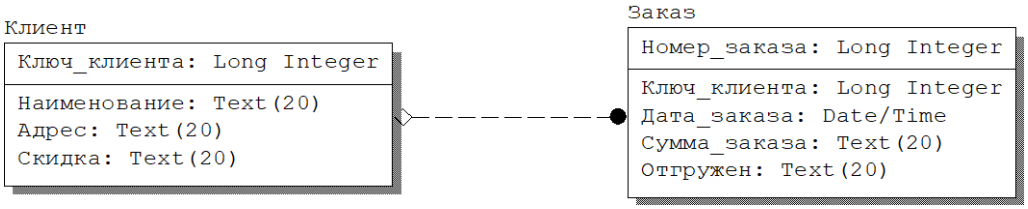


Рис. 2.11. Неидентифицирующая связь

Связь «многие ко многим» (рис. 2.12). Неспецифичный тип связи, во многом свидетельствующий о незавершенности анализа. На конечных этапах моделирования данных все связи «многие-ко-многим» преобразуются в другие (идентифицирующие и неидентифицирующие) типы связей. В связи «многие-ко-многим» не выделяются родительская и дочерняя сущность и, как правило, используются две глагольных фразы.

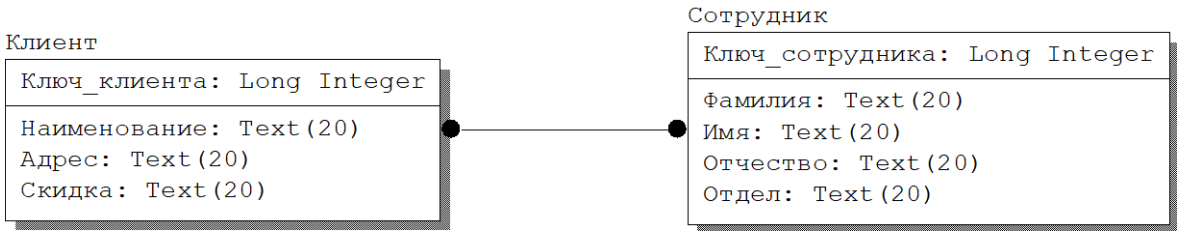


Рис. 2.12. Связь «многие-ко-многим»

Иерархическая связь (рис. 2.13). Указывает на то, что связь связана сама с собой.

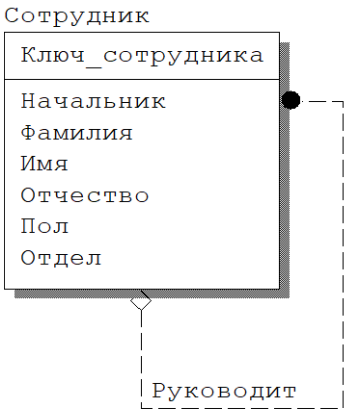


Рис. 2.13. Иерархическая связь

Связь «иерархия категорий». В процессе моделирования данных могут быть выявлены сущности, часть атрибутов и связей которых одинаковы. В этом случае используется «иерархия категорий» (рис. 2.14). Все общие атрибуты выделяются в сущность называемую супертипом, а отличающиеся атрибуты помещаются в сущности-подтипы, связанные с супертипом. При помощи дискриминанта определяется, с экземпляром какого подтипа связан экземпляр супертипа.

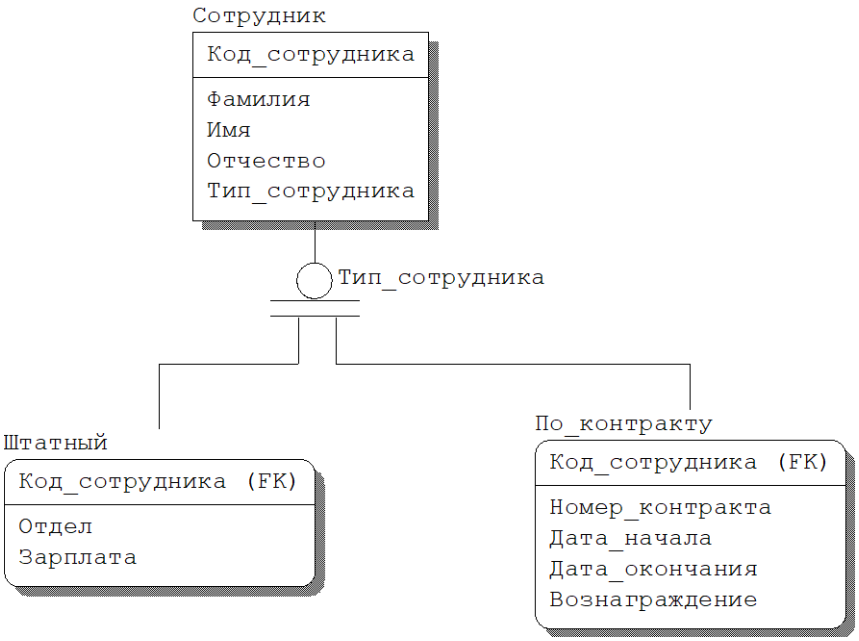


Рис. 2.14. Связь «иерархия категорий»

В нотации IDEF1X иерархия категорий может быть двух типов – полная и неполная. Определить, какой тип иерархии категорий представлен в модели данных можно по стилю отображения дискриминанта: дискриминант с двумя горизонтальными линиями свидетельствует о полноте иерархии категорий, с одной горизонтальной линией – о ее неполноте. Полная иерархия категорий говорит о завершенности анализа. В примере с сотрудником полная иерархия категорий говорит о том, что любой сотрудник является либо штатным сотрудником, либо работающим по контракту и никаких других вариантов быть не может. Если в примере мы изменим иерархию категорий на неполную, это будет свидетельствовать о том, что помимо сотрудников, зачисленных в штат и работающих по контракту, могут быть и другие типы сотрудников, но на данном этапе моделирования данных они еще не выявлены.

Процесс разработки информационной модели системы включает следующие основные шаги:

*идентификацию сущностей***.** Каждая сущность должна обладать уникальным идентификатором. Каждый экземпляр сущности должен однозначно идентифицироваться и отличаться от других экземпляров в данной сущности по ключевым признакам. Каждая сущность обладает одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через связь и являются так называемыми внешними ключами. Каждая сущность может обладать любым количеством связи с другими сущностями;

идентификацию связей и указание типов отношений. Связи может присваиваться имя, выражаемое глаголом или грамматическим оборотом;

идентификацию атрибутов. Атрибуты бывают обязательные и необязательные. Обязательные атрибуты не могут принимать неопределенных значений. Обязательными являются все атрибуты первичного ключа, а также некоторые из неключевых атрибутов.

Методика построения информационной модели данных или модели «сущность – связь»

Разработка диаграммы «сущность–связь» включает следующие основные этапы:

идентификацию сущностей, их атрибутов, первичных и альтернативных ключей;

идентификацию отношений между сущностями и указание типов отношений;

разрешение неспецифических отношений.

Для реляционной модели данных неспецифическими являются отношения типа «многие-ко-многим».

Первый этап является определяющим при построении модели данных. Исходная информация для данного этапа – содержимое хранилищ данных функциональной модели системы. На первом этапе осуществляется упрощение схемы отношений за счет ее нормализации путем избавления от повторяющихся строк таблицы. Нормализация всегда выполняется путем расщепления сущности на две или более простых сущностей.

Методы нормализации схемы базы данных были предложены Коддом в работах, посвященных реляционной модели данных. Кодд определил для схемы отношений существование трех нормальных форм: первой, второй и третьей.

Примеры диаграмм для предметной области «Сервисный центр по гарантийному обслуживанию компьютерной техники» представлены на рис. 2.15–2.20.

Рис. 2.15. Верхний уровень в стандарте IDEF0

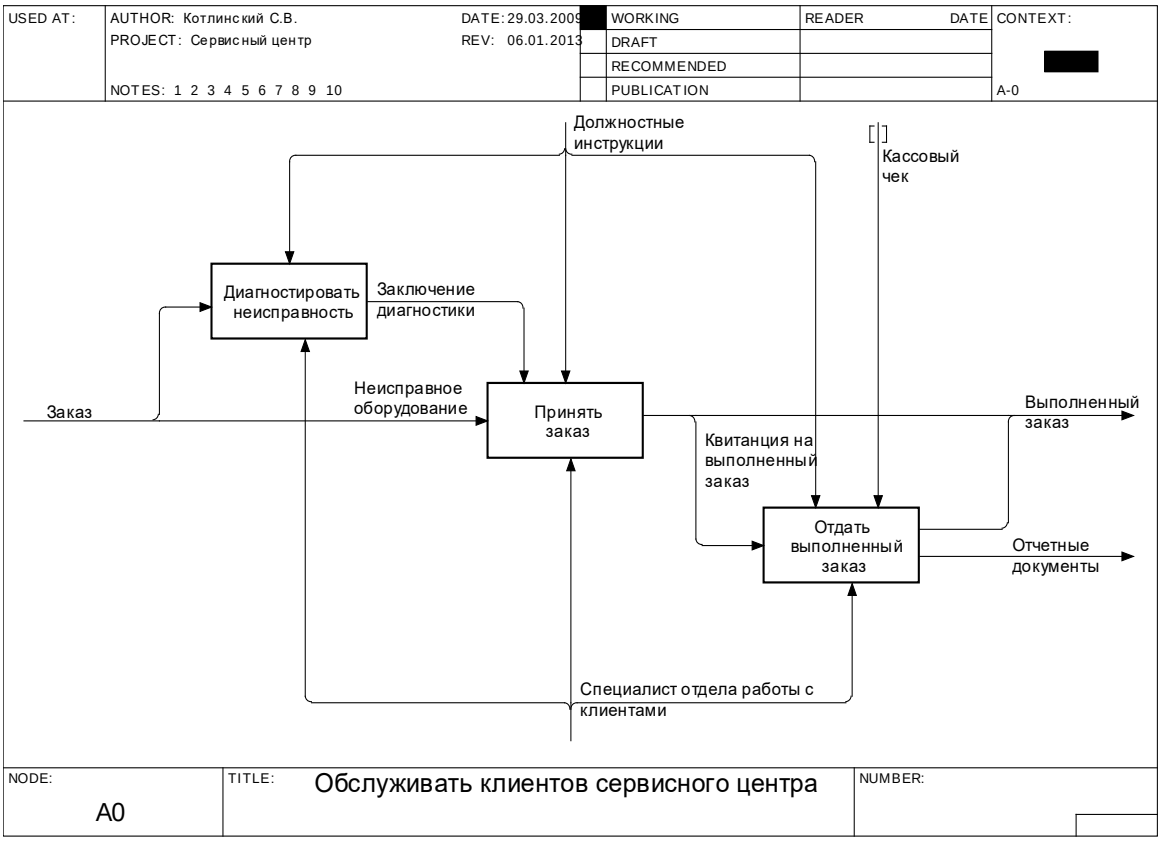


Рис. 2.16. Декомпозиция в стандарте IDEF0

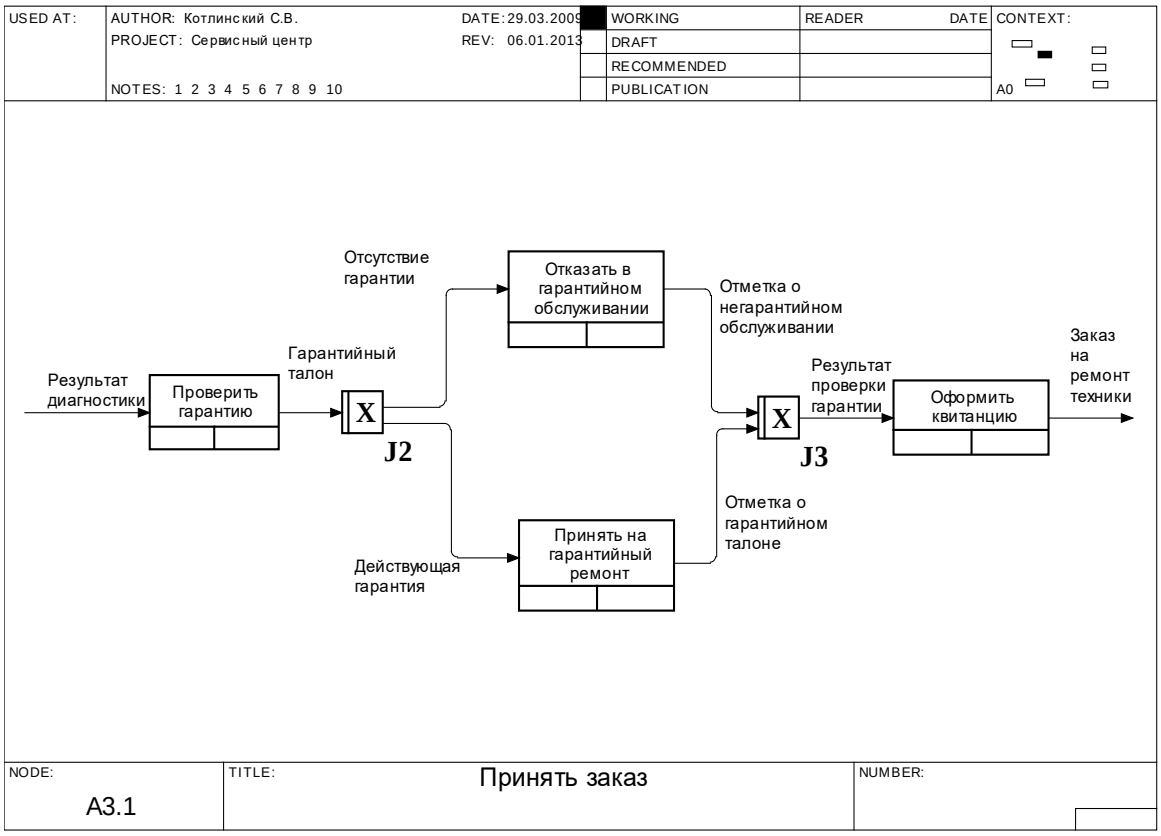


Рис. 2.17. Дополнение стандарта IDEF0 стандартом IDEF3

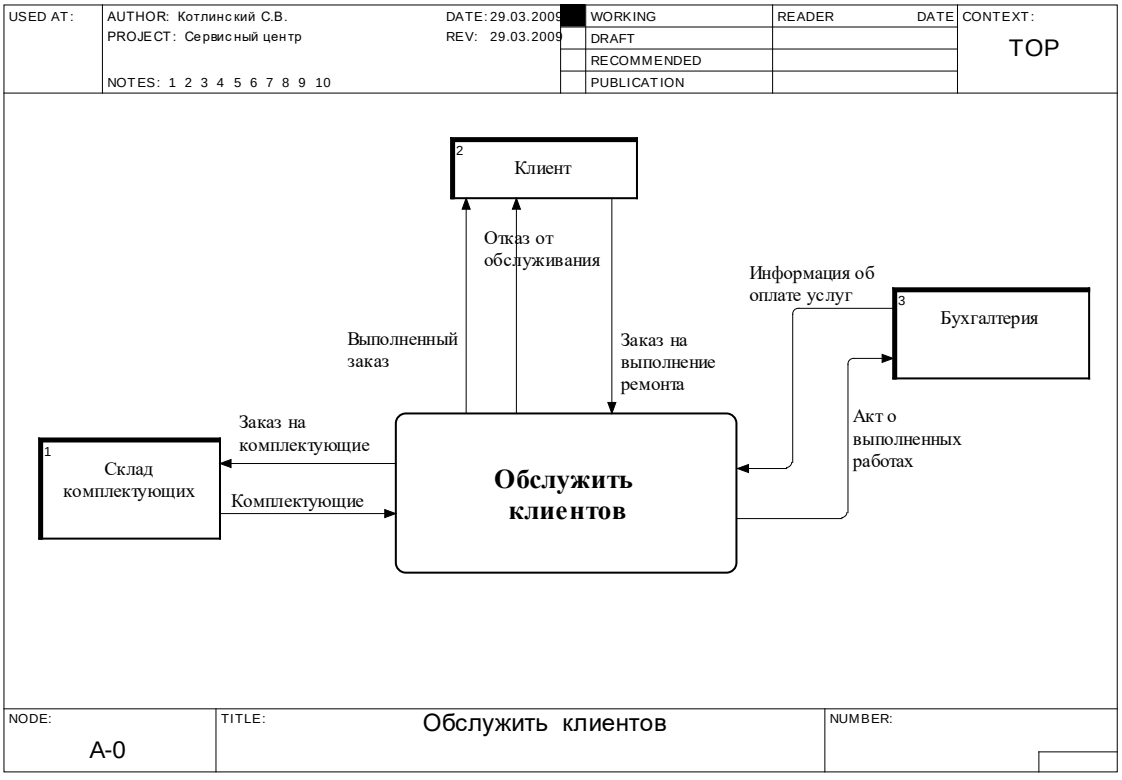


Рис. 2.18. Верхний уровень в стандарте DFD

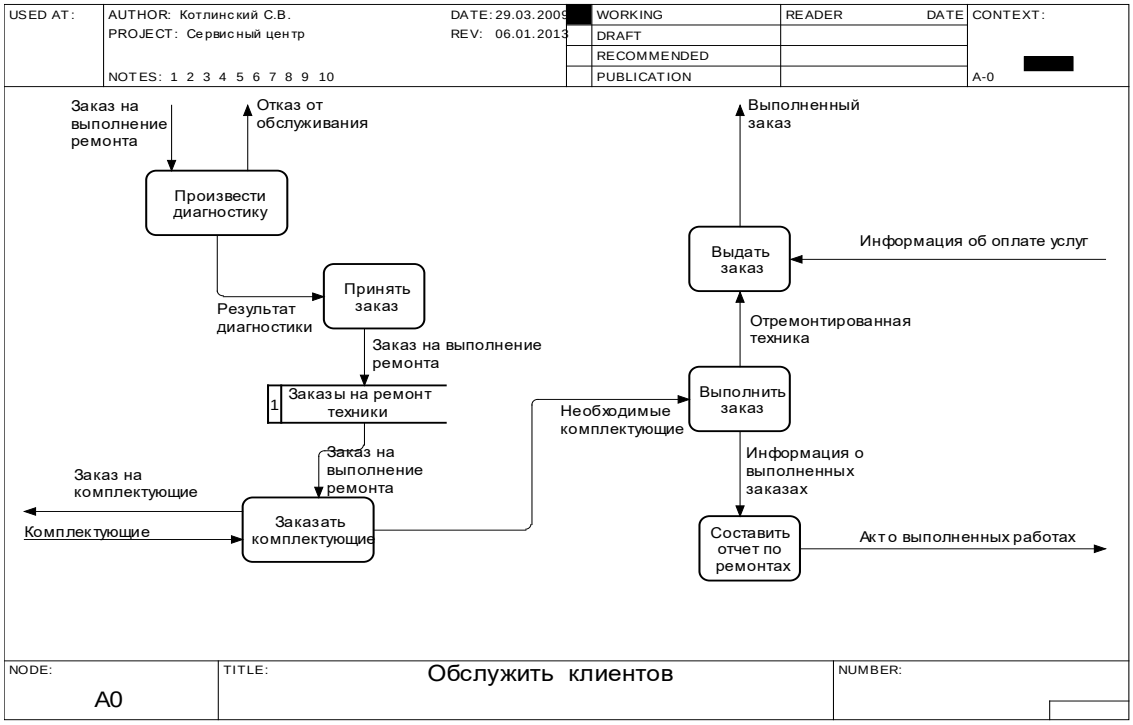


Рис. 2.19. Декомпозиция в стандарте DFD

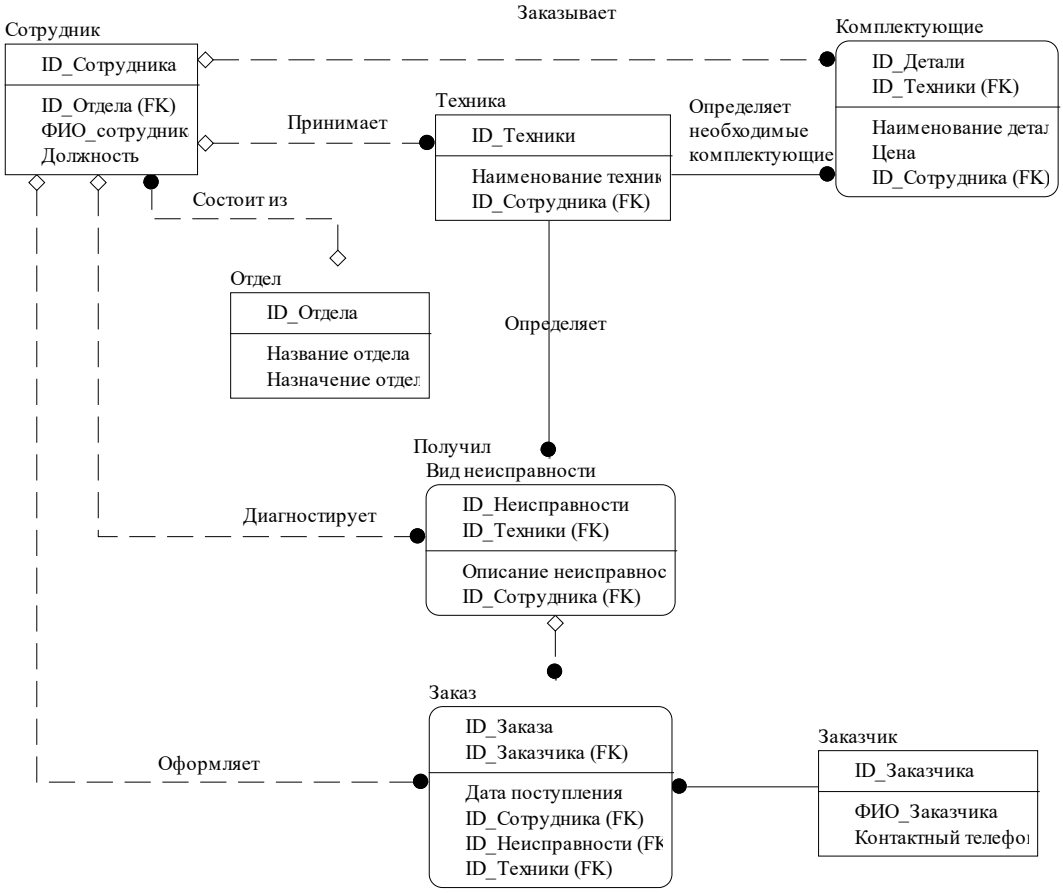


Рис. 2.20. Модель данных в стандарте IDEF1X, выполненная при помощи инструмента Erwin (аналог – в характеристике на Power Designer и приложении на Enterprise Architect)

2.7. Методологии, применяемые для разработки средних и крупных информационных систем

Методология ARIS

Одной из современных методологий бизнес-моделирования, получившей широкое распространение в России, является методология ARIS, которая расшифровывается как Architecture of Integrated Information Systems – проектирование интегрированных информационных систем.

Методология ARIS на данный момент времени является наиболее объемной и содержит около 100 различных бизнес-моделей, используемых для описания, анализа и оптимизации различных аспектов деятельности организации [1]. Часть моделей методологии ARIS используются в настроечном модуле интегрированной информационной системы SAP/R 3, который применяется при внедрении системы и ее настройке на деятельность компании. Ввиду большого количества бизнес-моделей методология ARIS делит их на четыре группы (рис. 2.21):

1. «Оргструктура». Состоит из моделей, с помощью которых описывается организационная структура компании, а также другие элементы внутренней инфраструктуры организации.

2. «Функции». Включает модели, используемые для описания стратегических целей компании, функций и прочих элементов функциональной деятельности организации.

3. «Информация». Состоит из моделей, с помощью которых описывается информация, используемая в деятельности организации.

4. «Процессы». Сюда входят модели, используемые для описания бизнес-процессов, а также различных взаимосвязей между структурой, функциями и информацией.

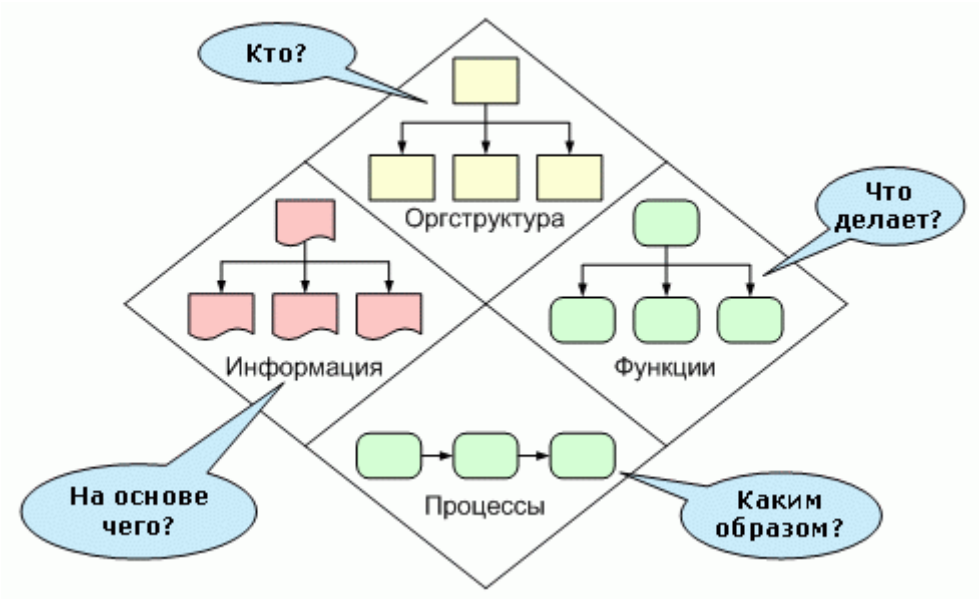


Рис. 2.21. Группы моделей методологии ARIS

Большими преимуществами методологии ARIS являются эргономичность и высокая степень визуализации бизнес-моделей, что делает данную методологию удобной и доступной в использовании всеми специалистами компании, начиная от топ-менеджеров и заканчивая рядовыми сотрудниками. В методологии ARIS смысловое значение имеет цвет, что повышает восприимчивость и читабельность схем бизнес-моделей. Например, структурные подразделения по умолчанию изображаются желтым цветом, бизнес-процессы и операции – зеленым. Помимо большего количества моделей по сравнению с другими методологиями, ARIS имеет наибольшее количество различных объектов, используемых при построении бизнес-моделей, что увеличивает их аналитичность. Например, материальные и информационные потоки на процессных схемах обозначаются разными по форме и цвету объектами, что позволяет быстро определить тип потока (рис. 2.22).

Несмотря на большее количество моделей в методологии ARIS, в проектах по описанию и оптимизации деятельности в общем случае их используется не более десяти. Методология *ARIS* позиционирует себя как конструктор, из которого под конкретный проект в зависимости от его целей и задач разрабатывается локальная методология, состоящая из небольшого количества требуемых бизнес-моделей и объектов. Практика показала, что в проектах наиболее часто используются модели, приведенные в табл. 2.3.

Таблица 2.3. Наиболее часто используемые на практике модели методологии ARIS

№	Название модели		Описание и предназначение модели
	Английский вариант	Русский вариант	
1	*2*	*3*	*4*
1	OD-Objective diagram	Диаграмма целей	Модель описывает стратегические цели компании и их взаимосвязь с другими элементами организации
2	PST-Product/Service tree	Дерево продуктов и услуг	Модель описывает продукты и услуги, производимые компанией, и их взаимосвязь с другими элементами организации
3	FT-Function tree	Дерево функций	Модель описывает функции, выполняемые в компании и их иерархию
4	FAD-Function allocation diagram	Диаграмма окружения процесса	Процессная модель описывает окруже-ние бизнес-процесса
5	VACD-Value added chain diagram	Диаграмма цепочки добавленной стоимости	Процессная модель – аналог классического стандарта *DFD*. Применяется для описания бизнес-процессов верхнего уровня

Окончание табл. 2.3

1	*2*	*3*	*4*
6	PSM – Process selection matrix	Матрица выбора процесса	Процессная модель – аналог классического стандарта *DFD*. Является альтернативой модели *VACD* и применяется для описания бизнес-процессов верхнего уровня
7	eEPC – Extended Event driven Process Chain	Расширенная цепочка процессов, управляемая событиями	Процессная модель – аналог классического стандарта *WFD*. Применяется для описания бизнес-процессов нижнего уровня
8	ORG – Organizational chart	Модель организационной структуры	Модель описывает организационную структуру компании
9	ASTD-Application system type diagram	Диаграмма типов информационных систем	Модель описывает структуру информационных систем, используемых в компании

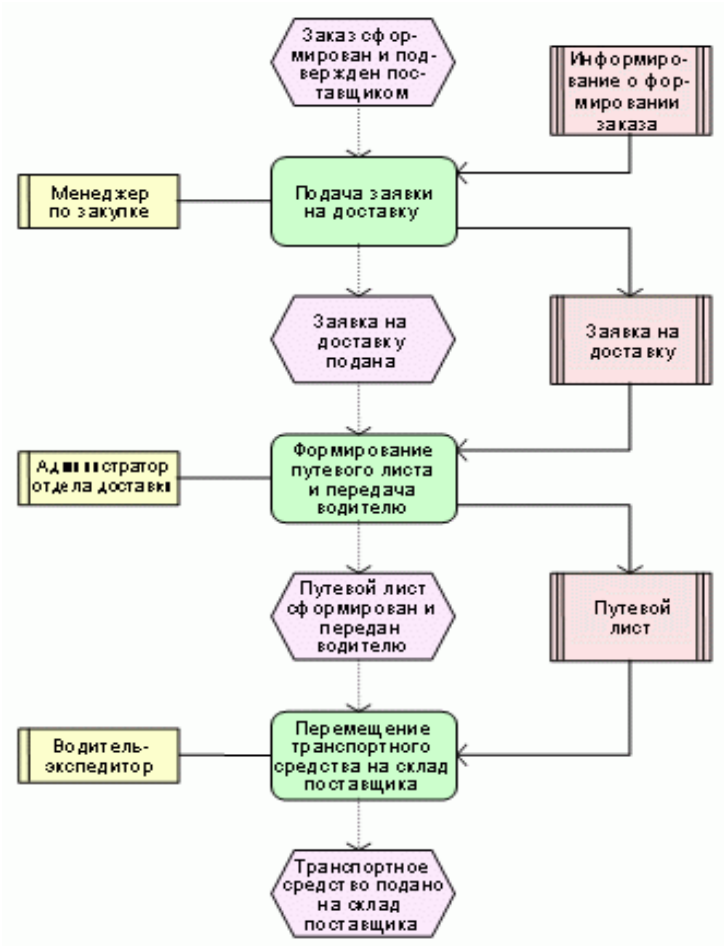


Рис. 2.22. Модель «Расширенная цепочка процессов, управляемая событиями» – eEPC/ARIS

Говорить о преимуществе той или иной методологии бессмысленно, пока не определены тип и рамки проекта, основные задачи, которые данный проект должен решить. В зависимости от решаемых задач эти преимущества могут как усиливаться, так и ослабевать.

Можно сделать следующие выводы:

1. Использовать метод DFD как единственный инструмент описания процессов при разработке системы нецелесообразно. DFD гораздо больше подходит для проектирования информационных систем вообще, баз данных, для описания документооборота и обработки информации. Он позволяет проанализировать информационное пространство системы. Использовать данную методологию рекомендуется в качестве дополнения модели бизнес-процессов, выполненной в IDEF0.

2. ARIS – несомненно, инструмент более функциональный и удобный, чем другие инструменты. Он преодолевает перечисленные недостатки IDEF. Однако следует отметить, что ARIS рационально применять для «сплошной» документации процессов организации, при его использовании для документирования отдельных процессов потребуется предварительное выявление значимых для решения поставленной задачи процессов. Кроме того, программный продукт ARIS имеет цену, на порядок превышающую стоимость инструментов аналогичного класса, и требуются огромные трудозатраты на его разработку.

3. Наиболее приемлемой из рассмотренных методологий процессного моделирования при разработке систем управления на предприятиях России является методология IDEF0 по следующим причинам: приемлемая стоимость разработки моделей; применяя методологию IDEF, можно построить модель такой системы управления, которая в случае необходимости позволит быстро и эффективно скорректировать систему менеджмента в соответствии с новыми условиями и требованиями; четко прослеживается логика и взаимодействие процессов организации, что является основой процессного подхода; возможность получения полной информации о каждой работе (процедуре) благодаря жестко регламентированной структуре.

ГЛАВА 3. ВВЕДЕНИЕ В УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ

3.1. История унифицированного языка моделирования

Унифицированный язык моделирования (Unified Modeling Language – UML) – это универсальный язык визуального моделирования систем, представляющий собой основанную на диаграммах стандартную систему обозначений.

Язык не привязан к какой-либо конкретной методологии или жизненному циклу, но лучше всего адаптирован к методологии унифицированного процесса. UML предоставляет возможность создавать и разбираться в правильно построенных моделях, но не говорит, какие модели и когда нужно создавать. Разработка UML началась в компании Rational Software в 1995 году с объединения метода Booch'93 Гради Буча, техники объектного моделирования OMT (Object Modeling Technique) Айвара Якобсона и методологии объектно-ориентированной разработки программного обеспечения Objectory, или OOSE (Object-Oriented Soft-ware Engineering) Джима Рамбо.

В 1997 году язык UML утвержден консорциумом по технологии манипулирования объектами OMG в качестве открытого стандарта UML 1.1 (http://www.omg.org/uml) для представления объектно-ориентированных моделей. В 2004 г. одобрен второй выпуск UML 2.0.

UML – язык для визуализации, специфицирования, конструирования и документирования компонентов программных средств.

Визуализация (зрительное восприятие). Для многих программистов время между обдумыванием и написанием кода равно нулю. Код получается прекрасный, но программист при этом моделирует в уме. В связи с этим возникает несколько проблем, которые можно решить, используя UML:

1. Чтение кода. Существуют некоторые вещи в ПС, которые невозможно понять по тексту кода (даже хорошо прокомментированного). UML – графический язык и, следовательно, решает проблему.

2. Потеря информации. Если разработчик уничтожил часть кода и никогда не записывал в каком-либо виде модель, тогда этот код будет утрачен навсегда. UML обеспечивает восстановление кода по модели.

3. Интерпретация модели. Как правило, в проектной группе вырабатывается некоторый внутренний язык (совершенно непонятный извне). UML обладает корректноопределенной семантикой, и поэтому разные разработчики будут одинаково трактовать модель.

Специфицирование. UML позволяет определить все важные решения по анализу, проектированию и реализации, которые принимаются в процессе создания и внедрения ПС.

Конструирование. Созданные с помощью UML модели могут быть переведены на различные языки программирования. Модель можно отобразить на такие языки, как C++, Java, Visual Basic и даже на таблицы реляционной базы данных. Возможно, как прямое (forward engineering), так и обратное (reverse engineering) отображение.

Документирование. При создании ПС создается много вторичных по отношению к исполняемому коду продуктов: требования к системе, архитектура системы, проект, исходный код, проектные планы, тесты, прототипы, версии. UML предоставляет возможности документирования принятых решений.

Язык UML предназначен, прежде всего, для разработки программных средств. Его использование наиболее эффективно в следующих областях: информационные системы масштабов предприятия; транспорт, в том числе железнодорожный; банковские и финансовые услуги; распределенные Web-системы; оборонная промышленность, авиация и космонавтика; розничная торговля; медицинская электроника; телекоммуникации; наука.

Если попытаться проследить историю возникновения и развития элементов UML как на уровне основополагающих идей, так и на уровне технических деталей, то пришлось бы назвать сотни имен и десятки организаций (рис. 3.1). Мы не будем этого делать потому, что история развития UML отнюдь не завершена – язык постоянно совершенствуется, обогащается и расширяется.

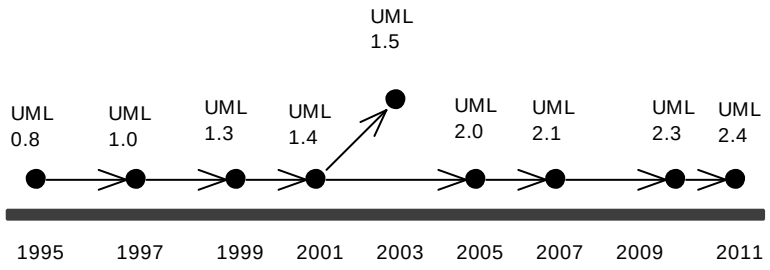


Рис. 3.1. История развития UML

Как видно из рис. 3.1, на особом положении оказалась версия 1.5. Версия 1.5 содержит некоторые элементы версии 2.0, в частности, набор элементарных действий, достаточно широкий для того, чтобы применять UML не только как язык моделирования, но и как язык программирования. Но «генеральная линия» развития инструментальных средств прошла мимо этого явления. Все крупные поставщики инструментов предпочли заявить о поддержке версии 2.0.

3.2. Структура унифицированного языка моделирования

Унифицированный язык моделирования (UML) в настоящий момент является стандартом де-факто при описании (документирования) результатов проектирования и разработки объектно-ориентированных систем. Общая структура UML показана на рис. 3.2.

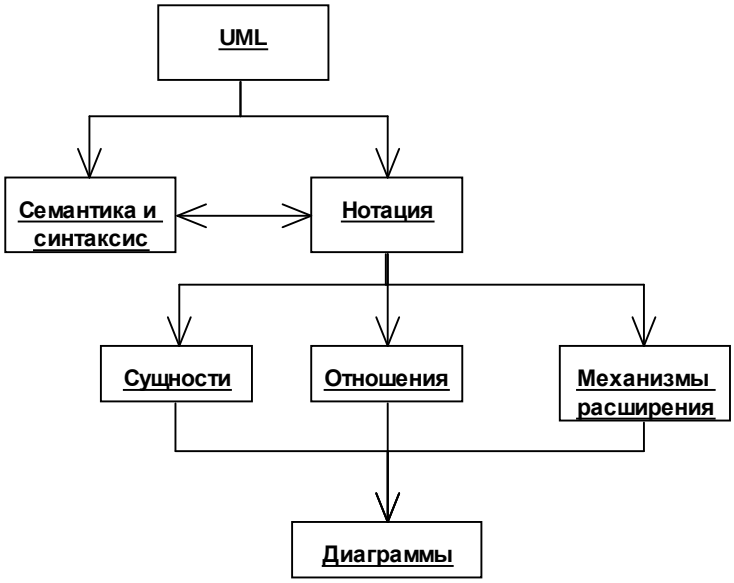


Рис. 3.2. Структура UML

Семантика и синтаксис UML

Семантика – раздел языкознания, изучающий значение единиц языка, прежде всего его слов и словосочетаний

Синтаксис – способы соединения слов и их форм в словосочетания и предложения, соединения предложений в сложные предложения, способы создания высказываний как части текста. Таким образом, применительно к UML, семантика и синтаксис определяют стиль изложения (построения моделей), который объединяет естественный и формальный языки для представления базовых понятий (элементов модели) и механизмов их расширения.

Нотация UML

Нотация представляет собой графическую интерпретацию семантики для ее визуального представления.

В UML определено три типа сущностей:

структурная – абстракция, являющаяся отражением концептуального или физического объекта;

группирующая – элемент, используемый для некоторого смыслового объединения элементов диаграммы;

поясняющая (аннотационная) – комментарий к элементу диаграммы.

Авторы UML определяют его как графический язык моделирования общего назначения (т. е. его можно применять как для проектирования

простой качели, так и сложного аппаратно-программного комплекса или даже космического корабля), предназначенный для спецификации, визуализации, проектирования и документирования всех артефактов, создаваемых в ходе разработки.

Итак, UML в первую очередь – это спецификации. В глоссарии *спецификация* определяется как подробное описание системы, которое полностью определяет ее цель и функциональные возможности. Различают следующие спецификации:

словесные на естественном языке;

модельные;

формальные.

В настоящее время все вопросы дальнейшей разработки языка UML сконцентрированы в рамках консорциума OMG. При этом статус языка UML определен как открытый для всех предложений по его доработке и совершенствованию. Сам язык UML не является чьей-либо собственностью и не запатентован кем-либо, хотя указанный выше документ защищен законом об авторском праве. В то же время аббревиатура UML, как и некоторые другие (OMG, CORBA, ORB), является торговой маркой их законных владельцев, о чем следует упомянуть в данном контексте.

3.3. Типы диаграмм UML 2.0

Авторы UML выделяют следующие типы диаграмм (diagram types) (рис. 3.3):

1. Структурные диаграммы:

классов (class diagrams). Предназначены для моделирования структуры объектно-ориентированных приложений – классов, их атрибутов и заголовков методов, наследования, а также связей классов друг с другом;

компонент (component diagrams). Используются при моделировании компонентной структуры распределенных приложений; внутри каждая компонента может быть реализована с помощью множества классов;

объектов (object diagrams). Применяются для моделирования фрагментов работающей системы, отображая реально существующие в runtime экземпляры классов и значения их атрибутов;

композитных структур (composite structure diagrams). Используются для моделирования составных структурных элементов моделей – кооперации, композитных компонент и т. д.;

развертывания (deployment diagrams). Предназначены для моделирования аппаратной части системы, с которой ПО непосредственно связано (размещено или взаимодействует);

пакетов (package diagrams). Служат для разбиения объемных моделей на составные части, а также (традиционно) для группировки классов моделируемого ПО, когда их слишком много.

2. Поведенческие диаграммы:

активностей (activity diagrams). Используются для спецификации бизнес-процессов, которые должны автоматизировать разрабатываемое ПО, а также для задания сложных алгоритмов;

случаев использования (use case diagrams). Предназначены для «вытягивания» требований из пользователей, заказчика и экспертов предметной области;

конечных автоматов (state machine diagrams). Применяются для задания поведения реактивных систем;

3. Диаграммы взаимодействий (interaction diagrams):

последовательностей (sequence diagrams). Используются для моделирования временных аспектов внутренних и внешних протоколов ПО;

схем взаимодействия (interaction overview diagrams). Служат для организации иерархии диаграмм последовательностей;

коммуникаций (communication diagrams). Являются аналогом диаграмм последовательностей, но по-другому изображаются (в привычной, графовой, манере);

временные (timing diagrams). Являются разновидностью диаграмм последовательностей и позволяют в наглядной форме показывать внутреннюю динамику взаимодействия некоторого набора компонент системы.

Не все узлы обозначают типы диаграмм – некоторые изображают лишь группы диаграмм, например, «Структурные», «Поведенческие», «Взаимодействий» (рис. 3.3).

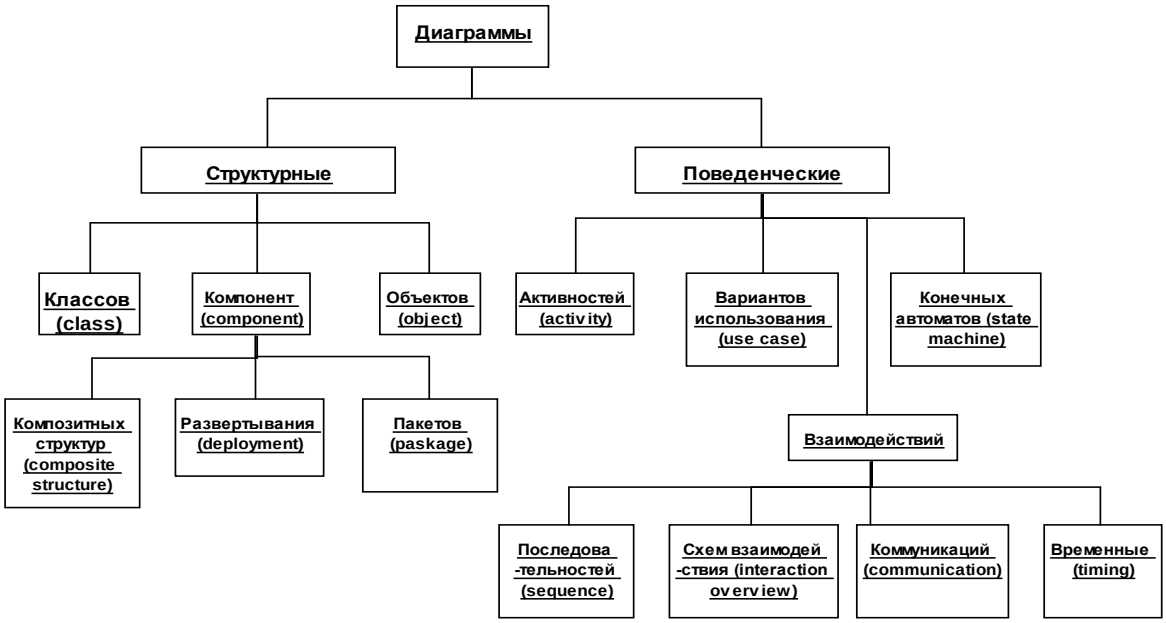


Рис. 3.3. Типы диаграмм UML 2.0

Отметим новые типы диаграмм, которые появились в UML 2.0 по сравнению с версией UML 1.5:

композитных структур (composite structure diagrams);

схем взаимодействий (interaction overview diagrams) – прообразом явились диаграммы MSC overview;

коммуникаций (communication diagrams) – это упрощенный вариант диаграмм кооперации UML 1.5;

временные (timing diagrams) – это новый тип диаграмм, предназначенный для наглядного изображения потока изменения состояний нескольких объектов.

Структура UML включает: строительные блоки – основные элементы, отношения и диаграммы; общие механизмы – пути для достижения определенных целей; архитектуру – представление системной архитектуры.

3.4. Строительные блоки UML

В UML задействованы три вида блоков: сущности (things); отношения (relationships); диаграммы (diagrams).

Сущности. Сущность – это абстракция, являющаяся основным элементом UML-модели. Графические обозначения некоторых сущностей показаны на рис. 3.4.

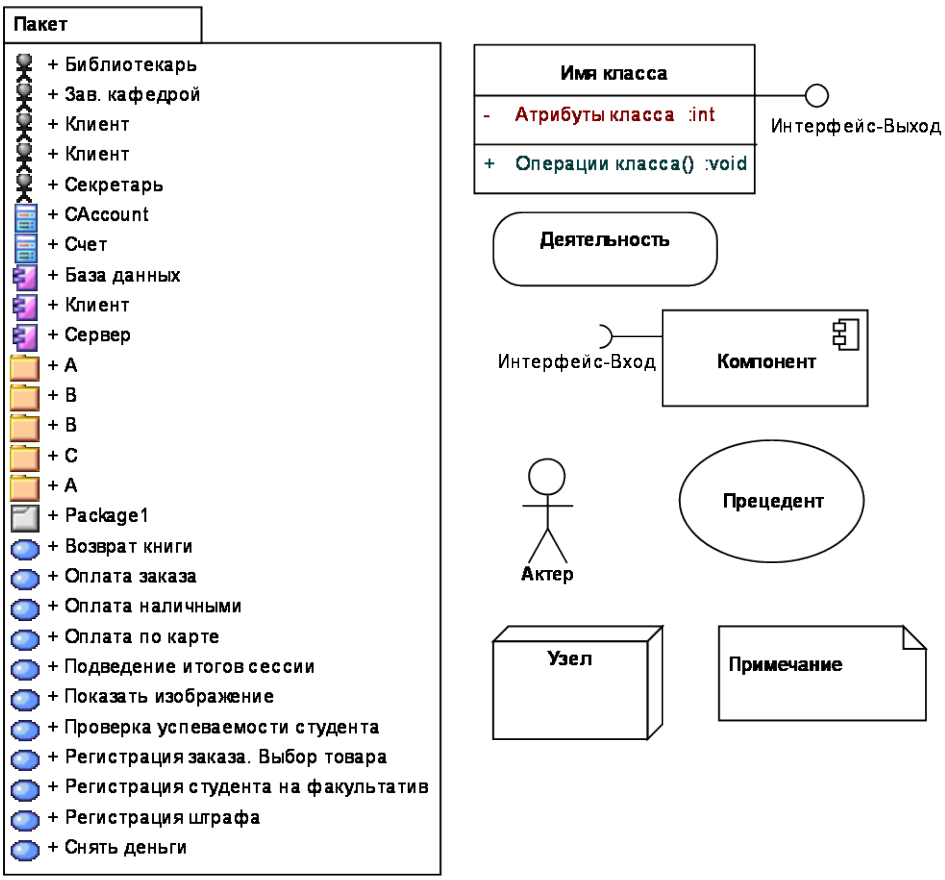


Рис. 3.4. Графические обозначения сущностей в UML

Все сущности можно разделить на следующие группы:

структурные сущности. Представляют собой статические части модели. Структурными сущностями являются: класс (class), интерфейс (interface), актер (actor), прецедент (use case), компонент (component), узел (node);

поведенческие сущности – динамические части UML-модели. К поведенческим сущностям относятся: взаимодействия (interaction), деятельности (activity), автоматы (state machine);

группирующая сущность – пакет (package). Объединяет семантически связанные элементы UML-модели в единое целое;

аннотационная сущность – примечание (note). Добавляется к UML-модели для записи специальных уточняющих сведений.

Класс (class)

Класс – это описание набора объектов с общими атрибутами, операциями и семантикой. Графически класс изображается в виде разделенного на три части прямоугольника, где записаны его имя, атрибуты и операции (рис.3.5).

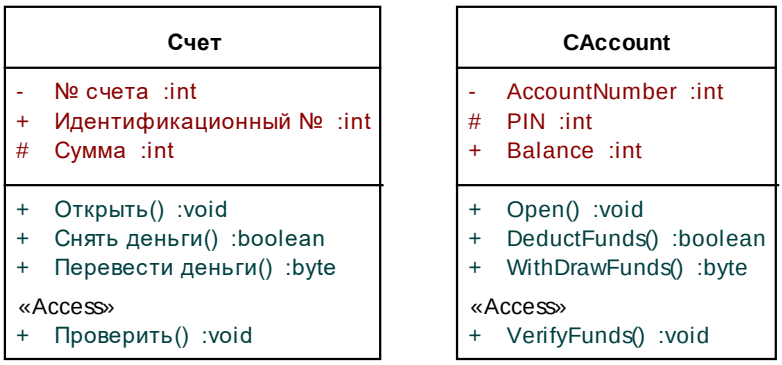


Рис. 3.5. Графическое изображение класса

Объединение объектов в классы вводит в процесс разработки ПС абстракцию, цель которой – ограничить универсальность, всеобщность понятия (вещи). Абстракция – это наиболее существенные характеристики объекта, которые отличают его от всех других видов объектов и четко определяют особенности данного объекта с точки зрения дальнейшего анализа. Абстракция концентрирует внимание на внешних свойствах объекта и позволяет отделить существенные особенности поведения от деталей их реализации. Такое разделение поведения от реализации называется *барьером абстракции*, который основывается на двух принципах:

наименьшей выразительности, по которому абстракция должна охватывать лишь самую суть объекта, не больше (но и не меньше);

минимизации связей, когда интерфейс объекта содержит только существенные аспекты поведения.

Следует заметить, что на разных этапах создания ПС разработчики имеют дело с разными классами:

на этапе анализа ПС классы – это концептуальные абстракции, являющиеся частью предметной области;

проектирования – логические (программные) абстракции;

реализации – это классы выбранного для реализации логических абстракций языка программирования.

Одна из целей разработки ПС заключается в том, что необходимо описать классы/объекты, составляющие в совокупности проектируемую систему. Основными характеристиками класса/объекта являются:

1. Уникальное имя. Позволяет отличить объекты друг от друга.

2. Атрибут – именованное свойство объектов класса. Каждый атрибут имеет свое значение для каждого объекта. Среди атрибутов различают:

статические (условно постоянные свойства);

динамические (условно переменные свойства);

производные (вычисляемые свойства), значения которых определяются при помощи значений статических и динамических атрибутов объекта.

3. Состояние объекта – ситуация в ходе жизни объекта, в течение которой он удовлетворяет некоторому условию, выполняет некоторую деятельность или ожидает некоторого события. Выражается состояние всеми текущими значениями объекта. При изменении значений атрибутов меняется состояние.

4. Поведение – это то, как объект действует и реагирует (т. е. как он меняет свое состояние). Поведение выражается в терминах состояния объекта и передачи сообщений. Поведение может изменять состояние. Понятие *поведение* совпадает с понятием *операция*.

5. Операция – действие, которое должен выполнить объект для реализации своего поведения, или сервис, который может быть востребован одним объектом у другого. У каждой операции есть свой объект-получатель, т. е. объект, для которого операция применяется. Операция может иметь дополнительные аргументы, которые ее параметризуют. Некоторые операции могут приобретать разные формы в различных классах, такие операции называют полиморфными. К стереотипным операциям для всех классов относятся:

создать и инициировать *(create and initialize)* объект;

удалить (delete) объект;

получить (get…) значение атрибута;

установить (set…) значение атрибута;

добавить объект (add…) – добавить связь с другим объектом;

исключить объект (remove…) – исключить связь с другим объектом.

К наиболее распространенным операциям относятся:

модификатор – меняет состояние объекта;

запрос (селектор) – считывает состояние объекта, но не меняет состояние;

итератор – позволяет организовать доступ ко всем частям объекта-контейнера для последующего их использования в строго определенной последовательности (например, просмотр очереди);

преобразование – производит объект другого типа.

6. Метод – конкретная реализация операции.

Актер (actor)

Актер – это любая сущность, которая взаимодействует с системой извне. Графически изображается так, как показано на рис. 3.6.

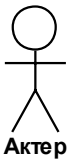


Рис. 3.6. Графическое изображение актера

Актеры не подлежат контролю со стороны ПС, потому что определяют вct то, что находится вне системы. Актеры делятся на четыре основных типа:

1. *Физические личности*. Например, в банковской системе к актерам относятся клиенты и обслуживающий персонал.

2. *Другая система*. Когда другая система становится действующим лицом, предполагается, что она не будет меняться вообще. Например, у банка имеется кредитная система для работы с кредитными счетами клиентов. Банковская система должна иметь возможность взаимодействовать с кредитной системой, в таком случае последняя становится актером.

3. *Время*. Так как время не подлежит контролю, оно является актером. Время становится актером, если от него зависит запуск каких-либо процессов в системе. Например, банковская программная система может каждую полночь выполнять служебные процедуры по согласованию своей работы.

4. *Электромеханические устройства*. Для последующей обработки ПС может получать сигналы от этих устройств, а также само управлять их работой.

Прецедент – use case

Прецедент (вариант использования) представляет собой описание системной функции, которая производит конкретный результат, значимый (полезный) для конкретного пользователя (участника-актера). Графическое изображение показано на рис. 3.7.

Рис. 3.7. Графическое изображение прецедента

Прецедент – это документ с текстовым описанием последовательности действий, которые должны быть выполнены системой для актера по его запросу. Последовательности действий называют *потоками событий*. *Нормальная последовательность действий* (без отклонений и без ошибок), которая должна быть выполнена ПС для

достижения конкретного результата, *называется основным потоком событий*. *Альтернативные потоки событий* отражают отклонения от основного потока событий, а также потоки ошибок. Основной поток событий всегда один, альтернативных потоков может не быть или может быть несколько альтернативных потоков для одного основного.

Предполагается, что никакой поток событий не может быть прерван и должен быть обязательно выполнен от начала до конца. В итоге выполнения основного потока участник-актер получает от системы результат. Если основной поток не может быть выполнен, то по какому-либо альтернативному потоку система сообщает о невозможности получения результата. С основным потоком событий связаны два понятия: предусловие и постусловие. *Предусловие* – это другой прецедент, который должен быть выполнен прежде, чем вариант использования начнет свою работу. *Постусловие* – это прецедент, который должен быть выполнен после завершения текущего варианта использования. Не у всех вариантов использования бывают предусловия и постусловия. Пример основного потока для прецедента *Снятие денег со счета*:

1. Клиент банка вставляет карту в прорезь банкомата.
2. Клиент отвечает на вопрос о коде.
3. Система проверяет соответствие карты и кода.
4. Клиент вводит сумму.
5. Система проверяет наличие соответствующей суммы на счету клиента.
6. Клиент получает деньги.

Для этого же прецедента альтернативные потоки для строк 3 и 5: введенный код не соответствует карте; система сообщает клиенту о неверном коде; система предлагает ввести другой код или завершить сеанс; баланс счета меньше введенной суммы; система сообщает клиенту о невозможности снятия денег со счета; система завершает сеанс работы с клиентом.

Компонент – component

Компонент – это физический элемент реализации с четко определенным интерфейсом, предназначенный для использования в качестве заменяемой части системы. Компонент изображается в виде прямоугольника с вкладками (рис. 3.8).

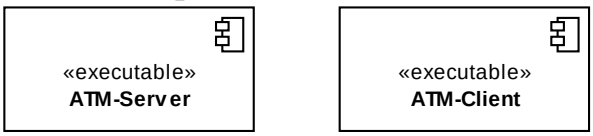


Рис. 3.8. Графическое изображение компонента

В отличие от классов, являющихся логическими абстракциями, компоненты представляют собой их физическую реализацию. Любой

компонент может быть заменен другим, который поддерживает точно такой же интерфейс. К наиболее распространенным стереотипам компонентов относятся:

1. <<EXE>> – исполняемый *(executable)* компонент, который может быть выполнен компьютером.

2. <<DLL>> – динамическая объектная библиотека *(library)*.

3. <<DB>> – реляционная или объектная база данных *(database)*.

Компоненты имеют следующие особенности:

включают отношения тех классов ПС, которые они реализуют;

способ реализации компонентов зависит от используемого языка программирования.

Узел – node

Узел – физически существующий элемент системы, который представляет собой процессор или устройство. *Процессор (processor)* – узел, способный обрабатывать данные, т. е. исполнять компонент. *Устройство (device)* – узел, не способный обрабатывать данные. Графическое обозначение показано на рис. 3.9.

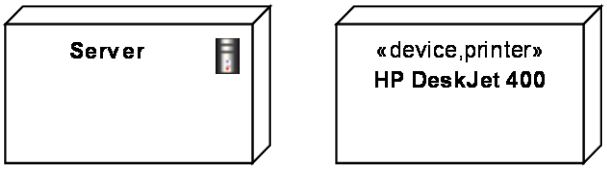


Рис. 3.9. Графическое обозначение узлов

В настоящее время понятие узла включает в себя не только вычислительный ресурс (процессор, некоторый объем памяти), но и другие электромеханические или электронные устройства: датчики; модемы; сканеры; принтеры; цифровые камеры; манипуляторы.

Между узлами и компонентами существует соответствие:

узлы исполняют компоненты. Компоненты исполняются на узлах;

узлы – это средства физического развертывания компонентов. Компоненты включают в спецификацию узла как процесс. Один и тот же компонент можно разворачивать на одном или нескольких узлах системы.

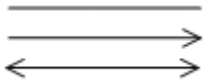
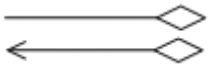
Множество компонентов, приписанных на узел как группа, называется элементом распределения (Distribution Unit). Каждый узел должен иметь уникальное имя, которое может быть произвольной последовательностью символов. Например, Сервер, Принтер HP DeskJet 400. Дополнительно к узлу можно использовать стереотипы. Несмотря на то что стандартных стереотипов в UML не существует, они приписываются узлам. Например, *процессор, консоль, сеть, модем, факс, принтер,*

плоттер, датчик. Чаще всего используются стереотипы: *процессор* и *устройство*.

3.5. Отношения

Отношением (relationship) называется набор однотипных связей между отдельными элементами – актерами и прецедентами, классами, компонентами (табл. 3.1).

Таблица 3.1. Отношения

Наименование	Обозначение	Определение (семантика)
Ассоциация (association)		Отношение, описывающее значимую связь между двумя и более сущностями. Наиболее общий вид отношения
Агрегация (aggregation)		Подвид ассоциации, описывающей связь «часть» – «целое», в котором «часть» может существовать отдельно от «целого». Ромб указывается со стороны «целого». Отношение указывается только между сущностями одного типа
Композиция (composition)		Подвид агрегации, в которой «части» не могут существовать отдельно от «целого». Как правило, «части» создаются и уничтожаются одновременно с «целым»
Зависимость (dependency)		Отношение между двумя сущностями, в котором изменение в одной сущности (независимой) может влиять на состояние или поведение другой сущности (зависимой). Со стороны стрелки указывается независимая сущность
Обобщение (generalization)		Отношение между обобщенной сущностью (предком, родителем) и специализированной сущностью (потомком, дочкой). Треугольник указывается со стороны родителя. Отношение указывается только между сущностями одного типа
Реализация (realization)		Отношение между сущностями, где одна сущность определяет действие, которое другая сущность обязуется выполнить. Отношения используются в двух случаях: между интерфейсами и классами (или компонентами), между вариантами использования и кооперациями. Со стороны стрелки указывается сущность, определяющая действие (интерфейс или вариант использования)

Отношения между классами

Связь (link) – это физическое или концептуальное соединение между объектами. Например, *Максим учится в ТвГТУ*. Чаще всего связь соединяет ровно два объекта. Отношение – это набор связей, которые обладают общей структурой и общей семантикой. Например, *студент может учиться в каком-либо учебном заведении*. Отношение описывает множество потенциальных однотипных связей точно так же, как класс описывает множество объектов. Связи – экземпляры отношения, соединяющие объекты тех классов, между которыми отношение установлено. Самым распространенным типом отношений является *ассоциация (association)*, которая отражает значимые и полезные связи объектов. По своей природе ассоциация – двусторонняя, имеет два конца и может быть прослежена как в одном, так и в обоих направлениях. В UML двунаправленная ассоциация обозначается при помощи прямой линии. Когда ассоциация однонаправленная, линия заканчивается стрелкой, указывающей направление ассоциации. Оба вида обозначений показаны на рис. 3.10.

Рис. 3.10. Обозначение ассоциаций

Один конец ассоциации называется *полюсом*. Полюс обладает кратностью, ограничивающей количество связанных между собой объектов. *Кратность (multiplicity)* – это количество объектов одного класса, которые могут быть связаны с одним объектом класса противоположного конца ассоциации. Кратность указывается рядом с полюсом под линией ассоциации и может принимать одно из следующих значений:

1 – точно один объект;

0…1 – ноль или один объект;

0…* – ноль или больше объектов;

1…* – один или больше объектов;

* – много объектов.

Двунаправленная ассоциация с кратностью на одном полюсе – *много*, а на другом – *один* показана на рис. 3.11. Поскольку эта ассоциация двунаправленная, справедливы оба утверждения: *много людей работает на одном предприятии* и *на одном предприятии работает много людей*.

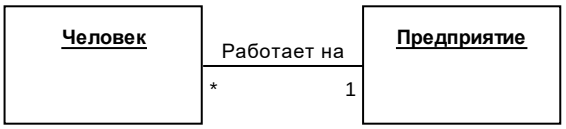


Рис. 3.11. Кратность ассоциации

Полюс может иметь не только кратность, но и располагать именем роли, которое описывает роль объектов. Представление ролей в ассоциации не является обязательным, однако значительно улучшает понимание диаграмм классов, а также обеспечивает простоту добавления уникальных атрибутов при программировании классов. Имя роли указывается около конца ассоциации. Роли в двух однонаправленных ассоциациях: *управляет* и *назначает* показаны на рис. 3.12. Роль объектов класса *человек* в ассоциации *назначает* определена как *менеджер*: *одно предприятие назначает некоторых из многих людей менеджерами*. В ассоциации *управляет* повторно используется роль *менеджер* и назначается вторая – *рабочий*: *один человек, являющийся менеджером, управляет многими людьми, которые являются рабочими предприятия*.

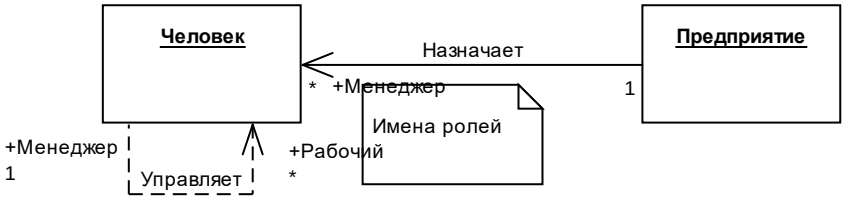


Рис. 3.12. Роли в ассоциации

Агрегация (aggregation) – специализированный вид ассоциации, который обозначает отношение «часть целого» *(part of has)*, где каждая пара классов определяется отдельно. Агрегация имеет два важных свойства:

транзитивность *(если A является частью B, а B – частью C, то A является частью C);*

асимметричность *(если A является частью B, то B не является частью A)*.

Для отношения «часть целого» в UML существует две формы: *общая (агрегация)* и *частная (композиция)*. Композиция *(composition)* – особый случай агрегации с дополнительными ограничениями:

часть может принадлежать не более чем одному целому;

приписанная к некоторому целому часть автоматически получает срок жизни, совпадающий со сроком жизни целого.

Агрегация допускает независимую обработку объектов-частей и объекта-целого. Обработка частей в композиции возможна только через объект целого, равно как и доступ внешних объектов к частям.

Кратность присутствует в каждой форме отношений «часть целого» и ставится на стороне частей. Если ее значение не указано, то предполагается, что в отношении участвует много частей и одно целое. В UML агрегация обозначается линией с полым ромбом около полюса целого, у композиции ромб закрашен (рис. 3.13).

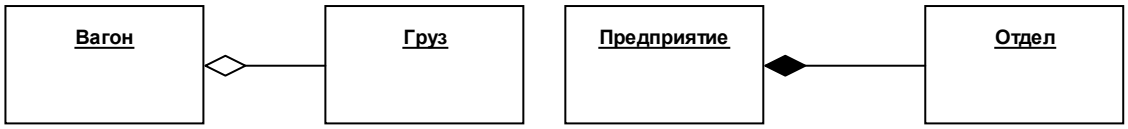


Рис. 3.13. Обозначение агрегации и композиции

Из диаграммы видно, что *в одном вагоне много грузов* (при потере вагона не обязательно исчезнут все грузы), *на одном предприятии много отделов* (при ликвидации предприятия отделы перестают существовать). Пример агрегации, где тепловоз *(агрегат)* состоит из секций *(частей)* и дизелей *(частей)* показан на рис. 3.14. Каждый дизель *(как агрегат)* имеет в своем составе топливные насосы *(части)*. По свойству транзитивности насосы – части тепловоза. По свойству асимметричности агрегации тепловоз не является частью топливных насосов.

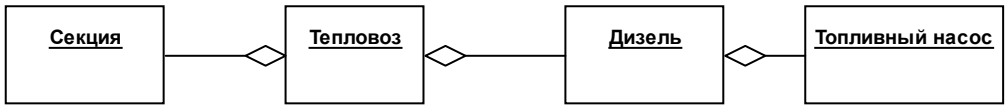


Рис. 3.14. Агрегация

Обобщение (generalization) является частным случаем ассоциации и обозначает отношение типа «общее – частное» *(is – a)*. Обобщение указывает на наличие общих характеристик между классами. Класс, имеющий наиболее общие для некоторой группы классов характеристики, называется *суперклассом*, а его специализированная версия – *подклассом*. Открытые и защищенные атрибуты и операции суперкласса наследуются всеми подклассами. Любой подкласс может добавлять уточняющие характеристики к унаследованным. Дополнительные атрибуты и/или операции в подклассах могут отсутствовать. Отношение обобщения не является набором связей между объектами, поэтому направление и кратность для обобщения не указываются. Обобщение допускает многоуровневую иерархию классов, но лучше ограничиться тремя уровнями, чтобы не усложнять модификацию программ, так как изменение суперкласса требует корректировки подклассов дополнительного тестирования.

Программируется обобщение при помощи механизма наследования. Суперкласс становится базовым классом (как правило, абстрактным), подклассы реализуются как производные классы. Следует заметить, что

абстрактный класс может иметь чистые виртуальные функции, следовательно, во всех его производных классах должны быть определены методы для реализации таких функций. В UML обобщение обозначается линией с треугольником на конце. Вершина треугольника направлена в сторону суперкласса (рис. 3.15).

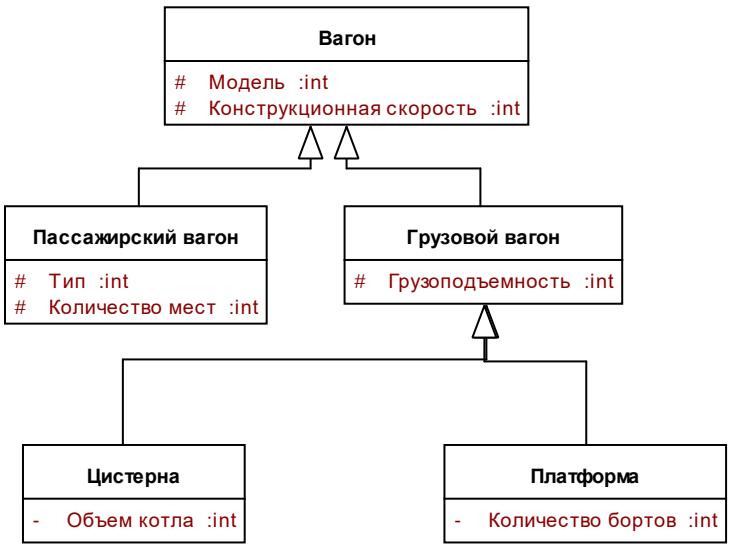


Рис. 3.15. Обобщение

Первый уровень иерархии на диаграмме содержит класс *Вагон*, являющийся суперклассом для двух классов второго уровня: *Грузовой вагон* и *Пассажирский вагон*. Оба класса наследуют от суперкласса *Вагон* защищенные атрибуты *(модель и конструкционная скорость)*, а также имеют дополнительные. *Грузовой вагон* служит суперклассом для классов третьего уровня иерархии: *Цистерна* и *Платформа*. К их закрытым атрибутам добавляются атрибуты классов *Грузовой вагон* и *Вагон*. Дискриминатор *(discriminator)* показывает признак, указывающий, по какому свойству объектов сделано обобщение.

Зависимость (dependency) – это однонаправленное отношение использования между двумя классами. На одном конце отношения находится зависимый класс, на втором – независимый. Объект-клиент зависимого класса для своего корректного функционирования пользуется услугами объекта-сервера независимого класса. Зависимость отражает связь между объектами по применению, когда изменение поведения сервера может повлиять на поведение клиента. В UML отношение зависимости изображается пунктирной стрелкой, всегда направленной в сторону независимого класса (рис. 3.16).

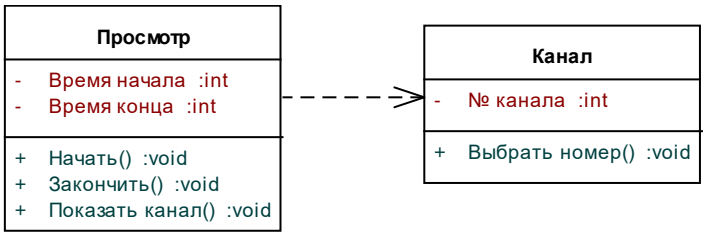


Рис. 3.16. Зависимость

На диаграмме показана зависимость просмотра тех или иных передач от выбора телевизионного канала зрителем. Объект-клиент *просмотр* использует объект-сервер *канал* для реализации операции *показать ()*. Результатом выполнения операции объектом *просмотр* станет смена его состояния.

Класс ассоциации (association class) позволяет назначить для ассоциации атрибуты и операции, которые являются принадлежностью связей и не могут быть приписаны ни к одному из объектов-участников. Для одной ассоциации может существовать только один класс ассоциации. Важная особенность класса ассоциации заключается в том, что для объектов-участников создается только один объект этого класса. Обозначается класс ассоциации как обычный класс, соединенный пунктирной линией с той ассоциацией, которую он уточняет (рис. 3.17).

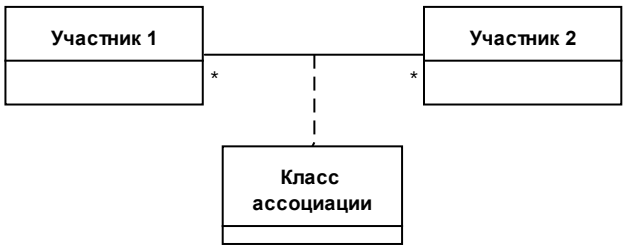


Рис. 3.17. Обозначение класса ассоциации

Основанием для введения классов ассоциации послужили ассоциации с кратностью *многие–ко–многим*, где возникает неоднозначность в определения атрибутов классов-участников. Например, в сессию основной экзамен по дисциплине проводится для студента один раз. Между классами *Студент* и *Дисциплина* существует ассоциация *многие–ко–многим*: *один студент экзаменуется по многим дисциплинам*, и *по одной дисциплине – много студентов*. Полученные студентом оценки не могут быть атрибутами ни одного названного класса. Диаграмма с классом ассоциации для регистрации результатов экзаменов показана на рис. 3.18. Дата и оценка экзамена конкретного студента по конкретной дисциплине хранятся объектом класса ассоциации *экзамен*. Для каждой пары объектов студент – дисциплина будет существовать один объект экзамен.

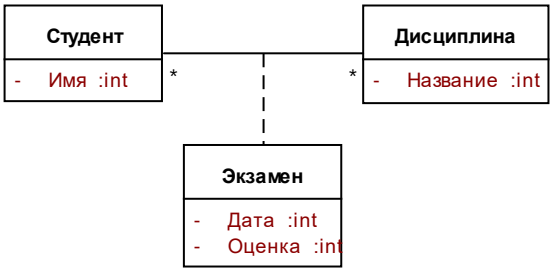


Рис. 3.18. Диаграмма с классом ассоциации для регистрации результатов экзаменов

Класс ассоциации может участвовать в другой ассоциации, как показано на рис. 3.19. Теперь объект *экзамен* знает об объекте *преподаватель*, чтобы сохранить данные о педагоге, принявшем у студента экзамен по дисциплине. Так как не все преподаватели участвуют в приеме экзаменов, для объектов класса определена роль *экзаменатор*.

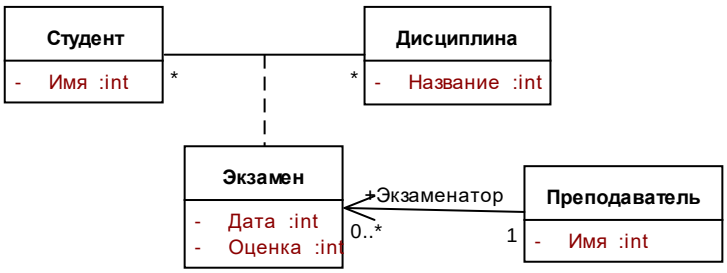


Рис. 3.19. Класс ассоциации как участник другой ассоциации

Классы ассоциации существенно отличаются от обычных классов, потому что для них разрешается только один объект. Если требуется создание нескольких или многих объектов для одной и той же пары участников, используется отдельный класс. Пусть необходимо регистрировать результаты не только основного экзамена, но еще двух дополнительных, которые могут сдавать некоторые студенты. Для решения задачи класс ассоциации преобразован в обычный класс, как показано на рис. 3.20.

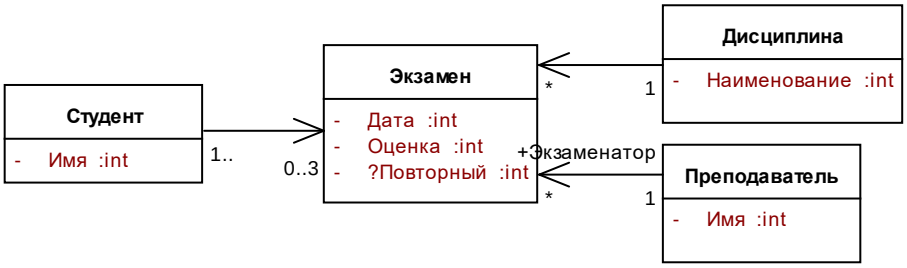


Рис. 3.20. Диаграмма с обычным классом для регистрации результатов экзаменов

Здесь класс *Экзамен* уже не является классом ассоциации. Это обычный класс, который участвует в трех следующих ассоциациях:

1. Студент – Экзамен: *один студент не сдает экзаменов вообще или экзаменуется не более трех раз*.

2. Дисциплина – Экзамен: *по одной дисциплине проводится много экзаменов*.

3. Преподаватель – Экзамен: *один преподаватель (являющийся экзаменатором) принимает много экзаменов*.

Для делегирования композиция и обобщение используются совместно, что повышает устойчивость системы.

Наличие отношений между классами добавляет в классы дополнительные атрибуты, которые, как правило, реализуются посредством указателей.

Отношения между актерами и вариантами использования

Отношение между актером и прецедентом называют *коммуникативной ассоциацией (communicate association)*. Обычно отношение направлено от актера к прецеденту (рис. 3.21).

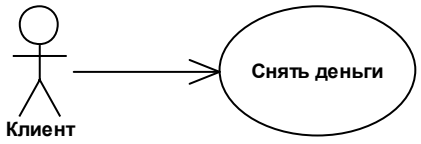


Рис. 3.21. Коммуникативная ассоциация

Между прецедентами существует отношение зависимости двух типов:

1. Включение *(include relationship)*.

2. Расширение *(extend relationship)*.

Отношение включения используется, когда разные прецеденты имеют общее поведение, т. е. их основные потоки содержат одну и ту же последовательность. Общее поведение может быть представлено в виде *отдельного* прецедента, благодаря чему удается избежать многократного описания одного и того же потока событий. Отношение включения между прецедентами означает, что в некоторой точке базового прецедента агрегировано поведение другого прецедента. Включаемый прецедент никогда не существует автономно, он всегда рассматривается как часть базового прецедента. В UML отношение включения обозначается в виде зависимости со стереотипом <<include>>, которая направлена от базового к включаемому прецеденту. Пример отношения включения показан на рис. 3.22.

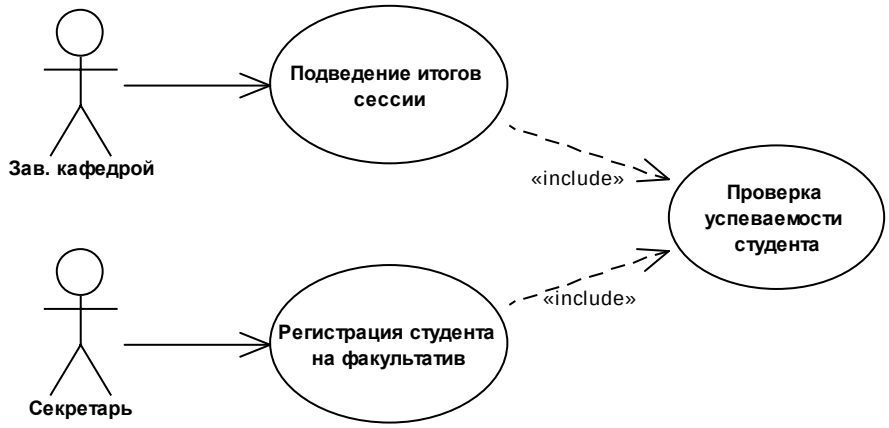


Рис. 3.22. Пример включения

Если *разные* прецеденты включают *один и тот* же прецедент, он должен выполняться для обоих абсолютно одинаково. Базовый прецедент не завершится до тех пор, пока от начала и до конца не будет выполнен включаемый. Следует помнить, что базовый прецедент не полон без включаемого прецедента. *Отношение расширения* добавляет к базовому прецеденту дополнительное поведение другого прецедента. Базовый прецедент в отношении расширения должен быть самостоятельным, допускающим применение без всяких дополнений (в отличие от включения). Дополняющий прецедент выполняется, когда реализация базового прецедента доходит до точки вставки расширения, и никогда не рассматривается в качестве прецедента, выполнение которого является обязательным. Расширение применяется для моделирования:

1. Отдельных потоков событий, которые выполняются в некоторой точке базового прецедента только при определенных условиях.

2. Нескольких потоков событий, которые могут быть выполнены в некоторой точке базового прецедента в результате явного взаимодействия с актером.

3. Частей прецедента, которые актер воспринимает как необязательное поведение системы.

Базовый прецедент в отношении расширения дополняется одной или несколькими точками расширения *(extension points)*.

В UML отношение расширения изображается в виде зависимости со стереотипом <<extend>>, которая направлена от расширяющего прецедента к базовому. На рис. 3.23 показаны примеры отношения расширения.

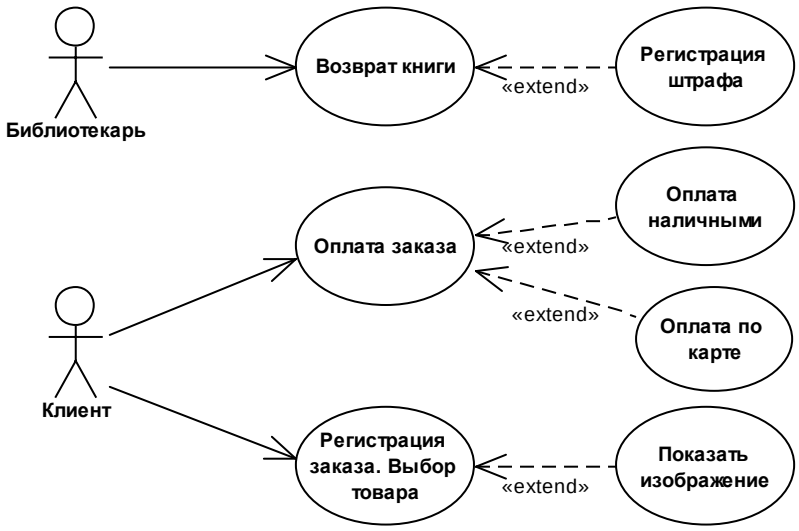


Рис. 3.23. Примеры расширения

Следует заметить, что не имеет смысла связывать друг с другом актеров, поскольку они действуют вне системы.

В UML имеется несколько стандартных видов отношений между *актерами* и *вариантами использования* (табл. 3.2, 3.3).

Таблица 3.2. Виды отношений между актерами и вариантами использования

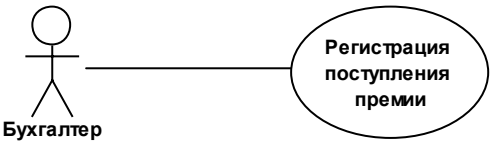
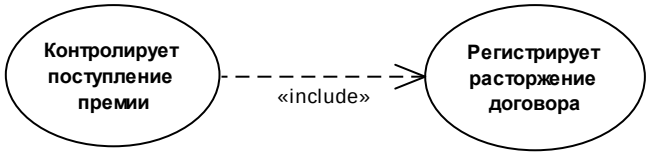
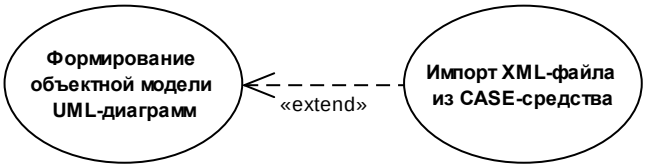
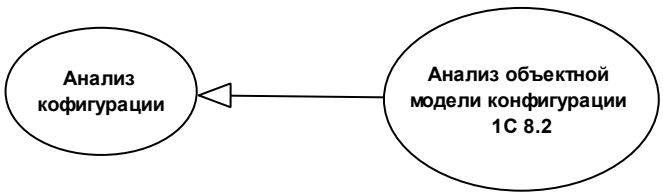
Отношение	Пример
Ассоциации (association relationship)	
Включения (include relationship)	
Расширения (extend relationship)	
Обобщения (generalization relationship)	

Таблица 3.3. Сравнения основных типов отношений между вариантами использования

Отношение	Описание
1	*2*
Обобщение (Generalization), непрерывная линия с белым треугольником от варианта *B* к варианту *A*	*B* является разновидностью *A*. Другими словами, дочерний сценарий получает шаги из родительского и некоторые может переопределить
A <<*includes*>> *B*	В процессе выполнения сценария *A* вызывается сценарий *B*. Когда выполнение *B* завершается, продолжается выполнение *A*. Можно сказать, что *B* – часть *A*. *A* зависит от *B* и без него выполнятся не может. Стрелка рисуется от *B* к *A* аналогично агрегации

Окончание табл. 3.3

1	*2*
A <<*precedes*>> *B*	Сценарий *A* должен быть выполнен полностью до начала выполнения *B*
A <<*extends*>> *B*	Все шаги сценария *A* выполняются в некоторой точке (точке расширения) при выполнении сценария *B*. Можно сказать, что это отношение <<includes>> наоборот <<includes>> и <<extends>> являются разновидностями <<invokes>>, но <<invokes>> не является разновидностью <<includes>> или <<extends>> Стрелка рисуется от *A* к *B* аналогично наследованию *B* должен знать, что его расширяет *A* (отличие от наследования). *B* узнает об этом через точки расширения. Таким образом, это опциональное дополнение сценария *B* шагами из сценария *A*
A <<*invokes*>> *B*	Сценарий *B* необходим во время выполнения сценария *A*

Написание текстов вариантов использования

При написании текстов необходимо:

стремиться к наименьшей неоднозначности;

использовать повествование от третьего лица (пользователь сделал то и то, система ответила тем то);

использовать термины и понятия из предметной области;

дополнять при необходимости модель новыми понятиями;

описывать сценарий варианта использования не более, чем двумя абзацами;

полностью описывать действия пользователя из альтернативных сценариев в случаях, если что-то мешает ему выполнить основной сценарий;

использовать в тексте вариантов использования названия страниц или окон интерфейса.

Например, для варианта использования Write Reader Review текст основного сценария может быть таким:

Пользователь вводит *Обзор книги*, задает *Рейтинг книги*, выбирая из пяти возможных баллов, и нажимает кнопку *Отправить*.

Этот текст подразумевает, что пользователь ввел все данные правильно, не превысил максимальную длину текста, указал допустимое количество баллов и т. д.

Однако нужно постоянно задавать себе вопрос: что может произойти еще, кроме действий основного сценария?

Например, пользователь может быть не авторизован. Он может ввести слишком длинный или слишком короткий текст. Что значит слишком короткий? Пустой или не превышающий десяти символов? На все эти вопросы должны быть даны ответы в альтернативных сценариях.

Пример.

Основной сценарий. Пользователь щелкает на кнопке *Написать отзыв для текущей книги*. Система показывает страницу для написания отзыва. Пользователь вводит *Отзыв*, задает *Рейтинг*, выбирая из возможных пяти баллов, и нажимает кнопку *Отправить*. Система проверяет, что *Отзыв* не слишком короткий и не слишком длинный и баллы входят в допустимый диапазон. Система показывает страницу с подтверждением, и отзыв отправляется на проверку модератору. Варианты использования не являются алгоритмами выполнения задач (табл. 3.4).

Альтернативный сценарий:

1. Пользователь не авторизован. Система перенаправляет пользователя на страницу авторизации и если он авторизуется, то возвращает его на страницу написания отзыва.

2. Пользователь ввел слишком длинный текст (> 1 Mb). Система отвергает отзыв и выводит сообщение, объясняющее, почему отзыв был отвергнут.

3. Пользователь слишком короткий (< 10 символов). Система отвергает отзыв.

Таблица 3.4. Основные отличия варианта использования от алгоритма

Вариант использования	Алгоритм
Вариант использования описывает диалог между пользователем и системой.	Описывает вычисления
Последовательность событие/реакция на событие	Последовательность шагов
Основной/альтернативные сценарии	Алгоритм соответствует одному шагу варианта использования
Множество участвующих понятий из модели предметной области	Действия производятся над классами объектов
Участники: пользователь и система	Участники: все элементы системы

Отношения между компонентами и пакетами

Между компонентами и пакетами существуют только *зависимости*. Если компонента зависит от второй, а вторая от первой, между ними будет существовать два отношения зависимости (рис. 3.24).

Рис. 3.24. Отношения между компонентами

В модели пакетов следует избегать циклических зависимостей. Если один пакет зависит от другого и наоборот, то лучше перестроить отношения зависимости. Следует или объединить пакеты в один, или попытаться вынести общие элементы в отдельный пакет так, как показано на рис. 3.25.

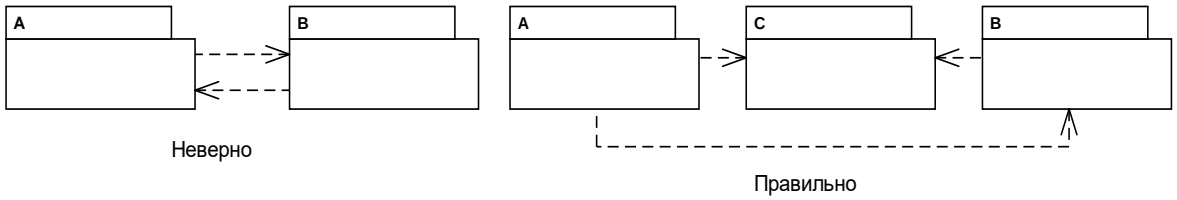


Рис. 3.25. Устранение циклических зависимостей

3.6. Диаграммы

Любая диаграмма состоит из набора элементов и является частью UML-модели. Все диаграммы делятся на две группы: диаграммы структуры, диаграммы поведения.

3.6.1. Диаграммы структуры

Диаграммы структуры представляют собой статическую модель системы, которая фиксирует сущности и структурные отношения между ними. Язык UML включает следующие базовые диаграммы структуры:

классов *(Class diagram);*

компонентов *(Component diagram);*

развертывания *(Deployment diagram);*

объектов *(Object diagram);*

пакетов *(Package diagram).*

Диаграмма классов

Представляет собой граф с вершинами-классами, между которыми установлены разные отношения. Как правило, для каждого прецедента создается своя диаграмма классов, которая уточняется по мере работы над проектом. Существует три различные точки зрения на диаграмму классов:

1. *Концептуальная точка зрения*. Диаграмма классов отображает понятия предметной области и не имеет никакого отношения к реализующему ее программному обеспечению. Классы показывают без атрибутов и операций. Эти диаграммы полезны на стадии анализа. Пример концептуальной диаграммы классов показан на рис. 3.26.

2. *Точка зрения спецификации*. Моделирование спускается на уровень ПО; рассматриваются только интерфейсы классов, но безотносительно ПО. Логические абстракции (классы пользовательского интерфейса, управления и классы сущностей) показывают с атрибутами и операциями, а также уточняют отношения. Эти диаграммы полезны на стадии проектирования.

3. *Точка зрения реализации*. Моделирование спускается на уровень реализации, и в этом случае логические абстракции проектирования будут представлены на диаграмме классами выбранного объектно-ориентированного языка программирования.

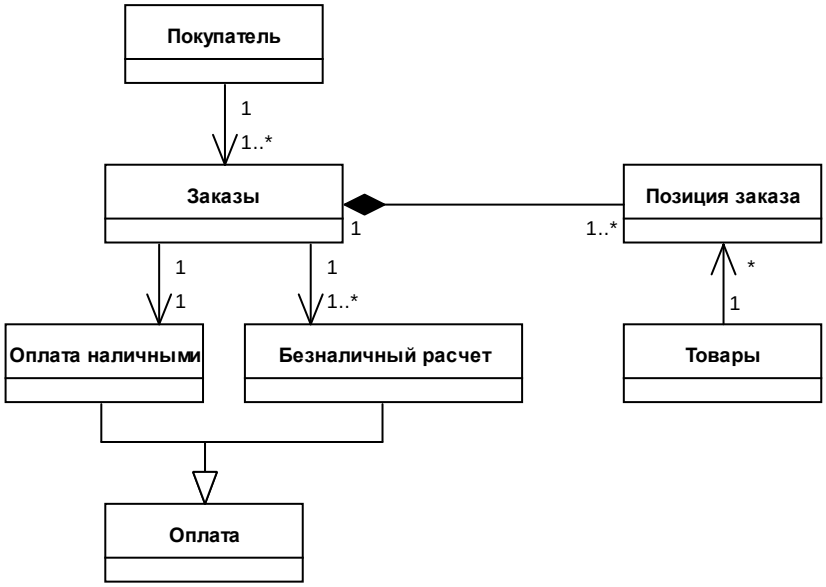


Рис. 3.26. Пример концептуальной диаграммы классов

Для представления архитектуры системы разрабатывается столько диаграмм классов, сколько требуется. С их помощью разработчики могут видеть и планировать архитектуру еще до фактического написания кода.

Для расширения семантики классов могут использоваться стандартные стереотипы и ограничения (табл. 3.5).

Таблица 3.5. Стандартные стереотипы и ограничения

Стереотипы и ограничения	Комментарий	Пример
{abstract}	Абстрактный класс, для которого не предусмотрено создание объектов-потомков. Имя абстрактного класса отображается курсивом	«abstract» ***Платформа1С8_2*** {root}
{root}	Ограничение, обозначающее корневой класс, находящийся в самом верхнем уровне иерархии	**Прикладной класс** {root} + Имя :int + Синоним :int + Комментарий :int
{leaf}	Ограничение, обозначающее конечный класс, находящийся в конце иерархии. У данного класса не может быть классов-потомков	**СправочникВида1** {leaf}

В UML имеется несколько стандартных видов отношений между классами (табл. 3.6):

Таблица 3.6. Стандартные виды отношений между классами:

Отношение	Пример
Ассоциации (association)	**ДокументВида1** {leaf} - Справочник1 :int — 1 «link» 1 — **СправочникВида1** {leaf}
Обобщения (generalization)	**Прикладной класс** {root} + Имя :int + Синоним :int + Комментарий :int; **Справочник**, **Документ**
Отношение агрегации (aggregation)	**Документ**: 1 — 0..* **Форма**; 1 — 0..* **ТабличнаяЧасть**
Отношение композиции (composition)	**ДокументВида1** {leaf} — **ФормаВида3** {leaf}, **ТабличнаяЧасть2** {leaf}

Ассоциация (association) – семантическое отношение между двумя и более классами, которое специфицирует характер связи между соответствующими экземплярами этих классов.

Отношение ассоциации соответствует наличию произвольного отношения или взаимосвязи между классами, обозначается сплошной линией со стрелкой или без нее и с дополнительными символами, которые характеризуют специальные свойства ассоциации. В качестве дополнительных специальных символов могут использоваться стереотип, имя ассоциации, символ навигации, а также имена и кратность классов-ролей ассоциации.

Наиболее простой случай данного отношения – бинарная ассоциация (binary association), которая служит для представления произвольного отношения между двумя классами. Частный случай бинарной ассоциации – рефлексивная ассоциация, которая связывает класс с самим собой.

Ненаправленная бинарная ассоциация изображается линией без стрелки, направленная – сплошной линией (рис. 3.27) с простой стрелкой; направление стрелки указывает на то, какой класс ассоциации классов является первым, а какой – вторым.

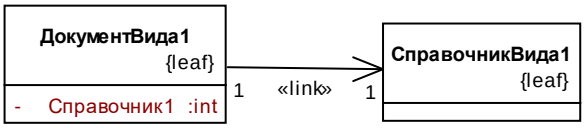


Рис. 3.27. Пример направленной ассоциации

Обобщение (generalization) – таксономическое отношение между более общим и менее общим понятием. Менее общий элемент модели должен быть согласован с более общим элементом и может содержать дополнительную информацию. Обобщение используется для представления иерархических взаимосвязей между различными элементами UML, такими как пакеты, классы, варианты использования. На диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения.

С этим видом отношения связаны такие понятия ООП, как наследование, родитель, потомок.

Наследование (inheritance) – специальный концептуальный механизм, посредством которого более специальные элементы включают в себя структуру и поведение более общих элементов.

Родитель, предок (parent) – в отношении обобщения более общий элемент.

Потомок (child) – специализация одного из элементов отношения обобщения, называемого в этом случае родителем. Согласно методологии ООАП класс-потомок обладает всеми свойствами и поведением класса-

предка, а также имеет собственные свойства и поведение, которые могут отсутствовать у класса-предка.

На диаграммах классов отношение обобщения обозначается сплошной линией с треугольной стрелкой на одном из концов. Стрелка указывает на более общий класс, а ее начало – на класс-потомок (рис. 3.28).

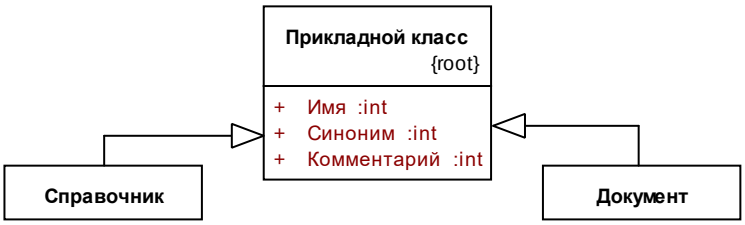


Рис. 3.28. Пример отображения отношения обобщения на диаграмме классов

Агрегация (aggregation) – специальная форма ассоциации, которая служит для представления отношения типа «часть-целое» между агрегатом (целое) и его составной частью. Данное отношение применяется для представления системных взаимосвязей типа «часть-целое» и по своей сути описывает декомпозицию или разбиение сложной системы на более простые составные части, которые также могут быть подвергнуты декомпозиции, если в этом возникнет необходимость.

Деление системы на составные части представляет собой иерархию, принципиально отличную от иерархии, формируемой отношением обобщения. Отличие заключается в том, что части системы никак не обязаны наследовать ее свойства и поведение, поскольку являются самостоятельными сущностями. Более того, части целого обладают собственными атрибутами и операциями, которые, как правило, существенно отличаются от атрибутов и операций целого.

Графически отношение агрегации изображается сплошной линией, один из концов которой представляет собой не закрашенный внутри ромб.

Ромб указывает на класс, который представляет собой «целое» или класс-контейнер. Остальные классы агрегации являются ее «частями» (рис. 3.29).

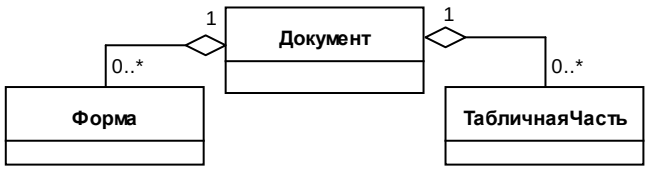


Рис. 3.29. Пример отображения отношения агрегации на диаграмме классов

Композиция (composition) – разновидность отношения агрегации, при которой составные части целого имеют такое же время жизни, что и само целое. Эти части уничтожаются вместе с уничтожением целого. Композиция – частный случай агрегации. Это отношение служит для спецификации более сильной формы отношения «часть – целое», при которой составляющие части тесно взаимосвязаны с целым. Особенность этой взаимосвязи заключается в том, что части не могут выступать в отрыве от целого, т. е. с уничтожением целого уничтожаются и все его составные части.

Графически отношение композиции изображается сплошной линией, один из концов которой представляет собой закрашенный внутри ромб. Ромб указывает на класс, который представляет собой класс-композит. Остальные классы являютсяего «частями» (рис. 3.30).

Рис. 3.30. Пример отображения отношения композиции на диаграмме классов

На концах стрелок, отображающих отношение, может отображаться кратность. Кратность (multiplicity) – спецификация области значений допустимой мощности, которой могут обладать соответствующие множества.

Кратность атрибута характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса. В общем случае кратность записывается в форме строки текста из цифр в квадратных скобках после имени соответствующего атрибута, при этом цифры разделяются двумя точками: [нижняя граница .. верхняя граница], где нижняя и верхняя границы – положительные целые числа. Каждая пара обозначает отдельный замкнутый интервал целых чисел, у которого нижняя (верхняя) граница равна значению нижней границы (верхняя). В качестве верхней границы может использоваться специальный символ «*» (звездочка), который означает произвольное положительное целое число, т. е. не ограниченное сверху значение кратности соответствующего атрибута. В рамках примера для отношения агрегации между классами *Форма* и *Документ* установлены значения кратности. Интерпретация данной кратности следующая: 1 экземпляр класса Документ может содержать от 0 до неограниченного количества экземпляров класса *Форма* (рис. 3.31).

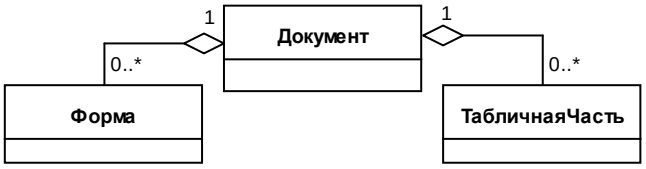


Рис. 3.31. Пример отображения кратности
для отношения агрегации

В специальной методологии проектирования систем Rational Unified Process (RUP – рациональный унифицированный процесс), основанной на UML, диаграмма классов может использоваться не только для проектирования логической модели системы, но и для представления результатов анализа бизнес-процессов и предметной области. В данной методологии для классов анализа введены дополнительные стереотипы (табл. 3.7):

Таблица 3.7. Дополнительные стереотипы классов

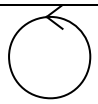
Стереотип	Обозначение	Определение
Boundary	 Boundary	Граничный класс. Класс, который располагается на границе системы с внешней средой и непосредственно взаимодействует с актерами, но является составной частью системы
Control	 Control	Управляющий класс. Класс, отвечающий за координацию действий других классов. Управляющий класс – это, как правило, варианты использования
Entity	 Entity	Класс-сущность. Класс-сущность (entity class) – пассивный класс, информация которого должна храниться постоянно

Диаграмма компонентов

Эта диаграмма изображает состав типов системных компонентов (исходного кода, исполняемых компонентов) и зависимости между ними. Пример диаграммы компонентов показан на рис. 3.32.

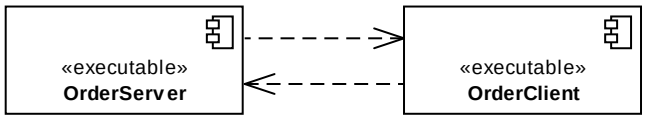


Рис. 3.32. Пример диаграммы компонентов

Диаграмма развертывания

Эта диаграмма имеет дескрипторную и экземплярную форму. Первая форма изображает конфигурацию узлов, где производится обработка данных, и развернутые на узлах исполняемые компоненты (рис. 3.33).

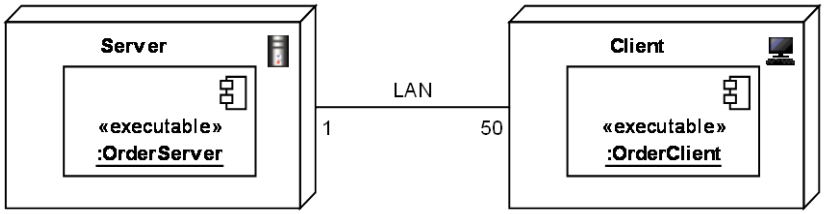


Рис. 3.33. Дескрипторная форма диаграммы развертывания

Вторая форма показывает конфигурацию для работающих узлов и экземпляры компонентов, а также существующие на узлах объекты (рис. 3.34).

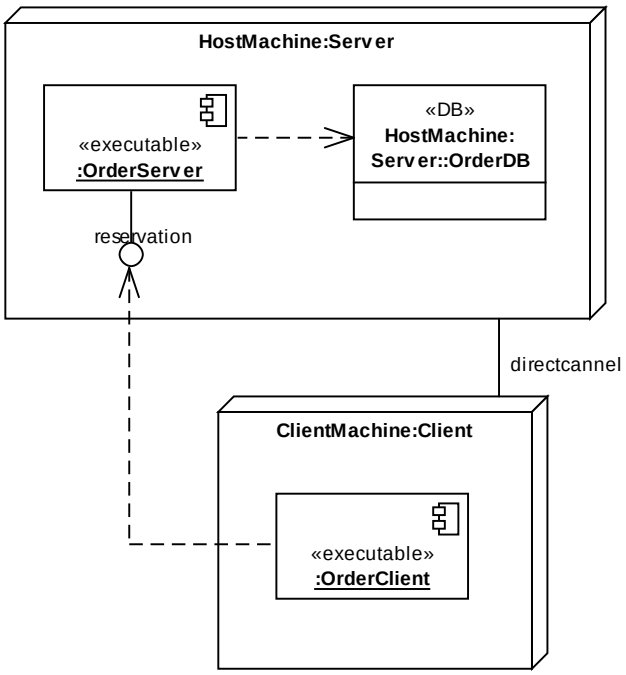


Рис. 3.34. Экземплярная форма диаграммы развертывания

Диаграмма объектов

Эта диаграмма изображает объекты и их связи в некоторый момент времени. Ее используют для иллюстрации сложности структуры данных или чтобы показать поведение в виде последовательности примеров. Диаграмму объектов можно считать особым случаем диаграммы классов. Пример диаграммы приведен на рис. 3.35.

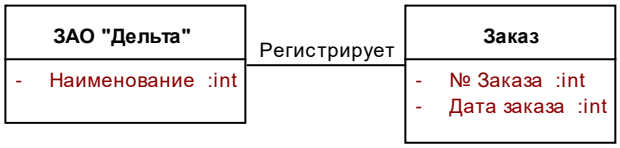


Рис. 3.35. Пример диаграммы объектов

Диаграмма пакетов

Диаграмма изображает пакеты и зависимости между ними. Пакеты позволяют разбить UML-модель на части, что упрощает ее понимание и поддержку. Пакеты образуют дерево, уровень абстракции которого увеличивается в направлении корня – системы (пакет верхнего уровня). Пример диаграммы пакетов показан на рис. 3.36.

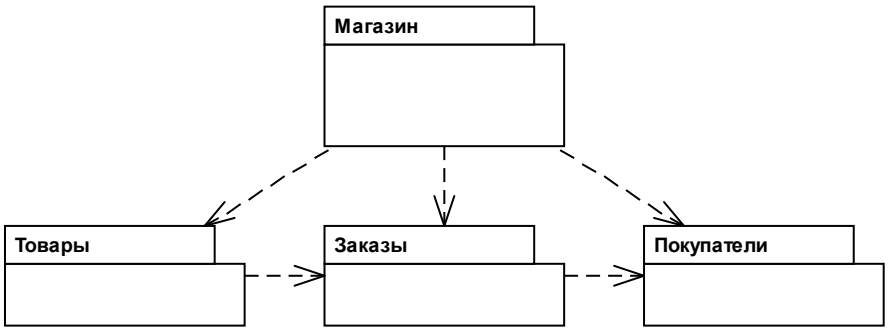


Рис. 3.36. Пример диаграммы пакетов

3.6.2. Диаграммы поведения

Диаграммы поведения представляют динамическую модель системы, которая отображает взаимодействие сущностей для реализации требуемого поведения. UML включает следующие базовые диаграммы поведения:

прецедентов (Use Case diagram);

деятельности (Activity diagram);

последовательности (Sequence diagram);

состояний (Statechart diagram).

Диаграмма прецедентов

Диаграмма является графической нотацией для отображения вариантов использования, актеров и отношений между ними. Между актером и прецедентом существует отношение коммуникативной ассоциации, которое показывает, что между ними существует какое-то взаимодействие. Между двумя прецедентами устанавливается отношение зависимости, чтобы показать использование <<*include*>> включаемого прецедента базовым или расширение <<*extend*>> базового прецедента. Между актерами отношений не существует, поскольку они находятся вне границ системы. Пример диаграммы прецедентов показан на рис. 3.37.

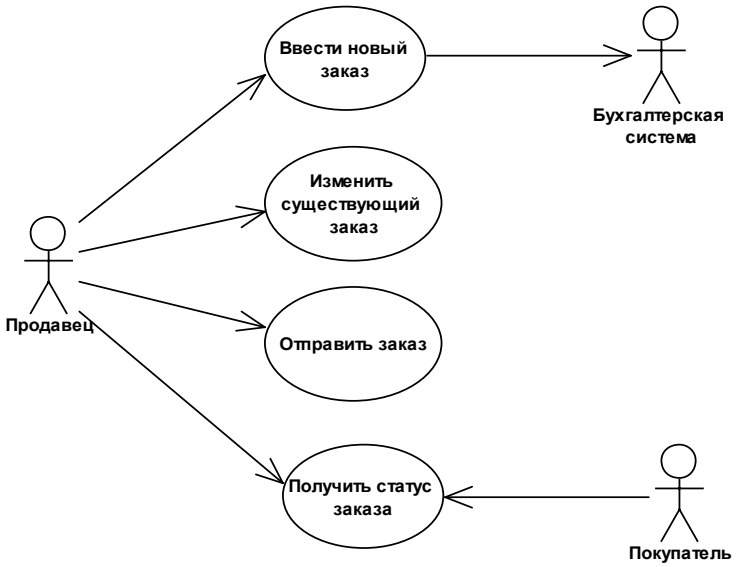


Рис. 3.37. Пример диаграммы прецедентов

В представлении *Use Case View* диаграмма отражает требования к системе с точки зрения пользователя.

В представлении *Logical View* – отражает требования к системе с точки зрения разразработчика и детализирует диаграмму из представления *Use Case View*. Диаграмма прецедентов используется:

для управления процессом разработки ПС;

определения границ ПС;

определения требований к ПС:

выявления основных функций *(Use Case View);*

спецификации функций *(Logical View);*

спецификации дополнительных функций *(Logical View);*

составления тестов.

Следует помнить, что *use case (прецедент)* – это документ, который описывает последовательность связанных с актером событий (а не отдельное событие). Эта последовательность используется системой для выполнения требуемого сервиса. Каждый прецедент определяет относительно большой завершенный процесс, состоящий из многих шагов, и реализуется как неделимая транзакция, которая не может быть прервана никаким другим прецедентом. После того как система выполнит функцию (сервис), она должна вернуться в исходное состояние, в котором готова к выполнению новых запросов. В представлении *Logical View* для каждой диаграммы прецедентов должна быть создана своя диаграмма классов.

Диаграмма видов деятельности

Это диаграмма, на которой изображается граф деятельности с потоком работ или работа отдельной процедуры. Диаграмму деятельности можно дополнять потоками объектов (рис. 1–32). Диаграмма деятельности используется везде, где возникает необходимость моделирования

поведения. На ней обязательно указывается одно начальное состояние *(закрашенный круг)* и одно или несколько конечных состояний. Действия *(прямоугольники с закругленными углами)* моделируют поведение участников. Для участников на диаграмме создаются отдельные дорожки *(swimlane)*, или участки *(partition)*. Диаграмма, которая моделирует обслуживание клиентов для получения ими заказа, показана на рис. 3.38.

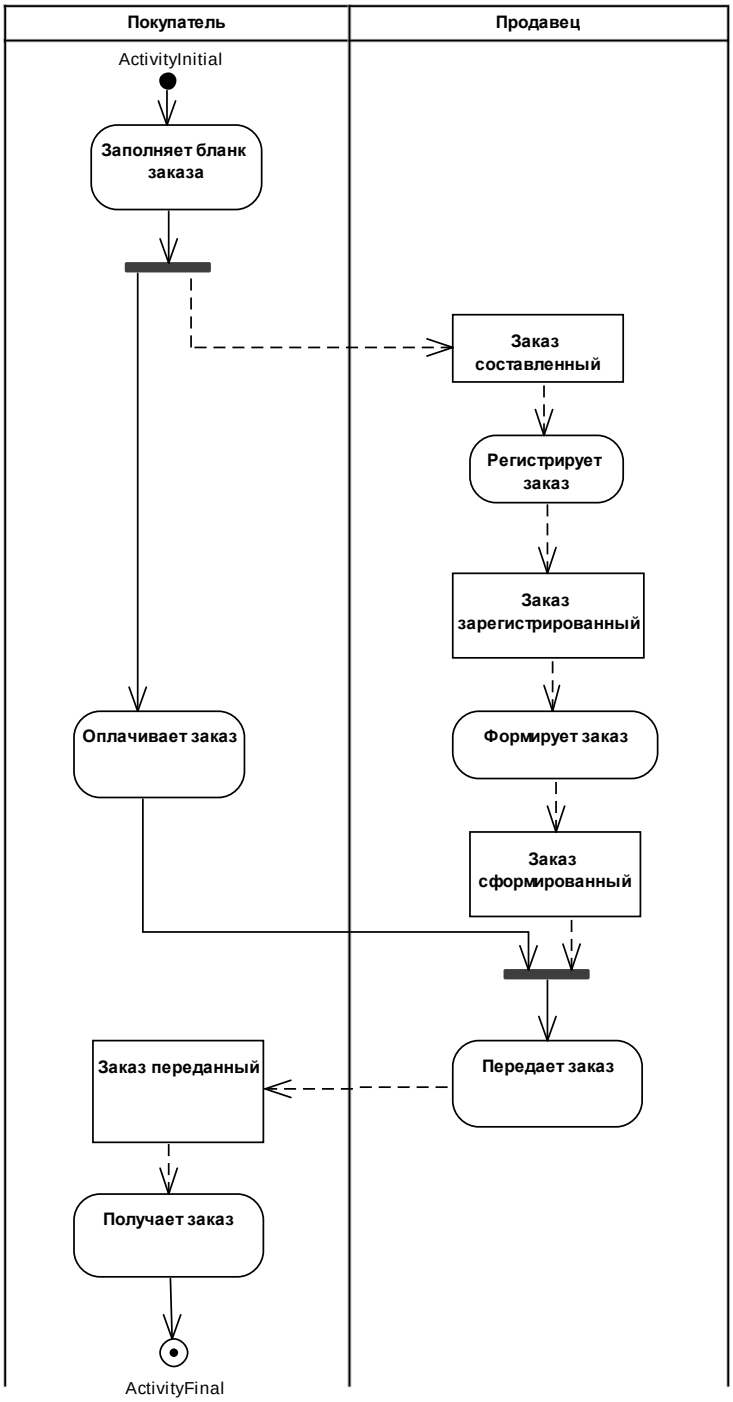


Рис. 3.38. Диаграмма видов деятельности с потоком объектов

Диаграмма последовательности

Эта диаграмма изображает упорядоченное во времени взаимодействие объектов. На этой диаграмме могут быть показаны либо объекты и сообщения, которые они посылают друг другу, либо соотнесенные с объектами классы и соотнесенные с сообщениями операции классов. Диаграммы последовательности особенно полезны для моделирования операций классов. Пример диаграммы приведен на рис. 3.39.

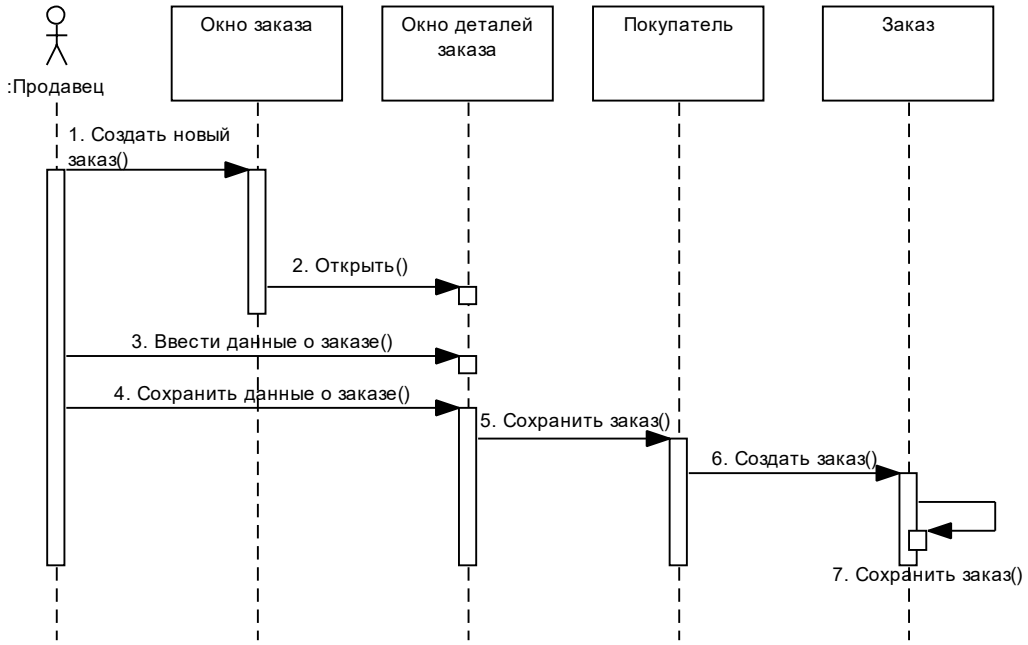


Рис. 3.39. Пример диаграммы последовательности

В верхней части диаграммы изображаются актер и объекты (классы). Стрелки соответствуют сообщениям (операциям). Как правило, диаграмма иллюстрирует одно единственное действие, происходящее в рамках прецедента. Если *use case* достаточно прост, то может быть показан весь поток событий. Чаще всего, для основного потока событий каждого прецедента создается набор диаграмм последовательности, которые отображают во времени последовательность действий, реализующих системный сервис. Создание диаграммы последовательности проходит в два этапа:

1. Сначала на диаграмме отображаются:
   объекты, которые участвуют в выполнении действия;
   сообщения, которые посылают объекты друг другу, для выполнения действия.
2. Затем на полученной диаграмме:
   объекты соотносятся с классами;
   сообщения соотносятся с операциями.

Глядя на эту диаграмму, пользователи знакомятся со спецификой своей работы. Аналитики видят поток действий, разработчики – классы, которые надо создать, и их операции. Специалисты по контролю качества поймут детали процесса и смогут разработать тесты для их проверки.

Диаграмма состояний

Это граф специального вида, который изображает некоторый конечный автомат *(state machine)*. Вершинами графа являются состояния автомата, а дуги обозначают переходы из одного состояния в другое.

С помощью диаграммы состояний моделируют поведение для классов, прецедентов, подсистем, целевой системы. Пример такой диаграммы показан на рис. 3.40.

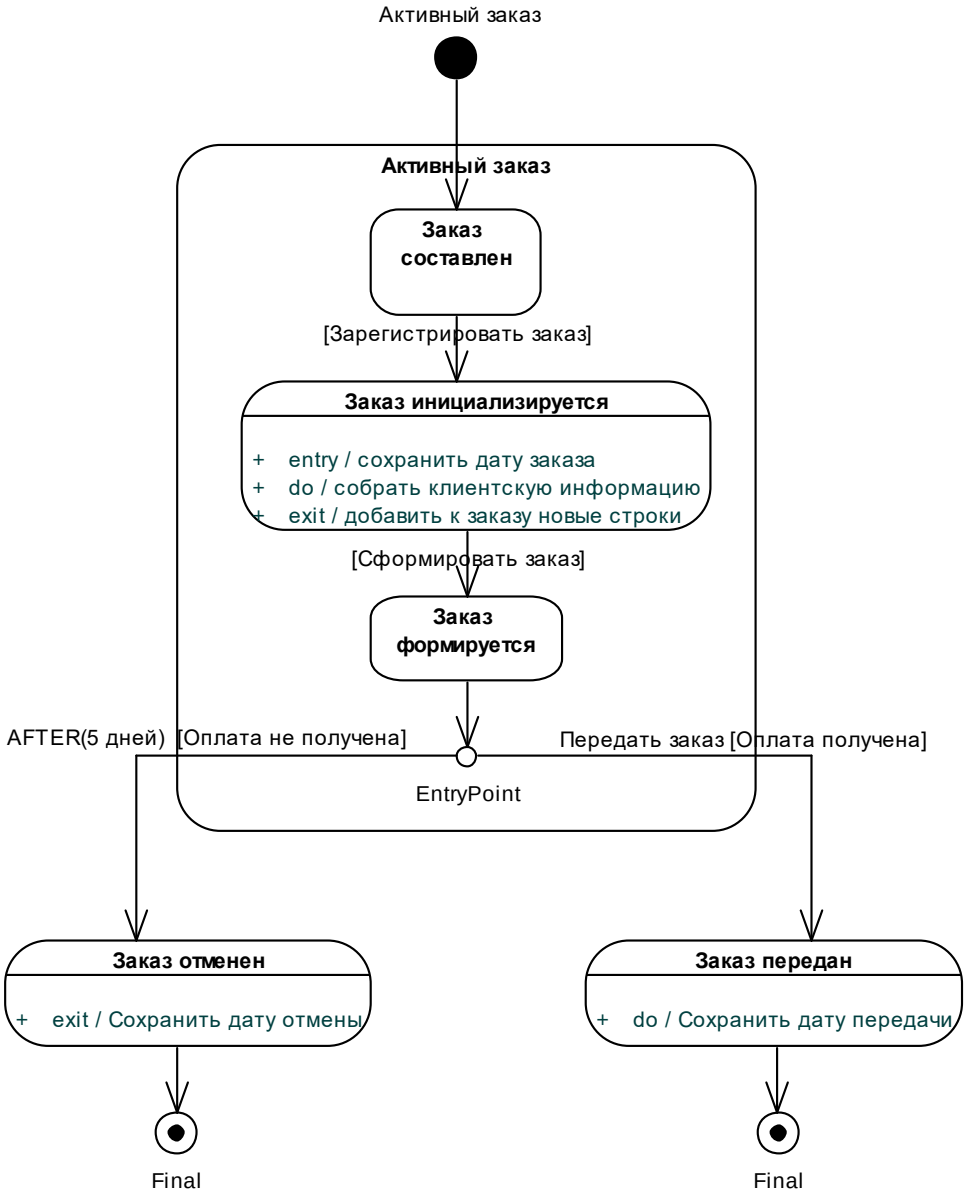


Рис. 3.40. Пример диаграммы состояний

На диаграмме состояний для моделирования поведения класса отображают жизненный цикл одного объекта, начиная с момента его создания и заканчивая разрушением. Диаграмма определяет все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате влияния некоторых событий. *Состояние (State)* – это ситуация в жизни объекта, на протяжении которой он удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает какого-то события. Состояние изображается в виде прямоугольника с закругленными углами. Существует два специальных состояния объекта: начальное и конечное. *Начальным (Start)* называется состояние, в котором объект находится сразу же после своего создания, *конечным (Stop)* – состояние, в котором объект находится непосредственно перед уничтожением.

Начальное состояние обязательно. На диаграмме состояний может быть только одно начальное состояние, конечных состояний может быть сколько угодно.

Находясь в конкретном состоянии, объект может выполнять определенные действия. Например, он может осуществлять некоторые вычисления или посылать сообщение другому объекту. В состоянии объекта различают входное действие, деятельность и выходное действие. *Входное действие (Entry action)* – непрерываемое действие, которое выполняется при переходе объекта в данное состояние. Например, при переходе в состояние *Заказ инициализируется* выполняется действие *Сохранить дату заказа*. *Деятельность (Activity)* – реализуемое объектом действие, когда он находится в данном состоянии. *Выходное действие (Exit action)* – осуществляется как составная часть процесса выхода объекта из данного состояния. От одного состояния объекта к последующему состоянию проводится переход *(Transition)*. На диаграмме все переходы изображают в виде стрелки, начинающейся в первоначальном состоянии и заканчивающейся в последующем состоянии. При переходе из первого состояния во второе объект выполняет некоторое *действие (Сохранить дату оплаты)* под воздействием определенного *события (Передать заказ)* и при выполнении заданного *условия (Оплата получена)*. *Событие (Event)* – это то, что вызывает переход из одного состояния в другое. Действие *(Action)* – это атомарное вычисление, которое приводит к смене состояния или возврату значения. *Ограждающее условие (Guard conditions)* определяет, когда переход может быть выполнен, а когда нет. Диаграмма состояний служит для спецификации состояний объекта, меняющихся при различном обороте событий. Эта диаграмма является динамической и особенно важна для моделирования систем реального времени, поскольку показывает упорядоченное по событиям поведение объекта. При создании диаграммы нужно учитывать следующие обязательные условия:

1. *Отсутствие конфликтующих переходов*. Элемент не может одновременно переходить в более чем два последующих состояния. Для разрешения конфликта следует пользоваться ограждающими условиями.

2. *Отсутствие изолированных состояний и переходов*. Каждый переход должен соединять два состояния (исключением являются начальное и конечное). Разрешаются рефлексивные переходы.

3. *Конечное количество состояний*. По определению диаграмма состояний представляет конечный автомат. Каждое состояние на диаграмме должно быть специфицировано явным образом (исключением являются начальное и конечное состояния).

4. *Индивидуальность состояния*. В каждом состоянии объект должен вести себя по-разному.

5. *Определенность состояния*. В каждый момент времени элемент может находиться только в единственном состоянии. Если это не так, то это или ошибка, или признак наличия параллельности поведения.

3.7. Общие механизмы UML

В UML существуют следующие общие механизмы, которые описывают стратегии подхода к моделированию и многократно и последовательно применяются в языке: спецификации; способы представления; дополнения; расширения.

Спецификации

Спецификация *(Specification, Definition)* – декларативное описание того, чем является или для чего служит некоторая сущность. Составление спецификации обязательно для каждого элемента (сущностей, отношений, диаграмм) модели. Набор спецификаций формирует семантический задний план *(semantic backplane)* модели, который наполняет ее смыслом. Модель считается полной и полезной только в том случае, когда в спецификациях присутствует семантика. Спецификации используются для составления документации к проекту и программной системе, потому что содержат для этого все необходимые сведения. Инструментальные средства UML-моделирования поддерживают доступ, просмотр и изменение спецификации любого элемента модели.

Способы представления

В UML существует два способа представления: классификатор – экземпляр и интерфейс – реализация. *Классификатор (classifier)* – дискретная концепция UML-модели, которая представляет собой абстракцию и находится в определенных отношениях с другими элементами модели. Основными видами классификаторов являются: классы, интерфейсы и типы данных. *Экземпляр (instance)* – конкретная сущность времени выполнения, обладающая индивидуальностью, которая отличает ее от других таких сущностей. В каждый момент времени

экземпляр имеет некоторое значение. Типичным примером экземпляра служит объект класса. Основная идея понятия *интерфейс – реализация* заключается в отделении действий (интерфейс) от того, как они выполняются (реализация).

Интерфейс (interface) – именованное множество открытых операций, которые описывают поведение класса или компонента и определяют контракт для конкретных реализаций. В целом интерфейс соответствует абстрактному классу, где определены только абстрактные операции. *Реализация (realization)* – это указание на то, что программная реализация элемента модели наследует определенный интерфейс. Реализующий элемент должен поддерживать все имеющиеся в интерфейсе операции. На рис. 3.41 показан пример графического представления понятия *интерфейс – реализация*.

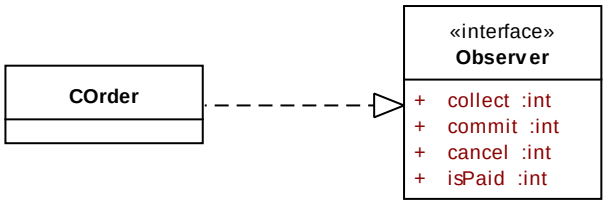


Рис. 3.41. Пример представления понятия интерфейс – реализация

Дополнения

Дополнения используются для того, чтобы подчеркнуть важные детали на UML-диаграммах. Элемент диаграммы может быть показан как в виде простого графического символа, так и в развернутом состоянии. Один и тот же класс *COrder* без дополнений и с дополнениями показан на рис. 3.42.

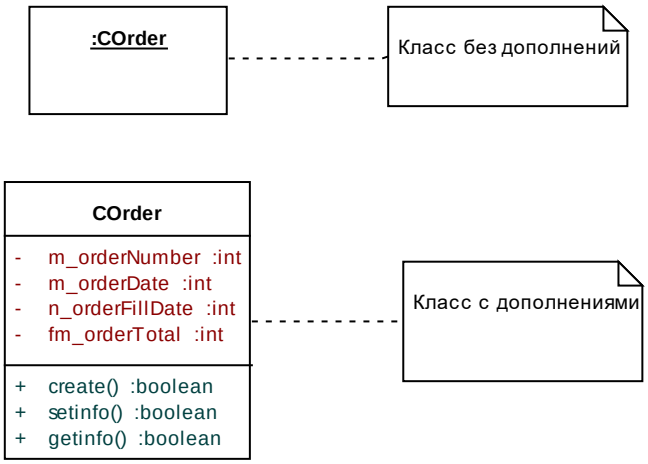


Рис. 3.42. Класс без дополнений и с дополнениями

Расширения

Расширения предназначены для использования в процессе создания программного обеспечения, чтобы удовлетворить новым требованиям разработчиков к представлению моделей. К расширениям относятся: ограничения, стереотипы и именованные значения. *Ограничение (constraint)* – это текстовое указание на семантическое ограничение для элемента модели, которое должно оставаться истинным. Пример графического изображения ограничения показан на рис. 3.43.

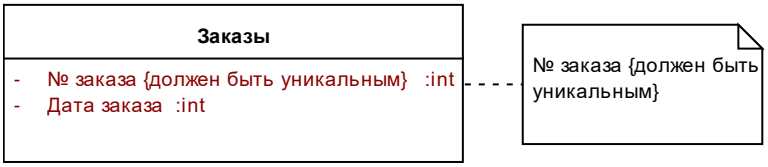


Рис. 3.43. Пример ограничения

Стереотип (stereotype) представляет категорию элемента модели. В UML определены некоторые стандартные стереотипы. Например, стандартными стереотипами для классов являются: <<*boundary*>> (пограничный класс), <<*control*>> (класс управления), <<*entity*>> (класс сущностей). Стандартные стереотипы поддерживаются специальными графическими обозначениями (рис. 3.44). UML позволяет создавать собственные стереотипы для проектируемой системы. В таком случае следует определить семантику новых стереотипов, используя примечания *(notes)*.

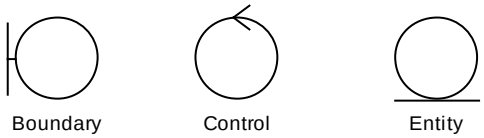


Рис. 3.44. Графическое изображение стандартных стереотипов классов

Пограничные классы расположены на границе системы. К ним относятся экранные формы, отчеты, интерфейсы с аппаратурой (с принтерами, сканерами, датчиками) и другими системами. Чтобы выявить пограничные классы, необходимо исследовать диаграмму прецедентов. Для каждого взаимодействия между актером и прецедентом должен существовать хотя бы один пограничный класс. Необязательно создавать уникальные пограничные классы для каждой пары *актер – прецедент*. Если актеры инициируют один и тот же *use case*, то они могут применять общий класс. *Классы управления* отвечают за координацию действий других классов. Этот класс не несет в себе никакой функциональности, он делегирует ответственность другим классам. Остальные классы посылают ему мало сообщений. Но он сам посылает множество сообщений. Выделение классов управления – одна из главных

задач логического проектирования системы. С помощью управляющих классов можно изолировать функциональность системы. Если верно выбраны классы управления, любые изменения логики ПС затронут только эти классы, но не классы сущностей. *Классы сущностей* содержат атрибуты, которые хранятся постоянно. Классы сущностей можно обнаружить в потоке событий и в диаграммах последовательностей. Эти классы имеют наибольшее значение для пользователя, и потому их названия часто являются терминами из предметной области. Как правило, для каждого класса сущностей создают таблицу в базе данных. Многие инструментальные средства проектирования позволяют создавать логическую схему базы данных для ПС на основе объектной модели. *Именованное значение (tagged values)* представляет собой ключевое слово, к которому прикреплено значение. Именованные значения позволяют добавлять собственные свойства к элементу модели.

3.8. Архитектура системы в RUP *(Rational Unified Process)*

Архитектура описывает высокоуровневую структуру системы. Унифицированный процесс RUP (Rational Unified Process) использует модель представления архитектуры «4 + 1» (рис. 3.45).

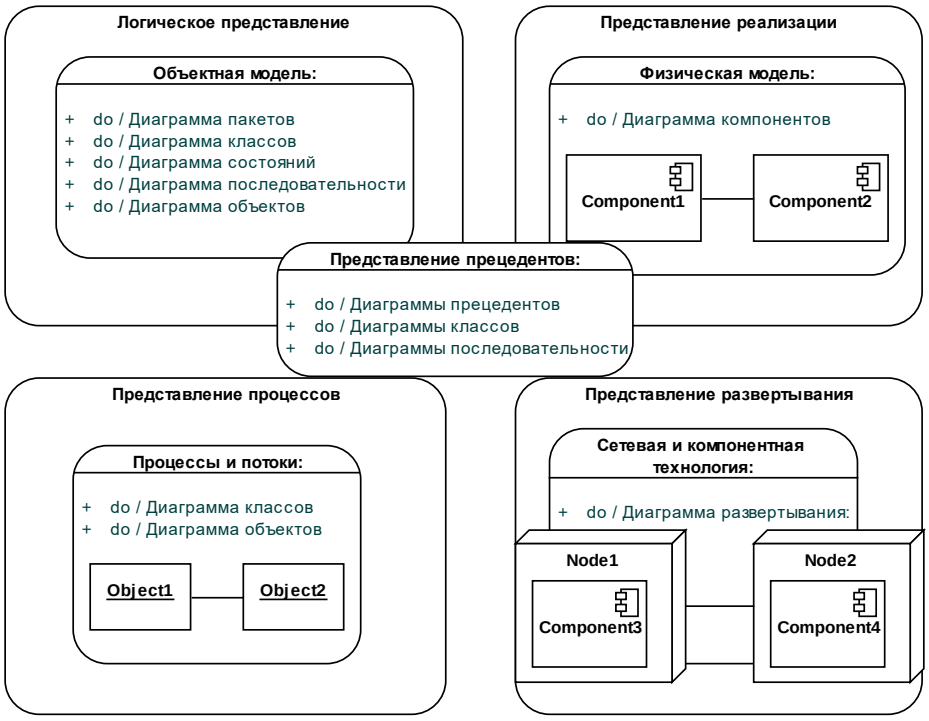


Рис. 3.45. Архитектура системы в RUP

Логическое представление описывает словарь предметной области как набор классов и объектов. Основное внимание уделяется отображению

того, как образующие систему объекты и классы реализуют требуемое поведение. *Представление процессов* моделирует исполняемые процессы и потоки системы как классы, имеющие собственный поток управления. Внимание акцентируется на производительности, масштабируемости и пропускной способности системы. *Представление реализации* моделирует файлы и компоненты, из которых интегрируется система, отображает зависимости между компонентами и управляет конфигурированием компонентов для определения версии системы. *Представление развертывания* моделирует размещение исполняемых компонентов и данных на физические вычислительные узлы. В этом представлении отображается топология, распространение, поставка и установка системы. *Представление прецедентов* описывает основные требования, предъявляемые к системе, как набор прецедентов. Это представление объединяет все остальные представления и является для них базой. Архитектура – это описание структуры программного средства. При проектировании архитектуры определяют подсистемы, а также управление подсистемами и их взаимодействие. Для создания архитектуры выполняются следующие шаги:

1. Структурирование системы. ПС структурируется в виде совокупности относительно независимых подсистем и определяет их взаимодействие. Обмен данными между подсистемами можно организовать двумя способами:

все совместно используемые данные хранятся в центральной базе данных, доступной всем подсистемам. Эту модель называют *моделью репозитория*;

каждая подсистема может иметь собственную базу данных. Взаимообмен данными происходит посредством передачи сообщений.

2. Моделирование управления. Разрабатывается базовая модель управления взаимоотношениями между частями системы. Можно выделить два основных типа управления программных систем:

а) централизованное управление. Одна из подсистем полностью отвечает за управление, запускает и завершает работу остальных подсистем. Управление от этой подсистемы может перейти к другой, но потом обязательно возвращается обратно. Примеры централизованного управления: *модель вызова-возврата* для последовательных систем, *модель диспетчера* для параллельных систем. В первой модели управление начинается на вершине иерархии и через вызовы передается на нижние уровни. Во второй – один системный компонент назначается диспетчером для координации процессов в системе, причем процессы могут быть параллельными;

б) управление, основанное на событиях. Здесь на внешние события может ответить любая из подсистем. События, на которые реагирует подсистема, могут происходить либо в других подсистемах, либо во

внешнем окружении. Примеры: *модели передачи сообщений* (подсистемы реагируют на определенные события и несут ответственность за их обработку); *модели, управляемые прерываниями* для систем реального времени (внешние прерывания регистрируются обработчиком прерываний, а обрабатываются другим системным компонентом).

3. Модульная декомпозиция. Каждая подсистема разбивается на отдельные модули. Определяются типы модулей и типы их взаимодействия. При модульной декомпозиции чаще всего используются объектно-ориентированная модель (ПС состоит из набора взаимодействующих объектов) и модель потоков данных (ПС состоит из функциональных модулей, которые на входе получают данные и преобразуют их некоторым образом в выходные данные).

Четкого различия между подсистемами и модулями нет. Как правило, модуль никогда не рассматривается как независимая компонента. Приняты такие определения:

подсистема – это часть системы, операции которой не зависят от сервисов, предоставляемых другими подсистемами. Подсистемы состоят из модулей и имеют определенные интерфейсы, с помощью которых и взаимодействуют;

модуль – это элемент системы, который предоставляет один или несколько сервисов для других модулей. Модуль может использовать сервисы, поддерживаемые другими модулями. Структурирование, моделирование управления и модульная декомпозиция перемежаются и накладываются друг на друга. Шаги повторяются до тех пор, пока архитектура не станет удовлетворять требованиям. При проектировании архитектуры разрабатывается четыре модели:

статическая структурная модель – представлены подсистемы или компоненты, разрабатываемые в дальнейшем независимо;

динамическая модель процессов – представлена организация процессов во время работы системы;

интерфейсная модель – определяет сервисы, предоставляемые каждой подсистемой через общий интерфейс;

модель отношений – показывает взаимоотношения между подсистемами.

ГЛАВА 4. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД К РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Объектно-ориентированный подход (object-oriented approach) подразумевает разделение системы на компоненты с разной степенью детализации. В основе этой декомпозиции лежат объекты и классы. Классы связаны разнообразными отношениями и обмениваются сообщениями, вызывающими операции над объектами. Объектный подход к разработке систем полностью соответствует требованиям итеративного и поступательного процесса разработки, в котором постепенно уточняется, изменяется и совершенствуется единая модель для реализации системы, удовлетворяющей пользователя. Преимущества объектно-ориентированного подхода:

1. Сложность принимает форму иерархии. Сложные ПС состоят из взаимозависимых подсистем (модулей), которые в свою очередь также могут быть разбиты на подсистемы (модули), вплоть до самого низкого уровня.
2. Иерархические системы состоят из немногих типов подсистем, которые комбинируются в различных сочетаниях.
3. Связи внутри компонентов обычно сильнее межкомпонентных, что позволяет относительно независимо разрабатывать каждый компонент.
4. Компоненты могут повторно использоваться в различных системах.
5. Любая работающая сложная система развивается из работающей простой системы.
6. Выбор примитивных элементов, из которых строится ПС, относительно произволен и оставляется на усмотрение разработчиков.
7. Гибкая архитектура объектно-ориентированных систем относительно легко поддается модификации.

Несмотря на перечисленные достоинства объектно-ориентированная парадигма (как способ создания высокоуровневых проектов) подвергается критике по следующим причинам, которые относят к ее недостаткам:

1. Усложнение методологии. Для успешного использования подхода требуется наличие определенного уровня квалификации у специалистов. Для небольших проектов более эффективным может оказаться применение структурного подхода, когда декомпозиция ПС осуществляется не по классам, а по функциям.
2. Сложность реализации. Реализация на объектно-ориентированном языке программирования, как правило, приводит к построению требовательного к ресурсам приложения.
3. Высокая стоимость инструментальных средств по автоматизации процесса разработки.

4.1. Трехуровневая модель приложения

Большинство ПС разрабатывается на основе трехзвенного архитектурного стиля (Three-tiered Architecture), состоящего из трех базовых уровней:

представления (User Service);

логики приложения или бизнес-правил (Business Service);

управления данными (Data Service).

Уровень представления

Классы этого уровня содержат логику приложения, которая получает входные данные от внешнего источника и предоставляет информацию этому источнику. В большинстве случаев в качестве внешнего источника выступает пользователь, хотя это может быть также некоторое устройство автоматизированного ввода-вывода. При помощи классов уровня представления осуществляются навигация пользователя через приложение и (как опция) контроль ограничений для вводимых данных.

Бизнес-правила

Классы уровня бизнес-правил содержат логику приложения, которая управляет функциями и процессами, выполняемыми ПС. Эти функции и процессы вызываются или объектами классов уровня представления (когда пользователь запрашивает сведения), или другими системными функциями. В общем, классы логики приложения выполняют манипулирование данными.

Уровень управления данными

Классы этого уровня содержат логику, которая по существу является интерфейсом с системой управления хранилищем данных, например, с системой управления базами данных (СУБД), иерархической файловой системой либо с другим источником данных типа внешней программной системы. Функции этого уровня вызываются объектами классов логики приложения, хотя в простых приложениях они могут вызываться непосредственно объектами классов уровня представления.

Распределенная вычислительная архитектура

Схема распределенной вычислительной архитектуры показана на рис. 4.1.

Рис. 4.1. Распределенная вычислительная архитектура

Свойства распределенной вычислительной архитектуры:

1. Может существовать любое количество компонентов каждого типа внутри одной программной системы.
2. Компоненты могут разделяться любым числом прикладных систем.
3. Каждый компонент разрабатывается с использованием оптимального для его структуры инструментального средства.
4. Компоненты взаимодействуют друг с другом на основе абстрактного фейса, который скрывает основную функцию, выполняемую компонентом.
5. Компоненты физически могут размещаться на одной или более аппаратных системах.
6. Распределенная вычислительная инфраструктура должна обеспечивать размещение, защиту и услуги связи для компонентов приложения.

Пакеты классической модели

Пакеты классической модели ПС показаны на рис. 4.2.

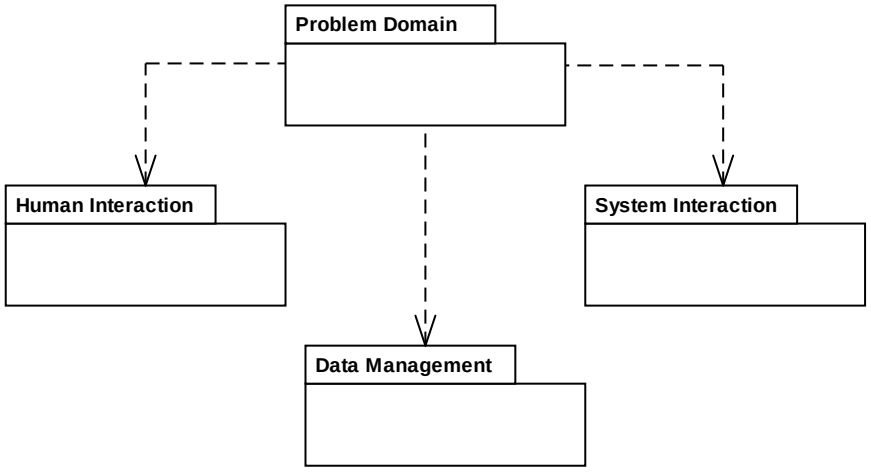


Рис. 4.2. Пакеты классической модели

Пакет *Human Interaction (HI) – Взаимодействие с человеком* содержит классы, обеспечивающие отображение, ввод и вывод данных. Пакет *Problem Domain (PD) – Проблемная область* содержит логические программные абстракции, точно соответствующие моделируемой предметной области и нейтральные по отношению к реализации (независимые от реализации). Они *знают* или *не знают совсем* о классах других пакетов. Пакет *Data Management (DM) – Управление данными* представляет классы, обеспечивающие интерфейс между классами предметной области и СУБД. Эти классы отвечают за доступ к хранимым данным и выполняют все необходимые операции над ними. Чаще всего они соответствуют классам проблемной области, которые нужно постоянно хранить во внешней памяти и искать. Пакет *System Interaction (SI) – Взаимодействие систем* содержит классы, которые поддерживают интерфейс между классами предметной области и другими системами (внешними по отношению к данной системе) или устройствами. Каждый класс ПС соответствует одному из пакетов *(HI, PD, DM, SI)*, причем разделение классов на пакеты не означает их обязательного физического разделения. Практическое применение этого шаблона для решения конкретных задач позволяет получить типовые решения типичных проблем в контексте решаемой задачи.

Пусть система реализуется на основе технологии клиент/сервер. Эта технология предполагает, что объекты клиента и объекты сервера будут разделены физически, т. е. будут находиться в разных исполняемых модулях. Объекты, реализующие интерфейс пользователя, выносятся в отдельный программный модуль *Client*. Объекты оставшихся трех пакетов *(PD, SI, DM)* помещаются в программный модуль *Server*, через который пользователи и получают доступ к данным. Шаблон организации архитектуры приобретает вид, который показан на рис. 4.3.

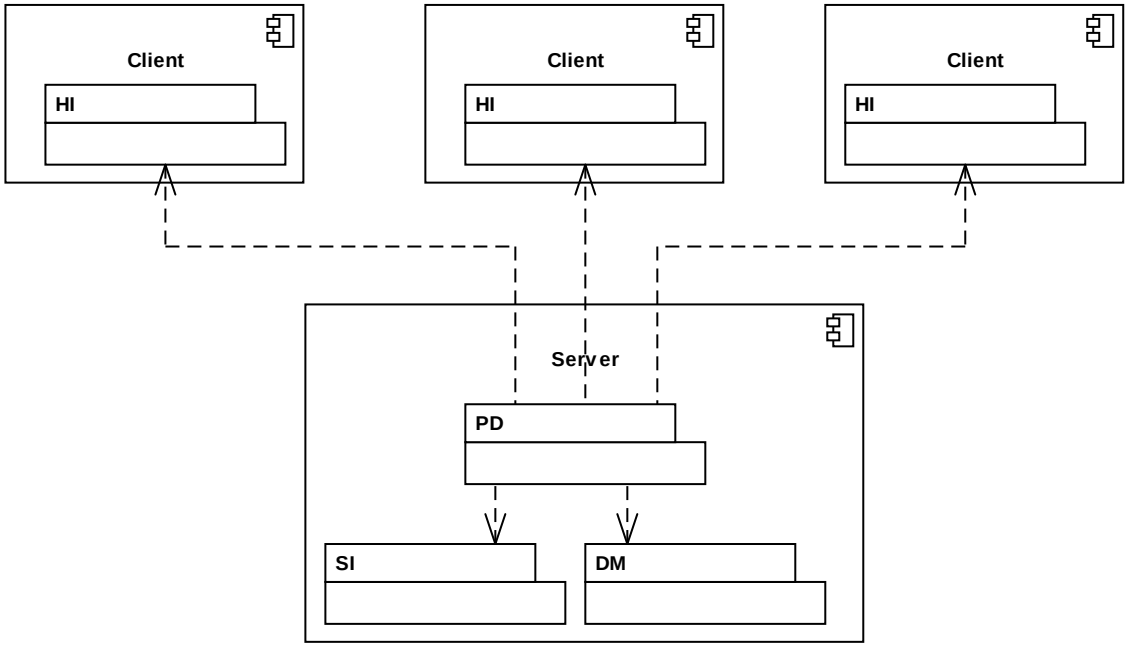


Рис. 4.3. Шаблон организации архитектуры

Такая организация общей архитектуры ПС упрощает моделирование (в рамках каждого пакета), а также повышает вероятность повторного использования и модулей, и отдельных классов. Для проектирования и реализации ПС в рамках одной предметной области достаточно воспользоваться упрощенным шаблоном (рис. 4.4).

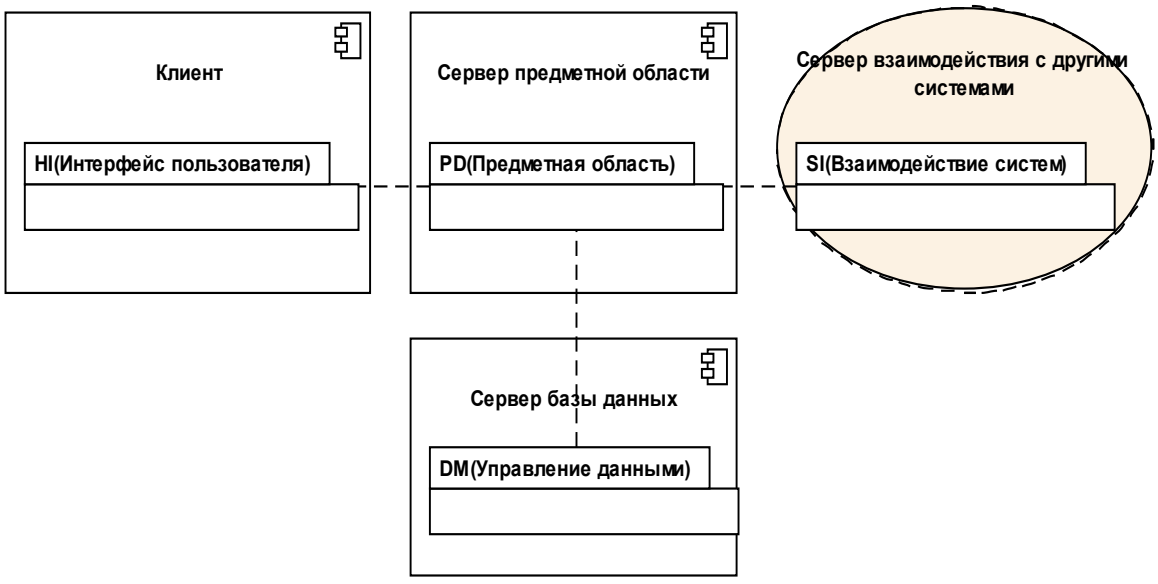


Рис. 4.4. Упрощенный шаблон

4.2. Методологии объектно-ориентированного подхода

Объектно-ориентированный подход к разработке ПС базируется на трех основополагающих методологиях:

1. Объектно-ориентированный анализ (Object Oriented Analysis). Предполагает исследование предметной области с точки зрения объектов реального мира и определяет границы ПС, а также требования к нему.

2. Объектно-ориентированное проектирование (Object Oriented Design). Акцентирует внимание на программных классах и ищет логические пути решения поставленных анализом задач.

3. Объектно-ориентированное программирование (Object Oriented Programming). Обеспечивает реализацию классов проектирования на выбранном языке программирования для получения конкретных результатов.

Объектно-ориентированный анализ и проектирование логически приводят к объектно-ориентированной декомпозиции на различных уровнях абстракции. При изменении уровня абстракции изменяется декомпозиция. Происходит итеративный процесс совершенствования модели ПС.

Объектно-ориентированный анализ

Объектно-ориентированный анализ (ООА) направлен на моделирование поведения системы. Модель анализа служит основой для последующего проектирования ПС. ООА – методология, при которой требования к системе воспринимаются с точки зрения пользователей и объектов/классов, выявленных в автоматизируемой предметной области. ООА решает три основные задачи:

1. Формализация требований, предъявляемых к ПС (определение назначения системы и создание модели функций системы). Требования обычно корректируются на протяжении всего времени работы над проектом.

2. Выявление объектов/классов, которые составляют словарь (глоссарий) предметной области.

3. Создание структур, которые обеспечивают взаимодействие объектов/классов для удовлетворения поставленных требований.

Результатом анализа являются определение назначения системы, ключевые абстракции и кооперации ключевых абстракций для реализации поведения ПС.

Определение назначения системы

Назначение системы должно состоять из одного предложения длиной не более 25 слов, содержащего подлежащее, сказуемое и прямое дополнение. Подлежащим всегда должна быть система. Например, «Эта система…», «Система АТМ…», «Система АРЕНДАТОР …», «Система РЕГИСТРАТОР …». Сказуемое описывает целевое предназначение

функций, которые будет выполнять система. Например, «Эта система помогает…», «Система АТМ обслуживает…», «Система РЕГИСТРАТОР поддерживает…», «Система АРЕНДАТОР облегчает…». Другие глаголы: управляет, повышает и т. п. *Дополнение* показывает объект, для которого работает система. Например, «Эта система помогает диспетчеру товарной конторы работать более эффективно при расчете с заказчиком», «Система АТМ обслуживает клиентов банка, пользующихся банковскими автоматами», «Система РЕГИСТРАТОР поддерживает ведение правильных записей о каждом факультативном курсе на факультете», «Система АРЕНДАТОР облегчает управление арендой недвижимости». Формулировка назначения должна быть настолько короткой и прозрачной, насколько можно. Максимальное упрощение поможет четко выдержать главное направление дальнейшей детализации.

Определение основных функций

Необходимо сначала ограничиться только самыми важными функциями. Следует помнить, что выявленные функции не должны перекрываться, т. е. две разные функции не должны решать одну и ту же задачу (например, регистрация документа и сохранение документа). При определении функций удобно придерживаться следующей классификации:

1. Регистрация важной информации. Например, регистрация документов, результатов бизнеса.
2. Ведение дела. Например, контроль и оценка состояния чего-либо на основе значений его атрибутов, различные вычисления, сортировка и поиск.
3. Анализ результатов бизнеса. Например, подсчет экономических показателей деятельности, оценка производительности труда.
4. Взаимодействие с другой системой. Например, прием сообщений от системы более низкого уровня и отправка сообщений системе более высокого уровня.

Например, для системы АРЕНДАТОР можно определить такие функции:

регистрировать договоры аренды;

регистрировать текущие эксплуатационные работы;

анализировать выполнение текущих эксплуатационных работ;

генерировать счета для арендаторов;

получать информацию об аренде каждого объекта недвижимости.

При использовании UML на уровне представления *(Use Case View)* полезно строить диаграммы прецедентов *(Use case diagram)* для идентификации акторов, системных сервисов и представления входных и выходных документов.

Пример. Вариант вузовской системы РЕГИСТРАТОР.

Назначение системы. Система РЕГИСТРАТОР поддерживает ведение правильных записей о каждом факультативном курсе на факультете. *Общие требования*. Система должна позволять секретарю факультета регистрировать студентов разных курсов факультета для прослушивания факультативных курсов, которые читают преподаватели разных факультетов. *Определение действующих лиц и функций системы*. *Актер* – секретарь факультета, который будет работать с системой. Варианты использования представлены на рис. 4.5.

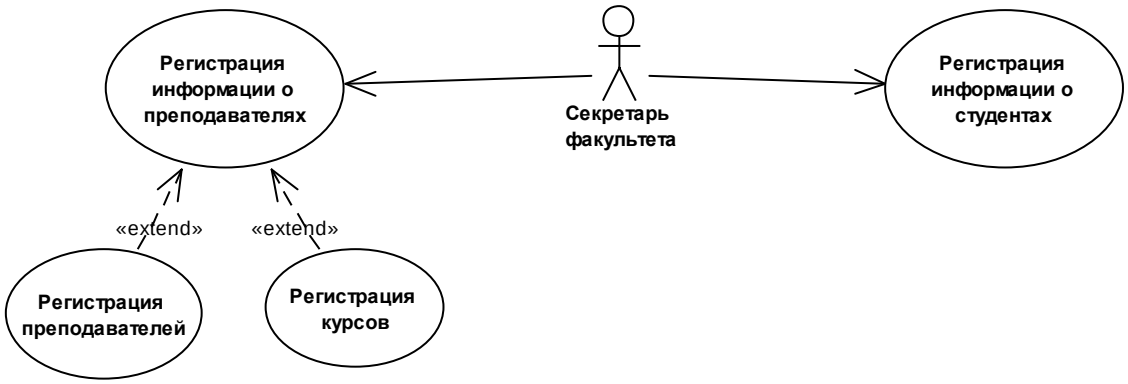


Рис. 4.5. Диаграмма прецедентов системы РЕГИСТРАТОР

Секретарь должен регистрировать читаемые преподавателями факультативы, студентов факультета, которые посещают факультативные курсы.

Выделение ключевых абстракций. Для определения классов можно воспользоваться следующими рекомендациями:

1. Перечислить всех кандидатов в классы, которые есть в описании задачи. Как правило, это имена существительные. Например, для системы РЕГИСТРАТОР к таким кандидатам можно отнести классы: *Факультет (Department), Студент (Student), Курс (Course), Факультативный курс (Elective Subject), Факультатив (Facultative), Преподаватель (Teacher), Регистрация (Registration).*

2. Добавить кандидатов в классы из анализа предметной области: *Университет (University), Декан (Decan).*

3. Отказаться от всех ненужных классов согласно критериям:

3.1. Избыточные классы: если два класса выражают одну и ту же информацию, сохранить класс с более подробным описанием (*Факультатив* – название читаемого предмета, *Факультативный курс* – название читаемого предмета, количество лекционных часов, часов лабораторных, практических занятий).

3.2. Несоответствующие объекты: если объект имеет немного или ничего, чтобы решить проблему, этот объект должен быть устранен *(Декан)*.

3.3. Неопределенные объекты: объекты с неточно определяемыми свойствами должны быть устранены.

3.4. Атрибуты: названия, которые описывают конкретные объекты с нечетко выраженным поведением, должны быть заявлены как атрибуты, а не как объекты *(Курс)*.

3.5. Операции: если название описывает операцию, которая применяется к объекту и не может быть выполнена отдельно от него, то это не объект, и эту операцию следует инкапсулировать в классе объекта *(Регистрация)*.

3.6. Роли: название объекта должно отражать его характер, а не роль, которую он может играть в ассоциации *(Декан)*.

3.7. Элементы реализации: все объекты, отдаленные от реальной проблемы, должны быть устранены из модели анализа.

Подготовка словаря системы

Глоссарий (словарь системы) – это центральное хранилище относящихся к системе абстракций. Глоссарий рекомендуется составлять в алфавитном порядке. Сначала в него помещаются все существенные классы этапа анализа, названные именами, которые отражают их смысл. По мере проектирования в глоссарий заносятся описания программных классов (их атрибутов, операций) и ассоциаций между классами. На начальном этапе он состоит из трех основных граф, в которые заносятся: термин, его обозначение в системе и точное описание

Термин	Обозначение	Описание

По мере разработки происходи расширение за счет детального описания абстракций. Для атрибутов добавляются графы: тип и значение по умолчанию

Имя класса	Имя атрибута	Тип атрибута	Значение по умолчанию	Описание атрибута

Для операций – название, назначение, тип и параметры (с указанием их типов)

Имя класса	Название операции	Назначение операции	Тип операции	Параметры операции	
				Название	Тип

Для отношений – полюсы, направление и описание отношения

Полюс-источник			Полюс-приемник			Направление		Описание отношения
Имя класса	Имя роли	Мощность	Имя класса	Имя роли	Мощность	Однонаправленная	Двунаправленная	

Словарь системы постепенно уточняется путем:

введения новых абстракций;

исключения лишних абстракций;

объединения схожих абстракций.

Создание глоссария дает три существенных выигрыша:

1. Единая терминология. Вырабатывается общепринятая и исчерпывающая терминология, которой можно и нужно пользоваться на протяжении всей работы над проектом.

2. Оглавление. Словарь является естественным оглавлением ко всем материалам проекта, и можно в произвольном порядке обратиться к любому из них.

3. *Обобщение* – возможность посмотреть на весь проект единым взглядом, что позволяет выявить пропущенные общности.

Пример (продолжение). Глоссарий системы РЕГИСТРАТОР:

Термин	Обозначение	Описание
Университет	University	Учебное заведение, в котором преподаватели читают факультативные курсы
Студент	Student	Учащийся университета, который слушает факультативный курс
Факультативный курс	Elective Subject	Дополнительный курс, выбираемый студентом
Преподаватель	Teacher	Сотрудник факультета университета, который читает факультативный курс

Создание структуры

Создание структуры предполагает идентификацию возможных отношений между классами предметной области с точки зрения решаемой задачи. Любое взаимодействие объектов указывает на наличие отношения между классами. Отношения могут обозначать:

1. Физическое расположение. Один объект является частью другого *(Университет – Факультет)*.

2. Направленные действия. Один объект воздействует на другой. В системе АРЕНДАТОР оплаченный клиентом *счет № 123* инициирует генерацию и отправку клиенту *счета-фактуры № 123Ф*, подтверждающую оплату.

3. Связь. Один объект связан с другим *(Преподаватель – Факультативный курс)*.

4. Права владения. Один объект является хозяином другого. Например, *Октябрьская железная дорога* владеет *вагоном № 64084356*.

5. Удовлетворение некоторого условия. Один объект запрашивает состояние другого объекта. Например, если *вагон № 64084356 неисправный*, то *отправить в ремонт*.

Идентификация ассоциаций производится следующим образом:

1. Определить ассоциацию для каждой пары классов, между объектами которых надо будет осуществлять навигацию. Это взгляд с точки зрения данных.

2. Если объекты одного класса должны будут взаимодействовать с объектами другого иначе, чем в качестве параметров операции, следует между этими классами определить ассоциацию. Это взгляд на ассоциацию с точки зрения поведения.

3. Для каждой из определенных ассоциаций задать мощность и имена ролей (особенно если это помогает понять модель).

4. Если один из классов ассоциации представляет собой целое в отношении классов на другом полюсе ассоциации, выделяющих его части, пометить такую ассоциацию как агрегацию.

В общем случае, ассоциации предполагают участие равноправных классов, следовательно, навигацию можно осуществлять как в одном, так и в обоих направлениях. Исключение составляют отношения обобщения и зависимости. Эти отношения применяются при моделировании классов, которые находятся на разных уровнях абстракции или имеют различную значимость. Обобщение и зависимость – односторонние ассоциации.

Пример (продолжение)

Идентификация отношений для системы РЕГИСТРАТОР: *Университет – Факультет*. Один университет состоит из одного или более факультетов. Факультеты не могут существовать без своего университета. *Факультет – Студент*. На одном факультете обучается много студентов. Факультет не может существовать без своих студентов. Факультеты в какой-то степени определяются своими студентами. В то же время студенты связаны со своим факультетом, и в какой-то мере он формирует их облик. *Факультет – Преподаватель*. На каждом факультете должен быть хотя бы один преподаватель. *Факультет – Факультативный*

курс. На факультете могут изучать любое количество дополнительных курсов, в том числе и ни одного. Один факультативный курс может изучаться на любом количестве факультетов, в том числе и ни на одном. *Студент – Факультативный курс*. Каждый студент может посещать любое число дополнительных курсов или не посещать их совсем. На каждый такой курс может приходить не более десяти студентов. *Преподаватель – Факультативный курс*. Каждый преподаватель может вести не более трех дополнительных курсов, в том числе и ни одного. Для каждого курса должен быть хотя бы один преподаватель. Структурные отношения для системы РЕГИСТРАТОР показаны на концептуальной диаграмме классов (рис. 4.6).

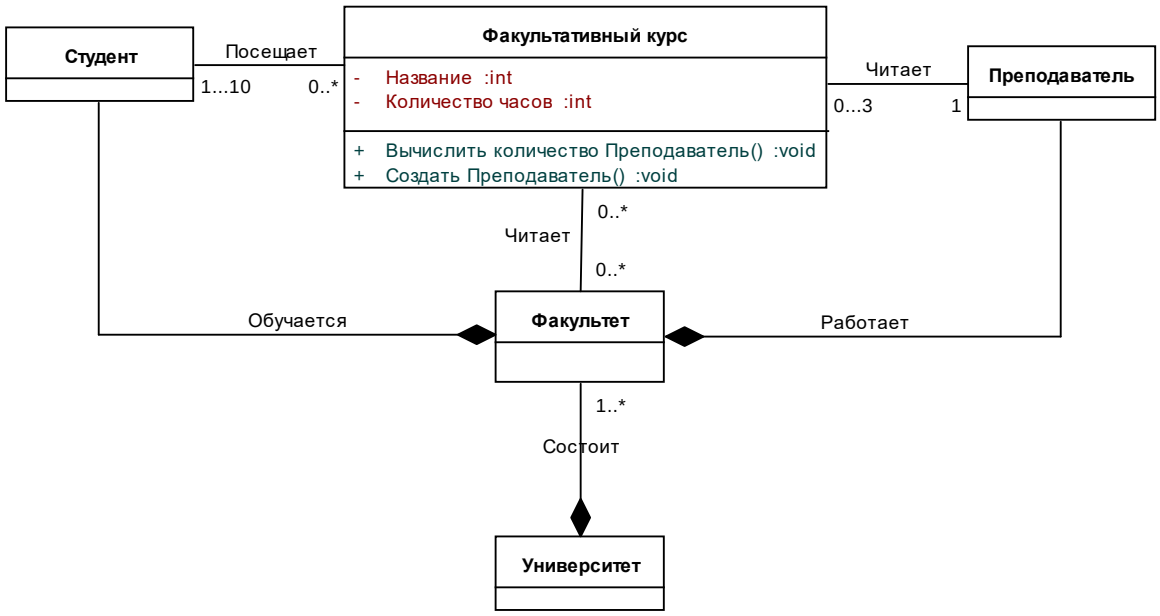


Рис. 4.6. Концептуальная диаграмма классов системы РЕГИСТРАТОР

4.3. Рекомендации по созданию модели анализа

При создании аналитической модели крайне важно ограничиться лишь теми классами, которые являются частью словаря предметной области. Такое ограничение накладывается потому, что требуется простое описание структуры и поведения системы. Модель анализа ПС средней сложности и размера включает до 100 классов. Для создания хорошей модели анализа необходимо помнить:

1. Модель анализа должна быть простой.

2. Модель анализа всегда использует язык предметной области. Абстракции должны формировать часть словаря бизнеса в соответствии с принятой в предметной области терминологией.

3. Каждая диаграмма должна раскрывать отдельную важную часть поведения системы.

4. Следует избегать частностей. Основное внимание должно быть направлено на абстракции предметной области. Здесь не должно быть классов для установления соединений или классов доступа к базе данных.

5. Необходимо стремиться к минимизации связанности, так как каждая ассоциация между классами увеличивает связанность объектов.

6. Если присутствует естественная и очевидная иерархия абстракций, то использовать обобщение. Наследование (как механизм реализации обобщения) – самая сильная форма связанности классов.

7. Модель анализа должна быть понятной всем заинтересованным лицам: пользователям, заказчикам, аналитикам-проектировщикам и тестерам.

Рабочий поток анализа в RUP показан на рис. 4.7.

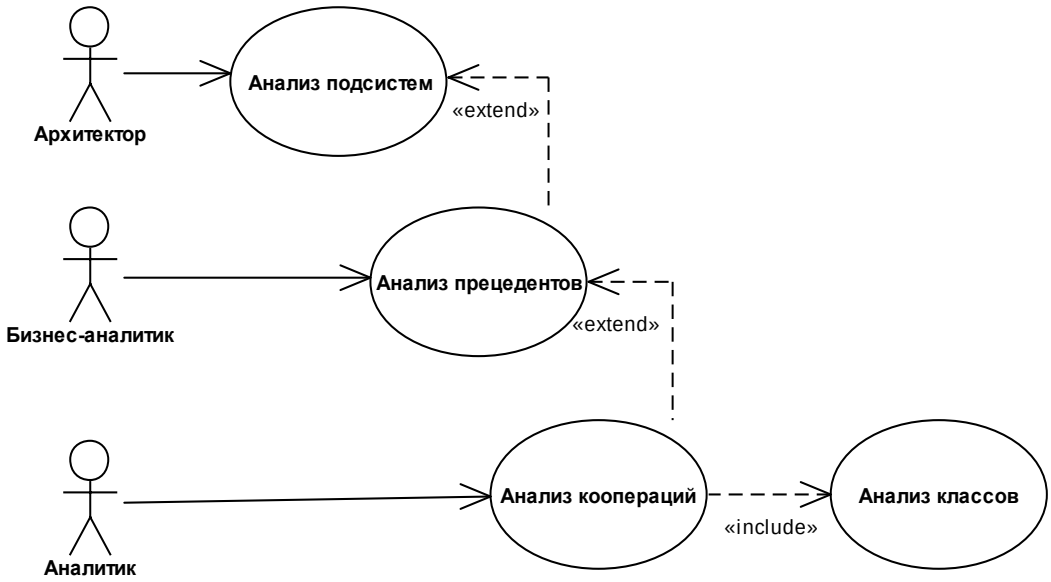


Рис. 4.7. Рабочий поток анализа

4.4. Объектно-ориентированное проектирование

Рассмотрим основные понятия объектно-ориентированного проектирования (OOD), которое направлено на создание моделей программных абстракций. Основой для создания модели проектирования служит модель анализа (рис. 4.8).

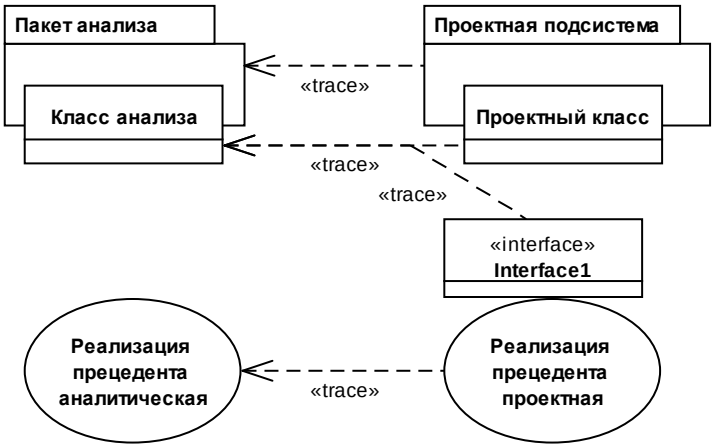


Рис. 4.8. Связь анализа и проектирования

Объектно-ориентированное проектирование – методология, при которой требования воспринимаются с точки зрения программных объектов/классов. Цель проектирования – определить то, как будет реализовываться функциональность, выявленная на этапе анализа. OOD так же, как и OOA использует принципы объектной абстракции и декомпозиции, и решает следующие основные задачи:

1. Определение логических программных классов.
2. Определение атрибутов и операций логических программных объектов.
3. Создание статической логической модели системы.
4. Создание динамической логической модели системы.

Наиболее существенным аспектом проектирования является создание диаграмм классов, которые будут отражать спецификации программных классов и их отношений, для последующей программной реализации. Важно помнить, что классы проектирования – это описания, а не классы языка программирования и они разрабатываются до конкретной реализации.

Определение программных классов

Большинство классов, которые должны участвовать в программном решении, определены в диаграммах классов этапа анализа и занесены в глоссарий. Может получиться так, что не все классы анализа станут программными. Один класс анализа может быть преобразован несколькими классами проектирования. Появятся классы управления для поддержки событий, которые будет инициировать система. Появятся классы, чтобы организовать соединение объектов, а также доступ к хранилищу данных. На этапе проектирования детально прорабатываются следующие абстракции:

1. Классы для управления последовательностью событий. Как правило, для каждого прецедента существует (как минимум) один управляющий класс.

2. Классы, моделирующие сущности предметной области.

3. Классы безопасности для обеспечения защиты данных.

4. Классы отчетов и документов, если эти концептуальные сущности играют важную роль в проектируемой системе.

5. Классы обслуживания (утилиты) для поддержки в программной системе общих для вариантов использования механизмов.

Пример (продолжение)

Для системы РЕГИСТРАТОР рассматривается прецедент *Регистрация информации о преподавателях* (см. рис. 4.5). В качестве управляющего класса определен класс *Управление*. После идентификации программных сущностей диаграмма классов примет вид, который показан на рис. 4.9.

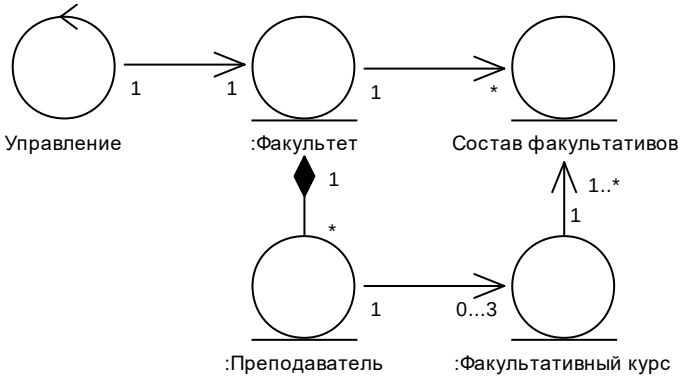


Рис. 4.9. Диаграмма классов

Определение атрибутов

Атрибут – это именованное свойство объекта. Вряд ли атрибуты полностью описаны в постановке задачи, так что следует извлечь их из знания предметной области и из анализа существа объекта. На атрибуты, как и на диаграммы классов, можно посмотреть с трех точек зрения: концепции, спецификации и реализации. Например, атрибут *имя* для объекта *преподаватель* на *уровне концепции* указывает на то, что преподаватели обладают именами; на *уровне спецификации* – на то, что объект *преподаватель* может сообщить свое имя и обладает некоторым механизмом определения имени. На *уровне реализации на языке C++* объект *преподаватель* содержит элемент данных *имя* со значением, соответствующим имени преподавателя. При определении атрибутов важно учитывать следующие обстоятельства:

атрибут должен отражать существенное и важное свойство объекта;

иметь подходящее название;

атрибут не должен быть объектом;

атрибутом реализации (например, указателем);

атрибут не обязательно указывает на свойство объекта реального мира (например, на статус).

После идентификации кандидатов в атрибуты необходимо устранить ненужные и неправильные атрибуты по следующим критериям:

1. Если сущность более важна, чем значение, то это объект, а не атрибут. Например, если система занимается составлением расписания курсов для Вуза, то лекции, практические занятия, семинары, экзамены – это отдельные объекты, а не атрибуты класса *Расписание*.

2. Если значение атрибута зависит от специфического контекста, то лучше заявить атрибут, как роль. Например, *декан*.

3. Идентификаторы реализации не должны быть заявлены как атрибуты. Например, если для какого-либо запроса нужно определить промежуточное значение, то переменную, которая определяет это значение, не следует объявлять в качестве атрибута.

4. Если свойство зависит от наличия связи, то это свойство является атрибутом связи, а не атрибутом объекта. Например, преподаватель может читать или не читать факультативный курс (существует связь *преподаватель – читает – факультативный курс*).

5. Если атрибут описывает внутреннее состояние объекта, которое является невидимым внешней стороной, его следует устранить. Например, если в системе вообще отсутствуют операции, использующие значение какого-либо атрибута, это значит, что без него вполне можно обойтись.

6. Атрибут, который не связан со всеми другими атрибутами, может указывать на класс, который должен быть разбит на два разных класса. Например, *имя, адрес, год рождения, номер кредитной карточки*.

В зависимости от степени детализации обозначение атрибута на диаграмме классов может включать имя атрибута, тип и значение по умолчанию.

Пример (продолжение)

Атрибуты классов для системы РЕГИСТРАТОР показаны на рис. 4.10.

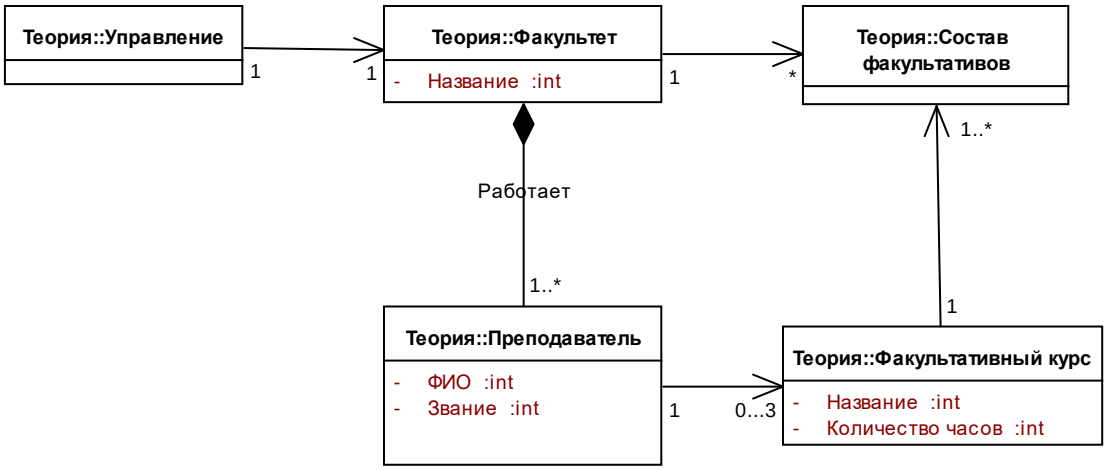


Рис. 4.10. Диаграмма классов с атрибутами

Упрощение классов путем обобщения

После идентификации атрибутов можно проанализировать классы на предмет отношения обобщения. Идентифицировать суперкласс и подклассы можно *снизу вверх* или *сверху вниз*. При проектировании справедливы соглашения о том, что следует по возможности избегать множественного наследования, а в отношениях обобщения не выделять более трех уровней иерархии классов. В противном случае будет достаточно сложно модифицировать систему в дальнейшем.

Пример (продолжение)

Для системы РЕГИСТРАТОР атрибут *фио* является общим для классов *Преподаватели* и *Студенты*. Определяется суперкласс *Персона* с атрибутом *фио*, который будет общим для подклассов. В каждом из подклассов, которые являются специализированными версиями суперкласса, определены уточняющие атрибуты. Диаграмма классов приобретает вид, показанный на рис. 4.11.

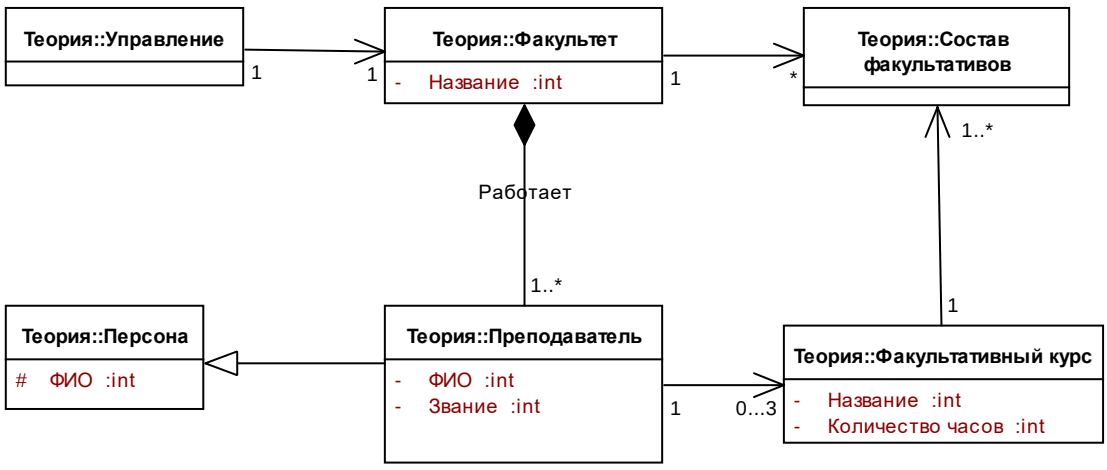


Рис. 4.11. Диаграмма классов с обобщением

Определение операций

Одной из главных задач логического проектирования является задача *назначение обязанностей классов*. Определение операций – это и есть назначение обязанностей. *Операции* – это некоторые сервисы (действия), которые один класс предоставляет другому либо самому себе.

Определение операций фокусирует внимание разработчиков на идентификацию полного и удобного для применения набора действий, который позволит выполнить предъявляемые к ПС требования. При выяснении того, какие операции надо обеспечить, следует стремиться к простоте. Можно выделить несколько общих операций:

1. Определить минимальный набор операций, который требует представляющая класс концепция. Как правило, эти операции при реализации становятся методами (функциями-членами).

2. Добавить к этому набору операции, которые необходимы для удобства.

3. Подумать об общности в именовании и функциональности для всех классов.

4. Думать не о реализации операций, а только о спецификации.

Чем больше операций, тем вероятнее, что они останутся неиспользованными, затруднят реализацию и дальнейшее развитие ПС. Например, функции, часто читающие и записывающие состояние объекта, жестко ограничивают реализацию класса и лимитируют возможность его перепроектирования, так как понижают уровень абстракции. Объявление операции виртуальной критически влияет на использование класса и на отношение между классами. Класс с виртуальной функцией потенциально действует как интерфейс к еще не определенному классу, а виртуальная функция подразумевает зависимость от еще не определенного класса. Стереотипные операции *(create and initialize, delete, get…, set…, add…, remove…)*, которые выполняются всеми объектами любого класса, в диаграммах классов, как правило, не показывают. Эти операции часто показывают в диаграммах последовательностей.

Пример (продолжение)

Для прецедента *Регистрация преподавателей* системы РЕГИСТРАТОР определены операции, которые показаны на диаграмме классов (рис. 4.12).

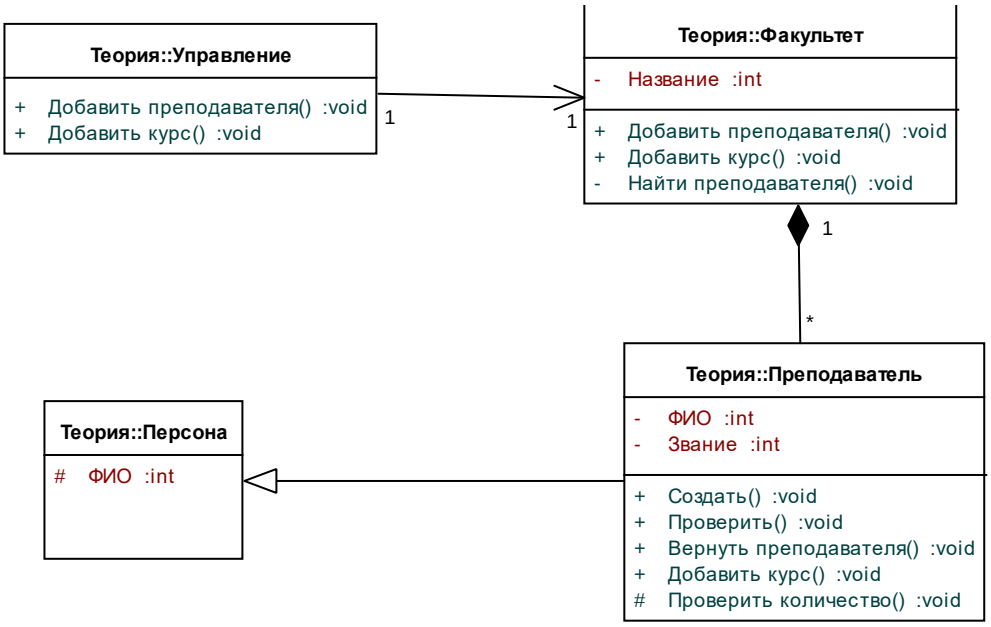


Рис. 4.12. Диаграмма классов с операциями для прецедента *Регистрация преподавателей*

Для прецедента *Регистрация курсов* дополнительно определены операции, которые показаны на диаграмме классов (рис. 4.13).

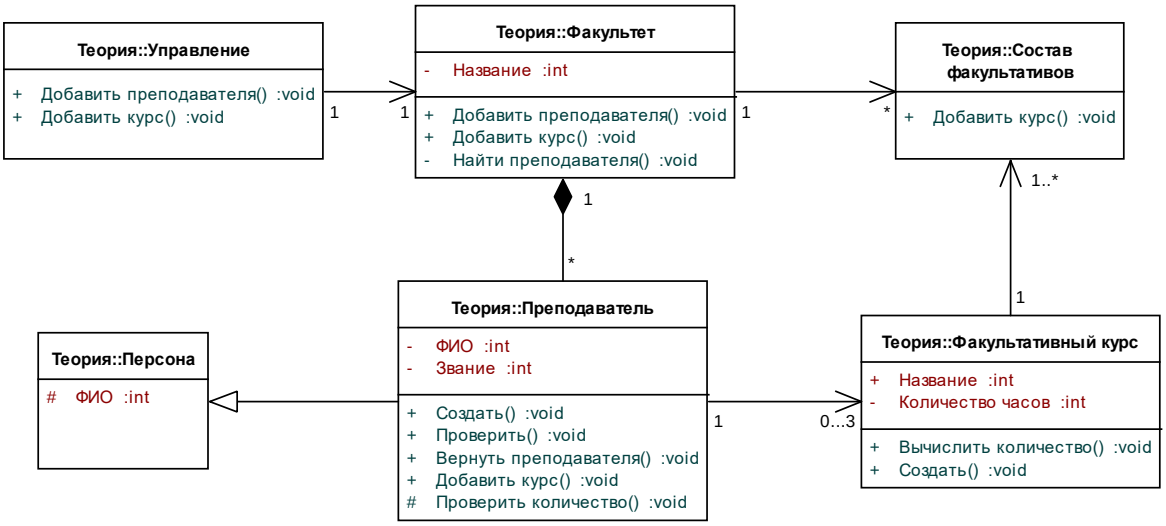


Рис. 4.13. Диаграмма классов с операциями для прецедента *Регистрация курсов*

В конце детального проектирования для каждой операции должны быть определены тип возвращаемого значения, параметры, а также пред- и постусловия. Окончательная диаграмма классов для прецедента *Регистрация информации о преподавателях* системы РЕГИСТРАТОР представлена на рис. 4.14

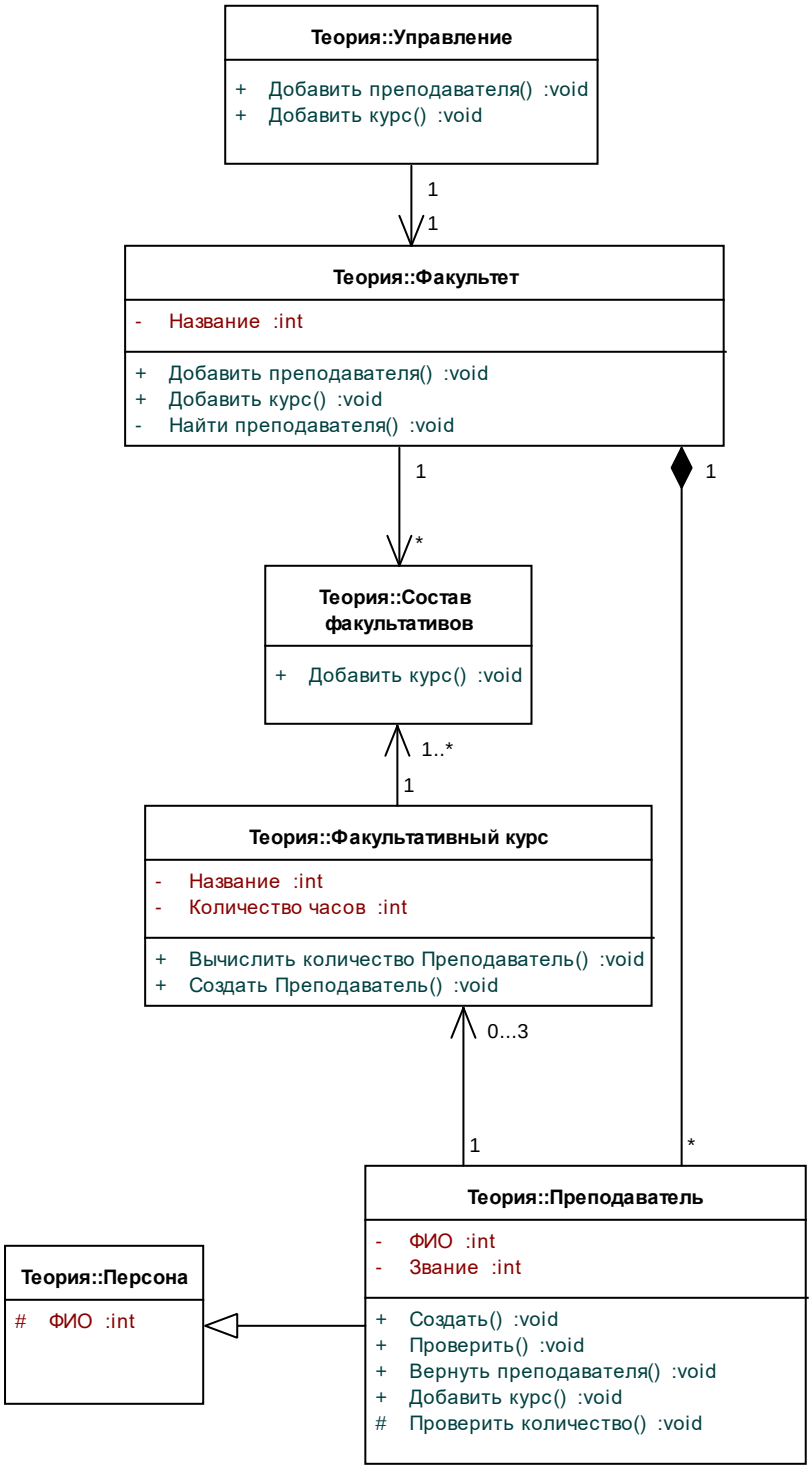


Рис. 4.14. Диаграмма классов для прецедента
Регистрация информации о преподавателях

На базе инструмента Enterprise Architect автоматически создаем общую диаграмму классов (рис. 4.15).

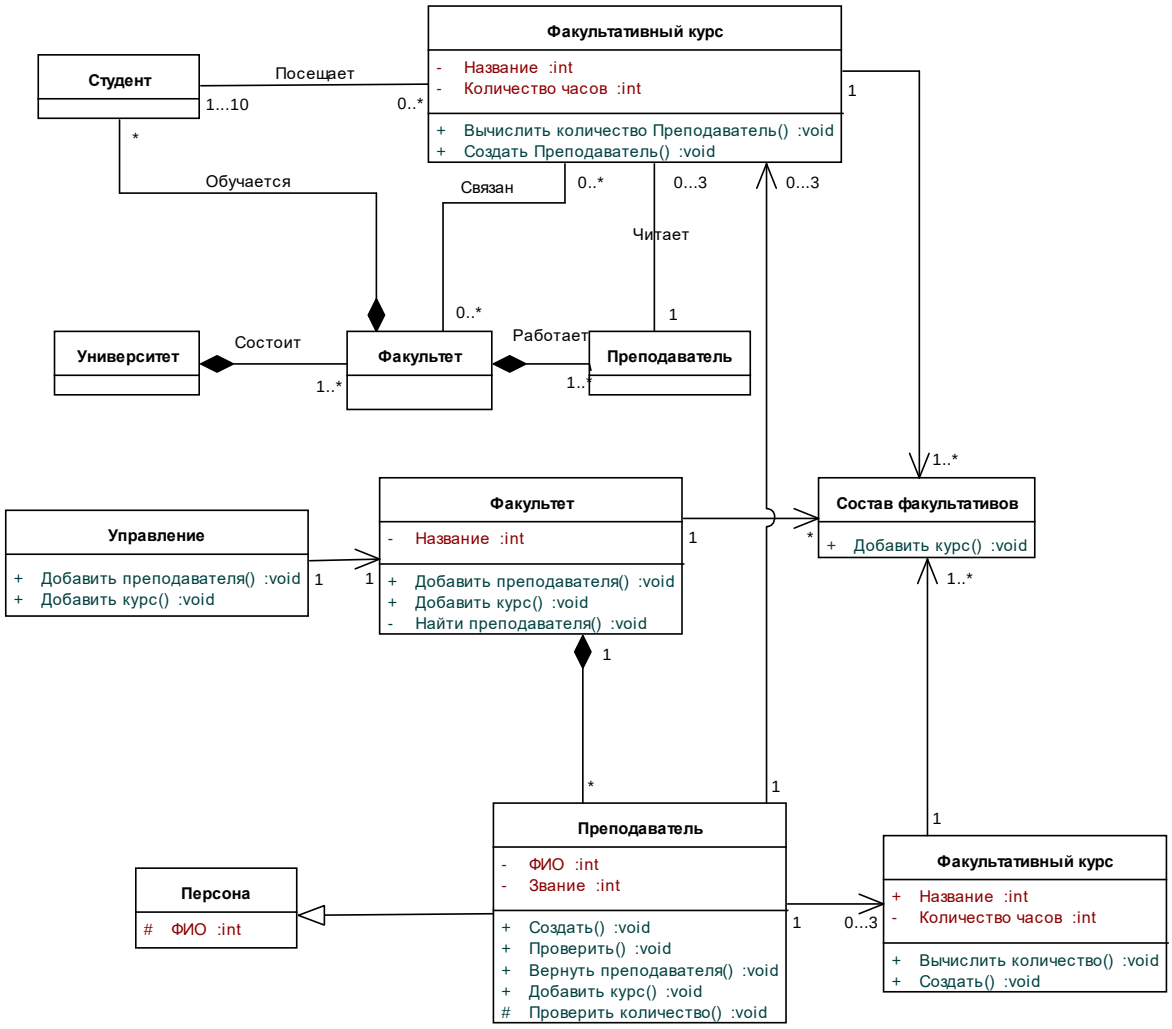


Рис. 4.15. Общая диаграмма классов

Рабочий поток проектирования в RUP

В данном пункте представлено описание стандартного процесса объектно-ориентированной разработки информационного (программного) обеспечения бизнеса – *Рационального Унифицированного Процесса* (*Rational Unified Process*) – *РУП*. Для его составления использованы работы [6, 54].

Проектирование – основная деятельность для моделирования реализации прецедентов на базе модели анализа. Анализ и проектирование в некоторой степени могут происходить параллельно. Следует поддерживать две отдельные модели (аналитическую и проектную), если система большая, сложная, с предположительно большим сроком службы и подверженная частым изменениям. Рабочий поток проектирования показан на рис. 4.16

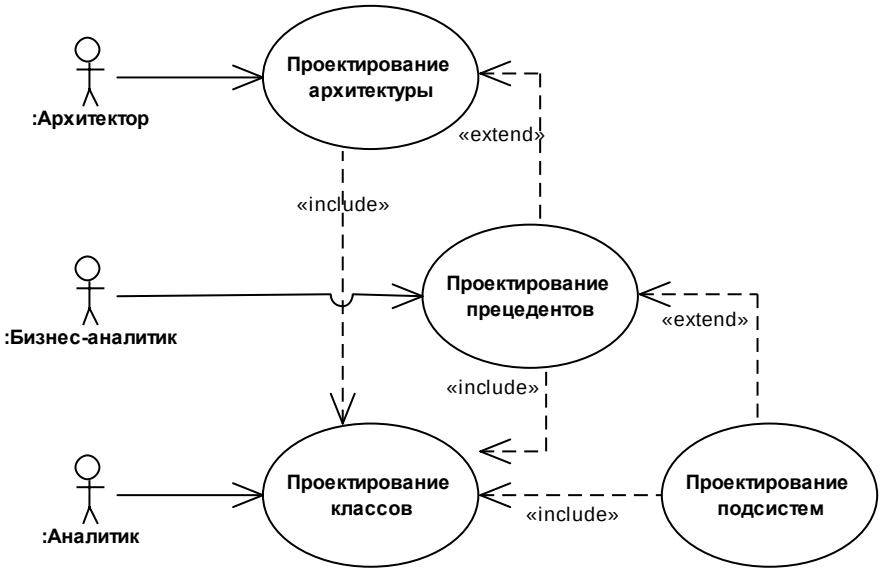


Рис. 4.16. Рабочий поток проектирования

Одни и те же участники проектной группы должны заниматься анализом и проектированием. Процесс проектирования – это итеративный процесс.

Проектная модель включает:

проектные подсистемы;

проектные классы;

интерфейсы;

проектные реализации прецедентов;

диаграмму развертывания в первом приближении.

4.5. Модели системы

Физическая модель тесно связана с тремя моделями: статической, динамической и функциональной (рис. 4.17). Три перечисленные модели делят ПС на ортогональные, но взаимосвязанные представления, каждое из которых описывает один аспект системы и содержит ссылки на другие модели. Статическая модель обеспечивает необходимую основу ПС. Эта модель определяет исходную структуру работы динамической и функциональной моделей.

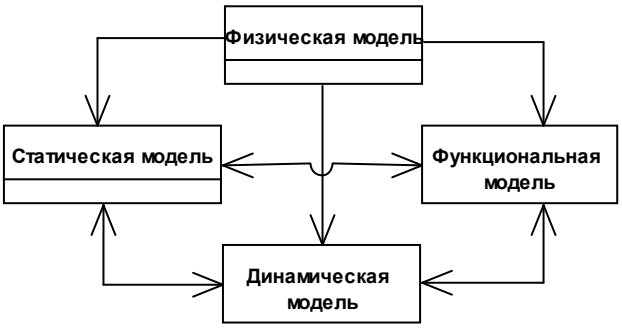


Рис. 4.17. Модели системы

В конце проектирования три модели дают *реализацию* (физическую модель), которая включает *данные* (статическая модель), *упорядочение* (динамическая модель) и *методы* (функциональная модель).

Статическая модель системы

Статическая модель системы отражает наличие и расположение классов и компонентов системы. При использовании UML модель представляется диаграммами классов, которые обеспечивают необходимую основу системы и фиксируют ее внутреннюю структуру. Эти диаграммы определяют основные элементы, из которых будет строиться система, и их возможные отношения. Элементы системы объединяются в компоненты, что специфицируется в диаграммах компонентов. Статическая модель тесно связана с динамической и функциональной моделями.

Динамическая модель системы

Динамическая модель представляет аспекты управления системой, временные аспекты и поведение системы. Как правило, динамика системы проектируется вместе с ее статикой. Она может строиться и после того, как построена и согласована статическая модель, в которой уже распределены обязанности между классами. В динамической модели:

1. Определяется структура управления при помощи описания последовательности операций во времени. Эти операции происходят в ответ на внешние стимулы, т. е. инициируются либо актерами, либо активными программными классами, которые посылают сообщения. Операции в диаграммах не отражают того, где они работают (статическая модель), что эти операции делают (функциональная модель), как они реализованы (физическая модель). При использовании UML временные аспекты отображаются диаграммами последовательностей.

2. Фиксируются изменения, происходящие с объектами, при этом создаются диаграммы состояний для классов.

3. Фиксируются изменения, происходящие со связями между объектами во время работы системы. Это может быть отражено в диаграммах состояний для вариантов использования.

Главные концепции динамического моделирования – события (внешние стимулы) и состояния (значения и связи с объектом).

Функциональная модель системы

Функциональная модель показывает различные производимые классами операции (вычисления) и функциональную структуру данных. Функциональная модель содержит четыре компонента:

1. Операции – преобразование данных.

2. Участники операций – производители и потребители значений.

3. Хранилища данных – создатели задержки между созданием и использованием данных.

4. Потоки данных – отношения между операциями, участниками и хранилищами данных.

Для операций могут быть построены UML-диаграммы деятельностей.

Хранилища и потоки данных, отношения между значениями при вычислении моделируют в диаграммах потоков данных (Data Flow Diagram). Эти диаграммы показывают потоки значений от внешних входов через операции и внутренние хранилища данных к внешним выходам.

Физическая модель системы

Физическая модель системы отражает использованные программные и аппаратные средства для реализации системы, а также расположение компонентов реализации по аппаратным устройствам. Элементы реализации специфицируются в UML-диаграммах классов, компонентов и развертывания.

Статическая и динамическая модели

Динамическая модель определяет:

1. Допустимую последовательность изменения состояний объекта класса (из статической модели). Состояния связаны со значениями атрибутов и конкретными связями объекта. Если имеют место существенные различия в состояниях объектов выделенного класса, то объекты следует моделировать как разные классы.

2. Сообщения, которые могут посылать объекты друг другу для того, чтобы управлять системой и менять состояние. Сообщения могут быть представлены операциями-модификаторами в статической модели.

Такие отношения из статической модели, как обобщение и агрегация, также применяются к динамической модели. *Обобщение состояния* подразумевает выделение однотипных подсостояний объекта. *Агрегация состояния* – это составное состояние, т. е. соединение частей более чем одного подсостояния.

Статическая и функциональная модели

Все четыре компонента функциональной модели (операции, участники, хранилища и потоки данных) могут быть связаны со статической моделью.

1. Операции в функциональной модели показывают то, что должно быть реализовано в методах.

2. Участники – это объекты из статической модели, которые связаны по операции. Как правило, один входной объект является поставщиком (сервером), а один выходной объект – потребителем (клиентом). Другие входные объекты – это параметры операции.

3. Хранилища данных – это структуры данных для хранения значений атрибутов объектов из статической модели.

4. Потоки данных – это движение значений атрибутов из статической модели:

движение к участникам – это операции над объектами (например, модификаторы, запросы);

движение от участников – это операции с объектами (например, преобразование объектов, возврат значений);

движение к хранилищу – это запросы к хранилищу на получение данных;

движение от хранилища данных – это получение данных из хранилища и, как правило, модификация данных.

Динамическая и функциональная модели

Взаимодействие между этими двумя моделями заключается в следующем:

динамическая модель определяет операции (без рассмотрения того, как они будут выполняться) и рассматривает состояния, когда операции выполнены;

функциональная модель определяет то, как выполнить операции, и какие параметры для этого необходимы.

Между операциями с участниками и операциями с хранилищем данных имеется различие. Поскольку участники – это активные объекты, то динамическая модель обязательно должна определять, когда они действуют, а функциональная – как они действуют. Хранилища данных – пассивные объекты (они только отвечают на модификацию и запросы). Для больших систем хранилище данных не что иное, как база данных. Организация, динамика и функциональность базы данных моделируются отдельно.

4.6. Методы проектирования

Почти все методы проектирования имели оригинальные графические нотации и нашли поддержку в виде CASE-средств. Попытка унификации привела к появлению UML, при создании которого авторы соединили идеи трех методов:

1. OMT (Object Modeling Technique) – технология объектного моделирования, разработанная в 1991 г. Джеймсом Рамбо (James Rumbaugh) в научно-исследовательском центре General Electric.

2. Booch’93 – метод визуального моделирования Гради Буча (Grady Booch).

3. OOSE (Object-Oriented Software Engineering) – метод, известный под названием Objectory, Ивара Якобсона (Ivar Jacobson), 1992 г.

По сложившейся практике, наряду со смысловыми названиями (или вместо них), методы получали имена своих создателей. Среди них:

CRC (Class – Responsibility – Collaborations): Класс – Ответственность – Сотрудничество. Метод создан в 1989 г. Разработчики метода: Уорд Каннингхем (Cunningham) и Кент Бек (Beck).

RDD (Responsibility-Driven Design) – проектирование по обязательствам. Метод в 1989 году разработан Ребеккой Вирс-Брок (Wirfs-Brock).

OOA/D (Object-Oriented Analysis and Design) – объектно-ориентированный анализ и проектирование. Авторы Питер Коад (Coad), Эдвард Йордон (Yourdon) создали метод в 1991 г. и развили в 1996 г.

Object-Oriented Systems Analysis and Recursive Design – рекурсивное проектирование. Авторы метода: Салли Шлаер и Стивен Меллор.

В настоящее время наиболее естественным является применение набора моделей, созданных с помощью UML, так как этот язык стандартизирован, широко используется и постоянно развивается.

4.7. Унифицированный процесс разработки программного обеспечения

Рациональный унифицированный процесс RUP (Rational Unified Process) – одна из наиболее совершенных методологий, которая формировалась с прицелом на поддержку управления качеством в рамках требований CMM – SEI (модели роста возможностей Capability Maturity Model Института разработки программ Software Engineering Institute). Унифицированный процесс – это методология создания ПС, оформленная в виде базы знаний, которая снабжена поисковой системой. RUP определяет: *кто* делает, *когда* делает, *что* делает, *как* достичь заданной цели. Методология широко используется для производства объектно-ориентированных программных систем и соответствует стандарту ISO 12207, связанному с процессами жизненного цикла. Унифицированный процесс, созданный в компании IBM Rational Software, представляет собой программный продукт, применение которого гарантирует получение как минимум третьего уровня технологической зрелости организаций-разработчиков согласно CMM – SEI. Возможности RUP:

1. Поддержка необходимого уровня хода разработки.

2. Обеспечение всестороннего контроля.

3. Установление последовательности различных видов деятельности, необходимых для преобразования требований пользователей в ПС.

4. Управление задачами как одного разработчика, так и проектной группы в целом.

Базовые понятия RUP

Основными понятиями унифицированного процесса являются:

1. Артефакт (artifact). Существенная часть информации, которая порождается, изменяется и используется разработчиками в ходе проекта для выпуска целевой системы. Артефактом может быть модель или ее часть, код программы, исполняемая программа, стандарт, документация и т. д. Весь процесс разработки рассматривается в RUP как последовательное создание и развитие артефактов.

2. Вариант использования (use case), или прецедент (precedent). Описывает законченную последовательность выполняемых системой действий для получения пользователем конкретного и значимого для него результата.

3. Роль (role). Видение системы, которое может быть поручено отдельному сотруднику или группе. Каждая роль требует ответственности и способностей для выполнения некоторой деятельности или разработки некоторого артефакта. Типичные роли для RUP: руководитель проекта, архитектор, аналитик, проектировщик, программист, тестер, пользователь.

4. Деятельность (Activity). Выполняемая сотрудником в ходе проекта существенная единица работы, соответствующая понятию технологической операции. Подразумевает четко определенную ответственность сотрудника. Дает точно определенный набор артефактов, который базируется на хорошо определенных исходных данных (другой набор артефактов). Каждый вид деятельности связан с конкретной ролью и обычно выполняется одним исполнителем. Любая деятельность должна планироваться и сопровождаться набором руководств (методик выполнения артефактов). Видом деятельности может быть планирование тестирования, выполнение теста на производительность, определение прецедентов.

5. Дисциплина (discipline). Соответствует технологическому процессу и представляет собой последовательность действий, приводящую к получению конкретного и значимого результата.

Связь между понятиями представлена на рис. 4.18.

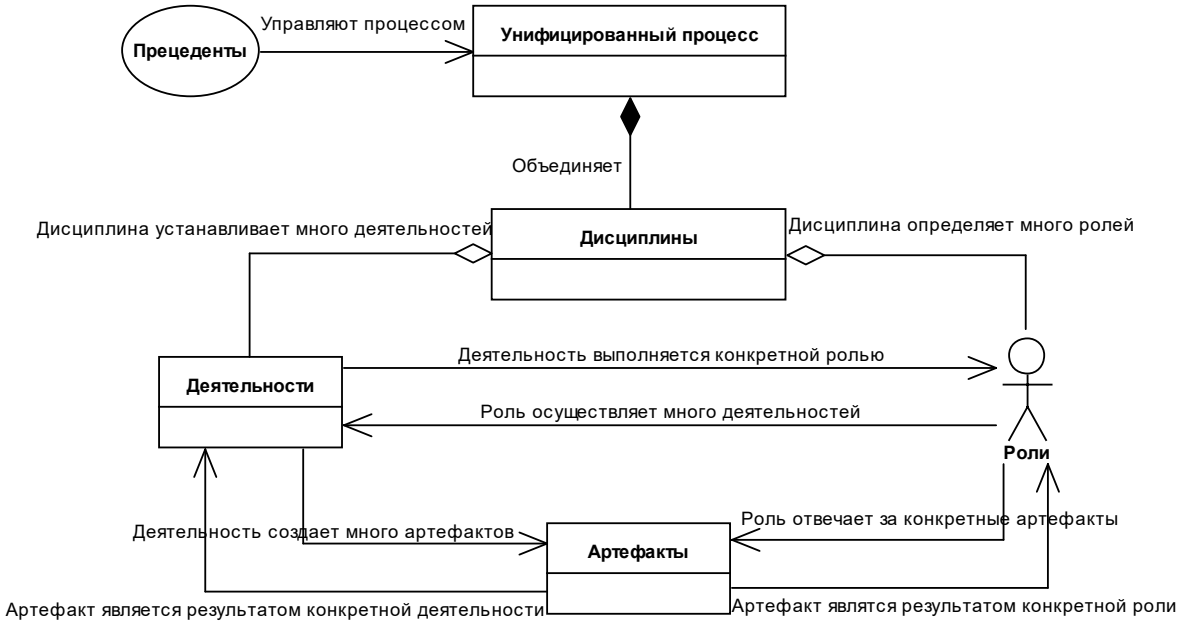


Рис. 4.18. Связь базовых понятий унифицированного процесса

Модели унифицированного процесса

Основой унифицированного процесса являются модели, которые создаются с использованием стандарта унифицированного языка моделирования UML. Визуальные модели позволяют разработчикам определять, конструировать и документировать артефакты, и все вместе полностью описывают системную архитектуру. Модели отображают работу проектируемой системы на уровнях взаимодействия:

между пользователями и системой;

объектов внутри системы;

между различными системами.

Основной набор моделей RUP включает следующие модели:

бизнес-модель. Описывает предметную область и контекст системы;

модель прецедентов. Показывает связи между пользователями и вариантами использования;

модель анализа. Уточняет детали прецедентов и создает первичное распределение поведения системы по набору объектов;

модель проектирования. Определяет статическую структуру системы и кооперации классов для реализации прецедентов;

модель реализации. Показывает представленные исходным кодом компоненты и распределение классов по компонентам;

модель развертывания. Определяет расположение оборудования и связи между отдельными устройствами, а также местонахождение хранилищ данных и компонентов в соответствии с топологией сети;

модель тестирования. Описывает варианты тестов для проверки прецедентов.

Принципы методологии RUP

Основными положениями методологии являются:

1. Планирование и управление проектом на основе функциональных требований к системе (use case driven).

2. Построение системы на базе архитектуры (architecture centric).

3. Итеративный и инкрементный подход к разработке системы (controlled iterative development).

Все три принципа RUP – управляемая прецедентами, ориентированная на архитектуру, итеративная и инкрементная разработка – неразрывны и одинаково важны. Архитектура предоставляет структуру, направляющую работу в итерациях. В каждой итерации определяют цели и направляют работу варианты использования.

Управляемая прецедентами разработка

Прецеденты позволяют установить и описать цели проекта. Концепция прецедентов применяется в RUP на всех стадиях разработки – от формулировки требований до поставки готовой системы. Варианты использования, определенные для системы, составляют основу всего процесса. Производство программного продукта является управляемым функциями предметной области, что обеспечивает согласованность стадий разработки. Прецеденты связывают воедино все входящие в разработку виды деятельности, позволяют оценить объем и сложность предстоящей работы, а также определить содержание итераций. Наиболее важное преимущество использования прецедентов заключается в том, что они управляют проектом. Использование прецедентов в разных системных моделях показано на рис. 4.19.

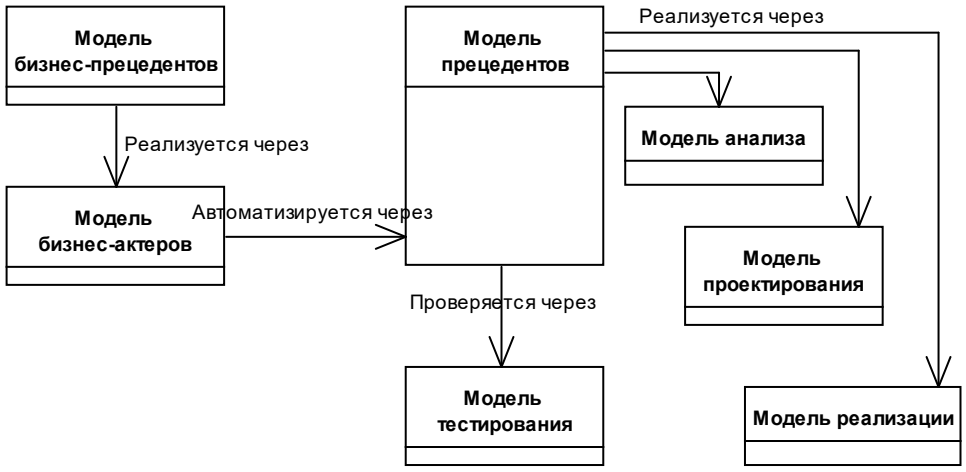


Рис. 4.19. Прецеденты и системные модели

Бизнес-моделирование определяет прецеденты производственного уровня, предоставляя среду для разработки системы. Модели бизнес-моделирования показывают пользователей и их функции, которые подлежат автоматизации. *Модель прецедентов* – это результат управления требованиями, где определяется, что (по мнению пользователей) должна делать система. Для *анализа и проектирования* прецеденты служат основой реализации функциональности системы. Прецеденты описывают реализацию через взаимодействующие объекты/классы моделей анализа и проектирования. В этих моделях определяется, как (по мнению разработчиков) система должна выполнять свои функции. В *реализации* согласно модели проектирования создается исполняемое приложение, в котором прецеденты реализуются через классы среды программирования. Для *тестирования* прецеденты являются источником при составлении контрольных задач и методик испытаний. Каждый прецедент тестируется для проверки системы.

Ориентированная на архитектуру разработка

Унифицированный процесс предлагает системный подход к созданию архитектуры. Он предусматривает установление архитектурного стиля, правил проектирования и ограничений. Архитектура системы используется в качестве базы для построения концепции, создания, управления и развития системы в ходе ее разработки. Важное внимание уделяется раннему определению архитектуры и формулированию основных ее особенностей. Архитектура складывается из подсистем, классов, компонентов и узлов, а также из их кооперации посредством интерфейсов. Распределение обязанностей и организация взаимодействия этих элементов должны обеспечить выполнение функциональных требований.

Архитектура разрабатывается итеративно в ходе определения требований, анализа, проектирования, реализации и тестирования. Для построения базового уровня архитектуры используются прецеденты, требования к системному программному обеспечению, унаследованные системы, нефункциональные требования. Описание архитектуры заключается в представлении моделей системы и разбито на части:

1. *Архитектурное представление модели прецедентов* включает полное описание самых существенных вариантов использования без учета остальных, обеспечивает базу для создания остальных архитектурных представлений.

2. *Архитектурное представление модели анализа* выражено в пакетах и классах анализа, а также в аналитических реализациях прецедентов. Как правило, структура системы сохраняется без изменений до начала проектирования. Это представление предназначено для уточнения и структурирования системных требований.

3. *Архитектурное представление модели проектирования* содержит большинство значимых подсистем, классов, интерфейсов и реализованных прецедентов проектирования, которые формируют словарь системных решений. С точки зрения проектирования рассматриваются механизмы параллелизма и синхронизации. Это представление ориентировано на принятие решений относительно нефункциональных требований к системе, включая требования к производительности, масштабируемости и пропускной способности.

4. *Архитектурное представление модели реализации* описывает компоненты, используемые при сборке и выпуске программного продукта. Представление предназначено для управления конфигурацией версий системы, состоящей из необходимых для создания работающего приложения относительно независимых компонентов.

5. *Архитектурное представление модели развертывания* определяет физическую архитектуру системы в понятиях связанных аппаратных устройств, на которых выполняются компоненты приложения. Часто сетевая топология известна еще во время определения требований, поэтому физическая архитектура процессоров и устройств может быть сформирована до начала проектирования. Это представление предназначено для распространения, поставки и установки системы.

Итеративная и инкрементная разработка

При итеративном подходе разработка делится на ряд коротких промежутков времени, а проект – на несколько меньших мини-проектов, которыми легче управлять. Каждая итерация выполняется согласно обычным этапам водопадной модели жизненного цикла и создает версию, включающую в себя частично завершенную версию целевой системы и документацию. В ходе последовательных итераций версии развиваются до тех пор, пока не будет получен окончательный вариант системы.

Такой подход позволяет обнаруживать просчеты на ранних стадиях, когда система более податлива к изменениям, а также помогает разработчикам вовремя устранить возможные проблемы.

Цели итеративной и инкрементной разработок

RUP использует такую разработку для достижения следующих целей:

получать на ранних итерациях описание критических и опасных рисков;

получать устойчивую архитектуру для создания программной системы;

вносить и контролировать неизбежно появляющиеся в ходе разработки дополнительные требования и изменения в уже имеющиеся требования;

улучшить систему и снизить риски в ходе последующих итераций;

снизить организационные риски и повысить эффективность работы участников проектной группы.

Преимущества итеративной и инкрементной разработок

Преимущества, которые обуславливают эффективность управления:

деление проекта на короткие временные итерации способствует быстрому принятию решений и фиксирует внимание на самых существенных задачах, а также повышает ответственность проектной группы за сдачу работ в срок;

регулярная поставка работающего приложения улучшает возможность получения обратной связи в процессе компиляции, интеграции, использования различных инструментов тестирования, а также от основных заинтересованных лиц, что помогает выявить проблемы тогда, когда их можно исправить с наименьшими затратами. Исправление ошибок на ранних этапах разработки повышает экономическую эффективность проекта;

автоматизированный сбор показателей от инструментов разработки и управления предоставляет в распоряжение точную информацию о состоянии итерации, что способствует объективной оценке хода процесса разработки и положительно сказывается на качестве;

итерации повышают шанс создать систему, которая удовлетворяет реальным потребностям заказчиков, потому что на протяжении всей разработки вносятся небольшие корректировки и тестирование проходит быстрее.

Другие важные принципы разработки

Необходимо проводить постоянный контроль качества, раннее и частое тестирование в реальных условиях эксплуатации, оценку рисков на ранних стадиях проекта, визуальное моделирование с помощью языка UML, поддерживать обратную связь с пользователями. Ткже требуется модификация требований.

ГЛАВА 5. ТРЕБОВАНИЯ ПРИ РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

5.1. Виды требований

Требования к продукту охватывают требования как пользователей (внешнее поведение системы), так и разработчиков (некоторые скрытые параметры).Термин *пользователи* относится ко всем заинтересованным лицам в создании системы [33, 58].

Требования к ПО состоят из трех уровней – бизнес-требования, требования пользователей и функциональные требования. Каждый уровень имеет свои нефункциональные требования.

Требования пользователей (user requirements) описывают цели и задачи, которые пользователям позволит решить система. К способам представления этого вида требований относятся варианты использования, сценарии и таблицы «событие–отклик».

Системные требования (system requirements) обозначают высокоуровневые требования к продукту, которые содержат многие подсистемы или вся система. Из требований для всей системы главными являются функциональные требования к ПО.

Функциональные требования включают описание требований к видам и типам реализуемых функций и документируются в *спецификации требований к ПО* (software requirements specification, SRS), где описано и ожидаемое поведение системы.

Спецификация требований к ПО используется при разработке, тестировании, гарантии качества продукта, управлении проектом и его функциями. В дополнение к функциональным спецификация содержит нефункциональные требования (защита данных, адаптивность, изменчивость и др.), где описаны цели и атрибуты качества.

Атрибуты качества (quality attributes) представляют собой дополнительное описание функций программного продукта, выраженных через описание его характеристик, важных для пользователей или разработчиков. К таким характеристикам относятся легкость и простота использования, легкость перемещения, целостность, эффективность и устойчивость к сбоям. Другие нефункциональные требования описывают внешние взаимодействия между системой и внешним миром, а также ограничения дизайна и реализации. *Ограничения* (constraints) касаются выбора возможности разработки внешнего вида и структуры продукта.

Бизнес-требования (business requirements) содержат высокоуровневые цели организации или заказчиков бизнес-системы

Бизнес-правила (business rules) включают корпоративные политики, правительственные постановления, промышленные стандарты и вычислительные алгоритмы. Они не являются требованиями к ПО, потому

что находятся снаружи границ любой системы ПО. Однако часто они налагают ограничения на выполнение конкретных вариантов использования или на функции системы, подчиняющиеся соответствующим правилам. Бизнес-правила становятся источником атрибутов качества, которые реализуются в функциональности.

Классификация требований

Формируемые требования к системе разделены на две категории:

функциональные требования, которые отображают возможности проектируемой системы;

нефункциональные требования, которые отображают ограничения, определяющие принципы функционирования системы и доступа к данным системы пользователей.

Первая из приведенных категорий дает ответ на вопрос «что делает система», а вторая определяет характеристики ее работы, например, что вероятность сбоев системы на некотором промежутке времени, доступ к ресурсам системы разных категорий пользователей и др.

Функциональные требования характеризуют функции системы или ее ПО, а также способы поведения ПО в процессе выполнения функций и методы передачи и преобразования входных данных в результаты.

Нефункциональные требования определяют условия и среду выполнения функций (например, защита и доступ к БД, секретность и др.). Разработка требований и их локализация завершается на этапе проектирования архитектуры и отражается в специальном документе, по которому проводится окончательное согласование требований для достижения взаимопонимания между заказчиком и разработчиком.

Функциональные требования связаны с семантическими особенностями предметной области (ПрО), для которой планируется разработка ПС. Важным фактором является проблема использования соответствующей терминологии при описании моделей ПрО и требований к системе. Одним из путей ее решения является стандартизация терминологии для нескольких ПрО (например, для информационных технологий, систем обеспечения качества и др.). Тенденция к созданию стандартизированного понятийного базиса для большинства ПрО отражает важность этой проблемы в плане обеспечения единого понимания терминов, используемых в документах, описывающих требования к системе и к ПО.

Нефункциональные требования могут иметь числовой вид (например, время ожидания ответа, количество обслуживаемых клиентов, базы данных (БД) и др.), а также содержать числовые значения показателей надежности и качества работы компонентов системы, период смены версий системы и др. Для большинства ПС, с которыми будет работать много пользователей, требования должны выражать такие ограничения на работу системы: конфиденциальность;

отказоустойчивость; одновременность доступа к системе пользователей; безопасность; время ожидания ответа на обращение к системе; ограничения на исполнительские функции системы (ресурсы памяти, скорость реакции на обращение к системе и т. п.); регламентация действующих стандартов, упрощающих процессы организации формирования требований и менеджмента.

Иными словами, для каждой системы формулируются нефункциональные требования, относящиеся к защите от несанкционированного доступа, к регистрации событий системы, аудиту, резервному копированию, восстановлению информации. Эти требования на этапе анализа и проектирования структуры системы должны быть определены и формализованы аналитиками. Для обеспечения безопасности системы необходимо определить категории пользователей системы, которые имеют доступ к тем или иным данным и компонентам.

Определяются объекты и субъекты защиты. На этапе анализа системы решается вопрос, какой уровень защиты данных необходимо предоставить для каждого из компонентов системы, какие выделить критичные данные, доступ к которым строго ограничен. Для этого применяется система мер по регистрации пользователей, т. е. идентификации и аутентификации, а также реализуется защита данных, ориентированная на регламентированный доступ к объектам данных (например, к таблицам, представлениям). Если же требуется сделать ограничение доступа к данным (например, к отдельным записям, полям в таблице), то в системе должны предусматриваться определенные средства (например, мандатная защита).

Для обеспечения восстановления и хранения резервных копий БД, архивов баз данных изучаются возможности, предоставляемые СУБД, и рассматриваются способы обеспечения требуемого уровня бесперебойной работы системы, а также организационные мероприятия, отображаемые в соответствующих документах по описанию требований к проектируемой системе.

В целях описания нефункциональных (защита, безопасность, аутентификация и др.) требований к системе и фиксации сведений о способности компонента обеспечивать несанкционированный доступ к нему неавторизованных пользователей языки спецификаций компонентов дополнились средствами описания нефункциональных свойств. Эта группа свойств целиком связана с сервисом среды функционирования компонентов, ее способностью обеспечивать меры борьбы с разными угрозами, которые поступают извне от пользователей, не имеющих прав доступа к некоторым данным или ко всем данным системы.

Процесс формализованного описания функциональных и нефункциональных требований, а также требований к характеристикам

качества с учетом стандарта качества ISO/IEC 9126-94 будет уточняться на этапах ЖЦ ПО и специфицироваться.

В спецификации требований отражается структура ПО, требования к функциям, качеству и документации, а также задается архитектура системы и ПО, алгоритмы, логика управления и структура данных. Специфицируются также системные, нефункциональные требования и требования к взаимодействию с другими компонентами и платформами (БД, СУБД, сеть и др.).

5.2. Анализ и сбор требований

В современных информационных технологиях процесс ЖЦ, на котором фиксируются требования на разработку ПО системы, является определяющим для задания функций, сроков и стоимости работ, а также показателей качества и др.

Анализ и сбор требований является довольно нетривиальной задачей, поскольку в реальных проектах приходится сталкиваться с общими трудностями:

требования не всегда очевидны и могут исходить из разных источников, их не всегда удается ясно выразить словами;

имеется много различных типов требований на различных уровнях детализации и их число может стать большим, требующим ими управлять;

требования связаны друг с другом, а также с процессами разработки ПС и постоянно меняются.

Требования имеют уникальные свойства или значения свойств. Например, они не являются одинаково важными и простыми для согласования. В требованиях к ПС должны отображаться проблемы системы и фиксироваться реальные потребности заказчика, касающиеся функциональных, операционных и сервисных возможностей разрабатываемой системы. В результате создается договор между заказчиком и исполнителем системы на ее создание.

Здесь цена ошибок и нечетких неоднозначных формулировок очень высока, поскольку время и средства могут расходоваться на не нужную заказчику систему или программу. Для внесения изменений в требования может понадобиться повторное проектирование и, соответственно, перепрограммирование всей системы или отдельных ее частей.

Существуют стандарты на разработку требований на систему и документы, в которых фиксируются результаты создания программного, технического, организационного и других видов обеспечения автоматизированных систем (АС) на этапах жизненного цикла (ГОСТ 34.601-90 «Стадии и этапы разработки АС» и ГОСТ 34.201-89 «Документация на разработку АС»).

В процессе формулирования требований на систему принимают участие:

представители от заказчика из нескольких профессиональных групп;

операторы, обслуживающие систему;

разработчики системы.

Процесс формулирования требований состоит из нескольких подпроцессов.

Сбор требований. Источниками сведений о требованиях могут быть:

цели и задачи системы, которые формулирует заказчик. Для однозначного их понимания разработчику системы необходимо их тщательно осмыслить и согласовать с заказчиком;

действующая система или коллектив, выполняющий ее функции. Система, которую заказывают, может заменять собой старую систему, переставшую удовлетворять заказчика или действующий персонал. Изучение и фиксация ее функциональных возможностей дает основу для учета имеющегося опыта и формулирования новых требований к системе.

Таким образом, требования к системе формулируются:

исходя из знаний заказчика относительно проблемной области, формулирующего свои проблемы в терминах понятий этой области;

ведомственных стандартов заказчика и требований к среде функционирования будущей системы, ее исполнительским и ресурсным возможностям.

Методами сбора требований являются:

интервью с носителями интересов заказчика и операторами;

наблюдение за работой действующей системы с целью отделения ее проблемных свойств от тех, которые обусловлены структурой кадров;

сценарии (примеров) возможных случаев выполнения функций системы, ролей лиц, запускающих сценарии или взаимодействующих с системой во время ее функционирования.

Продукт процесса сбора требований – неформализованное их описание – основа контракта на разработку между заказчиком и исполнителем системы. Такое описание является входом для следующего процесса инженерии требований – анализа требований. Исполнитель этого процесса производит дальнейшее уточнение и формализацию требований, а также их документирование в нотации, понятной коллективу разработчиков системы.

Анализ требований. Это процесс изучения потребностей и целей пользователей, классификация и их преобразование к требованиям системы, к ПО, установление и разрешение конфликтов между требованиями, определение приоритетов, границ системы и принципов взаимодействия со средой функционирования. Требования классифицируются по функциональному и нефункциональному принципу, а также по отношению к внешней или внутренней стороне системы.

Функциональные требования относятся к заданию функций системы или ее ПО, к способам поведения ПО в процессе выполнения функций и методам преобразования входных данных в результаты.

Нефункциональные требования определяют условия и среду выполнения функций (например, защита и доступ к БД, секретность, взаимодействие компонентов функций и др.).

Разработанные требования специфицируются и отражаются в специальном документе, по которому проводится *согласование требований* для достижения взаимопонимания между заказчиком и разработчиком.

Функциональные требования отвечают на вопрос «что делает система», а нефункциональные – определяют характеристики ее работы (вероятность сбоя системы, защита данных и др.). К основным нефункциональным требованиям, существенным для большинства ПС и выражающим ограничения, актуальные для многих проблемных областей, относятся: конфиденциальность; отказоустойчивость; несанкционированный доступ к системе; безопасность и защита данных; время ожидания ответа на обращение к системе; свойства системы (ограничение на память, скорость реакции при обращении к системе и т. п.).

Функциональные требования отражают семантические особенности проблемной области, и терминологические расхождения для них являются достаточно существенными. Имеется тенденция к созданию стандартизации понятийного базиса большинства проблемных областей, которые имеют опыт компьютеризации.

Следующий шаг анализа требований – установление их приоритетов и избежание конфликтов между ними.

Продукт процесса анализа – построенная модель проблемы, которая ориентирована на понимание этой модели исполнителем до начала проектирования системы.

К настоящему времени сложилось направление в инженерии программирования – инженерия требования, сущность которой достаточно подробно рассмотрена в соответствующей области знаний ядра SWEBOK (*Guide to the Software Engineering Body of Knowledge)* и приводится ниже.

5.3. Инженерия требований ПО

Инженерная дисциплина анализа и документирования требований к ПО, которая заключается в преобразовании предложенных заказчиком требований к системе в описание требований к ПО, ее спецификации и верификации. Она базируется на *модели процесса* определения требований [16], процессах актеров – действующих лиц, управлении и

формировании требований, а также на процессах верификации и повышения их качества.

Модель процесса – это схема процессов ЖЦ, которые выполняются от начала проекта и до тех пор, пока не будут определены и согласованы требования. При этом процессом может быть маркетинг и проверка осуществимости требований в данном проекте.

Управление требованиями к ПО заключается в планировании и контроле за выполнением требований и расходом проектных ресурсов в процессе разработки компонентов системы на этапах ЖЦ.

Качество и процесс улучшения требований – это процесс формулировки характеристик и атрибутов качества (надежность, реактивность и др.), которыми должна обладать система и ПО, методы их достижения на этапах ЖЦ и адекватности процессов работы с требованиями.

Управление требованиями к системе – это руководство процессами формирования требований на всех этапах ЖЦ, которое включает управление изменениями и атрибутами требований, отражающими программный продукт, а также проведение мониторинга – восстановления источника требований. Неотъемлемыми составляющими процесса управления являются *трассирование требований* для отслеживания правильности задания и реализации требований к системе и ПО на этапах ЖЦ и обратный процесс отслеживания от полученного продукта к требованиям.

При управлении требованиями выполняются процессы: управление версиями требований и рисками; разработка атрибутов требований; контроль статуса требований; измерение усилий в инженерии требований и др. Связь между разработкой и управлением требований представлена на рис. 5.1.

Управление рисками состоит из оценки, предотвращения и контроля появления риска определения отдельных требований. Планирование работ на проекте по управлению рисками проводится в разработке требований.

Рис. 5.1. Связь между разработкой требований, управлением требованиями и другими процессами проекта

5.4. Верификация и формализация требований

Сбор требований является начальным и неотъемлемым этапом процесса разработки ПС. Он заключается в определении набора функций, которые необходимо реализовать в продукте. Процесс сбора требований осуществляется в общении с заказчиком, а также разработчиками посредством метода «мозгового штурма». Результатом является формирование набора требований к системе, именуемой в отечественной практике техническим заданием.

Фиксация требований (Requirement Capturing), с одной стороны, определяется желаниями заказчика в реализации того или иного свойства. С другой стороны, в процессе сбора требований может обнаружиться ошибка, которая приведет к определенным последствиям, а их устранение – к непредвиденным затратам (дополнительное кодирование, перепланирование).

Ошибка может быть тем серьезней, чем позже она будет обнаружена, особенно если это связано с множеством спецификаций. Поэтому одной из составляющей этапа фиксации требований наряду со сбором является верификация требований, а именно проверка их на непротиворечивость и полноту. Автоматизированная верификация требований может производиться лишь после спецификации или формализации требований.

Спецификация требований к ПО – процесс формализованного описания функциональных и нефункциональных требований, требований к характеристикам качества в соответствии со стандартом качества ISO/IEC 12119-94, которые будут учитываться при создании программного

продукта на этапах ЖЦ ПО. В спецификации требований отражается структура ПО, требования к функциям, показателям качества, которых необходимо достигнуть, и к документации. В спецификации задается в общих чертах архитектура системы и ПО, алгоритмы, логика управления и структура данных. Специфицируются также системные требования, нефункциональные требования и требования к взаимодействию с другими компонентами (БД, СУБД, протоколы сети и др.).

Формализация включает в себя определение компонентов системы и их состояний, правил взаимодействия компонентов и определения условий в формальном виде, которые должны выполняться при изменении состояний компонентов. Для формального описания поведения системы используются языки инженерных спецификаций, например, UML. В качестве формальной модели для описания требований применяются базовые протоколы, которые позволяют использовать дедуктивные средства верификации в сочетании с моделированием поведения систем путем трассирования.

Валидация (аттестация) требований – это проверка требований, изложенных в спецификации для того, чтобы убедиться, что они определяют данную систему и отслеживают источники требований. Заказчик и разработчик ПО проводят экспертизу сформированного варианта требований на завершенность. Одним из методов аттестации является прототипирование, т. е. быстрая отработка отдельных требований на конкретном инструменте и исследование масштабов изменения требований, измерение объема функциональности и стоимости, а также создание моделей оценки зрелости требований.

Верификация требований – это процесс проверки правильности спецификаций требований на их соответствие стандартам, непротиворечивости, полноте и выполнимости. В результате проверки требований составляется согласованный выходной документ, устанавливающий полноту и корректность требований к ПО, а также возможность продолжить проектирование ПО.

5.5. Объектно-ориентированная инженерия требований

Структурирование проблемы на отдельные компоненты и обеспечение способов их взаимодействия определяют понятность и «прозрачность» описания требований, полученных в результате процесса анализа.

Типы составляющих компонентов и правила их композиции определяются в программной инженерии как *архитектура системы*. Процесс декомпозиции проблемы, определяющей архитектуру системы, называют *парадигмой программирования,* базирующейся на двух широкораспространенных моделях: функции-данные и объектная модель.

Модель функции-данные исторически появилась первой. Согласно этой модели, проблема декомпозируется на последовательность функций и обрабатываемых с их помощью данных. Элементами композиции служат данные и функции над ними, их представления должны быть согласованы между собой. Если изменяются некоторые данные, то пересматриваются функции, которые их обрабатывают, и определяются пути их изменений, т.е. внесение локальных изменений в постановку проблемы требует ревизии всех данных и функций для подтверждения того, что на них не повлияли внесенные изменения.

Объектно-ориентированный подход к разработке программных систем такого недостатка не имеет. В нем общее видение решения проблемы формирования требований осуществляется исходя из следующих постулатов:

мир составляют объекты, которые взаимодействуют между собой;

каждому объекту присущ определенный состав свойств или атрибутов, который определяется своим именем и значениями;

объекты могут вступать в отношения друг с другом;

значения атрибутов и отношения могут с течением времени изменяться;

совокупность значений атрибутов конкретного объекта в определенный момент времени определяет его состояние;

совокупность состояний объектов определяет состояние мира объектов;

мир и его объекты могут находиться в разных состояниях и порождать некоторые события;

события могут быть причиною других событий или изменений состояний.

Каждый объект может принимать участие в определенных процессах, разновидностями которых являются:

переходы из одного состояния в другое под влиянием соответствующих событий;

возбуждение определенных событий или посылка сообщений другим объектам;

операции, которые могут выполнять объекты;

возможные совокупности действий, которые задают его поведение;

обмен сообщениями.

Объект – это определенная абстракция данных и поведения. Множество экземпляров с общим набором атрибутов и поведением составляет класс объектов. Определение класса связано с известным принципом *сокрытия информации*, суть которого можно сформулировать так: сообщайте пользователю только то, что ему нужно.

Таким образом, определение объектов в соответствии с данным принципом состоит из двух частей – видимой и невидимой. Видимая часть

содержит все сведения, которые требуются для того, чтобы взаимодействовать с объектом и называется *интерфейсом объекта.* Невидимая часть содержит подробности внутреннего устройства объекта, которые «инкапсулированы» (т. е. находятся словно бы в капсуле). Так, например, если объектом является некоторый прибор, который регистрирует показатели температуры, то к видимой его части относится операция показа значения температуры.

Другим важным свойством определения объектов является *наследование*. Один класс объектов наследует другой, если он полностью вмещает все атрибуты и поведение наследуемого класса, но имеет еще и свои атрибуты и (или) поведение. Класс, который наследует свойства другого, называют *суперклассом*, а класс, который наследуют, называют *подклассом.* Наследственность фиксирует общие и отличающиеся черты объектов и позволяет явно выделять компоненты проблемы, которые можно использовать в ряде случаев при построении нескольких классов-наследников.

Классы могут образовывать иерархии наследников произвольной глубины, где каждый отвечает определенному уровню абстракции, являясь обобщением класса-наследника и конкретизацией класса, который наследует его самого. Например, класс «число» в качестве наследников имеет подклассы: «целые числа», «комплексные числа» и «действительные числа». Все эти подклассы наследуют операции суперкласса (сложения и вычитания), но каждый из них имеет свои особенности выполнения этих операций.

При объектно-ориентированном подходе модели определяются через взаимодействие определенных объектов. В модели требований фигурируют объекты, взаимодействие которых определяет проблему, решаемую с помощью программной системы, а в других моделях (проекта, реализации и тестирования) заданный принцип взаимодействия объектов определяет сущность решения этой проблемы (модели проекта и реализации) или проверки достоверности решения (модель тестирования).

В настоящее время предложен ряд современных методов объектно-ориентированного анализа требований, объектно-ориентированного проектирования программ, объектно-ориентированного программирования (C++, JAVA). Наибольшую ценность среди них имеет проблема согласованности между ними.

Если удается установить соответствие между объектами указанных моделей на разных стадиях (процессах) жизненного цикла продукта, то они позволяют провести *трассирование требований*, т. е. проследить за последовательной трансформацией требований объектов на этих стадиях. Трассирование заключается в контроле трансформаций объектов при переходе от этапа к этапу с учетом внесения изменений во все

наработанные промежуточные продукты разных стадий разработки и ее завершения.

Концептуальное моделирование проблемы системы происходит в терминах взаимодействия объектов:

онтология домена определяет состав объектов домена, их атрибуты и взаимоотношения, а также оказываемые услуги;

модель поведения – возможные состояния объектов, инциденты, события, инициирующие переходы из одного состояния в другое, а также сообщения, которые объекты посылают друг другу;

модель процессов – действия, которые выполняют объекты.

Все объектные методы имеют в своем составе приведенные модели, отличающиеся своими нотациями и некоторыми другими деталями.

Метод инженерии требований А. Джекобсона

Данный метод является методом с последовательным выявлением объектов, существенных для домена проблемной области [46]. Все методы анализа предметной области в качестве первого шага выполняют выявление объектов. Правильный выбор объектов обуславливает понятность и точность требований. Рассматриваемый метод Джекобсона дает рекомендации относительно того, с чего начинать путь к пониманию проблемы и поиску существенных для нее объектов.

Автор назвал свой метод как базирующийся на вариантах (примерах или сценариях – use case driven) системы, которую требуется построить. В дальнейшем термин «сценарий» используется для обозначения варианта представления системы.

Сложность проблемы обычно преодолевается путем декомпозиции ее на отдельные компоненты с меньшей сложностью. Большая система может быть достаточно сложной лоя представления ее компонентов программными модулями. Поэтому разработка системы с помощью данного метода начинается с осмысления того, для кого и для чего она создается.

Сложность общей цели выражается через отдельные подцели, которым отвечают функциональные или нефункциональные требования и проектные решения. Цели являются источниками требований к системе и способом оценки этих требований путем выявления противоречий между требованиями и установления зависимостей между целями.

Цели можно рассматривать как точку зрения отдельного пользователя или заказчика, или среды функционирования системы. Цели могут находиться между собой в определенных отношениях, например, конфликтовать, кооперироваться, зависеть одна от другой или быть независимыми.

Следующим шагом после выявления целей являются определение носителей интересов, которым отвечает каждая цель, и возможные варианты удовлетворения составных целей в виде сценариев работы

системы. Последние помогают пользователю получить представление о назначении и функциях системы. Эти действия соответствуют первой итерации определения требований к разработке.

Таким образом, производится последовательная декомпозиция сложности проблемы в виде совокупности целей, каждая из которых трансформируется в совокупность возможных вариантов использования системы, которые трансформируются в процессе их анализа в совокупность взаимодействующих объектов.

Определенная цепочка трансформации (проблема – цели – сценарии – объекты) отражает степень концептуализации в понимании проблемы, последовательного снижения сложности ее частей и повышения уровня формализации моделей ее представления.

Трансформация обычно выражается в терминах базовых понятий проблемной области, активно используется для представления актеров и сценариев или создается в процессе построения такого представления.

Каждый сценарий инициируется определенным пользователем, являющимся носителем интересов. Абстракция определенной роли личности пользователя – инициатора запуска определенной работы в системе, представленной сценарием, и обмена информацией с системой – называется *актером*. Это абстрактное понятие обобщает термин «действующее лицо системы». Фиксацию актеров можно рассматривать как определенный шаг выявления целей системы через роли, которые являются носителями целей и постановщиками задач, для решения которых создается система.

Актер считается внешним фактором системы, действия которого носят недетерминированный характер. Этим подчеркивается разница между пользователем как конкретным лицом и актером, роль которого может играть любое лицо в системе. В роли актера может выступать и программная система, если она инициирует выполнение определенных работ данной системы. Таким образом, актер – это абстракция внешнего объекта, экземпляр которого может быть человеком или внешней системой. В модели актер представляется классом, а пользователь – экземпляром класса. Одно лицо может быть экземпляром нескольких актеров.

Операции в системе, соотнесенные с поведением последовательности ее транзакций и называемые *сценарием*, запускаются лицом в роли. Когда пользователь как экземпляр актера вводит определенный символ для старта экземпляра соответствующего сценария, выполняется ряд действий с помощью транзакций, которые заканчиваются тогда, когда экземпляр сценария находится в состоянии ожидания входного символа.

Экземпляр сценария существует, пока выполняется и его можно считать экземпляром класса, в роли которого выступает описание

транзакции. Сценарий определяет протекание событий в системе и обладает состоянием и поведением. Каждое взаимодействие между актером и системой – это новый сценарий или объект. Если несколько сценариев системы имеют одинаковое поведение, то их можно рассматривать как класс сценариев. Вызов сценария – порождение экземпляра класса. Таким образом, сценарии – это транзакции с внутренним состоянием.

Модель системы по данному методу основывается на сценариях и внесение в нее изменений должно осуществляться повторным моделированием актеров и запускаемых ими сценариев. Такая модель отражает требования пользователей и легко изменяется по их предложению. Сценарий – это полная цепочка событий, инициированных актером, и спецификация реакции на них. Совокупность сценариев определяет все возможные варианты использования системы. Каждого актера обслуживает своя совокупность сценариев.

Можно выделить базовую цель событий, существенную для сценария, а также альтернативные.

Для задания модели сценариев используется специальная графическая нотация со следующими основными правилами:

актер обозначается пиктограммой человека, под которой указывается название;

сценарий изображается овалом, в середине которого указывается название пиктограммы;

актер связывается стрелкой с каждым запускаемым им сценарием.

Диаграмма сценариев для клиента банка как актера, который запускает заданный сценарий, представлена на рис. 5.2. Для достижения цели актер обращается не к кассиру или клерку банка, а непосредственно к компьютеру системы.

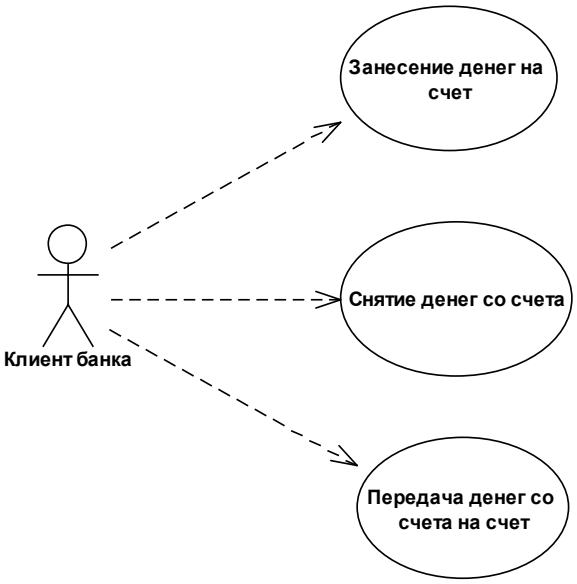


Рис. 5.2. Пример диаграммы сценариев для клиента банка

Отношения между сценариями. Между сценариями можно задавать отношения, которые изображаются на диаграмме сценариев пунктирными стрелками с указанием названий.

Таким образом, продуктом первой стадии инженерии требований (сбора требований) является модель требований, которая состоит из трех частей:

онтология домена;

модель сценариев, называемая диаграммой сценариев;

описание интерфейсов сценариев.

Модель сценариев сопровождается неформальным описанием каждого из них. Нотация такого описания не регламентируется. Как один из вариантов, описание сценария может быть представлено последовательностью элементов:

название, которое помечает сценарий на диаграммах модели требований и служит средством ссылки на сценарий;

аннотация (краткое содержание в неформальном представлении);

актеры, которые могут запускать сценарий;

определение всех аспектов взаимодействия системы с актерами (возможные действия актера и их возможные последствия), а также запрещенных действий актера;

предусловия, которые определяют начальное состояние на момент запуска сценария, необходимое для успешного выполнения;

функции, которые реализуются при выполнении сценария и определяют порядок, содержание и альтернативу действий, выполняемых в сценарии;

исключительные или нестандартные ситуации, которые могут появиться при выполнении сценария и потребовать специальных действий для их устранения (например, ошибка в действиях актера, которую способна распознать система);

реакции на предвиденные нестандартные ситуации;

условия завершения сценария;

постусловия, которые определяют конечное состояние сценария при его завершении.

На дальнейших стадиях сценарий трансформируется в сценарий поведения системы, т. е. к приведенным элементам модели добавляются элементы, связанные с конструированием решения проблемы и нефункциональными требованиями:

механизмы запуска сценария (позиции меню);

механизмы ввода данных;

поведение при возникновении чрезвычайных ситуаций.

Следующим шагом процесса проектирования является преобразование сценария в описание компонентов системы и проверка их правильности.

Описание интерфейсов делается неформально, они определяют взгляд пользователя на систему, с которым необходимо согласовать требования, собранные на этапе анализа системы и определения требований. Все вопросы взаимодействия отмечаются на панели экрана для каждого из актеров. Для этого предусмотрены соответствующие элементы интерфейса (меню, окна ввода, кнопки-индикаторы и т. п.).

Построение прототипа системы, моделирующего действия актера, помогает отработать детали и устранить недоразумения между заказчиком и разработчиком.

5.6. Модель анализа требований. Определение объектов

Модель требований дает обобщенное представление о будущих услугах системы для ее клиентов (актеров). Полученное представление является предметом анализа с целью дальнейшего структурирования проблемы. Основу составляет объектная архитектура, результатом структурирования которой должна быть совокупность объектов, полученная путем последовательной декомпозиции каждого из сценариев на объекты с действиями сценария, а также их взаимодействие, определяемое функциональностью системы.

Таким образом, стратегия выбора объектов в системе базируется на следующих принципах:

изменение требований неизбежно;

объект модифицируется вследствие изменения соответствующих требований к системе;

объект должен быть устойчивым к модификации системы (локальные изменения отдельного требования должны повлечь за собой изменения как можно меньшего количества объектов).

Исходя из этих принципов, в данном методе различаются типы объектов в зависимости от их способности к изменениям, система структурируется согласно следующим критериям:

наличие информации для обработки системы (для обеспечения ее внутреннего состояния);

определение поведения системы;

презентация системы (ее интерфейсов с пользователями и другими системами).

Выбор критериев обусловлен экспертными исследованиями динамики изменений действующих систем. Для каждого критерия функциональности системы имеется совокупность объектов, с помощью которых локализуются требования к наиболее стабильным фрагментам.

Рассматривается три типа объектов:

объекты-сущности;

объекты интерфейса;

управляющие объекты.

Объекты-сущности моделируют в системе долгоживущую информацию, хранящуюся после выполнения сценария. Обычно им соответствуют реальные сущности, которые находят свое отображение в БД. Большинство из них может быть выявлено из анализа онтологии проблемной области, но во внимание берутся только те из них, на которые ссылаются в сценариях.

Объекты интерфейса включают в себя функциональности, зависимые непосредственно от окружения системы и определенные в сценарии. С их помощью актеры взаимодействуют со сценариями системы. Такие объекты определяются путем анализа описаний интерфейсов сценариев модели требований и анализа действий актеров по запуску каждого из соответствующих ему сценариев.

Управляющие объекты – это объекты, которые превращают информацию, введенную объектами интерфейса и представленную объектами-сущностями, в информацию, что выводится интерфейсными объектами. Они являются своеобразным «клеем» для соединения объектов, связывая цепочки событий и задавая взаимодействие объектов.

Такое распределение служит целям локализации изменений в системе. При преобразовании модели требований в модель анализа каждый сценарий разбивается на эти три типа объектов. При этом один и тот же объект может присутствовать в нескольких сценариях и важно распознать такие объекты, чтобы унифицировать их функции и определить их как единый объект. Критерий распознавания состоит в том, что, если в различных сценариях имеется ссылка на один и тот же экземпляр объекта, то это один и тот же объект.

После выделения объектов формируется базис архитектуры системы как совокупности взаимодействующих объектов, для каждого из которых можно установить связь с соответствующим сценарием модели требований, а затем провести трассирование требований, идя от модели требований к модели анализа.

Атрибуты объектов представлены прямоугольниками, объединенными прямой линией с символом объекта, при этом на линии указывается название атрибута, а в прямоугольнике – его тип.

Между объектами определяются ассоциации, которые изображаются одно- или двунаправленной стрелкой, на которой указываются названия ассоциаций типа взаимодействует; составляется с; выполняет роль; наследует; расширяет; использует.

Эти ассоциации существенно отличаются от ассоциаций в моделях данных. Последние нацелены преимущественно на осуществление навигации в БД, тогда как ассоциации определяют взаимодействие объектов.

Исходя из известного метода анализа требований И. Джекобсона, на стадии анализа определяются: онтология домена; модель сценариев; неформальное описание сценариев и актеров; описание интерфейсов сценариев и актеров; диаграммы взаимодействия объектов сценариев.

Полученные требования трассируются, после чего проводится реализация функций системы на следующих этапах ЖЦ.

5.7. Трассирование требований

Одной из главных проблем сбора требований является проблема изменения требований. Требования появляются в процессе общения между группой заказчиков и аналитиками системы в виде интервью, обсуждений, которые не приносят желаемого результата. Это объясняется тем, что составляющие элементов требований последовательно изменяются, благодаря чему их содержание и форма постепенно становятся более точными и полными, т.е. соответствуют действительности.

Инструменты трассировки поддерживают развитие и обработку требований с сохранением их описания и внутренних связей между ними. Трассировка помогает проверять особенности системы на спецификациях требований, выявлять источники разнообразных ошибок и управлять изменениями требований. Если требования разрабатывались в объектной ориентации, то объектами трассировки являются классы и суперклассы и поддерживаемые отношения между ними.

Некоторые методы трассировки базируются на формальных спецификациях отношений (фреймы, соглашения сотрудничества и др.), другие ограничиваются описаниями действий, ситуаций, контекста и возможных решений.

Трассировку можно описать исходя из следующего:

требования изменяются во время функционирования системы;

возникновение требований и их расположение зависит от деталей практической ситуации и контекста их возникновения (требования можно изменить, изменяя эти детали);

трассировка требований должна поддерживаться и изменяться на протяжении всего ЖЦ программного продукта (так как изменяются сами требования, необходимо проводить изменение и промежуточных результатов, полученных при анализе, спецификации, кодировке и т. д.);

для удобства трассировки предпочтительнее использовать иерархическую структуру связей между требованиями, основу которой составляет информация об атрибутах требований.

Чтобы принять решение о возможных модификациях, необходимо иметь достаточно информации о частях и связях между ними. Более того, различные аспекты требований могут быть по-разному представлены и

изменены их контексты путем персонального вмешательства аналитиков или заказчика.

Механизмы трассировки предполагают, что необходимо учитывать следующее:

вместо простых связей рекомендуется вводить более сложные отношения (например, транзитивное отношение для выделения цепочек связей) или специфические отношения;

использовать сложные и гибкие пути трассировки;

поддерживать базы данных объектов трассировки и отношений между ними.

Желательно наличие механизмов поддержки возможностей объектно-ориентированного программирования, операций над классами, а также механизмов унификации функций и отношений (1:1, 1:N, N:M), уничтожение и изменение связей, установка стандартных связей.

Трассировка может быть выборочной для определенных объектов, беспорядочной проверкой объектов, связанных с другими объектами, а также с возможными переходами от одного объекта к другим. Она проводится с использованием механизмов поддержания контекста и отношений, соответствующих данной ситуации (например, трассировка с регулярными выражениями, когда контекст может быть изменен только при модификации соответствующего регулярного выражения).

5.8. Способ описания функциональных требований к системе

В данной главе представлен способ описания функциональных требований к АС и ее функций с использованием стандартов в области информационных технологий, современных методологий создания АС и диаграмм «Use Case Diagram» и «Actvity Diagram» универсального языка моделирования UML 2.0. Рассмотрим процесс определения требований согласно действующим отечественным стандартам.

При использовании стандартов создания АС на стадии «Техническое задание» (ТЗ) в документе фиксируются функциональные и нефункциональные требования к АС. Схема функциональной структуры АС разрабатывается на стадиях «Эскизное проектирование» и «Техническое проектирование», описание автоматизируемых функций АС производится на этапе «Техническое проектирование».

При разработке АС должны быть отслежены связи между функциональными требованиями к системе и функциями системы, их реализующими. Функции системы в свою очередь должны быть детально описаны.

Стадии работ по созданию АС и документы, формируемыми на стадиях, связанных с описанием требований и функций, представлены в табл. 5.1.

Создание АС на стадиях 3–5, подразумевает подготовку:

технического задания;

предварительной схемы функциональной структуры системы (эскизное проектирование);

окончательной схемы функциональной структуры (техническое проектирование);

описания автоматизируемых функций системы.

Таблица 5.1. Стадии создания АС и документы, связанные с требованиями к АС и функциями, их реализующими

№ стадии по ГОСТ 34.601-90	Наименование стадии по ГОСТ 34.601-90	Документы, модели, создаваемые на стадиях по ГОСТ 34.602-89, ГОСТ 34.201-89	ГОСТ, в котором описан документ
3	Техническое задание	Техническое задание (ТЗ)	ГОСТ 34.602
4	Эскизное проектирование	Схема функциональной структуры	РД 50-34.698-90 п. 2.3.
5	Техническое проектирование	Схема функциональной структуры	
		Описание автоматизируемых функций	РД 50-34.698-90 п. 2.5

ТЗ на АС – это документ, оформленный в установленном порядке и определяющий цели создания АС, требования к АС и основные исходные данные, необходимые для ее разработки, а также план-график создания АС.

Связи между требованием и функциями

Представлена модель, отображающая связи между требованиями и функциями (рис. 5.3), их реализующими, и если требуется, описание связей.

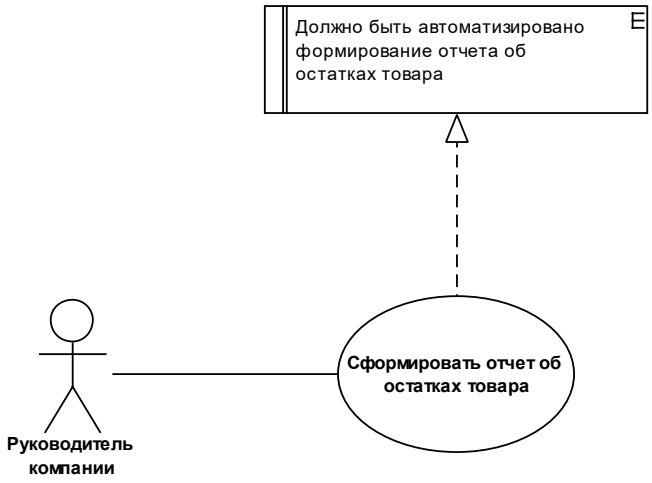


Рис. 5.3. Модель «Требование и функции»

Описание процесса выполнения функции

Представлены модели процесса выполнения функции (рис. 5.4, 5.5 и 5.6) и их описание.

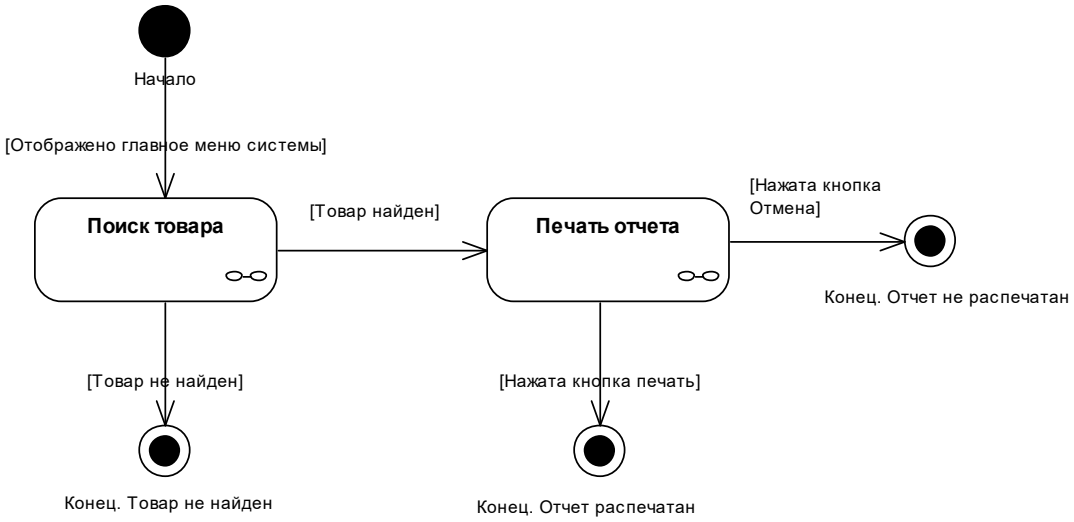


Рис. 5.4. Основные этапы формирования отчета

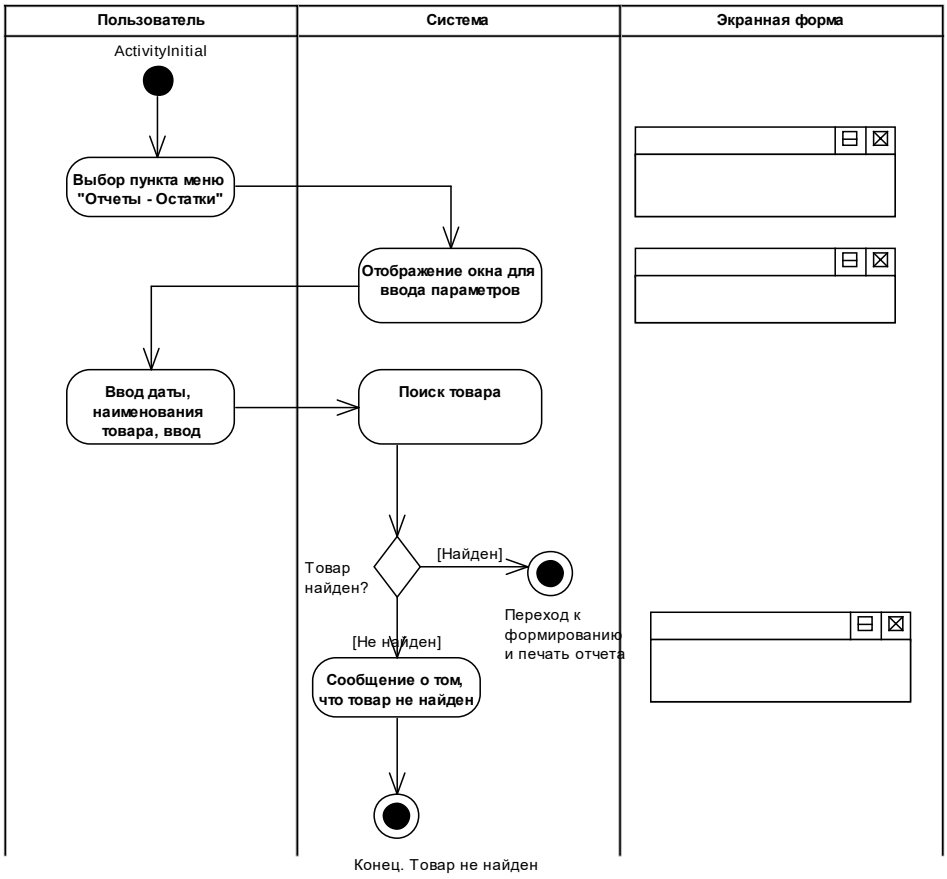


Рис. 5.5. Поиск товара

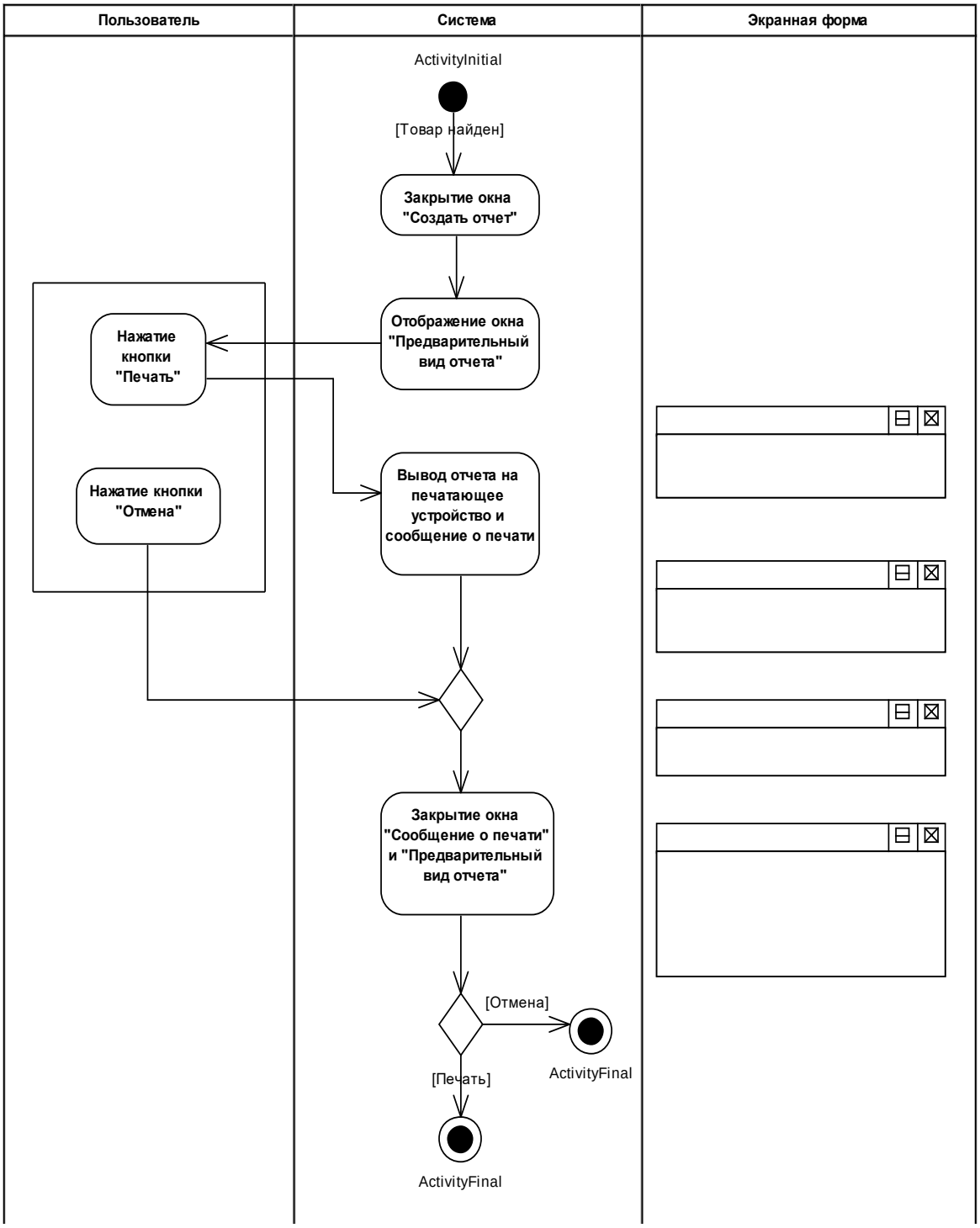


Рис. 5.6. Печать отчета

5.9. Управление требованиями на базе стандартов IBM Rational

Процесс управления требованиями традиционно считается одним из ключевых при создании автоматизированных систем – наибольшие риски проектов связаны с высокой изменчивостью требований и ошибками в их определении. Организация управления требованиями направлена на

снижение таких рисков, причем с помощью методики, основанной на международных и отечественных стандартах.

Проблемы при выполнении проектов создания автоматизированных систем (АС) могут возникать из-за неформального сбора информации, предполагаемой функциональности АС, ошибочных или несогласованных нефункциональных требований к системе, а также нерегламентированной процедуры их изменения.

Организация управления требованиями прежде всего направлена на дезавуирование таких проблем за счет усовершенствования способов сбора, документирования, согласования и модификации требований к системе, отслеживания требований от заинтересованных лиц и из прочих источников, их порождающих.

Регламентация процедур управления требованиями должна обеспечить высокое качество работы с ними в проектах, связанных с проектированием, сопровождением, созданием АС, разработкой их организационно-методического обеспечения, а также внедрения готовых систем за счет уменьшения типичных ошибок при работе с требованиями. Существенный момент предлагаемой методики – использование международных и отечественных стандартов в области управления жизненным циклом (ЖЦ) АС, позволяющее практически без адаптации применять методики в рамках уже существующих, хорошо зарекомендовавших себя на практике процессах разработки. В качестве инструментального средства, поддерживающего моделирование требований на основе UML 2.0., используется продукт *Enterprise Architect* компании *Sparx System.*

Подготовка к управлению требованиями

Под управлением требованиями обычно понимается систематический подход к выявлению, документированию, планированию реализации требований и отслеживанию их изменений. Методика призвана регламентировать следующие процедуры работы с требованиями: выявление, документирование, верификация и утверждение требований, планирование реализации и отслеживание изменений требований.

Методика предусматривает две стадии: подготовку управления требованиями и непосредственно управление ими. Процесс подготовки управления требованиями представлен на рис. 5.7, а его описание – в табл. 5.2.

Таблица 5.2. Описание процесса подготовки управления требованиями

№	Входная/Выходная информация	Шаг процесса	Участник	Бизнес-правила
01	Информация от заинтересованных лиц (заказчики, менеджер	Определение этапов ЖЦ системы, на	Специалист по управлению требованиями	ГОСТ 34.601-90, ГОСТ 19.102-77, ГОСТ Р

	проекта)/Этапы жизненного цикла системы	котором будет организована работа с требованиями		ИСО/МЭК 12207-99, RUP [3]
02	Этапы ЖЦ/Шаблоны документов	Подготовка шаблонов документов для каждого этапа	Специалист по управлению требованиями	ГОСТ 34.602-89, ГОСТ 19.201-78, РД 50-34.698-90, RUP [3]
03	Шаблоны документов/Коды типов требований	Кодирование типов требований в документах	Специалист по управлению требованиями	ГОСТ 34.602-89, ГОСТ 19.201-78, РД 50-34.698-90, RUP [3]
04	Коды типов требований/Атрибуты типов требований	Определение атрии-бутов типов требований	Специалист по управлению требованиями	Типовые атрибуты согласно RUP [3]
05	Коды типов требований/Зависимости между типами требований	Задание зависимостей между типами требований	Специалист по управлению требованиями	Выполняется для каждого конкретного проекта. Шаг процесса может быть пропущен
06	Зависимости между типами требований, типы требований с атрибутами, шаблоны документов/ План управления требованиями	Разработка плана управления требованиями	Специалист по управлению требованиями	ГОСТ 34.601-90, ГОСТ 19.102-77, ГОСТ 2.503-90, RUP [3]
07	Информация от заинтересованных лиц по процедурам работы с требованиями/Процедура выявления требований, анализа, верификации и утверждения	Разработка процедур выявления, верификации, утверждения, изменения требований	Специалист по управлению требованиями	ГОСТ 34.602-89, ГОСТ 19.201-78, РД 50-34.698-90, ГОСТ 2.503-90, RUP [3]
08	ГОСТ 34.602-89, ГОСТ 19.201-78/Шаблон модели и его описание	Создание шаблона модели требований и его описание	Специалист по управлению требованиями	ГОСТ 34.602-89, ГОСТ 19. 201-78

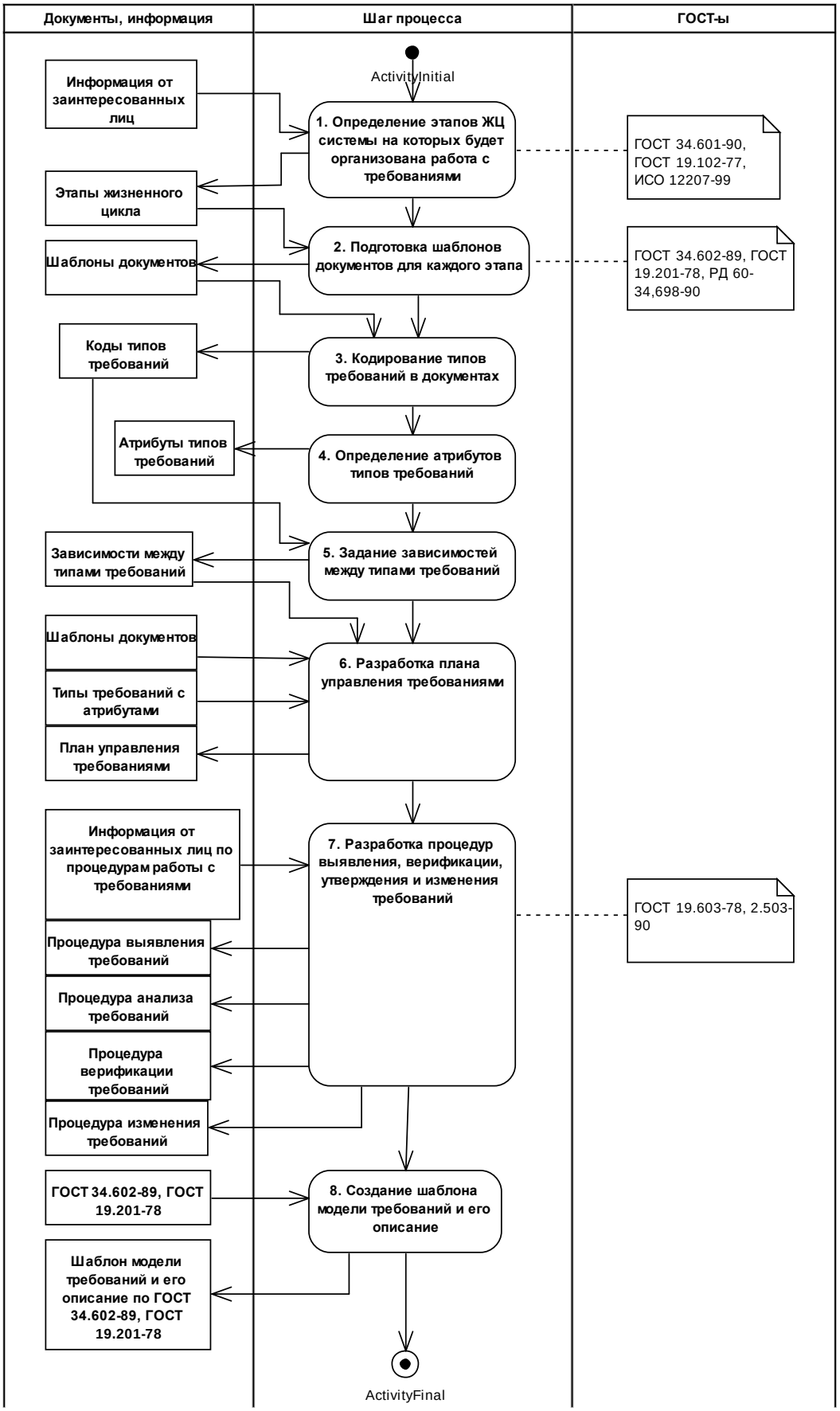


Рис. 5.7. Процесс подготовки управления требованиями

Определение этапов ЖЦ системы, на которых будет организована работа с требованиями, зависит от выбора его модели. Далее в качестве примера будем ориентироваться на определение согласно ГОСТ 34.601-90 «Автоматизированные системы. Стадии создания», в котором работа с требованиями в рамках процедур, регламентируемых методикой, осуществляется на следующих стадиях: формирование требований к АС, разработка концепции, техническое задание и сопровождение АС.

Подготовка шаблонов документов выполняется для каждого этапа ЖЦ. Состав документов, оформляемых для выделенных этапов ЖЦ системы, приведен в табл. 5.3. Часть документов, создаваемых и сопровождаемых при управлении требованиями, определена в соответствии с ГОСТ, а часть готовится по специальным шаблонам, которые разработаны непосредственно авторами методики и созданы на основе обобщения работ. В документах, формируемых с использованием подготовленных шаблонов, будут фиксироваться различные типы требований.

Далее выполняется кодирование типов требований в документах. Назначенные коды типов требований используются при определении их атрибутов и при задании зависимостей между ними.

Для каждого типа требований должен быть разработан атрибутный состав, позволяющий данное требование описать, а также дать возможность проектной группе оценить его с точки зрения ключевых аспектов разработки таких как объем работ, стабильность требования, влияние на архитектуру, приоритетность. Для большинства требований можно выделить общий блок атрибутов. При необходимости дополнительные атрибуты могут быть назначены индивидуально для каждого типа требований.

Различные типы требований могут оказывать взаимное влияние друг на друга, что должно быть учтено при реализации требований зависимых типов. Зависимости между типами требований могут носить различный характер. Например, требования по производительности могут конфликтовать с требованиями к пользовательскому интерфейсу в случае применения Web-интерфейса и его перегруженности графическими объектами. Установка взаимного влияния типов требований является задачей каждого конкретного проекта. При невозможности определить зависимости на начальных этапах разработки данный шаг процесса может быть пропущен.

Итогом работ, выполненных на предыдущих стадиях процесса подготовки управления требованиями, является разработка плана.

Следующим шагом процесса является разработка основных процедур управления требованиями: выявление, верификация и изменение требований. При этом учитывается информация, поступившая от заинтересованных лиц по данным процедурам, и требования ГОСТ 34.602-89, ГОСТ 19. 201-78.

В заключение создается шаблон модели требований и его описание. В качестве опорных документов используются ГОСТ 34.602-89, ГОСТ 19. 201-78.

Подготовительный этап является крайне важным, позволяя регламентировать организационные процедуры, определять шаблоны основных документов и роли основных участников процесса, а также планировать процесс управления требованиями. Типовой состав участников проектной группы в рамках процесса управления требованиями выглядит следующим образом: системный аналитик; специалист по требованиям; рецензент требований; архитектор. Кроме того, в процессе задействован эксперт предметной области. В большинстве проектов роль эксперта, как правило, играет представитель заказчика, но в ряде случаев эксперт предметной области входит в состав проектной группы со стороны разработчика.

Часто для серии типовых проектов могут применяться сходные процедуры, шаблоны, роли, однако игнорирование подготовительного этапа в «нетиповом» проекте может привести к использованию неадекватных процедур управления требованиями, что часто сказывается на качестве конечного продукта либо порождает другие проблемы в проекте, например, отсутствие или неактуальность проектной документации. Рассмотрим пример. Для типовых проектов, ограниченных по срокам реализации (обычно до 6 мес.), наличие подробной документации не обязательно. В случае краткосрочного проекта, не относящегося к типовому (например, ввиду его значимости и перспектив развития), недостаточная документированность изменений требований недопустима, что должно найти свое отражение в соответствующих процедурах процесса управления требованиями.

Если ориентироваться на ГОСТ 34.601-90, то можно использовать методику управления требованиями на определенных этапах ЖЦ (табл. 5.3). В таблице также перечислены документы, рекомендуемые для фиксирования требований. Часть документов, создаваемых и сопровождаемых при управлении требованиями, определена в соответствии с ГОСТ, а часть готовится по специальным шаблонам, ориентированным на применение для широкого класса проектов.

Таблица 5.3. Стадии и этапы создания АС и оформляемые документы

№ стадии создания АС	Стадии создания АС	Этапы создания АС	Документы
01	Формирование требований к АС	Обследование объекта и обоснование необходимости создания АС. Формирование требований пользователя к АС	Интервью заказчиков и пользователей о проблемах
02	Разработка концепции	Изучение объекта. Проведение необходимых научно-исследовательских работ. Разработка вариантов концепции АС, удовлетворяющих требованиям пользователя. Оформление отчета о выполненной работе	Концепция
03	Техническое задание	Разработка и утверждение технического задания на создание АС	ТЗ по ГОСТ 34.602-89 Описание автоматизируемых функций РД 50-34.698-90 п. 2.5 Словарь терминов Модель требований Описание модели требо-ваний
…	–	–	–
08	Сопровождение АС	Выполнение работ в соответствии с гарантийными обязательствами. Послегарантийное обслуживание	Интервью заказчиков и пользователей о проблемах ТЗ по ГОСТ 34.602 -89 Запрос на изменение требований Описание автоматизируемых функций РД 50-34.698-90 п. 2.5 Словарь терминов Модель требований Описание модели требований

Процесс управления требованиями

На основании регламентных процедур реализуется процесс управления требованиями (рис. 5.8. табл. 5.4.).

Таблица 5.4. Описание процесса управления требованиями

№	Входная/Выходная информация	Шаг процесса	Участник	Бизнес-правила
01	Методы выявления требований, результаты обследования объекта автоматизации, план управления требованиями/Список выявленных требований	Выявление требований и их структурирование	Системный аналитик	Процедуры выявления требований
02	Список выявленных требований/Модель требований и ее описание	Моделирование требований	Системный аналитик	Шаблон модели требований и его описание
03	Список выявленных требований/Документы с требованиями	Документирование требований	Системный аналитик, Специалист по требованиям	Шаблоны документов
04	Документы с требованиями, модель требований и ее описание/Документы и модели с требованиями верифицированные и утвержденные	Верификация и утверждение требований	Эксперт предметной области, Рецензент	Процедура верификации и утверждения требований
05	Документы и модели с требованиями верифицированные и утвержденные/Базовая версия требований	Определение базовой версии требований	Архитектор	Процедура верификации и утверждения требований (раздел, посвященный определению базовой версии требований)
06	Базовая версия требований/ Измененные требования	Изменение требований	Все заинтересованные лица	Процедура управления изменениями

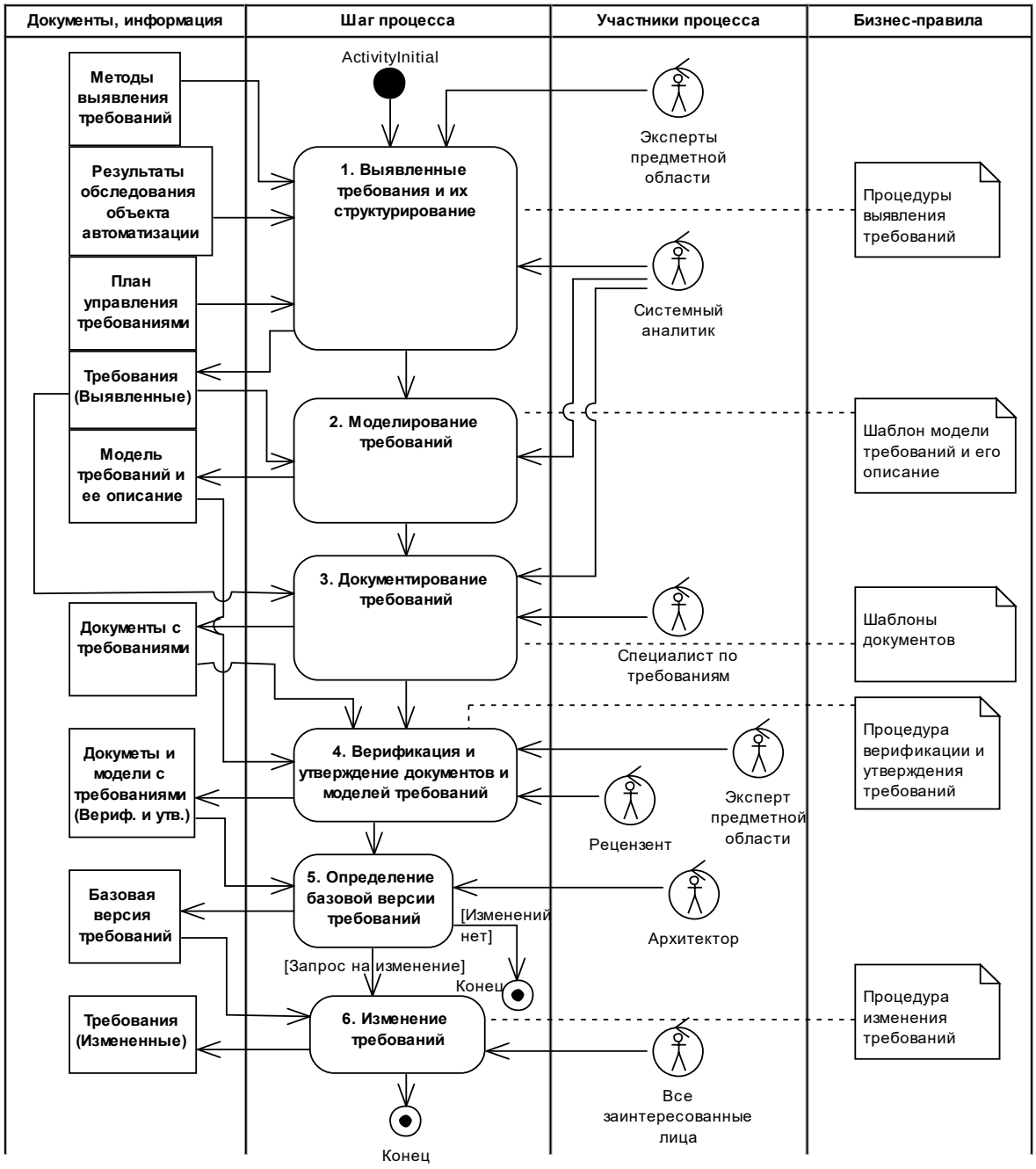


Рис. 5.8. Процесс управления требованиями

Пример управления требованиями: выявление

Выбор конкретного метода выявления требований (бизнес-моделирование, интервьюирование, создание прототипов) зависит от типа проекта. Выявление требований проводится системным аналитиком с привлечением экспертов по предметной области. Результатом этой деятельности являются требования, записанные в согласованном формате и структурированные в соответствии со своими типами, как представлено в плане управления требованиями. Например, выявление функциональных

требований на основе описания бизнес-процессов проводится следующим образом. Каждому бизнес-процессу ставится в соответствие подсистема в разрабатываемой системе, каждому шагу бизнес-процесса – функциональное требование. Состав бизнес-процессов представлен на рис. 5.9, а описание конкретного процесса «Зачисление студентов в университет» – на рис. 5.10.

Как видно из рис. 5.9, автоматизируемыми процессами являются: зачисление, перевод, отчисление, проведение сессии, подготовка отчетов. Шагами бизнес-процесса зачисления, подлежащими автоматизации являются: формирование списков групп, заполнение личной карточки студента, регистрация выдачи зачетной книжки в журнале.

На основе состава и шагов бизнес-процесса, подлежащих автоматизации, строится матрица трассировки (табл. 5.5, 5.6), которая позволяет проследить связи бизнес-процессов с реализующими их подсистемами и конкретных шагов бизнес-процессов с функциональными требованиями, а также контролировать полноту и целостность реализации: каждому автоматизируемому бизнес-процессу должна быть поставлена в соответствие подсистема (подсистемы), а подсистема, соответственно, реализовывать какой-либо процесс.

Таблица 5.5. Зависимость подсистем от бизнес-процессов

№	Бизнес-процесс	Подсистема
1	Зачисление студента	Зачисление студента
2	Перевод студента	Перевод студента
3	Отчисление студента	Отчисление студента
4	Подготовка отчётов	Подготовка отчётов
5	Проведение сессии	Проведение сессии

Таблица 5.6. Зависимость функциональных требований подсистемы «Зачисление студента» от шагов бизнес-процесса «Зачисление студента»

Шаг бизнес-процесса	Функциональное требование
Формирование списков групп	Формирование списков групп
Заполнение личной карточки студента	Ведение личных карточек студентов
Регистрация выдачи зачётной книжки	Ведение журнала регистрации выдачи зачетных книжек

На основе данных из табл. 5.5, 5.6 создается модель подсистем и функциональных требований, как представлено на рис. 5.9, 5.10.

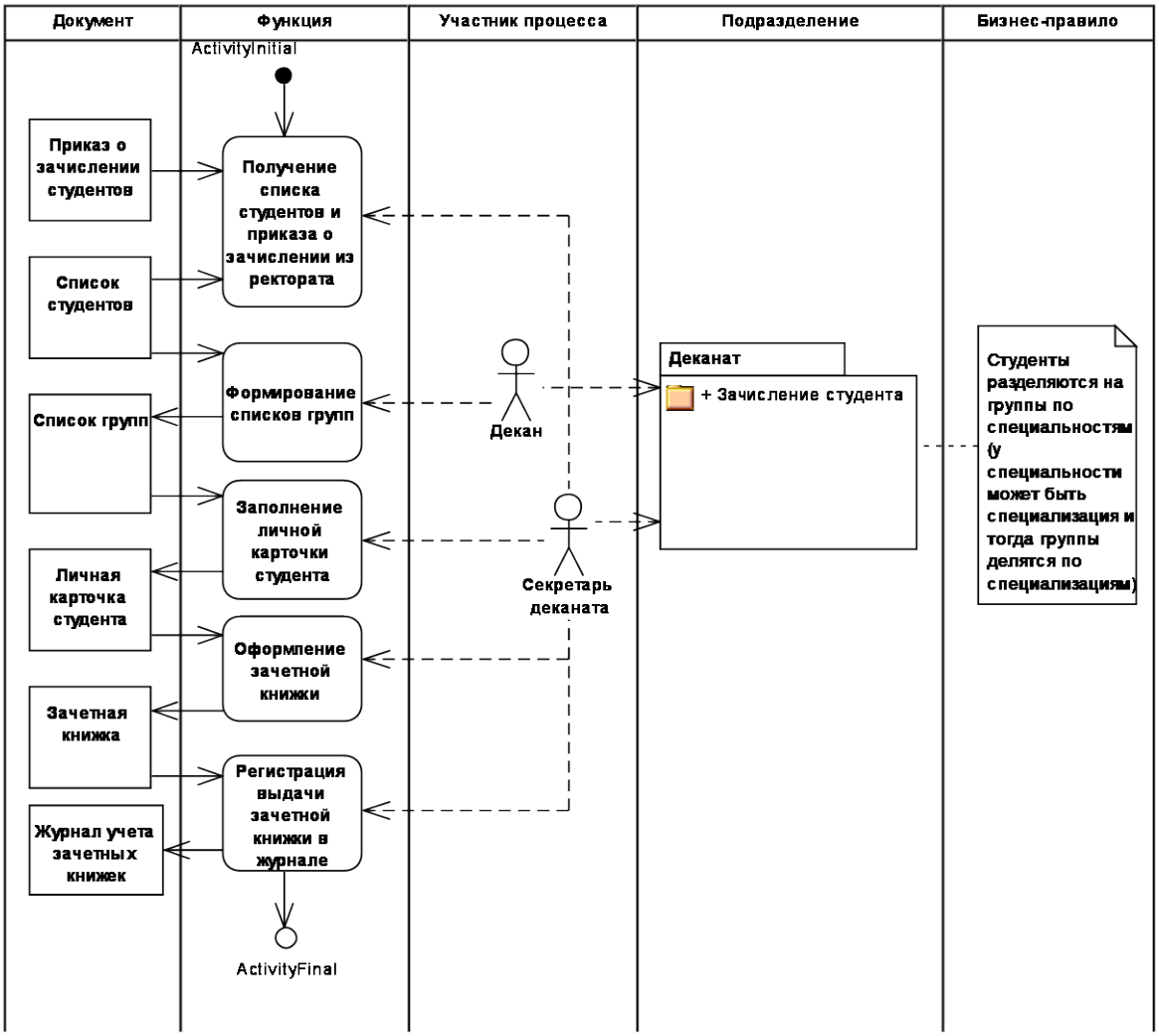


Рис. 5.9. Пример управления требованиями: выявление состава бизнес-процессов и описание бизнес-процесса «Зачисление студентов в университет»

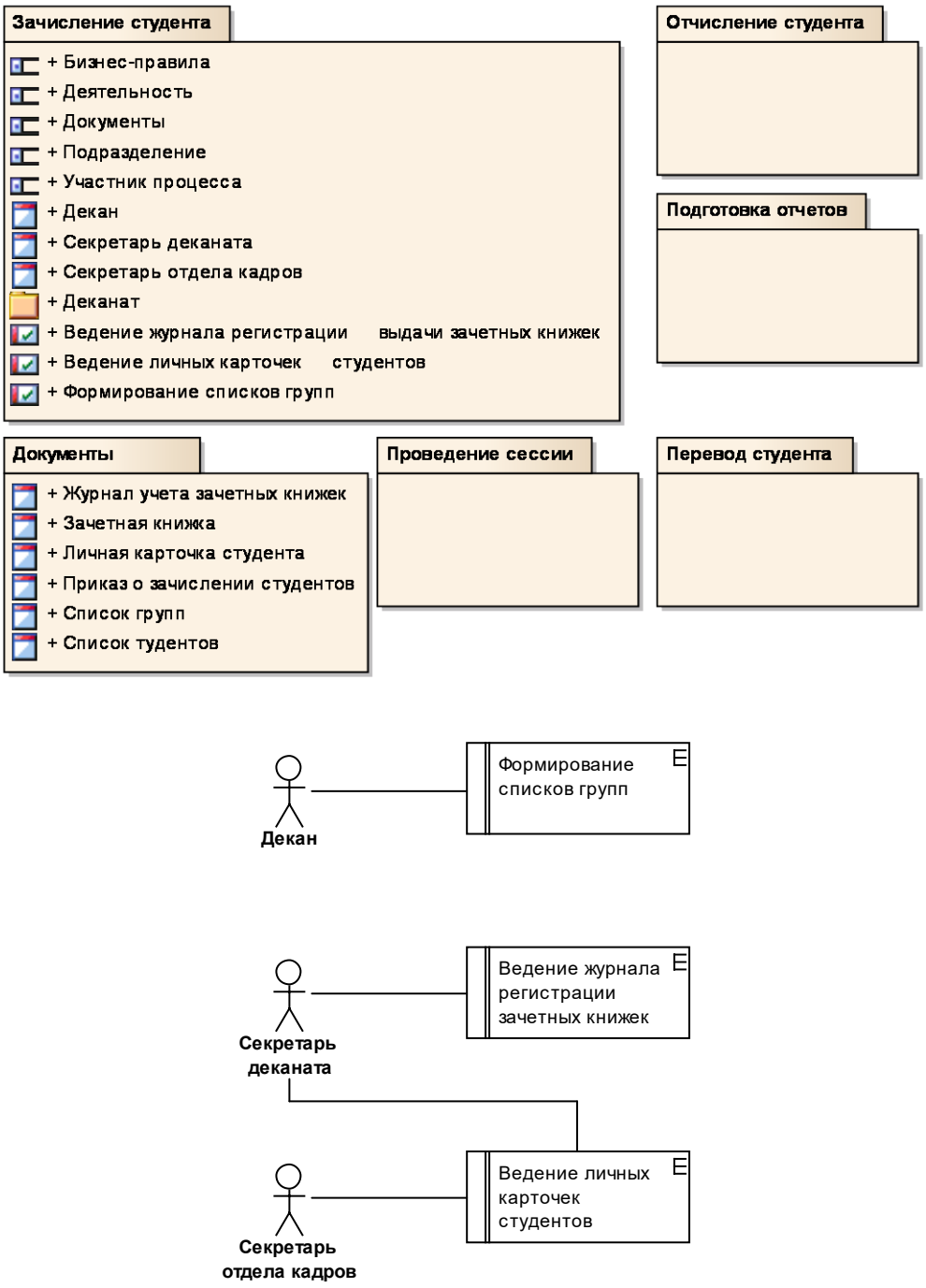


Рис. 5.10. Пример управления требованиями: выявление состава подсистем и состава функциональных требований подсистемы «Зачисление студентов»

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Шеер, Август-Вильгельм. ARIS-моделирование бизнес-процессов / Август-Вильгельм Шеер. М.: Вильямс, 2000. 175 с.

2. Биберштейн, Н. Компас в мире сервис-ориентированной архитектуры (SOA): ценность для бизнеса, планирование и план развития предприятия / Н. Биберштейн [и др.]. М.: КУДИЦ-Пресс, 2007. 256 с.

3. Брауде, Э. Технология разработки программного обеспечения / Э. Брауде; пер. с англ. Спб.: Питер. 2004.

4. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч. 2-е изд. М.: Бином, СПб.: Невский диалект 1998. 560 с.

5. Буч, Г. Язык UML. Руководство пользователя / Г. Буч, Д. Рамбо, А. Джекобсон. М.: ДМК, 2000. 432 с.

6. Вендров, А.М. Проектирование программного обеспечения экономических информационных систем / А.М. Вендров. М.: Финансы и статистика, 2000. 352 с.

7. Верников, Г. Основы методологии IDEF1, IDEF1X, IDEF3, IDEF5 [Электронный ресурс] / Г. Верников. Режим доступа: http://www.citforum.ru/cfin/vernikov.

8. Верников, Г. Основные методологии обследования организации. Стандарт IDEF0 [Электронный ресурс] / Г. Верников. Режим доступа: http:// www.consulting.ru/main/mgmt/texts/m7/079_idef.shtml.

9. Вигерс, К. Разработка требований к программному обеспечению / К. Вигерс; пер. с англ. М.: Издательско-торговый дом «Русская Редакция», 2004. 576 с.: ил.

10. Гецци, К. Основы инженерии программного обеспечения / К.Гецци, М. Джазайери, Д. Мандриоли; пер. с англ. СПб. БХВ-Петербург. 2005.

11. Дин, Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход / Д. Дин, У. Дон. М.: Вильямс. 2002.

12. Калянов, Г. CASE: все только начинается... [Электронный ресурс]. Режим доступа: http://www.osp.ru/cio/2001/03/016.htm.

13. Кватрани, Т. Rational Rose 2000 и UML. Визуальное моделирование / Т. Кватрани. М.: ДМК Пресс, 2001. 176 с.

14. Коберн, А. Современные методы описания функциональных требований к системам / А. Коберн. М.: ЛОРИ, 2002.

15. Кознов, Д.В. Визуальное моделирование: теория и практика [Электронный ресурс] / Д.В. Кознов. Режим доступа: http://www.intuit.ru/department/se/vismodtp/3/.

16. Кознов Д.В. Языки визуального моделирования. Проектирование и визуализация программного обеспечения: учеб. пособие / Д.В.Кознов. СПб.: Изд-во С.-Петерб. ун-та, 2004. 171 с.

17. Корпорация: языки управления бизнес-процессами. BPML [Электронный ресурс]. Режим доступа: http://citforum.ru/internet/xml/bpml/.

18. Лаврищева, Е.М. Методы и средства инженерии программного обеспечения / Е.М. Лаврищева, В.А. Петрухин. М., 2006 г.

19. Ларман, Крег. Применение UML и шаблонов проектирования / Крег Ларман. М.: Вильямс, 2001. 496 с.

20. Леоненков, А.В. Объектно-ориентированный анализ и проектирование с использованием UML [Электронный ресурс] / А.В.Леоненков. Режим доступа: www.intuit.ru.

21. Леоненков, А.В. Самоучитель UML 2 / А.В. Леоненков. СПб.: БХВ-Петербург, 2007. 576 с.

22. Маклаков, С.В. Создание информационных систем с AllFusion Modeling Suite / С.В. Маклаков. М.: Диалог-МИФИ, 2007. 400 с.

23. Мацяшек, Л.А. Анализ требований и проектирование систем: разработка информационных систем с использованием UML / Л.А. Мацяшек. М.: Вильямс, 2002. 432 с.

24. Машков, Д.А. Моделирование бизнес-процессов при проектировании КИС [Электронный ресурс] / Д.А. Машков. Режим доступа: http:// www.soft.implozia.ru/news/seminar02_04.html.

25. Маторин, С.И. Моделирование организационных систем в свете нового подхода «Узел-Функция-Объект» / С.И. Маторин, А.С. Попов, В.С.Маторин // Научно-техническая информация. Сер. 2. 2005. № 1.с. 1–8.

26. Маторин, С.И. Теория систем и системный анализ: учебное пособие / С.И. Маторин., О.А. Зимовец. Белгород: Изд-во НИУ «БелГУ», 2012.

27. Михеев, А. Война стандартов в мире workfiow [Электронный ресурс] / А. Михеев, М. Орлов // PC Week/RE. 2004. № 28. Режим доступа: http://wf.runa.ru/rus/images/c/ce/Stat_ya2.pdf.

28. Михеев, А. Перспективы WorkFlow систем. Сравнение WorkFlow языков [Электронный ресурс] / А. Михеев, М. Орлов // PC Week/RE. 2005. № 36. Режим доступа: http://wf.runa.ru/rus/images/c/c1/Stat_ya4.pdf.

29. Месарович, М. Общая теория систем: математические основы / М. Месарович, Д. Михайло, Я. Такахара. М.: Мир, 1978. 311 с.

30. Репин, В.В. Процессный подход к управлению. Моделирование бизнес-процессов / В.В. Репин, В.Г. Елиферов. М.: РИА «Стандарты и качество», 2004. 408 с.

31. Сапегин, А. Реорганизация бизнес-процессов: как выглядит наше будущее? [Электронный ресурс] / А. Сапегин. Режим доступа: http://www.interfase.ru.

32. Сомерсвиль, И. Инженерия программного обеспечения / И.Сомерсвиль; пер. с англ. 6-е изд. М.: Вильямс, 2002. 624 с.

33. Спольски, Д. Лучшие примеры разработки ПО / Д. Спольски. СПб.: Питер, 2007. 208 с.

34. Туманов, В.Е. Проектирование хранилищ данных для приложений систем деловой осведомленности (Business Intelligence Systems) [Электронный ресурс] / В.Е. Туманов. Режим доступа: http://www.intuit.ru/department/database/bispowerd/14/4.html.

35. Деревянко, А.С. Технологии и средства консолидации информации: учеб. пособие / А.С. Деревянко, М.Н. Солощук. Харьков: НТУ «ХПИ», 2008. 432 с.

36. Орлов, С. Технологии разработки программного обеспечения: учебник / С. Орлов. СПб.: Питер, 2002. 464 с.: ил.

37. Технология моделирования бизнес-процессов [Электронный ресурс] / НИЦ CALS-технологий «Прикладная логистика». Режим доступа: http:// http://www.cals.ru.

38. Фаулер, М. Архитектура корпоративных программных приложений / М. Фаулер. М.: Вильямс, 2004. 544 с.

39. Фаулер, М. UML в кратком изложении. Применение стандартного языка объектного моделирования / М. Фаулер, К. Скот. М.: Мир, 1999. 191 с.

40. Халл, Э. Разработка и управление требованиями (Практическое руководство пользователя) / Э.Халл. М., 2005.

41 Якобсон, А. Унифицированный процесс разработки программного обеспечения / А. Якобсон, Г. Буч, Дж. Рамбо. СПб.: Питер, 2002. 496 с: ил.

59. IDEF [Электронный ресурс] / Режим доступа: http:// www.idef.com.