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

Кафедра информатики

Курс лекций по предмету

«ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»

Для студентов специальности**: 1-40 81 01**
«Информатика и технологии разработки программного обеспечения»
всех форм обучения
факультет компьютерных систем и сетей,
кафедра информатики

к.ф.-м.н., доцент
Пилецкий И.И.

Минск, 2015

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ В ТЕХНОЛОГИЮ РАЗРАБОТКИ ПРОМЫШЛЕННОГО ПО 7

1.1. Основные понятия 7
1.2. Жизненный цикл разработки программного обеспечения 9
1.3. Модели жизненного цикла разработки программного обеспечения 10
1.3.1 Каскадная модель жизненного цикла разработки ПО 11
1.3.2 Итеративная и инкрементальная модель – эволюционный подход 13
1.3.4 Спиральная модель жизненного цикла разработки ПО 15
1.3.5 Основы методологии Rational Unified Process 18
1.3.6 ЖЦ ПО по методологии Rational Unified Process 21
1.3.7 Обобщенная модель ЖЦ ПО 23
1.4 Жизненный цикл тестирования программного обеспечения и его роль в жизненном цикле разработки 25
1.4.1 Современные модели качества программного обеспечения 25
1.4.1.1 Основные принципы стандарта CMM 26
1.4.1.2 Основные принципы стандарта SPICE 28
1.4.1.3 Концепция качества продукта в RUP. Критерии качества 29

2. МЕТОДОЛОГИИ И ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ИС 30

2.1. Общие требования к методологии и технологии 30
2.2. Структура комплекта документов 33
2.3. Наиболее перспективные и приемлемые технологии разработки ПО (обновить раздел) 36
2.3.1. Технологии, базирующиеся на CASE–средствах Computer Associates 37
2.3.2. Технологии, базирующиеся на CASE–средствах IBM Rational 41
2.3.2.1. Краткая характеристика основных технологических программных продуктов IBM Rational 42

3. МЕТОДОЛОГИЯ ФУНКЦИОНАЛЬНОГО МОДЕЛИРОВАНИЯ IDEF0 58

3.1. Концепция методологии функционального моделирования IDEF0 59
3.2. Основные определения (понятия) методологии и языка IDEF0 61
3.3. Синтаксис графического языка IDEF0 64
3.4. Семантика языка IDEF0 65
3.5. Имена и метки 68
3.6. Отношения блоков на диаграммах 69
3.7. Диаграммы IDEF0 72
3.8. Дочерняя диаграмма 75
3.9. Родительская диаграмма 76
3.10. Свойства диаграмм 80
3.10.1. Стрелки как ограничения 80
3.10.2. Параллельное функционирование 81
3.10.3. Ветвление и слияние сегментов стрелок 82
3.11. Создание диаграмм IDEF0 в среде AllFusion Process Modeler 84
3.12. Диаграммы DFD 90
3.12.1 Терминология DFD-нотации 94
3.12.2 Построение диаграмм 98
3.12.3 Примеры DFD диаграмм 102
3.13. Пример проектирования функций подсистемы обработки и хранения данных 110

4. IDEF3 – МЕТОДОЛОГИЯ ОПИСАНИЯ И МОДЕЛИРОВАНИЯ ПРОЦЕССОВ 120

4.1. Функциональный элемент 122
4.2. Элемент связи 123
4.2.1. Связи старшинства 123
4.2.2. Содерживаемые связи старшинства 124
4.2.3. Относительные связи 124
4.2.4. Связь поток объектов 124
4.3. Перекресток 125
4.3.1. Типы перекрестков 126
4.3.2. Значения комбинаций перекрестков 126
4.4. Декомпозиция описания процесса 132
4.5. Примеры 134

5. ЯЗЫК МОДЕЛИРОВАНИЯ БАЗ ДАННЫХ IDEF1X 142

5.1. Сущности....142
5.2. Связи и отношения....144
5.2.1. Мощность связей....145
5.3. Ключи....147
5.3.1 Внутренние и внешние ключи....148
5.3.2. Ссылочная целостность....150
5.4. Домены....151
5.5. Представления....155
5.6. Нормализация данных....160
5.7. Примеры построения диаграмм....161
5.8. Общие сведения о среде проектирования AllFusion Erwin Data Modeler....172
5.8.1. Построение логической модели....177
5.8.1.1. Диаграмма сущность – связь....177
5.8.1.2. Модель данных на основе ключа....179
5.8.1.3. Полная атрибутивная модель....182
5.8.2. Создание новой модели....184
5.8.3. Создание физического уровня базы данных на основе логического....185
5.8.4. Редактирование таблиц....185
5.8.5. Редактирование столбцов таблицы....187
5.8.6. Редактирование ключей и индексов таблицы....188
5.8.7. Редактирование связей таблиц....189
5.8.8. Сохранение модели базы данных....190
5.8.9. Генерация операторов для создания базы данных....191
5.8.10. Подготовка исходных данных для разработки новой версии БД....194

6. ЯЗЫК UML, МОДЕЛИ ПО, ОБЪЕКТНО–ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ ПО....197

6.1. Основные элементы языка UML....199
6.1.1. Сущности....199
6.1.2. Отношения....207
6.1.3. Диаграммы....209
6.2. Диаграмма вариантов использования как концептуальное представление бизнес–системы в процессе ее разработки....211
6.2.1. Базовые элементы диаграммы вариантов использования....212
6.2.2. Отношения на диаграмме вариантов использования....215
6.2.2.1. Отношение ассоциации....215
6.2.2.2. Отношение включения....216
6.2.2.3. Отношение расширения....217
6.2.2.4. Отношение обобщения....219
6.2.3. Дополнительные обозначения языка UML для бизнес–моделирования....219
6.2.4. Примеры USE CASE и их реализация....221
6.3. Диаграммы последовательности....225
6.3.1. Сообщения на диаграмме последовательности....228
6.3.2. Ветвление потока управления....230
6.3.3. Пример диаграммы последовательности....232
6.4. Диаграмма кооперации....232
6.4.1. Объекты диаграммы кооперации и их графическое изображение....233
6.4.2. Кооперация объектов....235
6.4.3. Пример совместного использования диаграмм кооперации и последовательности....237
6.5. Сравнение диаграммы последовательности и диаграммы кооперации....238
6.6. Диаграммы состояний....242
6.6.1. Составное состояние и подсостояние....246
6.6.1.1. Последовательные подсостояния....247
6.6.1.2. Параллельные подсостояния....248
6.6.1.3. Несовместимые подсостояния....249
6.6.2. Исторические состояния....250
6.6.3. Сложные переходы и псевдосостояния....251
6.6.4. Состояние синхронизации....253
6.6.5. Рекомендации по построению диаграмм состояний....254
6.6.6. Примеры диаграмм состояний....255
6.7. Диаграммы деятельностей....258
6.7.1. Примеры диаграмм деятельностей....265
6.8. Классы....267

6.8.1. Области видимости и действия, кратность и иерархия классов *270*
6.8.2. Отношения между классами *273*
6.8.2.1. Отношение ассоциации 273
6.8.2.2. Отношение обобщения 278
6.8.2.3. Отношение агрегации 280
6.8.2.4. Отношение композиции 282
6.8.3. Примеры диаграмм классов *283*
6.9. Компоненты 287
6.9.1. Виды компонентов *288*
6.9.2. Отношения между компонентами *289*
6.9.3. Компоненты и классы *291*
6.9.4. Компоненты и интерфейсы *292*
6.9.5. Варианты графического изображения компонентов *294*
6.9.6. Пример диаграммы компонентов *296*
6.10. Диаграмма развертывания 297
6.10.1. Узел диаграммы развертывания *297*
6.10.2. Отношения между узлами диаграммы *301*
6.10.3. Пример диаграммы развертывания *302*

7. СЕРВИС-ОРИЕНТИИРОВАННАЯ АРХИТЕКТУРА (SOA) 303

7.1. Основные проблемы ИТ 303
7.2. Основные факторы внедрения SOA 304
7.3. Определения, цели и принципы SOA 304
7.4. Жизненный цикл сервисов 306
7.4.1. Управление жизненным циклом сервиса *309*
7.4.2. Формирование требований и анализ *310*
7.4.3. Проектирование *310*
7.4.4. Разработка сервиса *311*
7.4.5. Эксплуатация в ИТ-среде *311*
7.4.6. Мониторинг процессов *312*
7.4.7. Эталонная модель сервис-ориентированных архитектур *312*

8. WEB-СЕРВИСЫ 313

8.1. Понятия технологии WEB-Сервисов 315
8.2. Основные технологии Web-сервисов 316
8.3 Использование WEB-Сервисов 319
8.3.1 Пример XML *320*
8.3.2 Описание SOAP *321*
8.3.3 WSDL: Определение Web-сервисов *324*
8.3.3.1 Структура документа WSDL 324
8.3.3.2 Пример WSDL 326
8.3.3.3 Типы Операций 327
8.3.3.4 Синтаксис WSDL 330
8.3.3.5 Примеры WSDL 331
8.3.4 UDDI *339*
8.4 Безопастность WEB-сервисов 341
8.5 WEB-Сервисы и SOA 342
8.6 Свойства WEB-Сервисов 343
8.7 Заключение 344

9. ЯЗЫК МОДЕЛИРОВАНИЯ БИЗНЕС ПРОЦЕССОВ BPMN 346

9.1 Назначение BPMN 346
9.1.1 Частные (Внутренние) бизнес процессы *347*
9.1.2 Публичные (Абстрактные) бизнес процессы *348*
9.1.3 Совместные (хореографии) процессы *349*
9.2 Основные типы схем 351
9.2.1 Хореография бизнес процессов *351*
9.2.3 Точка зрения, бизнес взгляд *353*
9.3 Отображение BPMN на другие языки моделирования 354
9.4 Основные элементы схемы бизнес процесса 355
9.5 Использование текста, цвета, размера и линий на схеме 371
9.6 Правила соединения объектов 372
9.7 Примеры 373

10. ОТ МОДЕЛИРОВАНИЯ НА BPMN К МОДЕЛИРОВАНИЮ НА BPEL И ГЕНЕРАЦИИ КОДА 386

10.1 Моделирование заказа путешествия 386
10.2 Определение BPEL информации.....388
10.3 Старт процеса 392
10.4 Создание параллельного потока.....396
10.5 Нанесение на карту петли.....399
10.6 Синхронизация параллельных потоков.....403
10.7 Конец потока 404
10.8 Обработка ошибок.....406
10.9 Заключение 407

11. ЯЗЫК WS BPEL.....408

11.1 Введение 408
11.2 История BPEL.....408
11.3 Пример практического применения BPEL.....409
11.4 Нотация BPEL.....412
11.5 Взаимоотношения с WSDL 413
11.6 Определение бизнес процесса 413
11.7 Структура бизнес процесса.....423
11.8 Основные конструкции деятельности 426
11.5 Расширяемость языка WS BPEL.....435
11.6 Взаимодействие бизнес парнеров 436

ПРИЛОЖЕНИЕ. ОПИСАНИЕ XML.....440

Аннотированные схемы SQLXML.....449

ЛИТЕРАТУРА.....456

1. Введение в технологию разработки промышленного ПО

1.1. Основные понятия

Трудозатраты, связанные с созданием программного обеспечения (ПО) прямо связаны с ***качеством и сложностью*** создаваемого ПО. Так трудозатраты на создание **программного продукта** в ***три раза больше***, ***чем обычной программы***, а ***трудозатраты*** на создание ***простого приложения*** (редактор текстов), ***среднего уровня сложности*** (трансляторы) и ***высокого уровня сложности*** (операционные системы) **возрастают еще в три раза**.

Поэтому крайне важны используемые технологии для разработки промышленного программного продукта.

Технология ПО включает в себя такие понятия, как *методы, инструменты*, *организационные мероприятия направленные* на *создания промышленного ПО*.

Под **технологией программирования понимают** организованную ***совокупность методов, средств и их программного обеспечения***, ***организационно–административных установлений, направленных на разработку, распространение и сопровождение программной продукции***.

Создание ПО включает:

Существенные задачи – моделирование сложных концептуальных структур объектов реального мира большой сложности с помощью абстрактных программных объектов;

Второстепенные задачи – создание представлений этих абстрактных объектов в программе и описание их поведения.

И *если 90% усилий* разработчиков (затрат) связано *не с существенными задачами*, то сведя все затраты к нулю *не получим роста в порядки* (сравните с электроникой).

Если просмотреть историю, то все усилия были направлены на преодоление второстепенных задач.

Пути преодоления существенной сложности:

Массовый рынок (главное не скорость, а качество и сопровождение);

Лучший способ повысить производительность труда – купить;

Быстрое макетирование для установления технических требований к ПО;

Наращивать программы постепенно, добавляя функциональность (хорошо протестированную);

Хорошая команда.

Трудности создания ПО:

1. **Внутренние**, присущие ему (задание технических требований, проектирование и проверка):

Сложность – размеры, одинаковых нет, они не просто объединяются, масштабирование это непростое увеличение, это добавление новых элементов и

связей, нелинейный рост сложности – сложность присуща (существенна) программным объектам и от нее абстрагироваться нельзя;

Согласованность – между людьми, между интерфейсами, которые появились не понятно, как и когда (10 и более лет);

Изменяемость – постоянно изменяются требования и применение ПО;

Незримость – ПО невидимо, различные графовые модели дают некоторое представление только с определенной точки зрения, но само ПО не плоское и его трудно передать такими моделями.

2. **Сопутствующие**, внутренне ему не присущие.

Движущей силой использования принципов программной инженерии было *опасение крупных аварий*, к которым могла привести (и привела) разработка все более сложных систем неуправляемыми художниками (не производительность в разы, хотя она и возросла от 3 до 5 раз).

Данные МО США по цене исправления *одной ошибки* следующие: обнаруженные и исправленные на *стадии требований – 139$*, на *стадии кодирования – 1000$*, на *стадии тестирования – 7000$,* на *стадии внедрения – 14000$.*

Рис. 1.0. Возможный сценарий разработки ПО

1.2. Жизненный цикл разработки программного обеспечения

Методологии, технологии и инструментальные средства составляют основу проекта любой информационной системы (ИС). Методология реализуется через конкретные технологии и поддерживающие их стандарты, методики и инструментальные средства, которые обеспечивают выполнение процессов ЖЦ.

Под технологией программирования понимают организованную совокупность методов, средств и их программного обеспечения, организационно-административных установлений, *направленных на разработку, распространение и сопровождение программной продукции.*

Одним из базовых понятий методологии проектирования информационных систем (ИС) является понятие жизненного цикла ее программного обеспечения (ЖЦ ПО). ***ЖЦ ПО – это непрерывный процесс****, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.*

Основным нормативным документом, регламентирующим ЖЦ ПО, является международный *стандарт ИСО/МЭК 12207 – 95 «Информационная технология. Процессы жизненного цикла программных средств» или ISO/IEC 12207* (ISO – International Organization of Standardization – Международная организация по стандартизации, IEC – International Electrotechnical Commission – Международная комиссия по электротехнике). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

Структура ЖЦ ПО по стандарту ИСО/МЭК 12207 – 95 базируется на трех группах процессов:

– основные процессы ЖЦ ПО (приобретение, поставка, разработка, эксплуатация, сопровождение);

– вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем);

– организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого ЖЦ, обучение).

Разработка включает в себя все работы по созданию ПО и его компонент в *соответствии с заданными требованиями*, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала и т.д. Разработка ПО включает в себя, как правило, анализ, проектирование и реализацию (программирование).

Эксплуатация включает в себя работы по внедрению компонентов ПО в эксплуатацию, в том числе конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т.д., и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения, модификацию ПО

в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО, обучение персонала и т.п.

Обеспечение качества проекта связано с проблемами верификации, проверки и тестирования ПО.

Верификация – это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разработки с исходными требованиями. Проверка частично совпадает с тестированием, *которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям.* В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла ПО, прежде всего процессы разработки и сопровождения ПО. При создании проектов сложных ИС, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникает проблема учета их связей и функций, создания унифицированной структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации конфигурационного учета, планирования и управления конфигурациями ПО отражены в стандарте ISO 12207–2.

Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. Результатами анализа, в частности, являются функциональные модели, информационные модели и соответствующие им диаграммы. ЖЦ ПО носит итерационный характер: результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних этапах.

1.3. Модели жизненного цикла разработки программного обеспечения

Под моделью ЖЦ понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении ЖЦ. Модель ЖЦ зависит от специфики ИС и специфики условий, в которых последняя создается и функционирует. *Стандарт ISO/IEC 12207 не*

предлагает конкретную модель ЖЦ и методы разработки ПО. Его регламенты являются общими для любых моделей ЖЦ, методологий и технологий разработки. Стандарт описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

К настоящему времени наибольшее распространение получили следующие основные модели ЖЦ:

- *Каскадная (водопадная)* или *последовательная модель* (70-85 г.г.);
- *Итеративная и инкрементальная – эволюционная (гибридная, смешанная) модель* (86-90 г.г.);
- *Спиральная (spiral) модель* или *модель Боэма* (88-90-е г.г.).

Легко обнаружить, что в разное время и в разных источниках приводится разный список моделей и их интерпретация. Например, ранее, инкрементальная модель понималась как построение системы в виде последовательности сборок (релизов), определенной в соответствии с заранее подготовленным планом и заданными (уже сформулированными) и неизменными требованиями. Сегодня об инкрементальном подходе чаще всего говорят в контексте постепенного наращивания функциональности создаваемого продукта.

Может показаться, что индустрия пришла, наконец, к общей “правильной” модели. Однако, каскадная модель, многократно “убитая” и теорией и практикой, продолжает встречаться в реальной жизни. Спиральная модель является ярким представителем эволюционного взгляда, но, в то же время, представляет собой единственную модель, которая уделяет явное внимание анализу и предупреждению рисков. Коротко рассмотрим каждую из моделей жизненного цикла.

1.3.1 Каскадная модель жизненного цикла разработки ПО

В изначально существовавших однородных ИС каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся *каскадный* способ. Его основной характеристикой является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем (рис. 1.1). Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена командой специалистов на следующем этапе.

Положительные стороны применения *каскадного* подхода заключаются в следующем:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

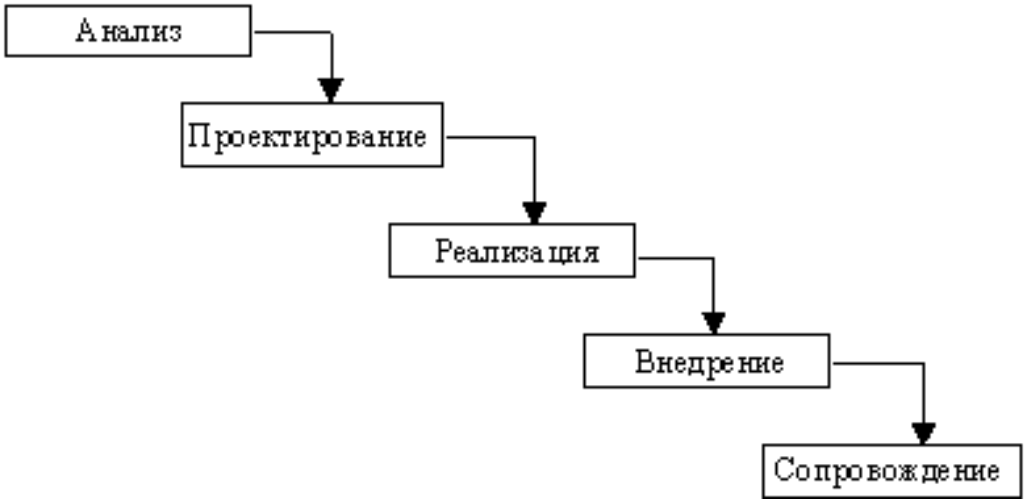


Рис. 1.1. Каскадная схема разработки ПО

Трудозатраты:
Оценка трудозатрат по Бруксу:
1/3 – планирование
1/6 – написание программ
1/4 – тестирование компонент
1/4 – системное тестирование
В настоящее время трудозатраты составляют:
на стадиях разработки – 20%
и на сопровождение – 80%
Примерное распределение трудозатрат:

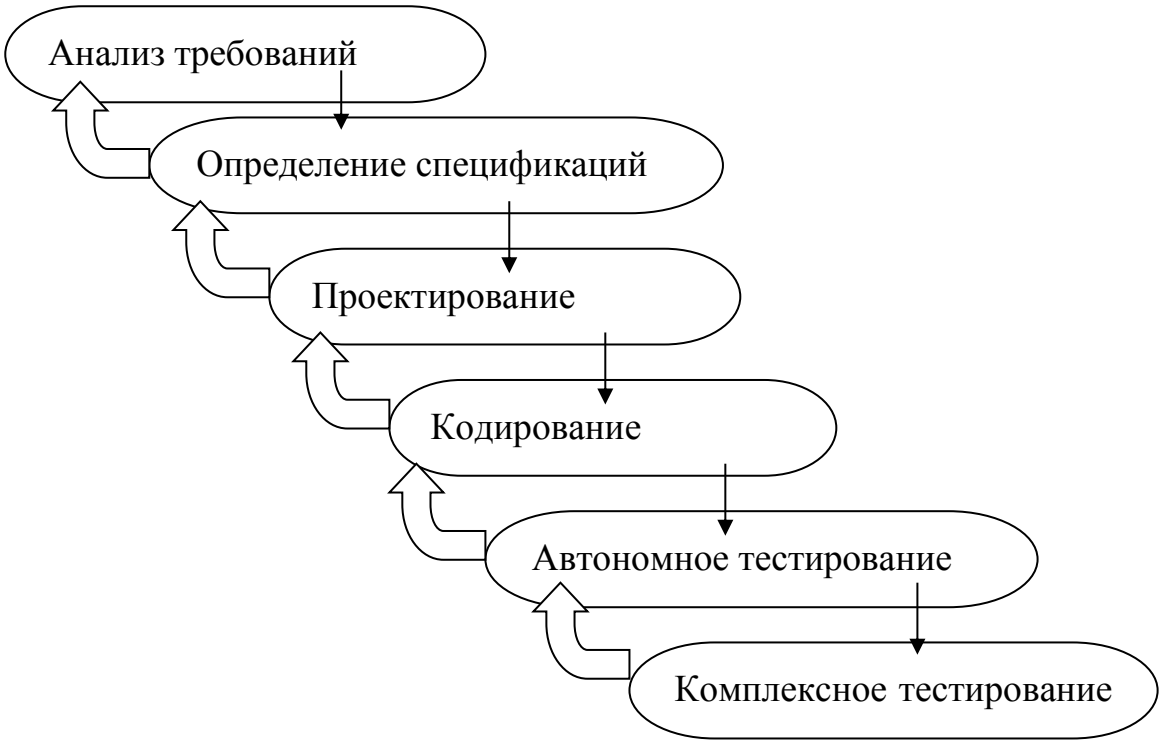


Рис. 1.2. Распределение трудозатрат при построении ИС

Каскадный подход хорошо зарекомендовал себя при построении ИС, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения. В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи. Однако, в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимал вид, представленный на рис. 1.3.

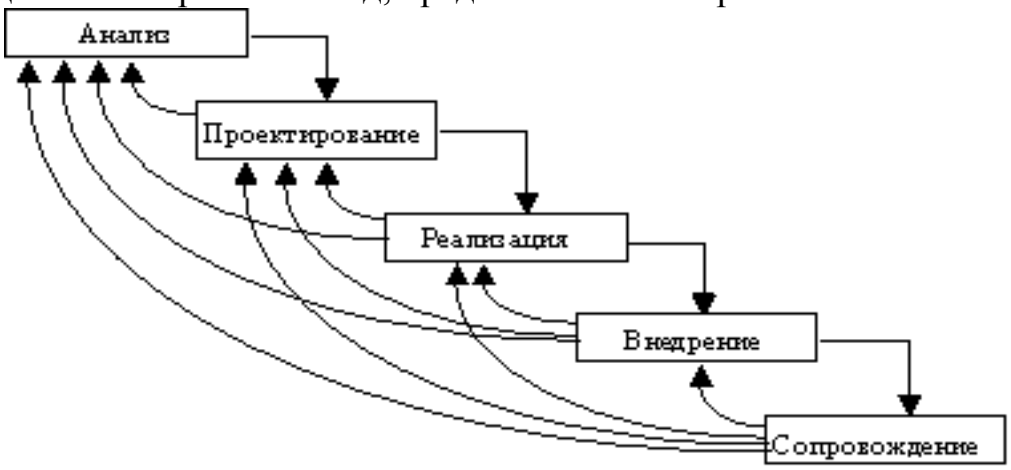


Рис. 1.3. Реальный процесс разработки ПО по каскадной схеме

Основным недостатком каскадного подхода является существенное запаздывание с получением результатов. Согласование результатов с пользователями производится только в точках, планируемых после завершения каждого этапа работ, требования к ИС "заморожены" в виде технического задания на все время ее создания. Таким образом, пользователи могут внести свои замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО, пользователи получают систему, не удовлетворяющую их потребностям. Модели (как функциональные, так и информационные) автоматизируемого объекта могут устареть одновременно с их утверждением.

1.3.2 Итеративная и инкрементальная модель – эволюционный подход

Для преодоления перечисленных проблем была предложена *итеративная модель*. Она предполагает разбиение жизненного цикла проекта на последовательность итераций, каждая из которых напоминает “мини-проект”, включая все фазы жизненного цикла в применении к созданию меньших

фрагментов функциональности, по сравнению с проектом, в целом. Цель каждой итерации – получение работающей версии программной системы, включающей функциональность, определенную интегрированным содержанием всех предыдущих и текущей итерации. Результата финальной итерации содержит всю требуемую функциональность продукта. Таким образом, с завершением каждой итерации, продукт развивается инкрементально.

С точки зрения структуры жизненного цикла такую модель называют *итеративной* (*iterative*). С точки зрения развития продукта – *инкрементальной* (*incremental*). Опыт индустрии показывает, что невозможно рассматривать каждый из этих взглядов изолировано. Чаще всего такую смешанную эволюционную модель называют просто итеративной (говоря о процессе) и/или инкрементальной (говоря о наращивании функциональности продукта).

Эволюционная модель подразумевает не только сборку работающей (с точки зрения результатов тестирования) версии системы, но и её развертывание в реальных операционных условиях с анализом откликов пользователей для определения содержания и планирования следующей итерации. “Чистая” инкрементальная модель не предполагает развертывания промежуточных сборок (релизов) системы и все итерации проводятся по заранее определенному плану наращивания функциональности, а пользователи (заказчик) получает только результат финальной итерации как полную версию системы. С другой стороны, итеративную разработку называют по-разному: инкрементальной, спиральной, эволюционной и постепенной. Разные идеологи вкладывают в эти термины разный смысл, но эти различия не имеют широкого признания и не так важны, как противостояние итеративного метода и метода водопада.

Поскольку, в идеале, на каждом шаге мы имеем работающую систему, итеративная модель имеет следующие преимущества:

- можно очень рано начать тестирование пользователями;
- можно принять стратегию разработки в соответствии с бюджетом, полностью защищающую от перерасхода времени или средств (в частности, за счет сокращения второстепенной функциональности).

Таким образом, значимость эволюционного подхода на основе организации итераций особо проявляется в снижении неопределенности с завершением каждой итерации. В свою очередь, снижение неопределенности позволяет уменьшить риски. Рис. 1.4 иллюстрирует некоторые идеи эволюционного подхода, предполагая, что итеративному разбиению может быть подвержен не только жизненный цикл в целом, включающий перекрывающиеся фазы – формирование требований, проектирование, конструирование и т.п., но и каждая фаза может, в свою очередь, разбиваться на уточняющие итерации, связанные, например, с детализацией структуры декомпозиции проекта – например, архитектуры модулей системы.

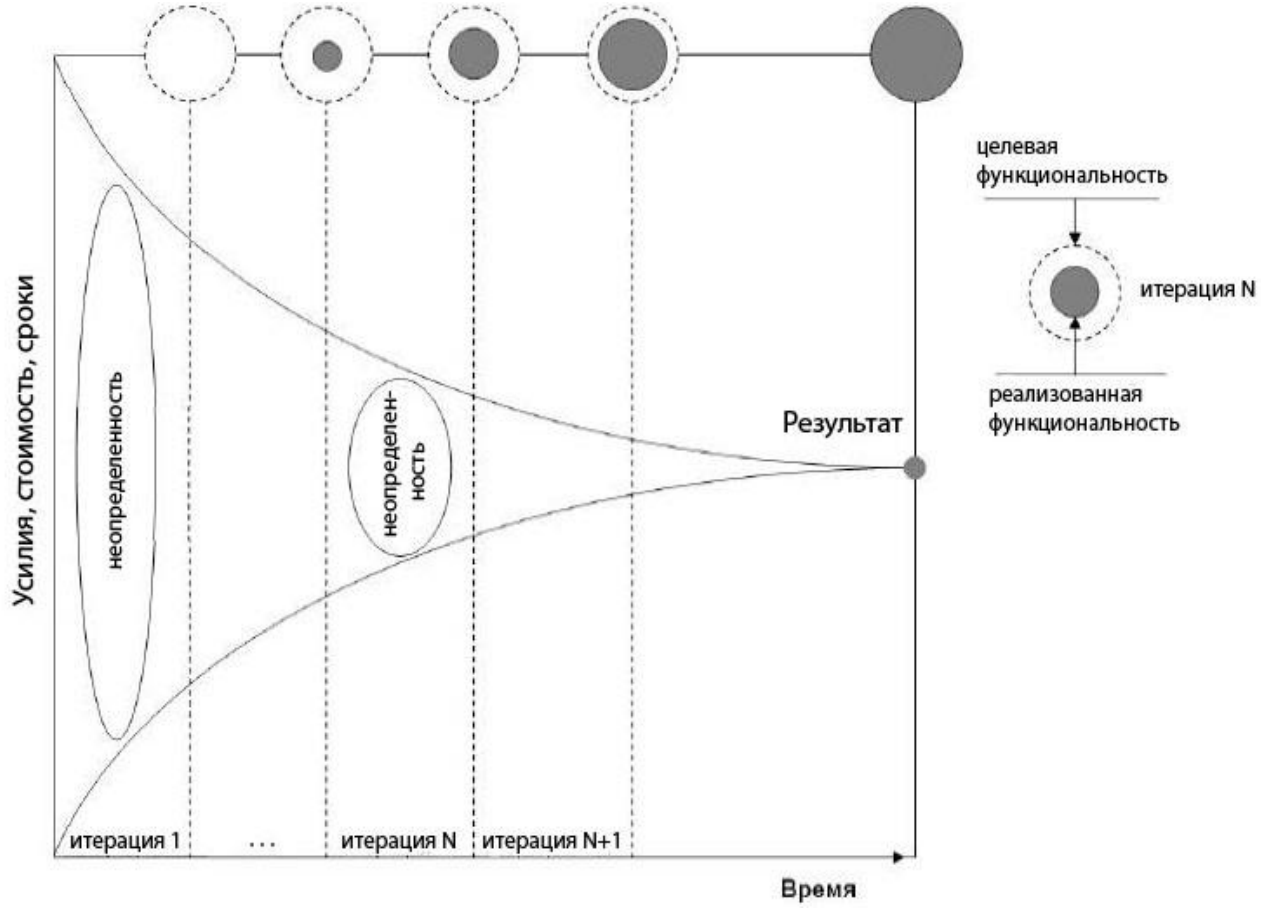


Рис. **1.4**. Снижение неопределенности и инкрементальное расширение функциональности при итеративной организация жизненного цикла

Следует добавить, что внедрение любой новой методологии или нового подхода разработки ПО существенно упрощается, если есть поддерживающий ее набор инструментов, позволяющий, как избежать тяжелого рутинного ручного труда, так и обеспечить выполнение процессов и задач, выполнить которые без средств автоматизации невозможно или несоизмеримо с затратами. Применение итерационной модели очень трудно реализовать без применения инструментов автоматизации: CASE-инструментов планирования и моделирования, средств автоматизации управления основными и вспомогательными поцессами разработки, средств автоматизации выполнения процессов разработки, и т.д.

Наиболее известным и распространенным вариантом эволюционной модели является *спиральная* модель.

1.3.4 Спиральная модель жизненного цикла разработки ПО

Спиральная модель (представлена на рис. 1.5) была впервые сформулирована Барри Боэмом (Barry Boehm) в 1988 году. Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию жизненного цикла. Большая часть этих рисков связана с организационными и процессными аспектами взаимодействия специалистов в проектной команде.

Также спиральная модель ЖЦ делает упор на начальные этапы ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный вариант, который доводится до реализации.

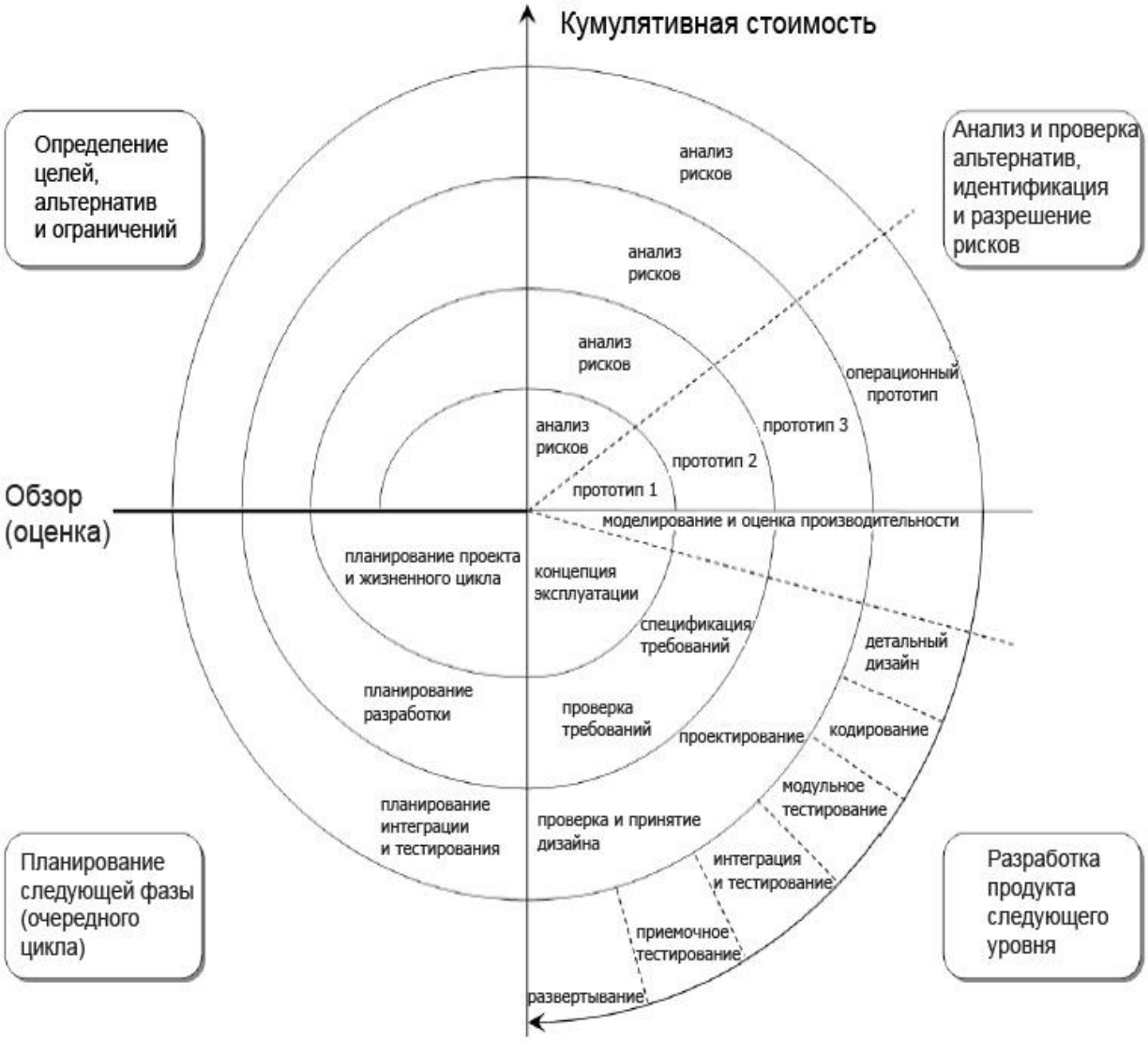


Рис. 1.5. Спиральная модель ЖЦ ПО

Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача – как можно быстрее показать Заказчику работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Перечислим основные преимущества спиральной модели:

- модель уделяет специальное внимание раннему анализу возможностей повторного использования;
- модель предполагает возможность эволюции жизненного цикла, развитие и изменение программного продукта;
- модель предоставляет механизмы достижения необходимых параметров качества как составную часть процесса разработки программного продукта;
- модель уделяет специальное внимание предотвращению ошибок и отбрасыванию ненужных, необоснованных или неудовлетворительных альтернатив на ранних этапах проекта;
- модель позволяет контролировать источники проектных работ и соответствующих затрат;
- модель не проводит различий между разработкой нового продукта и расширением (или сопровождением) существующего;
- модель позволяет решать интегрированный задачи системной разработки, охватывающей и программную, и аппаратную составляющие создаваемого продукта.

Основная проблема спирального цикла – определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков. Действительно, сам Боэм, описывая созданную спиральную модель, обращает внимание на то, что наряду с явными преимуществами по сравнению с другими взглядами на жизненный цикл, при использовании спиральной модели необходимо уточнить, детализировать шаги, т.е. циклы спиральной модели для обеспечения целостного контекста для всех лиц, вовлеченных в проект.

Однако, организация ролей (ответственности членов проектной команды), детализация этапов жизненного цикла и процессов, определение активов (артефактов), значимых на разных этапах проекта, практики анализа и предупреждения рисков – все это вопросы уже конкретного процессного фреймворка или, как принято говорить, *методологии* разработки. Ниже мы коротко рассмотрим наиболее успешные на сегодняшний день методологии разработки качественного программного обеспечения и более подробно остановимся на методологии Rational Unified Process (RUP).

1.3.5 Основы методологии Rational Unified Process

На основе существующих стандартов и моделей обеспечения качества создаются своды правил, описывающих правильную организацию и управление процессами создания современных программных продуктов. В них сформирован и документирован набор проверенных на практике принципов, методов и процессов качественной и производительной работы над проектами по созданию программного обеспечения.

Так как взглядов на детализацию описания жизненного цикла разработки ПО может быть много – безусловно, методологии тоже различаются. Наиболее распространенные на сегодняшний день методологии разработки ПО приведены ниже:

- Rational Unified Process (RUP);
- Enterprise Unified Process (EUP);
- Microsoft Solutions Framework (MSF) в двух представлениях: MSF for Agile и MSF for CMMI (анонсированная изначально как “MSF Formal”);
- Agile-практики: eXtreme Programming (XP), Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM), SCRUM, и другие.

Одной из наиболее успешных в данной области рынка на сегодняшний день является корпорация IBM Rational Software и ее идеологи, столпы объектно-ориентированного подхода Grady Booch, Ivar Jacobson и James Rumbaugh. IBM Rational Software выпустила Rational Unified Process (сокращено RUP, в переводе – Универсальный процесс разработки программных систем фирмы IBM Rational) – базу знаний, представляющую собой путеводитель для всех участников проекта по разработке большой программной системы.

RUP – методологическая основа для всего, что выпускает Rational. То есть данный продукт является энциклопедией (методологическим руководством) того, как нужно строить эффективное информационное производство. Также RUP регламентирует этапы разработки ПО, документы, сопровождающие каждый этап, и продукты самой Rational для каждого этапа. В RUP заложены все самые современные идеи. Продукт постоянно обновляется, включая в себя все новые и новые возможности. К достоинством данной методологии стоит отнести чрезвычайную гибкость, то есть RUP не диктует, что необходимо сделать, а только рекомендует использовать то или иное средство.

Как уже говорилось выше, *внедрение любой методологии существенно упрощается с применением автоматизации процессов данной методологии.* Что возможно только в случае, когда присутствуют: набор технологий, поддерживающих данную методологию, и набор инструментов, реализующих данные технологии. Методология RUP в этом смысле является одной из наиболее «благополучных», поскольку *ее поддерживает набор инструментов IBM Rational*.

Ниже перечислены основные инструментальные средства, входящие в поставку RUP 2003 – 2007 года:

- RUP Builder;
- Rational Process Workbench;
- Rational Administrator;
- Rational Suite AnalystStudio;
- Rational ClearCase;
- Rational ClearQuest;
- Rational ProjectConsole;
- Rational PurifyPlus;
- Rational QualityArchitect;
- Rational RequisitePro;
- Rational Robot;
- Rational Rose;
- Rational Rose RealTime;
- Rational SoDA;
- Rational TestManager;
- Rational Test RealTime;
- Rational TestFactory;
- Rational XDE Developer - Java Platform Edition;
- Rational XDE Developer - .NET Edition.

Методология RUP представляется в виде понятного и легко доступного каждому участнику ИТ проекта Web-сайта, содержимое которого может быть настроено под требования команды разработчиков любого размера (средствами RUP Process Workbench и RUP Builder, входящими в состав RUP) и индивидуально под каждого члена проектной команды (MyRUP).

Как уже упоминалось выше, на пути выпуска ПО существует ряд проблем. Вот что предлагает RUP для решения подобных проблем:

- *Выпускать программное обеспечение, пользуясь принципом промышленного подхода.* То есть так, как поступают любые заводы и фабрики: определяя стадии, потоки, уточняя обязанности каждого участника проекта. Именно промышленный

подход позволит достаточно оперативно выпускать новые версии ПО, которые при этом будут надежными и качественными.

- *Расширять кругозор специалистов для снятия барьеров*. Ведь в подавляющем большинстве случаев специалисты из разных отделов просто говорят на разных языках. Соответственно, снятие языкового барьера должно вести к ускорению работы над программным обеспечением.

- *Использовать итеративную разработку вместо каскадной*, существующей в настоящее время. Принцип итерации заключается в повторяемости определенной последовательности процессов с целью доведения элемента до безошибочного состояния.

- *Обязательное управление требованиями*. Всем известно, что по ходу разработки в систему вносятся изменения (самой группой разработчиков или заказчиком – неважно). Rational предлагает мощную систему контроля управления требованиями: их обнаружение и документирование, поддержку соглашений между разработчиками и заказчиками.

- *Полный контроль всего происходящего в проекте* посредством создания специальных архивов.

- *Унифицированный документооборот*, приведенный в соответствие со всеми известными стандартами. Это значит, что каждый этап в разработке (начало, работа и завершение) сопровождаются унифицированными документами, которыми должен пользоваться каждый участник проекта.

- *Использование визуального моделирования*.

- Применение не только механизмов *Объектно-ориентированного программирования, но и ОО-мышления* и подхода.

Методология RUP основана на следующих основных принципах современной программной инженерии:

- ***Итеративная разработка;***

- ***Управление требованиями;***

- ***Компонентная архитектура;***

- ***Визуальное моделирование;***

- ***Управление изменениями;***
- ***Постоянный контроль качества.***

1.3.6 ЖЦ ПО по методологии Rational Unified Process

Rational Unified Process во всех тонкостях описывает процесс разработки программного обеспечения.

В соответствие с RUP, жизненный цикл программного продукта состоит из серии относительно коротких итераций (рис. 1.6.).

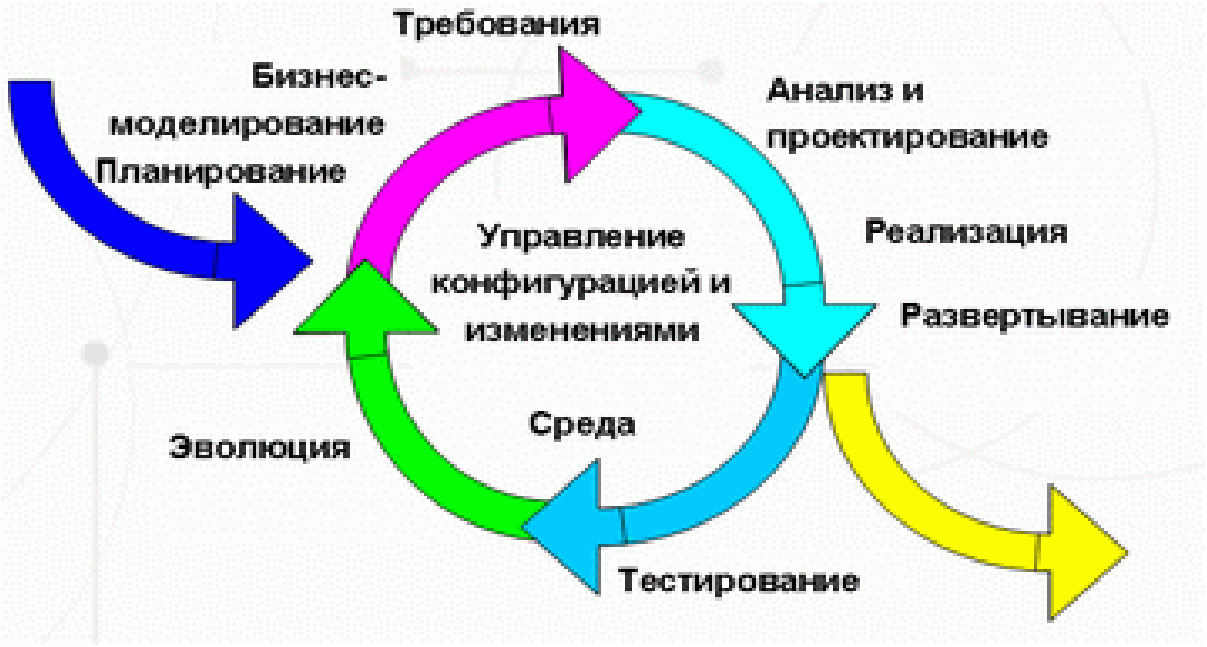


Рис. 1.6. Итерационный жизненный цикл программного продукта

Итерация – это законченный цикл разработки, приводящий к выпуску конечного продукта или некоторой его сокращенной версии, которая расширяется от итерации к итерации, чтобы, в конце концов, стать законченной системой.

Каждая итерация включает, как правило, задачи планирования работ, анализа, проектирования, реализации, тестирования и оценки достигнутых результатов. Однако соотношения этих задач могут существенно меняться. В соответствие с соотношением различных задач в итерации они группируются в фазы. В первой фазе — Начало — основное внимание уделяется задачам анализа. В итерациях второй фазы — Разработка — основное внимание уделяется проектированию и опробованию ключевых проектных решений. В третьей фазе — Построение — наиболее велика доля задач разработки и тестирования. А в последней фазе — Передача — решаются в наибольшей мере задачи тестирования и передачи системы Заказчику.

Rational Unified Process представляет процесс разработки программной системы в двух измерениях (рис. 1.7):

- по содержанию действий участников групп разработки (по основным потокам работ);

- во времени (по стадиям жизненного цикла разрабатываемой системы).

Первое измерение представляет *статический аспект* процесса: оно описано в терминах основных потоков работ (исполнители, действия, их последовательность и Второе измерение представляет *динамический аспект* процесса, поскольку оно выражено в терминах циклов, стадий, итераций и этапов.

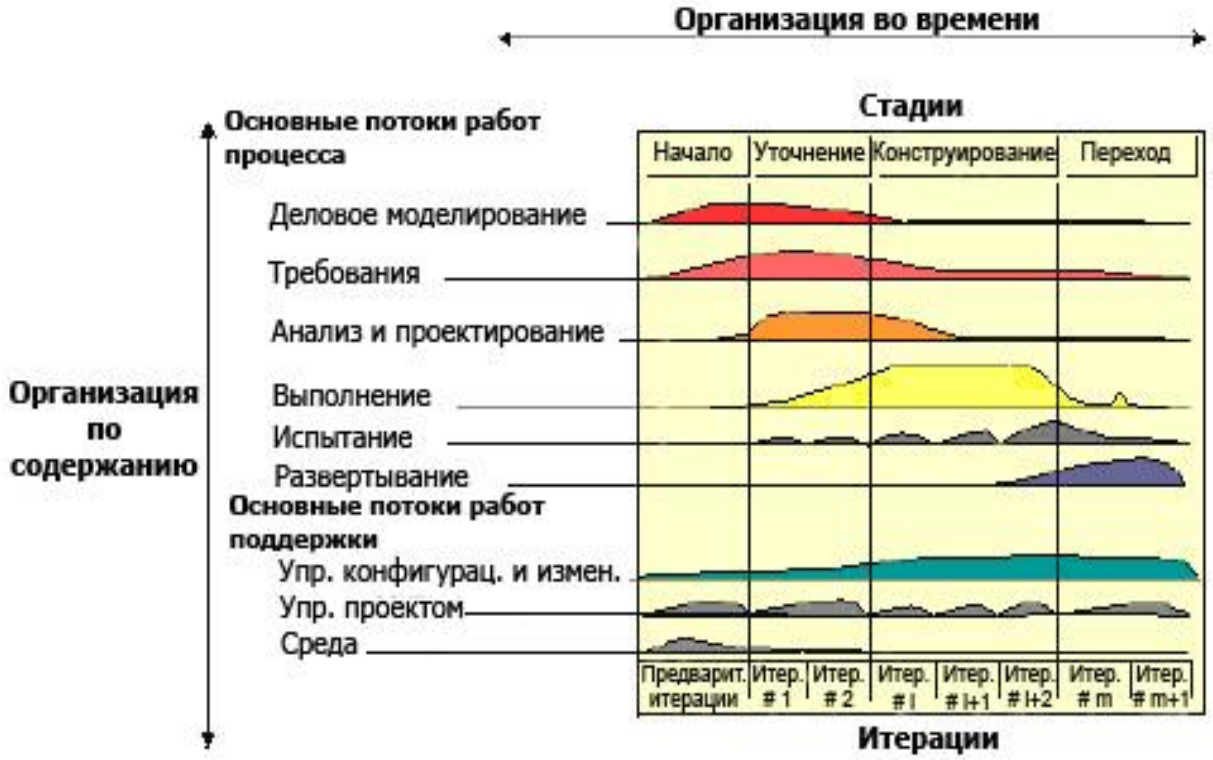


Рис. 1.7. Организация процесса разработки программной системы в двух измерениях

Все итерации, кроме итераций фазы Начало, завершаются созданием функционирующей версии разрабатываемой системы.
Описание процесса выполняется с двух различных точек зрения:

- Техническая точка зрения фокусируется на артефактах и представлениях, связанных с разрабатываемым изделием;

- Организационная точка зрения фокусируется на времени, бюджете, людях и других экономических соображениях.

Иными словами, в RUP четко обозначены следующие вещи: задачи, роли, средства, артефакты процессов; и однозначно определены взаимосвязи между всеми перечисленными элементами в рамках ограничения времени и ресурсов.

Таким образом, использование принципов продукта Rational Unified Process обеспечивает строгий подход к назначению задач и ответственности в пределах группы разработки. Его цель состоит в том, чтобы гарантировать высокое качество программного продукта, отвечающего потребностям конечных пользователей, в пределах предсказуемого временного графика и бюджета.

1.3.7 Обобщенная модель ЖЦ ПО

На основе рассмотренных выше моделей жизненного цикла разработки ПО можно сделать вывод, что методологии разработки в IT-индустрии стремительно развиваются и оптимизируются, находя новые методы и технологии для повышения качества производимого ПО без увеличения временных и/или материальных затрат. При этом все больше возрастает роль автоматизации процессов разработки: без использования техник и иснструментов автоматизации немыслима ни одна разработка ПО по современным методологиям, таким как RUP.

Но по какой бы методологии и модели ЖЦ ПО не велась разработка – можно выделить общие стадии и этапы, присущие каждой из них и состовляющие основу любой разработки. Причем, каждый из этапов имеет свой строго определенный набор задач и функций.

К основным стадиям обобщенной модели ЖЦ ПО можно отнести:

1) анализ требований и проектирование;
2) разработка;
3) эксплуатация;
4) развитие.

Ниже перечислены ***основные этапы обобщенной модели ЖЦ ПО***:

1) анализ требований и разработка технического задания (ТЗ);

2) проектирование системы;

3) детальное проектирование компонент;

4) кодирование и отладка компонент;

5) интеграция и комплексная отладка;

6) тестирование системы: верификация и валидация, приемочные испытания;

7) изготовление поставочного пакета ПО и документации, доставка заказчику;

8) внедрение и поддержка процесса эксплуатации;

9) сопровождение и развитие базовой версии.

Исходя из вышесказанного, обобщенную модель ЖЦ ПО можно представить рис. 1.8.

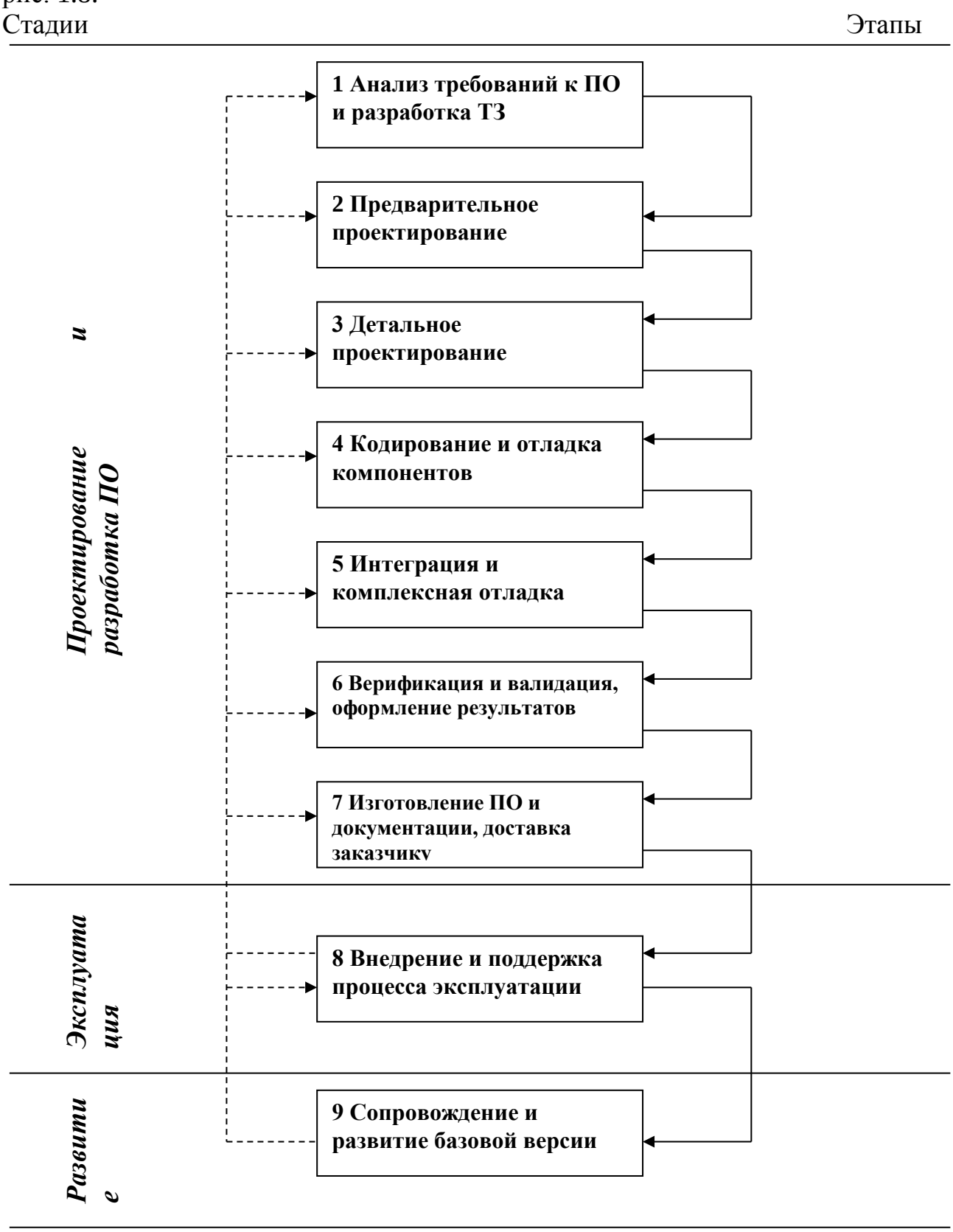


Рис. 1.8 Обобщенная модель жизненного цикла ПО

Обобщенная модель ЖЦ ПО напоминает собой модернизированную каскадную модель разработки. Но, в отличии от последней, она не подразумевает, что каждый этап должен быть завершен до начала последующего. Также обобщенная модель не утверждает, что уже пройденные этапы не могут быть пересмотрены позже в процессе разработки. Модель, представленную на рис. 1.8, можно считать отображением того, как каждый из основополагающих процессов разработки взаимодействует с остальными.

Как видно из обобщенной модели ЖЦ ПО, тестирование является неотъемлемой дисциплиной любой современной методологии. Тестирование имеет свои цели, задачи, роли, виды, методы, критерии, свою методологию и технологию. На основе анализа современных методологий и моделей качества, можно сделать вывод, что тестирование имеет свой собственный жизненный цикл – жизненный цикл тестирования программ (ЖЦ ТП).

Однако, тестирование – это не изолированный процесс. Он тесно связан с другими процессами, входящими в состав методологии разработки. Поэтому ЖЦ ТП тесно завязан на ЖЦ ПО и накладывается на него.

В существующей литературе не производится описания общей методологии и технологии тестирования и не описывается ЖЦ ТП. Чаще всего упор производится на определенный вид тестирования, например, функциональное тестирование. Целью данной главы является объединение и систематизация существующих знаний для описания общей методологии тестирования, а также определению и описанию ЖЦ ТП и выявлению его роли в ЖЦ ПО.

1.4 Жизненный цикл тестирования программного обеспечения и его роль в жизненном цикле разработки

При описании основных понятий разработки и тестирования программных продуктов мы будем основываться на принципах организации процессов создания программного обеспечения, приведенных в RUP, т.к. данные понятия наиболее полно отражают современные стандарты создания качественного ПО (CMM, SPICE, и др.). Коротко рассмотрим современные стандарты качества.

1.4.1 Современные модели качества программного обеспечения

В настоящее время существуют десятки различных подходов к обеспечению качества ПО, и у всех есть свои преимущества. Одной из первых моделей качества стал стандарт ISO (Международной организации по стандартизации) серии 9000, первая версия которого была выпущена в 1987 году. С тех пор сертификаты ISO серии 9000 сохраняют неизменную популярность и признаются во всем мире.

Однако время не стоит на месте, и методики, положенные в основу стандартов серии ISO 9000, постепенно устарели. Это заставило экспертов разрабатывать более совершенные решения в области обеспечения качества, что привело к

созданию в начале 90-х годов целого ряда новых стандартов и методологий. К ним относятся такие стандарты, как: *Bootstrap*, *Trillium*, ориентированный на разработку продуктов в области телекоммуникаций и *ISO 12207*, посвященный жизненному циклу программного обеспечения. Два же наиболее удачных и содержательных стандарта – *Capability Maturity Model (CMM)* и *ISO/IEC 15504 (SPICE)*. Коротко остановимся на описании их основ.

Но для начала приведем краткую сводку терминов, используемых в дальнейшем изложении.

Возможность процесса разработки ПО (software process capability) описывает ожидаемый результат, который можно достигнуть, следуя процессу разработки.

Зрелость процесса разработки ПО (software process maturity) – это степень определенности, управляемости, измеряемости и эффективности процесса разработки ПО.

Качество (quality) – степень соответствия системы, компоненты, процесса или службы потребностям и ожиданиям клиента или пользователя.

Количественное управление процессом (quantitative process management) заключается в выявлении причин для особых отклонений процесса от планируемого и подобающего исправления этих причин (таким образом, качество процесса измеряется не в количестве написанных строк кода или найденных ошибок, а в соответствии процесса исходному плану).

Обеспечение качества ПО (software quality assurance) предназначено для информирования руководства об успешности и зрелости процесса разработки ПО и конечных продуктах.

Определение процесса (process definition) – это рабочее определение набора мер, необходимых для достижения намеченных целей. Характеризуется стандартами.

1.4.1.1 Основные принципы стандарта CMM

Главным понятием стандарта *CMM* (Capability Maturity Model, что обычно переводят как "модель зрелости процесса разработки ПО", хотя более верным по смыслу, по мнению авторов, был бы перевод "модель совершенствования возможностей") является зрелость организации. Незрелой считается организация, в которой процесс разработки программного обеспечения зависит только от конкретных исполнителей и менеджеров, и решения зачастую просто импровизируются "на ходу". В этом случае велика вероятность превышения бюджета или заваливания сроков сдачи проекта, и потому менеджеры вынуждены заниматься только разрешением ближайших проблем.

С другой стороны, в зрелой организации имеются четко определенные процедуры создания программных продуктов и управления проектами. Эти процедуры по мере необходимости уточняются и совершенствуются в пилотных проектах или с помощью анализа стоимость/прибыль. Оценки времени и стоимости выполнения работ основываются на накопленном опыте и достаточно точны.

Наконец, в компании существуют стандарты на процессы разработки, тестирования и внедрения ПО, правила оформления конечного программного кода, компонент, интерфейсов и т.д. Все это составляет инфраструктуру и корпоративную культуру, поддерживающую процесс разработки программного обеспечения.

Таковы базовые посылки стандарта CMM. Можно сказать, что стандарт в целом состоит из критериев оценки зрелости организации и рецептов улучшения существующих процессов. В модели CMM определено пять уровней зрелости организаций. В результате аттестации компании присваивается определенный уровень, который в дальнейшем может повышаться или (теоретически) понижаться. На рис. 1.9 перечислены некоторые технологии, внедрение которых необходимо для достижения различных уровней зрелости организации. Отметим, что каждый следующий уровень включает в себя все ключевые характеристики предыдущих.

Рисунок 1.9 Пять уровней зрелости в модели CMM

Но, к сожалению, использование CMM затрудняют следующие проблемы:

- стандарт CMM является собственностью Software Engineering Institute и не является общедоступным (в частности, дальнейшая разработка стандарта ведется самим институтом, без заметного влияния остальной части программистского сообщества);

- оценка качества процессов организаций может проводиться только специалистами, прошедшими специальное обучение и аккредитованными SEI;

- стандарт ориентирован на применение в относительно крупных компаниях.

1.4.1.2 Основные принципы стандарта SPICE

Задачей *SPICE* (сокращение от Software Process Improvement and Capability dEtermination – определение возможностей и улучшение процесса создания программного обеспечения) является создание международного стандарта, в котором был бы учтен весь накопленный опыт в области разработки ПО. Стандарт SPICE унаследовал многие черты более ранних стандартов, в том числе и уже упоминавшихся ISO 9001 и CMM. Для этого пришлось прибегнуть к повышению уровня детализации стандарта. Следствием такого основательного подхода является большой объем стандарта: документация к нему содержит около 500 страниц.

Больше всего SPICE напоминает CMM. Точно так же, как и в CMM, основной задачей организации является постоянное улучшение процесса разработки ПО. Кроме того, в SPICE тоже используется схема с различными уровнями возможностей (в SPICE определено 6 различных уровней), но эти уровни применяются не только к организации в целом, но и к отдельно взятым процессам. Процессы в модели SPICE могут быть представлены рисунком 1.10.

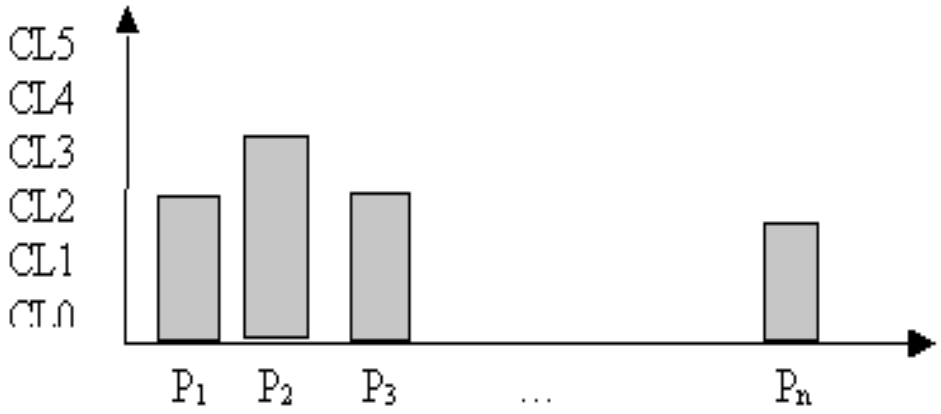


Рис. 1.10. Упрощенная модель оценки процессов в стандарте SPICE

Здесь Pn обозначает процессы, а CL (Capability Levels) обозначает уровни способностей этих процессов. Поэтому иногда говорят, что модель SPICE является двумерной – на одной оси откладываются процессы, а на другой оси – достигнутый уровень возможностей для этих процессов.

Самыми важными достоинствами SPICE являются его открытость и свободное распространение (полный текст SPICE можно получить на официальном сайте SPICE); наличие большого набора средств по обеспечению качества и улучшению процессов; возможность использовать SPICE и в небольших компаниях. В качестве недостатков можно упомянуть большой объем документации стандарта и его все еще неширокая популярность в мире.

Описанные стандарты уже сегодня представляют наиболее развитые модели качества, прошедшие применение на практике. Таким образом, соответствие стандарту перестает быть простым свидетельством достижения некоторого уровня качества и становится способом реального улучшения существующих на предприятии процессов.

1.4.1.3 Концепция качества продукта в RUP. Критерии качества

RUP не ставит своей целью добиться абсолютного качества разрабатываемого продукта. RUP вообще не призывает к достижению недостижимых целей. В частности, этим объясняются и принятая в RUP концепция фаз. В идеале, хорошо бы сначала все проанализировать, потом спроектировать и только потом взяться за программирование. И быть уверенным, что ничего не придется переделывать, потому, что все качественно проанализировано и спроектировано. К сожалению, это недостижимый идеал, как и абсолютное качество программного обеспечения. Как гласит программистская мудрость: «каждая обнаруженная в программе ошибка, в лучшем случае, предпоследняя...»

Приведем несколько концепций хорошего качества программного продукта, которые предлагаются в RUP. Они основаны на следующих утверждениях:

1) Не слишком плохо. Качество программного продукта должно позволить оставаться этому продукту на рынке.

2) Непогрешимость. Следует считать, что проектная команда, выпускающая продукт является лучшей в мире, поэтому и продукт, который она выпускает, есть лучший в мире.

3) Совершенство. Проектная команда делает все, чтобы создать совершенный программный продукт.

4) Клиент всегда прав. Если клиенту нравится создаваемый продукт, то этого достаточно.

5) Хороший процесс разработки. Качество продукта определяется хорошим процессом разработки.

6) Удовлетворение требований. Если продукт удовлетворяет требованиям – это хороший продукт.

7) Ответственность. Качество продукта определяется контрактом. Если проектная команда выполнила контракт, то качество продукта хорошее.

8) Защита. Проблемы, возникающие при разработке программного продукта, должны быть определены, зафиксированы и предотвращены.

9) Учет многих факторов. Продукт является хорошим, когда он имеет достаточно преимуществ и с ним не возникает никаких критических проблем.

Перечисленное выше есть ни что иное, как критерии качества. Они, конечно же, сильно размыты и их нужно детализировать (что и делается в конкретных проектах). Пример более конкретного критерия: количество находимых ошибок достаточно мало в сравнении со временем, затрачиваемым на тестирование (например, когда за день, используя одни и те же тесты, находится всего одна несерьезная ошибка, а ранее находилось 20). Другими словами, когда подавляющее большинство тестов больше не находят ошибки, можно прекратить тестирование.

Можно привести еще ряд критериев. И главная мысль – именно во многом на основании критериев качества и делается заключение о том, достаточно тестировать продукт или еще нет.

RUP предлагает использовать при оценке качества произведенного продукта понятие "достаточно хорошего качества". Использование этого понятия означает, что Разработчик с открытыми глазами оценивает качество продукта, который он представляет Заказчику. Прежде чем браться за дальнейшее улучшение продукта с дополнительными затратами на поиск и устранение ошибок, необходимо рационально оценивать поставленные цели – достижимы они или нет и оправдывают ли затраты. Должна быть четко определена цель (критерий) для обеспечения определенного качества продукта. Для критических систем критерий качественного продукта может быть более жестким. Важно, что критерий есть, и что он должен быть выполнен. И что при этом качественный продукт — это не обязательно продукт без единой ошибки.

Достижение достаточно хорошего качества невозможно без тщательного анализа статистических характеристик результатов тестирования. И это тоже одна из принципиальных особенностей подхода RUP.

2. Методологии и технологии проектирования ИС

2.1. Общие требования к методологии и технологии

Методологии, технологии и инструментальные средства интегрированной системы проектирования (CASE–средства) составляют основу проекта любой ИС. Методология реализуется через конкретные технологии и поддерживающие их

стандарты, методики и инструментальные средства, которые обеспечивают выполнение процессов ЖЦ.

Под технологией программирования понимают организованную совокупность методов, средств и их программного обеспечения, организационно–административных установлений, направленных на разработку, распространение и сопровождение программной продукции.

Технология проектирования определяется как совокупность трех составляющих:

– пошаговой процедуры, определяющей последовательность технологических операций проектирования (рис. 2.1.);

– критериев и правил, используемых для оценки результатов выполнения технологических операций;

– нотаций (графических и текстовых средств), используемых для описания проектируемой системы.

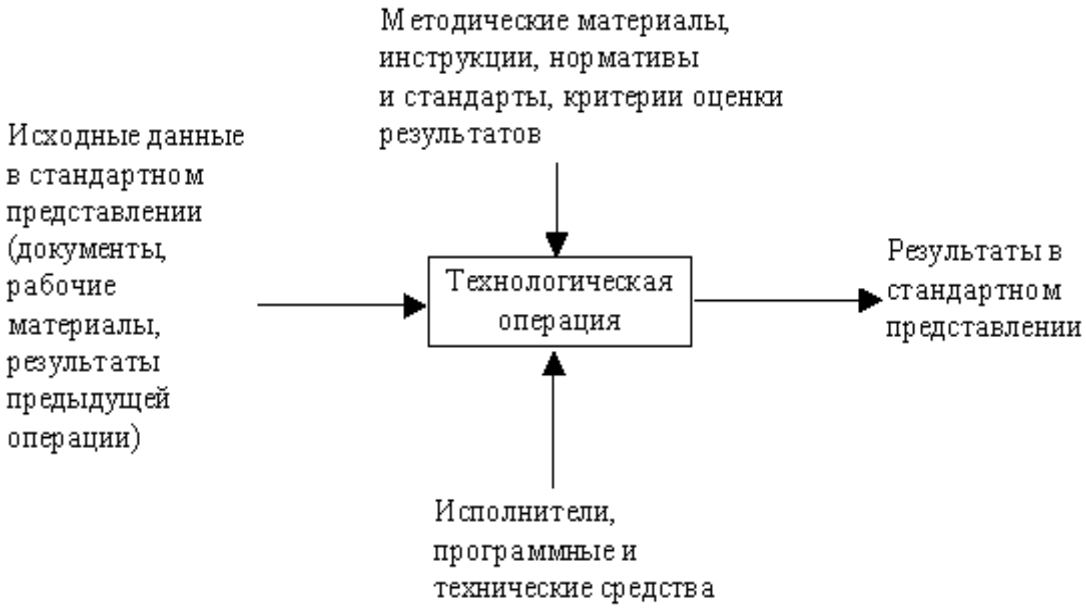


Рис. 2.1. Представление технологической операции проектирования

Технологические инструкции, составляющие основное содержание технологии, должны состоять из описания последовательности технологических операций, условий, в зависимости от которых выполняется та или иная операция, и описаний самих операций.

Технология проектирования, разработки и сопровождения ИС должна удовлетворять следующим общим требованиям:

– технология должна поддерживать полный ЖЦ ПО;

– технология должна обеспечивать гарантированное достижение целей разработки ИС с заданным качеством и в установленное время;

– технология должна обеспечивать возможность выполнения крупных проектов в виде подсистем (т.е. возможность декомпозиции проекта на составные

части, разрабатываемые группами исполнителей ограниченной численности с последующей интеграцией составных частей). Опыт разработки крупных ИС показывает, что для повышения эффективности работ необходимо разбить проект на отдельные слабо связанные по данным и функциям подсистемы. Реализация подсистем должна выполняться отдельными группами специалистов. При этом необходимо обеспечить координацию ведения общего проекта и исключить дублирование результатов работ каждой проектной группы, которое может возникнуть в силу наличия общих данных и функций;

– технология должна обеспечивать возможность ведения работ по проектированию отдельных подсистем небольшими группами (3–7 человек). Это обусловлено принципами управляемости коллектива и повышения производительности за счет минимизации числа внешних связей;

– технология должна обеспечивать минимальное время получения работоспособной ИС. Речь идет не о сроках готовности всей ИС, а о сроках реализации отдельных подсистем. Реализация ИС в целом в короткие сроки может потребовать привлечения большого числа разработчиков, при этом эффект может оказаться ниже, чем при реализации в более короткие сроки отдельных подсистем меньшим числом разработчиков. Практика показывает, что даже при наличии полностью завершенного проекта, внедрение идет последовательно по отдельным подсистемам;

– технология должна предусматривать возможность управления конфигурацией проекта, ведения версий проекта и его составляющих, возможность автоматического выпуска проектной документации и синхронизацию ее версий с версиями проекта;

– технология должна обеспечивать независимость выполняемых проектных решений от средств реализации ИС (систем управления базами данных (СУБД), операционных систем, языков и систем программирования);

– технология должна быть поддержана комплексом согласованных CASE–средств, обеспечивающих автоматизацию процессов, выполняемых на всех стадиях ЖЦ.

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

– стандарт проектирования;

– стандарт оформления проектной документации;

– стандарт пользовательского интерфейса.

Стандарт проектирования должен устанавливать:

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

– правила фиксации проектных решений на диаграммах, в том числе: правила именования объектов (включая соглашения по терминологии), набор атрибутов для

всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм, включая требования к форме и размерам объектов, и т. д.;

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

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

Стандарт оформления проектной документации должен устанавливать:

– комплектность, состав и структуру документации на каждой стадии проектирования;

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

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

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

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

Стандарт интерфейса пользователя должен устанавливать:

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

– правила использования клавиатуры и мыши;

– правила оформления текстов помощи;

– перечень стандартных сообщений;

– правила обработки реакции пользователя.

2.2. Структура комплекта документов

Стандарты ISO 12207 и ISO 9000–3 оставляют право выбора содержания комплекта документов в каждом конкретном случае за разработчиком и заказчиком. Некоторые документы могут объединяться в один, а некоторые вообще исключаться. Но в любом случаи документы, схемы, модели, исходный код и др. являются основой управления проектом. Ниже приведен максимально возможный набор документов в ЖЦ базовой версии сложного ПС на базе стандартов ISO 12207 и ISO 9000–3.

Этап 1. Системный анализ проекта ПС.

Результаты обследования и описание объекта и целей его информатизации.

Отчет о результатах предварительного технико–экономического анализа проекта ПС, оценки сроков, бюджета, рентабельности и риска разработки ПС.

Концепция и предложения по созданию типовой, базовой версии ПС.

Описание постановки задач и предварительная спецификация требований к ПС в целом, к крупным функциональным компонентам и описания данных.

Формализованное описание модели ЖЦ проектируемого ПС.

Предварительный состав стандартов и дополнительных нормативных документов для формирования профиля ЖЦ ПС.

Предварительный укрупненный план проектирования и разработки базовой версии ПС.

Предварительное распределение специалистов по функциональным и технологическим компонентам и по этапам разработки ПС, а также оценка потребности в субподрядчиках и поставщиках компонентов.

Предварительный состав возможных для применения готовых компонентов и версий ПС в целом.

Системный проект, общее описание базовой версии ПС.

Контракт (договор) с заказчиком на проведение предварительного и детального проектирования типовой, базовой версии ПС.

Этап 2. Предварительное (пилотное) проектирование базовой версии ПС.

Уточненная схема архитектуры ИС, взаимодействия программных и информационных компонентов, организации вычислительного процесса и распределения ресурсов среды.

Описание функционирования ПС с объектами внешней среды и человеко–машинного диалога.

Общее описание ПС и комплект спецификаций требований к функциональным программным компонентам и описания данных.

Описание системы управления базами данных комплекса программ, структуры и распределения программных и информационных объектов версии ПС.

Предварительный вариант руководства администратора и пользователя (оператора) ИС и по применению базовой версии ПС,

Состав документации на технологию, средства ее автоматизации и документирование при разработке базовой версии ПС.

Описание предварительного распределения компонентов в базе данных проектирования версии ПС.

Проект руководства по техническому проектированию, программированию, тестированию и отладке функциональных программных компонентов версии ПС,

Описание требований к составу и формам отчетных документов по этапам, работам и компонентам базовой версии ПС.

Таблица распределения специалистов по компонентам версии ПС и по этапам работ.

Аттестаты разработчиков на право использования технологии, профилей стандартов и средств автоматизации разработки базовой версии ПС.

Описание показателей качества компонентов, профилей стандартов и требований к ним по этапам ЖЦ базовой версии ПС.

Пояснительная записка к предварительному проекту базовой версии ПС.

Уточненное и утвержденное техническое задание на проектирование и разработку базовой версии ПС.

Уточненный контракт (договор) с заказчиком на техническое проектирование базовой версии ПС.

Этап 3. Детальное проектирование базовой версии ПС.

Схема архитектуры ПС, взаимодействия компонентов и распределения вычислительных ресурсов среды.

Описания функционирования ПС, потоков данных и человеко–машинного диалога.

Утвержденные спецификации требований и алгоритмы на функциональные группы программ, программные и информационные компоненты.

Руководства программистам по применению технологии и средств автоматизации при разработке программных и информационных компонентов версии ПС.

Руководство по управлению обеспечением качества, надежности и безопасности версии ПС.

Руководство по применению профиля стандартов в ЖЦ базового ПС.

Детальный откорректированный и утвержденный план разработки и распределения ресурсов проекта базовой версии ПС.

Детальный план обеспечения средствами генерации тестов, а также обработки результатов тестирования и отладки модулей и функциональных компонентов и руководство по их применению.

Комплект апробированных программных и информационных компонентов типовой версии ПС и формирования адаптивных версий ПС пользователей.

Пояснительная записка технического проекта базовой версии ПС.

Уточненное техническое задание на разработку и внедрение базовой версии ПС.

Уточненный договор с заказчиком разработку и внедрение базовой версии ПС.

Этап 4. Кодирование (программирование), отладка и разработка документации компонентов базовой версии ПС.

Исходные тексты программных компонентов и описаний данных.

Планы тестирования и отладки программных компонентов.

Сценарии тестирования, спецификации тестов, используемых при тестировании и отладке компонентов.

Отчеты о результатах тестирования, достигнутых показателях качества, откорректированные после отладки программ и описаний данных.

Тексты программных и информационных компонентов на языке программирования и в объектном коде реализующей ЭВМ после завершения отладки и испытаний.

Этап 5. Интеграция (комплексирование) и комплексная отладка базовой версии ПС.

План, средства и руководство для комплексирования и сборки программных и информационных компонентов базовой версии ПС.

Руководства по применению средств автоматизации тестирования программ, обработке результатов отладки и проведению изменений в базовой версии ПС.

Результаты тестирования и полные характеристики функционирования базовой версии ПС в имитированной внешней среде.

План и средства автоматизации интеграции базовой версии ПС с аппаратными средствами в реальной операционной и внешней среде.

План и средства автоматизации и руководства для комплексной отладки и квалификационного тестирования базовой версии ПС в реальной операционной и внешней среде.

Этап 6. Испытания и документирование базовой версии ПС.

Программа, методики и описание средств обеспечения приемо–сдаточных испытаний базовой версии ПС, согласованные с заказчиком.

Результаты определения показателей качества ПС в процессе комплексной отладки и предварительных приемо–сдаточных испытаний.

Отчет о результатах опытной эксплуатации базовой версии ПС.

План адаптации, поставки и переноса на платформы пользователей базовой версии ПС. Акт по результатам приемо–сдаточных испытаний базовой версии ПС и руководство пользователя.

Комплект эксплуатационной документации, описание версии ПС и руководство пользователя.

Исходные тексты программ, описания данных и полные спецификации требований к программным компонентам и версии ПС в целом.

Тексты и генераторы текстовых данных для тестирования программных и информационных компонентов и версии ПС в целом.

Руководство по установке, адаптации и генерации пользовательской версии ПС и загрузке базы данных в соответствии с условиями и характеристиками внешней среды пользователя.

Сертификат на применение и сопровождение версии ПС и область его действия.

Акт о завершении приемо–сдаточных и сертификационных испытаний и результатах выполнения контракта на разработку типовой базовой версии ПС.

2.3. Наиболее перспективные и приемлемые технологии разработки ПО (обновить раздел)

Технология создания крупных информационных систем предъявляет особые требования к методикам реализации и программным инструментальным средствам, а именно:

– реализацию крупных проектов принято разбивать на стадии анализа (прежде чем создавать ИС необходимо понять и описать бизнес–логику предметной области), проектирования (необходимо определить модули и архитектуру будущей

системы), непосредственного кодирования, тестирования и сопровождения. Известно, что исправление ошибок, допущенных на предыдущей стадии, обходится примерно в десять раз дороже, чем на текущей, откуда следует, что наиболее критичными являются первые стадии проекта. Поэтому крайне важно иметь эффективные средства автоматизации ранних этапов реализации проекта;

– крупный проект невозможно реализовать в одиночку. Коллективная работа существенно отличается от индивидуальной, поэтому при реализации крупных проектов необходимо иметь средства координации и управления коллективом разработчиков;

– жизненный цикл создания сложной ИС сопоставим с ожидаемым временем ее эксплуатации. Компании регулярно перестраивают свои бизнес – процессы и, если работать в традиционной технологии для создания ИС может оказаться, что к моменту сдачи ИС она уже никому не нужна, поскольку компания, ее заказавшая, вынуждена перейти на новую технологию работы. Следовательно, для создания крупной ИС жизненно необходим инструмент значительно (в несколько раз) уменьшающий время разработки ИС;

– вследствие значительного жизненного цикла может оказаться, что в процессе создания системы внешние условия изменились. Обычно внесение изменений в проект на поздних этапах создании ИС – весьма трудоемкий и дорогостоящий процесс. Поэтому для успешной реализации крупного проекта необходимо, чтобы инструментальные средства, на которых он реализуются, были достаточно гибкими к изменяющимся требованиям.

На современном рынке средств разработки ИС достаточно много систем, в той или иной степени удовлетворяющих перечисленным требованиям (например, Sybase PowerDesigner). Здесь будет рассмотрена вполне конкретная технология разработки, основывающаяся на *решениях фирм Computer Associates и IBM Rational Software, которые является одними из лучших на сегодняшний день по полноте охвата автоматизации и формализации технологического процесса производства программного обеспечения*.

2.3.1. Технологии, базирующиеся на CASE–средствах Computer Associates

Известно, что наиболее критичными являются ранние этапы создания информационных систем – этап анализа и этап проектирования, поскольку именно на этих этапах могут быть допущены наиболее опасные и дорогостоящие ошибки. Существуют различные методологии и CASE–средства, обеспечивающие автоматизацию этих этапов. Такие CASE–средства должны выполнять следующие задачи:

1. Построение модели бизнес–процессов предприятия и анализ этой модели, анализ эффективности бизнес–процессов с помощью имитационного моделирования.

2. Создание структурной модели предприятия и связывание структуры с функциональной моделью. Результатом такого связывания должно быть распределение ролей и ответственности участников бизнес–процессов.

3. Описание документооборота предприятия.

4. Создание сценариев выполнения бизнес–функций, подлежащих автоматизации и полного описание последовательности действий (включающее все возможные сценарии и логику развития).

5. Создание сущностей и атрибутов и построение на этой основе модели данных.

6. Определение требований к информационной системе и связь функциональности информационной системы с бизнес–процессами.

7. Создание объектной модели, на которой в дальнейшем может быть автоматически сгенерирован программный код.

8. Интеграцию с инструментальными средствами, обеспечивающими поддержку групповой разработки, системами быстрой разработки, средствами управления проектом, средствами управления требованиями, средствами тестирования, средствами управления конфигурациями, средствами распространения и средствами документирования.

Практика показывает, что одна отдельно взятая нотация или инструмент не могут в полной мере удовлетворить всем перечисленным требованиям. CASE–средства фирмы Computer Associates (CA) представляет собой набор связанных между собой инструментальных средств, обеспечивающих решение задач анализа, проектирования, генерации, тестирования и сопровождения информационных систем. На рис. 2.2. приведены основные CASE– средства CA и их взаимодействие.

AllFusion Process Modeler (BPwin) позволяет создавать модели процессов и поддерживает три стандарта (нотации) моделирования – IDEF0, DFD и IDEF3. Каждая из трех нотаций, поддерживаемых в BPwin, позволяет рассмотреть различные стороны деятельности предприятия.

http://www.interface.ru/fset.asp?Url=/ca/erwin.htm

Модель IDEF0 предназначена для описания бизнес–процессов на предприятии, она позволяет понять, какие объекты или информация служат сырьем или источником для процессов, какие результаты производят работы, что является управляющими факторами и какие ресурсы для этого необходимы. Методология структурного моделирования предполагает построение модели AS–IS (как есть), анализ и выявление недостатков существующих бизнес–процессов и построение модели TO–BE (как должно быть), то есть модели, которая должна использоваться при построении автоматизированной системы управлением предприятия.

Нотация IDEF0 позволяет наглядно представить бизнес–процессы и легко выявить такие недостатки, как недостаточно эффективное управление, ненужные, дублирующие, избыточные или неэффективные работы, неправильно использующиеся ресурсы и т.д. При этом часто выясняется, что обработка информации и использование ресурсов неэффективны, важная информация не доходит до соответствующего рабочего места и т.д. Признаком неэффективной организации работ является, например, отсутствие обратных связей по входу и

управлению для многих критически важных работ. Встроенная система стоимостного анализа (ABC) позволяет количественно оценить стоимость каждой работы и эффективность реализации той или иной технологии.

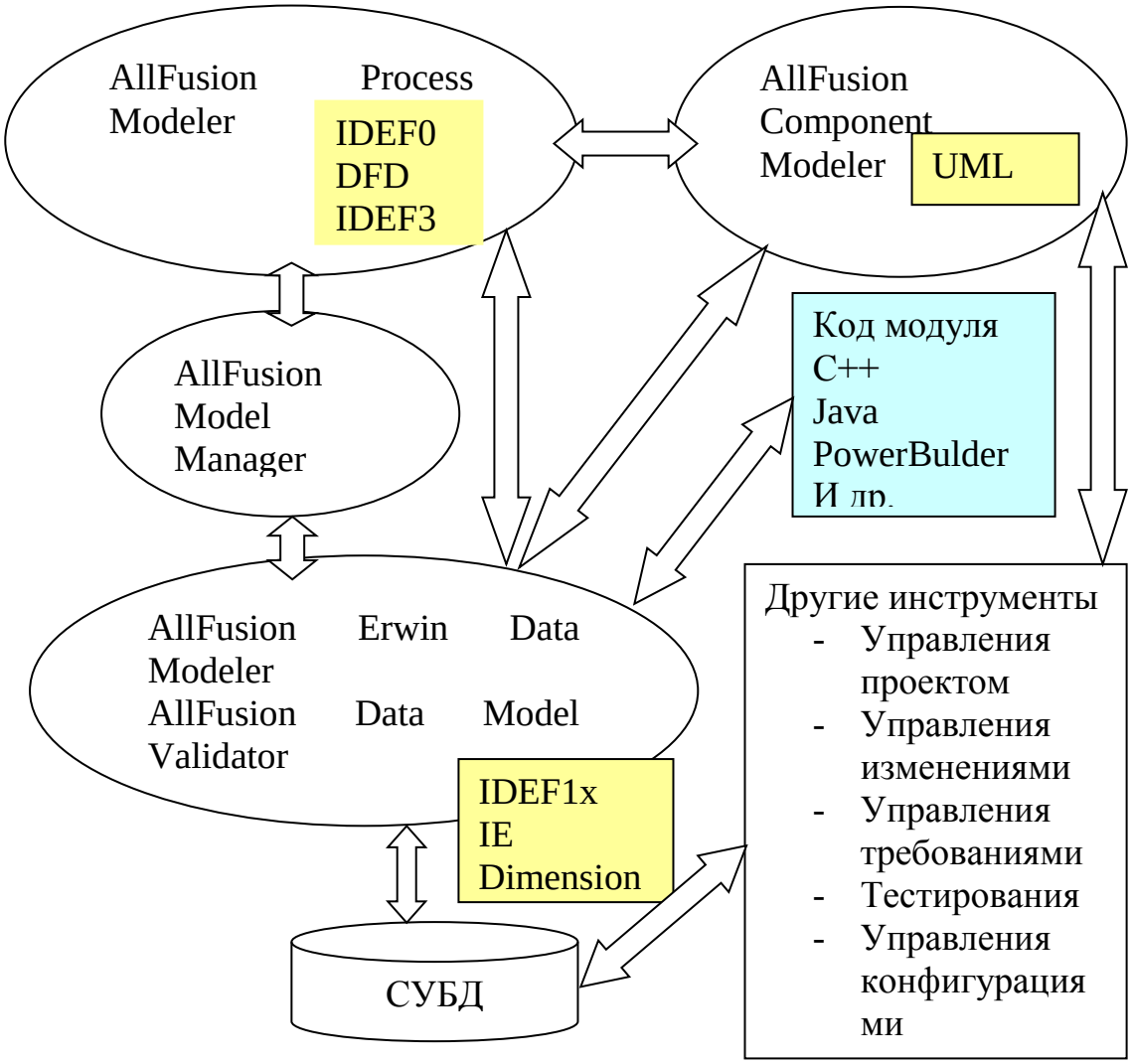


Рис. 2.2. Схема взаимодействия CASE–средств Computer Associates

Диаграммы потоков данных (Data flow diagramming, DFD) используются для описания документооборота и обработки информации. DFD описывают функции обработки информации, документы, объекты, а также сотрудников или отделы, которые участвуют в обработке информации. Наличие в диаграммах DFD элементов для описания источников, приемников и хранилищ данных позволяет более эффективно и наглядно описать процесс документооборота.

Для описания логики взаимодействия информационных потоков более подходит IDEF3, называемая также workflow diagramming, – нотация моделирования, использующая графическое описание информационных потоков, взаимоотношений между процессами обработки информации и объектов, являющихся частью этих процессов. Диаграммы IDEF3 позволяют описать как отдельные сценарии реализации бизнес–процессов, так и полное описание последовательности действий.

Организационные диаграммы (organization charts) позволяют описать структуру предприятия и создаются на основе предварительно созданных ролей. Благодаря организационным диаграммам можно отобразить как структуру организации, так и любую другую иерархическую структуру.

В BPwin возможен экспорт модели в систему имитационного моделирования Arena (Systems Modeling Corp.).

Имитационное моделирование – это метод, позволяющий строить модели, учитывающие время выполнения функций. Полученную модель можно "проиграть" во времени и получить статистику происходящих процессов так, как это было бы в реальности. В имитационной модели изменения процессов и данных ассоциируются с событиями. "Проигрывание" модели заключается в последовательном переходе от одного события к другому. Обычно имитационные модели строятся для поиска оптимального решения в условиях ограничения по ресурсам, когда другие математические модели оказываются слишком сложными. Экспорт модели процессов в Arena позволит аналитикам более качественно производить реорганизацию деятельности предприятий и оптимизировать производственные процессы.

BPwin поддерживает словари сущностей и атрибутов, что позволяет создавать объекты модели данных непосредственно в среде BPwin, связывать их с объектами модели процессов и экспортировать в систему моделирования данных AllFusion Erwin Data Modeler. Такая связь гарантирует завершенность анализа, гарантирует, что есть источник данных (Сущность) для всех потребностей данных (Работа) и позволяет делить данные между единицами и функциями бизнес–процессов. Каждая стрелка в модели процессов может быть связана с несколькими атрибутами различных сущностей. Связи объектов способствуют согласованности, корректности и завершенности анализа.

Для построения модели данных Computer Associates предлагает мощный и удобный инструмент – AllFusion Erwin Data Modeler.

ERwin имеет два уровня представления модели – логический и физический. На логическом уровне данные представляются безотносительно конкретной СУБД, поэтому могут быть наглядно представлены даже для неспециалистов. Физический уровень данных – это отображение системного каталога, который зависит от конкретной реализации СУБД.

ERwin позволяет проводить процессы прямого и обратного проектирования для СУБД более 20 типов. Это означает, что по модели данных можно сгенерировать схему БД или автоматически создать модель данных на основе информации системного каталога с учетом реализации конкретной СУБД. Кроме того, ERwin позволяет выравнивать модель и содержимое системного каталога после редактирования того, либо другого. ERwin поддерживает три нотации (IDEF1X, IE и DIMENSIONAL), что делает его незаменимым как для проектирования оперативных баз данных, так и для создания хранилищ данных.

Более полная информация приведена на официальном сайте:
http://www.ca.com/us/default.aspx
http://www.interface.ru/fset.asp?Url=/ca/erwin.htm http://www.interface.ru/

2.3.2. Технологии, базирующиеся на CASE–средствах IBM Rational

IBM Rational Unified Process – база знаний, представленная в виде гипертекстового справочника, оформленного как Web–сайт.

RUP – методологическая основа для всего, что выпускает IBM Rational Corp. То есть данный продукт является энциклопедией (методологическим руководством) того, как нужно строить эффективное информационное производство. Также RUP регламентирует этапы разработки ПО, документы, сопровождающие каждый этап, и продукты самой IBM Rational Corp. для каждого этапа. В RUP заложены все самые современные идеи. Продукт постоянно обновляется, включая в себя все новые и новые возможности. К достоинством данной методологии стоит отнести чрезвычайную гибкость, то есть RUP не диктует, что необходимо сделать, а только рекомендует использовать то или иное средство.

Методология RUP основана на следующих основных принципах современной программной инженерии:

– итеративная разработка,
– управление требованиями,
– компонентная архитектура,
– визуальное моделирование,
– управление изменениями,
– постоянный контроль качества.

В методологии RUP каждый типовой проект разработки или внедрения ПО схематично представляется в виде диаграммы, на временной оси которой показана динамическая структура процесса. В соответствие с RUP работа над проектом разбивается на четыре фазы жизненного цикла (рис. 2.3.):

– начало проекта (эскизное проектирование)
– детализация системы (разработка технического задания)
– создание системы (рабочее проектирование)
– внедрение системы (приемо–сдаточные испытания) или переход.

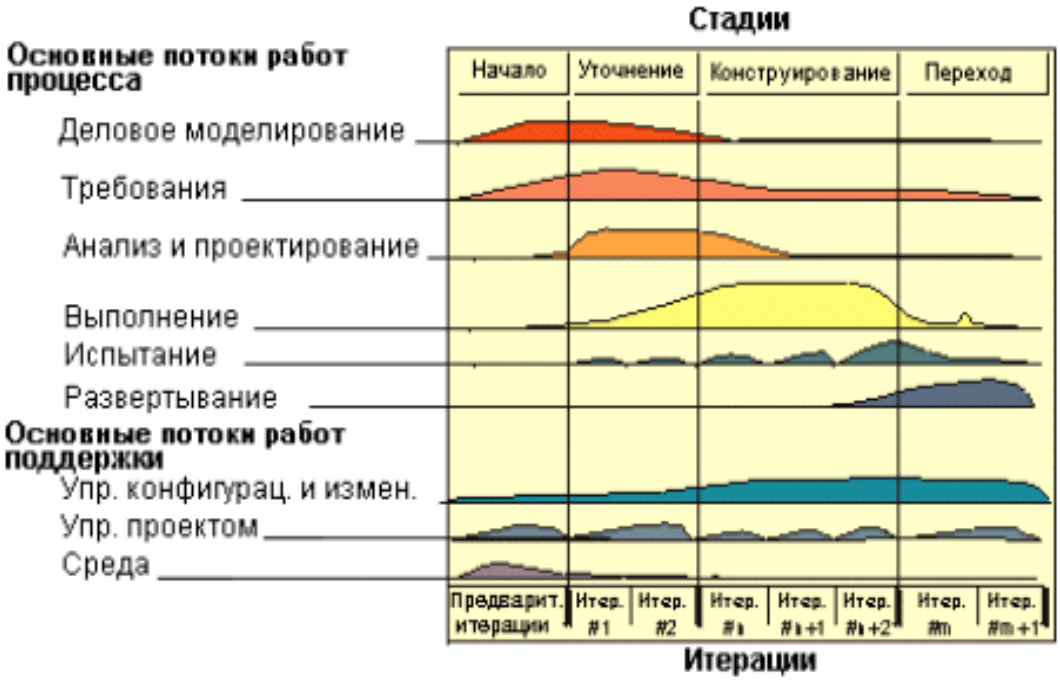


Рис. 2.3. Схема RUP по стадиям работы над проектом

Каждая фаза может быть разбита на итерации (итеративный подход). Каждая итерация завершается выпуском работающего прототипа конечной системы. Это позволяет правильно оценивать риски проекта и эффективно снижать или устранять их на ранних стадиях разработки.

Для каждого процесса разработки методология RUP определяет ролевой состав проектной команды и описывает регламент действий, потоки событий, получаемые результаты и документы (артефакты процесса). Интегральная интенсивность этих действий в зависимости от времени показана на диаграмме RUP для всех основных и вспомогательных процессов. Из диаграммы видно, что в отличие от каскадной модели в методологии RUP все процессы выполняются практически во всех фазах жизненного цикла проекта. Однако в зависимости от фазы меняются текущие цели проекта и, соответственно, соотношение между объемами работ, соответствующих различным процессам.

2.3.2.1. Краткая характеристика основных технологических программных продуктов IBM Rational

Решения IBM Rational прежде всего адресованы крупным компаниям и организациям, имеющим выделенный ИТ бюджет и ведущим большое количество проектов разработки, приобретения, внедрения или сопровождения ПО, аутсортинговым компаниям–разработчикам заказного ПО и системным интеграторам.

Технология IBM Rational может служить основой для построения корпоративной стратегии в области разработки, внедрения и сопровождения ПО

корпоративных ИС, удовлетворяющей требованиям современных международных (ISO 12207, 1504), национальных (ГОСТ 34, 19) и отраслевых стандартов. Вместе с тем она является очень гибкой и может использоваться для успешного выполнения проектов с минимальным объемом документирования.

Продуктовая линейка инструментов IBM Rational Corp. состоит из:

интегрированных функциональных наборов Rational Suite, представляющих эффективное и оптимизированное по цене решение для организации коллективной работы над ИТ проектами;

семейства продуктов IBM Rational XDE, которое дополняет возможности Rational Suite и предоставляет расширенный опыт разработки (eXtended Development Experience) для проектирования, разработки и тестирования Java и .NET приложений, включая Web–ориентированные решения;

специализированных инструментов для автоматизации отдельных процессов жизненного цикла ПО ИС (из этих инструментов комбинируются наборы Rational Suite).

Ориентировочный срок выполнения работ по внедрению процессов управления требованиями, визуального моделирования, конфигурационного управления и функционального тестирования составляет от 7–ми до 9–ти месяцев.

Более полная и свежая информация о платформе IBM Rational находиться на сайте http://www-01.ibm.com/software/ru/rational/.

IBM Rational Rose

IBM Rational Rose – CASE–средство визуального проектирования информационных систем, позволяющее моделировать как бизнес процессы, так и различные компоненты программного обеспечения. Поддерживает различные объектно–ориентированные методологии: язык моделирования (UML), нотации Гради Буча и Джеймса Рамбо.

Данный продукт позиционируется для использования проектировщиками, аналитиками, разработчиками. Rose является CASE средством, чьи графические возможности, основанные на языке UML (Universal Modeling Language – универсальном языке моделирования), способны решить любые задачи, связанные с любым проектированием и моделированием: от общей модели процессов (абстрактной) предприятия до конкретной (физической) модели класса в создаваемом ПО. Работа в Rational Rose заключается в проектировании определенного вида диаграмм, задавая при этом все свойства, отношения и взаимодействие друг с другом.

При разработке любой информационной системы в первую очередь возникает проблема взаимопонимания подрядчика и заказчика уже на стадии договоренности о структуре системы. Имея такой инструмент, как Rose, проектировщик (аналитик) всегда может показать заказчику не абстрактное словесное описание процесса, а его конкретную модель. Rose позволит быстрее утрясти с заказчиком все детали планируемой системы. Как говорилось выше, RUP описывает все артефакты (документы), возникающие как по ходу проекта, так и в Rose. Результатом моделирования является файл с моделью, которую проектировщик передает

следующему звену сотрудников – кодировщикам, которые дополняют полученную логическую модель системы моделями конкретных классов на конкретном языке программирования.

Rose предоставляет разработчикам возможность проектирования и моделирования систем на языке UML с последующей кодогенерацией скелетов программ языке C++, C#, Ada, Java, J#, Basic, Xml, Oracle и др. Возможность обратного проектирования – реинжениринга, когда готовую информационную систему (например, на C++) или базу данных (на Oracle) “закачивают” в Rose с целью получения наглядной визуальной (структурной) модели. Редакция Rose DataModeler – позволяет проектировать базы данных.

Rose RealTime – узкоспециализированная версия, способная проводить 100% кодогенерацию и реинжениринг. Имеет неполный набор диаграмм.

Rose Enterprise – наиболее полная версия, включает в себя все вышеописанные возможности и может быть использована проектировщиками, аналитиками, разработчиков широкого профиля.

При комплексной разработке ПО нужно использовать более мощные возможности моделирования данных, а именно использовать IBM Rational Software Architect, IBM Rational Software Modeler или IBM Rational Data and Application Modeling Bundle.

Семейство продуктов Rational Rose имеет в своем составе:

- IBM Rational Rose Developer for Java;
- IBM Rational Rose Developer for Visual Studio;
- IBM Rational Rose Modeler;
- IBM Rational Rose Data Modeler;
- IBM Rational Rose Developer for UNIX / Linux;
- IBM Rational Rose Enterprise;
- IBM Rational Rose Technical Developer.

IBM Rational SoDA

IBM Rational SoDa – система, автоматизирующая процесс создания и обновления проектной документации. Результатом любой деятельности, является документ или отчет заранее установленного образца. Еще лучше, когда “внутренние стандарты” как–то соотносятся с общепринятыми мировыми. Последнее особенно важно интернациональным командам, работающим совместно с зарубежными партнерами. На решение всех проблем с документооборотом направлен инструмент Rational SoDA. Его основная обязанность – подготовить отчет по заранее установленному шаблону. Данные для отчета берутся из любого инструмента Rational. Например, необходимо получить готовый документ по имеющейся модели в Rational Rose. SoDA позволит сгенерировать подобный отчет, представив результат в виде обычного документа в формате MS Word.

SoDA является макросом (шаблоном) для MS Word. Система вызовов и меню интегрирована в Word и позволяет генерировать шаблоны на базе имеющихся файлов. SoDA допускает к использованию, как стандартных шаблонов, так и

созданных пользователем при помощи специального Wizard, также встроенного в систему меню Word.

Инструмент автоматизированной подготовки документации позволяет:

- генерировать документы на основе прямого доступа к репозиториям данных инструментов IBM Rational;
- автоматически создавать документы и отчеты в формате HTML;
- использовать шаблоны для стандартизации работы в проекте или в организации в целом, настраивая их под требования используемых стандартов;
- повторно воспроизводить точные актуальные документы, предотвращая прямой ввод дополнительных данных в отчетные документы;
- являясь инструментом автоматизации документирования уровня организации, IBM Rational SoDA входит в состав пакетов коллективной работы, таких как IBM Rational Team Unifying Platform и IBM Rational Suite.

Rational SoDA генерирует документы, извлекая информацию из следующих проектных репозиториев:

- репозиторий требований Rational RequisitePro;
- репозиторий тестирования Rational TestManager;
- базы данных запросов на изменения Rational ClearQuest;
- версионного хранилища (VOB) Rational ClearCase;
- общий проектный репозиторий Rational Administrator.

IBM Rational Requisite PRO и Rational Doors

IBM Rational RequisitePro (Rational Doors) – средство управления требованиями, позволяющее организовать совместную работу большого числа специалистов и исполнителей над одним проектом. Позволяет всем членам проектной команды находиться в курсе событий в течение всего процесса разработки. RequisitePro – это удобный инструмент для ввода и управления требованиями, который может использоваться всеми участниками команды. Продукт позволяет в наглядной форме получать, выводить, структурировать наборы вводимых требований.

Для каждого требования поддерживается набор атрибутов, позволяющий эффективно управлять проектом на основе задания иерархий требований, установки их приоритетов, сортировки, назначения требований конкретным исполнителям. Для требований предусмотрен, предопределен набор атрибутов, который может быть расширен пользователем по своему усмотрению, что позволяет характеризовать требования в соответствии с представлениями пользователя.

Развитые возможности прослеживания требований позволяют визуально определять схожие требования в рамках одного или нескольких проектов. Это дает возможность применения готовых апробированных решений в новом проекте. Возможность задания связей между требованиями позволяет легко проследить, какие требования следует подвергнуть анализу (и, возможно, пересмотру) при

модификации некоторого конкретного требования или атрибута. Тем самым упрощается процесс внесения изменений.

Для каждого требования хранится его история, позволяющая отследить, какие изменения были внесены в требование, кем, когда и почему.

Выгоды эффективного управления требованиями с помощью RequisitePRO увеличиваются экспоненциально при использовании его всей командой разработчиков. RequisitePRO упрощает общение между разработчиками путем предоставления общего доступа ко всем требованиям проекта, либо к их части.

Все документы и данные, относящиеся к требованиям, централизованно организуются при помощи RequisitePRO. Требования заказчика, дизайн подсистем, сценарии, функциональные и нефункциональные спецификации и планы тестирования распределяются и связываются таким образом, чтобы максимально облегчить управление проектом.

Основные особенности:

- использование улучшенной интеграции с Microsoft Word обеспечивает удобную и знакомую рабочую среду для определения и структуризации требований;
- включает мощную базу данных требований, синхронизированную с документами Word, обеспечивая широкие возможности по организации требований, их контролю и анализу;
- позволяет детализировать требования набором настраиваемых атрибутов, которые используются для фильтрации требований при анализе и получении отчетов;
- предоставляет детальные представления взаимосвязей требований, включая связи типа родительское/дочернее требование и связи зависимых требований, изменение которых затрагивает другие требования;
- имеет функционал создания базовых версий проекта на основе XML с возможностью последующего сравнения таких версий и выявления отличий;
- интеграция со многими инструментами линейки IBM Software Delivery Platform позволяет расширить доступность, коммуникативность и возможности контроля взаимосвязи требований;
- предлагает полнофункциональный, масштабируемый Web-интерфейс, оптимизированный для использования в территориально распределенной среде.

IBM Rational Software Architect

Предназначен для проектирования и разработки приложений на основе моделей на языке UML, которые позволяют выполнять детальное проектирование информационной системы и обеспечивать создание качественной архитектуры. Включает такие средства как IBM Rational Software Modeler и IBM Rational Application Developer. Позволяет выполнить детальное проектирование информационной системы и обеспечить создание качественной архитектуры:

- расширяет открытую среду разработки Eclipse;

- прост в инсталляции и использовании как для Microsoft Windows так и для Linux;
- упрощает переход от модели к коду для Java/J2EE, Web Services, SOA и C/C++ приложений;
- включает все возможности IBM Rational Application Developer;
- использует последние достижения технологии объектного моделирования, предоставляет гибкие средства моделирования для UML 2, UML-подобных нотаций для Java и т.п.;
- обеспечивает гибкие возможности моделирования при параллельной разработке и архитектурном анализе, т.е. разбиение, слияние, комбинация, сравнение моделей и их фрагментов;
- упрощает переход от кода к архитектуре за счет трансформаций модель-модель и код-модель, включая обратную трансформацию.

IBM Rational Application Developer

Помогает разработчикам Java быстро проектировать, разрабатывать, тестировать и развертывать высококачественные продукты на базе Java/J2EE, Порталы, приложения Web, Web-сервисы и приложения SOA:

- увеличивает производительность и сокращает время разработки и тестирования на базе Eclipse;
- обеспечивает гибкий процесс инсталляции с возможностью выбора только необходимых функций и модулей;
- интегрирован и оптимизирован для IBM WebSphere Application Server и IBM WebSphere Portal Server включая среду тестирования для этих продуктов;
- сокращает время обучения Java за счет визуального проектирования с автоматической синхронизацией кода;
- мощное средство создания приложений SOA, включая возможность автоматического создания необходимых компонент SOA таких как WSDL и WSIL файлы;
- применяет визуальные техники разработки порталов;
- упрощает разработку и управление Web-приложениями.

IBM Rational Software Modeler

Позволяет архитекторам, системным аналитикам, проектировщикам и другим ролям специфицировать и передавать информацию по проекту разработки ПО из разных представлений и для разных заинтересованных лиц:

- работает на основе Eclipse;
- прост в инсталляции и использовании как для Microsoft Windows так и для Linux;
- поддерживает моделирование в UML 2.1;
- обеспечивает гибкие возможности моделирования при параллельной разработке и архитектурном анализе, т.е. разбиение, слияние, комбинация, сравнение моделей и их фрагментов;

- упрощает переход от кода к архитектуре за счет трансформаций модель-модель и код-модель, включая обратную трансформацию;
- позволяет применять встроенные шаблоны проектирования – и/или создавать собственные – для реализации утвержденных правил и методов;
- интегрируется с другими компонентами жизненного цикла – включая управление требованиями, управление изменениями; содержит Rational ClearCase LT.

IBM Rational Systems Developer

Использует преимущества платформы Eclipse и помогает группам разработчиков использовать язык UML 2 для создания хорошо структурированных C/C++, Java (J2SE) and CORBA-приложений на основе моделей:

- расширяет возможности Eclipse посредством улучшенной интеграции инструментов и доступа к экосистеме готовых встроенных решений (plug-in);
- прост в инсталляции и использовании как для Microsoft Windows так и для Linux;
- поддерживает моделирование в UML 2;
- обеспечивает гибкие возможности моделирования при параллельной разработке и архитектурном анализе, т.е. разбиение, слияние, комбинация, сравнение моделей и их фрагментов;
- позволяет применять встроенные шаблоны проектирования – и/или создавать собственные – для реализации утвержденных правил и методов;
- упрощает переход от кода к архитектуре за счет трансформаций код-модель и модель-код, например для переходов от UML к Java или от C++ к UML;
- использует возможности структурного контроля для обнаружения проблемных мест в приложениях Java.

IBM Rational Business Developer и EGL

IBM Rational Business Developer Extension (RBD Extension) предоставляет мощный инструментарий для разработки приложений на языке Enterprise Generation Language (EGL). Предлагаемый инновационный комплексный подход к быстрой разработке приложений обладает следующими особенностями:

- повышенная эффективность разработки за счет мощной, не привязанной к одной платформе бизнес-ориентированной спецификации и множества различных инструментов и программ-мастеров для быстрой разработки;
- быстрая и упрощенная поддержка SOA. Язык Enterprise Generation Language (EGL) включает понятие службы (Service), а ПО Rational Business Developer Extension предоставляет инструментарий для быстрого создания, тестирования и развертывания службы на всех поддерживаемых платформах, в том числе для автоматического создания служб на основе моделей;
- беспроблемная и быстрая сквозная разработка Web-приложений;

- возможность внедрения приложений и служб на самых разных платформах, включая серверы приложений J2EE и традиционные транзакционные среды мэйнфреймов (такие, как CICS-система для System z или ОС i5/OS для System i);
- простота обучения. Разработчики, имеющие общие навыки программирования, могут изучить EGL и приступить к работе с RBD Extension за несколько недель;
- расширение и модернизация традиционных стандартов. Встроенные возможности взаимодействия EGL с COBOL, RPG, PL/I или любой другой существующей программой идеально подходят для быстрого повторного использования существующих инвестиций при создании новых служб или Web-систем;
- гибкость и высокая оперативность реагирования. Подход к разработке, основанный на независимости создаваемых решений от определенной платформы, устраняет необходимость в привлечении специалистов различной квалификации и специализации, позволяя сформировать универсальную команду бизнес-ориентированных разработчиков, которую можно беспрепятственно переключать с одного проекта на другой в зависимости от потребностей бизнеса.

IBM Rational ClearQuest

IBM Rational ClearQuest – средство для управления запросами на изменения проекта и отслеживание дефектов в проекте на основе средств e–mail и Web–доступа.

ClearQuest является мощным средством управления запросами на изменение (change request management – CRM), специально разработанным с учетом динамической и сложной структуры процесса разработки ПО. ClearQuest отслеживает и управляет любым типом действий, приводящих к изменениям в течение всего жизненного цикла продукта, помогая, тем самым, организациям более предсказуемым (правильным) образом создавать качественное ПО.

Основные задачи, решаемые посредством ClearQuest:

– управлять изменениями, возникающими в ходе процесса разработки ПО.

– оптимизировать путь прохождения запросов на изменения, а также связанные с ним формы и процедуры.

– через World Wide Web поддерживать связь внутри команд, разделенных территориально.

– интегрироваться со средствами конфигурационного управления, такими как Rational’s ClearCase, позволяя создавать связи между запросами на изменение и развитие кода.

Каждый участник проекта может заходить в базу и определять собственные запросы.

Запросы на изменения проходят цикл из нескольких состояний (states), начиная с подачи и заканчивая их разрешением. Например, только что поданный запрос находится в состоянии “Подан” (Submitted). После передачи запроса

сотруднику он переходит в состояние “Назначен” (Assigned). Начало работы над запросом переводит его в “Открытое” состояние (Open), и вся команда может видеть, что кто–то обрабатывает запрос. Наконец, когда запрос проверен и закрыт, он проходит соответственно стадии “Проверка” (Verify) и “Закрыт” (Resolved).

Таким образом, любой участник проекта, от подчиненного до руководителя, может пользоваться базой в собственных целях.

ClearQuest обеспечивает автоматизацию процесса отработки ошибок и запросов на изменение в ходе жизненного цикла, отчетность, контроль состояния и прозрачность проекта для всех его участников:

- схемы жизненного цикла запросов предоставляются в готовом виде и могут быть настроены или созданы заново с учетом потребностей текущей реализации;
- контроль текущего состояния каждого запроса и широкие возможности отчетов обеспечивают прозрачность проекта от его начала до завершения;
- управление всеми материалами и задачами разработки и тестирования в едином интегрированном решении;
- улучшение коммуникаций и координации между разработчиками и тестировщиками;
- настройки безопасности, такие как аутентификация и авторизация пользователей, электронная подпись и аудит помогают соответствовать внутренним и внешним требованиям регуляторов;
- доступ через локальный или Web интерфейс и популярные IDE, включая IBM Rational Application Developer, IBM WebSphere Studio, Microsoft Visual Studio и Eclipse;
- масштабируемость от небольших рабочих групп до географически распределенных корпораций;
- полная интеграция с IBM Rational ClearCase для комплексных решений по управлению конфигурацией и изменениями;
- интеграция с инструментами управления требованиями, разработкой, сборкой, тестированием, развертыванием и упавлением портфелями облегчает коммуникации в организации и обеспечивает контроль по с обратной связью в ходе всего жизненного цикла программных средств;
- автоматизированные рабочие процедуры и оповещения по электронной почте улучшают коммуникации и обеспечивают согласованность групповой работы.

IBM Rational TestManager

IBM Rational TestManager – позволяет централизованно контролировать и управлять всеми задачами по тестированию. Все задачи тестирования и средства функционального, нагрузочного тестирования и тестирования надежности доступны всем разработчикам группы.

IBM Rational Quantify

IBM Rational Quantify – средство количественного определения узких мест, влияющих на общую эффективность работы программы.

Quantify – это простое, но в то же время мощное и гибкое средство учета производительности приложений, инструмент для сбора и анализа информации о производительности созданного программного продукта.

Quantify генерирует в табличной форме список всех вызываемых в процессе работы приложения функций, указывая временные характеристики каждой из них.

Quantify предоставляет полную статистическую выкладку по всем вызовам (внешним и внутренним), невзирая на размеры тестируемого приложения и время его тестирования. Сбор данных осуществляется посредством технологии OCI (Object Code Insertion). Суть способа состоит в подсчете всех машинных циклов путем вставки счетчиков в код для каждого функционального блока тестируемой программы (все циклы приложения просчитываются реально, а не при помощи произвольных выборок, как в большинстве пакетов тестирования). Уникальность данного подхода заключается в том, что, во–первых, тестируется не только сам исходный код, но и все используемые компоненты, (например: библиотеки DLL, системные вызовы), а во–вторых, для подобного анализа совсем необязательно иметь исходные тексты тестируемого приложения (правда, в этом случае нет возможности отслеживать внутренние вызовы).

Статистическая информация по вызовам может быть перенесена в Microsoft Excel, где можно построить как графики, так и сводные таблицы для разных запусков программы.

IBM Rational Purify

IBM Rational Purify – продукт для локализации труднообнаруживаемых Runtime–ошибок программы.

Данный продукт направлен на разрешение всех проблем, связанных с утечками памяти и Runtime–ошибками. Многие программные продукты замыкают на себя во время работы все системные ресурсы без большой на то необходимости. Это случай, который приведет готовую систему к краху в самый ответственный момент. Возникновение подобного рода ошибок достаточно трудно отследить стандартными средствами, имеющимися в арсенале разработчика. И дело тут не только в том, что разработчик может где–то недосмотреть, а в том, что в подавляющем большинстве случаев проектные сроки вынуждают смотреть “сквозь пальцы” на “мелкие” неточности.

В общих чертах работа Purify сводится к выводу детальнейшей статистики об использовании памяти приложением. Программа собирает данные о любых потерях в памяти. К ним можно отнести и банальное невозвращение блока, и не использование указателей, и остановку исполнения программы с выводом состояния среды при возникновении Runtime–ошибки.

Purify предоставляет возможность разработчику не только видеть состояние исполнения (предупреждения, ошибки), но и переходить к соответствующему исходному тексту (естественно, такая возможность существует только для внутренних вызовов, поскольку исходные тексты динамических библиотек пока программистам недоступны).

IBM Rational PureCoverage

IBM Rational PureCoverage – средство идентификации участков кода, пропущенных при тестировании.

Основное и единственное назначение продукта – выявление участков кода, пропущенного при тестировании приложения. Вполне очевидно, что при тестировании программы специалисту не удается оттестировать абсолютно все ее функции. А невозможно это, как правило, по двум причинам: во–первых, разработчик не может сделать все абсолютно правильно с учетом всех возможных нюансов, во–вторых, даже учитывая все возможные реакции приложения на внешние “раздражители” невозможно на 100% быть уверенным в том, что все оттестировано.

PureCoverage собирает статистику о тех участках программы, которые во время тестирования не были выполнены (пройдены). Дополнительно программа считает активно исполнявшиеся строки. Стало быть, разработчик может оценить не просто, сколько раз вызывалась та или иная функция, а сколько раз исполнилась каждая строка, составляющая ее. Имея подобную статистику, очень просто выявить не исполнившиеся строки и проанализировать причину, по которой они не получили управления.

Подобные строки PureCoverage подсвечивает красным цветом, четко указывая на наличие черных дыр в программе в виде не оттестированного кода.

PureCoverage позволяет организовать тестирование по одному из критериев белого ящика – выполнить все операторы в программе хотя бы один раз (это самый слабый критерий тестирования по критерию «белого ящика»).

IBM Rational Robot

IBM Rational Robot – модуль, предназначенный для разработки, записи и выполнения автоматизированных тестов для сборочного, функционального и регрессионного тестирования приложений. Обеспечивает накопление и повторное использование тестовых процедур.

Robot – средство функционального тестирования, базирующееся на объектно–ориентированной технологии, что позволяет существенно превзойти традиционные средства GUI–тестирования (тестирования графического интерфейса), так как здесь тестируются сотни и тысячи свойств всех объектов приложения (даже скрытых объектов), как вместе, так и каждого в отдельности.

Программа способна работать в двух режимах: в автоматическом и ручном. В ручном режиме пользователь сам задает на специальном языке (SQABasic) сценарий тестирования (скрипт), в автоматическом – пользователь тестирует приложение, а Robot автоматически генерирует необходимый скрипт для дальнейшего повторного тестирования.

Скрипты, создаваемые в Rational Robot, обеспечивают поиск ошибок в приложении, оставаясь виртуально независимыми от внесенных изменений и платформы. Объектное тестирование обеспечивает быстрое создание скриптов,

которые в дальнейшем можно легко изменять, создавать заново и воспроизводить. Rational Robot поддерживает широкий спектр языков программирования и ERP–решений. Rational Robot позволяет редактировать, отлаживать и настраивать скрипты. Допускает также тестирование сложных систем клиент/сервер на платформе Windows.

Robot предназначен для команд, выполняющих функциональное тестирования клиент-серверных приложений облегчает процесс обучения тестировщиков навыкам автоматизированного тестирования и позволяет опытным тестировщикам обнаруживать больше ошибок путем расширения тестовых скриптов условными переходами, покрывая большее количество функционала приложений.

Также возможно определять в теле тестов вызовы внешних библиотек DLLs или программ, обеспечивает работу с такими общими объектами, как меню, кнопки, списки и изображения, и специальные сценарии тестирования для специфических объектов среды разработки.

Имеет встроенные средства управления тестированием и интегрируется с инструментами, использующимися в IBM Rational Unified Process для процессов отслеживания ошибок, управления изменениями и управления требованиями, поддерживает все основные технологии интерфейса пользователя, от Java, Web и всех элементов управления (controls) VS.NET до приложений Oracle Forms, Borland Delphi и Sybase PowerBuilder.

IBM Rational Manual Tester

Средство подготовки и выполнения тестирования, повышающее производительность, охват и надежность ручного тестирования. Позволяет использовать рабочие материалы (планы, скрипты и т.п.) в распределенной среде, повторно использовать наработки в новых задачах:

- позволяет повторно использовать шаги тестирования для уменьшения воздействия изменений в ПО на подготовку новых тестовых испытаний;
- интеграция с Rational ClearQuest позволяет регистрацию ошибок в ClearQuest прямо из среды Manual Tester при создании, выполнении и в ходе анализа результатов тестирования;
- облегчает ввод и верификацию данных при тестировании и сокращает количество ошибок;
- импортирует существующие тестовые примеры из Microsoft Word и Excel;
- экспортирует результаты тестирования в файлы формата CSV, обеспечивая возможность их анализа с использованием приложений других производителей;
- входит в состав IBM Rational Functional Tester, предоставляя командам тестирования возможности автоматизированного и ручного тестирования;
- поддерживает работу через Citrix Server в географически распределенной среде.

IBM Rational Functional Tester

Мощное средство функционального тестирования для приложений Java, Web, VS.NET и WinForm, автоматизирующее процессы функционального и регрессионного тестирования:

- обеспечивает возможность автоматизации для тестирования, управляемого данными (datadriven testing) и выбор языка разработки скриптов вместе с мощным редактором для создания и настройки скриптов;
- облегчает начинающим тестировщикам автоматизацию тестирования благодаря таким возможностям, как тестирование, управляемое данными;
- предоставляет выбор языка разработки скриптов для опытных тестировщиков: Java для Eclipse или Microsoft Visual Basic .NET для Visual Studio .NET;
- дополнительно имеются инструменты функционального и регрессионного тестирования для работы с графическим интерфейсом пользователя (GUI);
- включает технологию ScriptAssure и возможности использования шаблонов для улучшения устойчивости скриптов к внесению частых незначительных изменений в пользовательский интерфейс;
- поддерживает контроль версий и параллельную разработку использование скриптов в том числе для географически распределенных команд;
- поддерживает тестирование приложений на базе 3270 (zSeries) и 5250 (iSeries) при использовании дополнительно IBM Rational Functional Tester Extension for Terminal-based Applications;
- при использовании расширений (Extention) беспечивает автоматизированное функциональное и регрессионное тестирование для приложений на основе Siebel 7.7, 7.8 и SAP;
- Поддерживает специализированные средства управления через промежуточную среду разработки (Java/.Net).

IBM Rational Performance Tester

Rational Performance Tester является многопользовательским инструментом тестирования, предназначенным для проверки масштабируемости приложений перед их развертыванием:

- создает и выполняет автоматизированные проверки, анализирует их результаты на предмет надежности бизнес-приложений. Имеет дополнительные расширения для Siebel и SAP Solutions;
- поддержка Windows, Linux и z/OS на уровне распределенных агентов;
- предоставляет высокоуровневый и детальный обзор тестов сширокими возможностями редактирования;
- выполняет планирование нагрузочного тестирования для оптимизации инвестиций в ИТ-инфраструктуру;
- предоставляет автоматическую идентификацию и поддерживает динамическую работу с сервером;

- обеспечивает многопользовательский режим тестирования с использованием минимальных ресурсов оборудования;
- интегрируется со средствами Tivoli для управления приложениями для идентификации источников проблем с производительностью.

IBM Rational Performance Tester Extension for SQL

Это расширение обеспечивает тестирование производительности серверов, предоставляющих SQL-интерфейсы клиентам в многопользовательских системах.

Модуль SQL-протокола, который может быть приобретен отдельно, обеспечивает непосредственное тестирование производительности серверов.

Могут использоваться те же наборы IBM Rational Performance Tester для тестирования Web-приложений.

IBM Rational Performance Tester Extension for SOA Quality

ПО IBM Rational Performance Tester Extension for SOA Quality предназначено для специалистов, которым необходимо выполнять нагрузочные испытания и тестирование производительности для Web-сервисов.

Это расширение дополняет ПО IBM Rational Tester for SOA и IBM Rational Performance Tester, обеспечивая выполнение нагрузочных тестов и тестирование производительности для SOA-приложений.

Помимо всех возможностей ПО Rational Tester for SOA Quality позволяет выполнять:

- Проверку масштабируемости систем SOA;
- Гибкое моделирование рабочих нагрузок для автоматизированного создания тестового клиента для Web-сервисов;
- Автоматизированное создание тестов производительности для Web-сервисов;
- Формирование отчетов о времени отклика и пропускной способности серверов в реальном времени.
- Обнаружение причин снижения производительности и определение проблем с эффективностью SOA благодаря поддержке мониторинга обширного спектра платформ для развернутых Web-сервисов, а также сбору и визуализации данных о ресурсах серверов;
- Гибкую настройку тестов путем включения Java-кода, обеспечивающего дополнительные возможности для анализа данных и синтаксического анализа.

IBM Rational Suite

IBM Rational Suite – это наборы основных программных продуктов, направленных на покрытие одного или нескольких этапов разработки программного обеспечения. Данный подход вполне оправдывает себя, поскольку, например, команде аналитиков незачем переплачивать за средства тестирования, которые им не нужны, и наоборот – тестировщикам ни к чему ставить Rose Modeler для проектирования баз данных. Второе преимущество наборов заключается в том,

что их легче устанавливать и администрировать, поскольку продукты идут комплектом, а не разрозненно.

В качестве базовых средств для организации процессов управления требованиями, визуального моделирования, функционального тестирования и управления изменениями предлагается использовать ролевые наборы инструментов, на пример: IBM Rational Suite Team Unifying Platform – руководитель проекта; IBM Rational Suite Enterprise, IBM Rational Suite Development Studio – системные аналитики и разработчики проекта, и генерального подрядчика по разработке ПС и ИС; и IBM Rational Suite TestStudio – группа тестирования и другие.

IBM Rational Suite обеспечивает:

Объединение усилий менеджеров, аналитиков, разработчиков и тестировщиков, снимая барьеры, которые существуют между ними, в том числе и при распределенной разработке

Оптимизацию работы исполнителей проекта, предоставляя набор инструментальных средств для выполнения специфических задач каждого члена команды

Простоту установки, поддержки и совместного использования продуктов, входящих в IBM Rational Suite, включая их обновление

Экономию средств – общая стоимость IBM Rational Suite значительно ниже стоимости приобретения по отдельности продуктов, входящих в пакет IBM Rational Suite

Ниже приведено краткое описание предлагаемых ролевых наборов IBM Rational Suite:

DevelopmentStudio – обеспечивает визуальное моделирование информационных систем, предоставляет необходимые инструменты для проектирования и создания программ высокого качества. Это средство, доступное на платформах Windows и UNIX, предназначенное для аналитиков, проектировщиков и разработчиков ИС

TestStudio – обеспечивает поддержку всех видов автоматического функционального и нагрузочного тестирования и тестирования надежности в течение всего жизненного цикла разработки информационной системы. Предназначено как для менеджеров проектов, тестировщиков приложений и проектировщиков тестов, так и для остальных членов тестовой команды

Enterprise – объединяет в себе весь спектр продуктов IBM Rational Software. Обеспечивает поддержку полного жизненного цикла разработки информационной системы. Ориентировано на использование, как менеджерами проектов, так и аналитиками, разработчиками и тестировщиками.

Для реализации конфигурационного управления и организации репозитория проектных материалов предлагается использовать комбинацию (Bundle) лицензий ролевых наборов инструментов с лицензиями инструментального средства IBM Rational.

Для создания инфраструктуры разработки и оснащения рабочих мест разработчиков представляется целесообразным использовать ролевой набор

инструментов IBM Rational Suite Development Studio в комбинации с инструментом конфигурационного управления ClearCase (80% от полного числа рабочих мест). Этот набор содержит оптимальный комплект инструментов для реализации процессов управления требованиями и изменениями, визуального моделирования, анализа и проектирования, конфигурационного управления и управления проектами разработки и сопровождения ПС и ИС.

Инструментальные средства IBM Rational позволяют автоматизировать все процессы, описанные в методологии RUP, обеспечивают выполнение проектов на платформах Windows, UNIX, Linux и частично на Mainframe. Эти инструменты поддерживают широкий спектр языков и сред разработки, включая Java, Eclipse, C/C++/C#, Visual Basic, J2EE, .NET, Microsoft.NET, COM/+, CORBA и программные оболочки для разработки встроенного ПО и Realtime–приложений.

Более полная информация приведена на официальном сайте:
http://www-01.ibm.com/software/ru/rational/

3. Методология функционального моделирования IDEF0

Постоянное усложнение производственно–технических и организационно экономических систем – фирм, предприятий, производств, и др. субъектов производственно–хозяйственной деятельности – и необходимость их анализа с целью совершенствования функционирования и повышения эффективности обусловливают необходимость применения специальных средств описания и анализа таких систем. Эта проблема приобретает особую актуальность в связи с появлением интегрированных компьютеризированных производств и автоматизированных предприятий.

В США это обстоятельство было осознано еще в конце 70–ых годов, когда ВВС США предложили и реализовали Программу интегрированной компьютеризации производства ICAM (ICAM – Integrated Computer Aided Manufacturing), направленную на увеличение эффективности промышленных предприятий посредством широкого внедрения компьютерных (информационных) технологий.

Реализация программы ICAM потребовала создания адекватных методов анализа и проектирования производственных систем и способов обмена информацией между специалистами, занимающимися такими проблемами. Для удовлетворения этой потребности в рамках программы ICAM была разработана методология IDEF (ICAM Definition), позволяющая исследовать структуру, параметры и характеристики производственно–технических и организационно–экономических систем. Общая методология IDEF состоит из трех частных методологий моделирования, основанных на графическом представлении систем:

– IDEF0 используется для создания функциональной модели, отображающей структуру и функции системы, а также потоки информации и материальных объектов, связывающие эти функции.

– IDEF1 применяется для построения информационной модели, отображающей структуру и содержание информационных потоков, необходимых для поддержки функций системы;

– IDEF3 позволяет построить динамическую модель меняющихся во времени поведения функций, информации и ресурсов системы.

К настоящему времени наибольшее распространение и применение имеют методологии IDEF0, IDEF1 (IDEF1X) и IDEF3 получившие в США статус федеральных стандартов.

Методология IDEF0 основана на подходе, разработанном Дугласом Т. Россом в начале 70–ых годов и получившем название SADT (Structured Analysis & Design Technique – метод структурного анализа и проектирования). Основу подхода и, как следствие, методологии IDEF0, составляет графический язык описания (моделирования) систем, обладающий следующими свойствами:

– Графический язык – полное и выразительное средство, способное наглядно представлять широкий спектр деловых, производственных и других процессов и операций предприятия на любом уровне детализации.

– Язык обеспечивает точное и лаконичное описание моделируемых объектов, удобство использования и интерпретации этого описания.

– Язык облегчает взаимодействие и взаимопонимание системных аналитиков, разработчиков и персонала изучаемого объекта (фирмы, предприятия), т.е. служит средством «информационного общения» большого числа специалистов и рабочих групп, занятых в одном проекте, в процессе обсуждения, рецензирования, критики и утверждения результатов.

– Язык прошел многолетнюю проверку и продемонстрировал работоспособность, как в проектах ВВС США, так и в других проектах, выполнявшихся государственными и частными промышленными компаниями.

– Язык легок и прост в изучении и освоении.

– Язык может генерироваться рядом инструментальных средств машинной графики; известны коммерческие программные продукты, поддерживающие разработку и анализ моделей – диаграмм IDEF0, например, продукт Design/IDEF 3.7 (и более поздние версии) фирмы Meta Software Corporation (США).

Перечисленные свойства языка предопределили выбор методологии IDEF0 в качестве базового средства анализа и синтеза производственно–технических и организационно–экономических систем, что нашло свое отражение в упомянутых федеральных стандартах США.

В Российской Федерации стандарт IDEF0 адаптирован в руководящий документ «Методология функционального моделирования IDEF0» (РД IDEF0–2000). Основные определения пунктов 3.1–3.7 используют информацию из данных стандартов.

3.1. Концепция методологии функционального моделирования IDEF0

M моделирует **A**, если **M** отвечает на вопросы относительно **A**. Здесь **M** – модель, **A** – моделируемый объект (оригинал).

Модель разрабатывают для понимания, анализа и принятия решений о реконструкции или замене существующей, либо проектировании новой системы.

Система представляет собой совокупность взаимосвязанных и взаимодействующих частей, выполняющих некоторую полезную работу. Частями (элементами) системы могут быть любые комбинации разнообразных сущностей, включающие людей, информацию, программное обеспечение, оборудование, изделия, сырье или энергию (энергоносители). Модель описывает, что происходит в системе, как ею управляют, какие сущности она преобразует, какие средства использует для выполнения своих функций и что производит.

Блочное моделирование и его графическое представление. Основной концептуальный принцип методологии IDEF – представление любой изучаемой системы в виде набора взаимодействующих и взаимосвязанных блоков, отображающих процессы, операции, действия, происходящие в изучаемой системе. В IDEF0 все, что происходит в системе и ее элементах, принято называть функциями. Каждой функции ставится в соответствие блок. На IDEF0–диаграмме

блок представляет собой прямоугольник. Интерфейсы, посредством которых блок взаимодействует с другими блоками или с внешней по отношению к моделируемой системе средой, представляются стрелками, входящими в блок или выходящими из него. Входящие стрелки показывают, какие условия должны быть одновременно выполнены, чтобы функция, описываемая блоком, осуществилась.

Лаконичность и точность. Документация, описывающая систему, должна быть точной и лаконичной. Многословные характеристики, изложенные в форме традиционных текстов, неудовлетворительны. Графический язык позволяет лаконично, однозначно и точно показать все элементы (блоки) системы и все отношения и связи между ними, выявить ошибочные, лишние или дублирующие связи и т.д.

Передача информации. Средства IDEF0 облегчают передачу информации от одного участника разработки модели (отдельного разработчика или рабочей группы) к другому. К числу таких средств относятся:

– диаграммы, основанные на простой графике блоков и стрелок, легко читаемые и понимаемые;

– метки на естественном языке для описания блоков и стрелок, а также глоссарий и сопроводительный текст для уточнения смысла элементов диаграммы;

– последовательная декомпозиция диаграмм, строящаяся по иерархическому принципу, при котором на верхнем уровне отображаются основные функции, а затем происходит их детализация и уточнение;

– древовидные схемы иерархии диаграмм и блоков, обеспечивающие обозримость модели в целом и входящих в нее деталей.

Строгость и формализм. Разработка моделей IDEF0 требует соблюдения ряда строгих формальных правил, обеспечивающих преимущества методологии в отношении однозначности, точности и целостности сложных многоуровневых моделей. Все стадии и этапы разработки и корректировки модели должны строго, формально документироваться с тем, чтобы при ее эксплуатации не возникало вопросов, связанных с неполнотой или некорректностью документации.

Итеративное моделирование. Разработка модели в IDEF0 представляет собой пошаговую, итеративную процедуру. На каждом шаге итерации разработчик предлагает вариант модели, который подвергают обсуждению, рецензированию и последующему редактированию, после чего цикл повторяется. Такая организация работы способствует оптимальному использованию знаний системного аналитика, владеющего методологией и техникой IDEF0, и знаний специалистов – экспертов в предметной области, к которой относится объект моделирования.

Отделение «организации» от «функций». При разработке моделей следует избегать изначальной «привязки» функций исследуемой системы к существующей организационной структуре моделируемого объекта (предприятия, фирмы). Это помогает избежать субъективной точки зрения, навязанной организацией и ее руководством. Организационная структура должна явиться результатом использования (применения) модели. Сравнение результата с существующей структурой позволяет, во–первых, оценить адекватность модели, а во–вторых – предложить решения, направленные на совершенствование этой структуры.

3.2. Основные определения (понятия) методологии и языка IDEF0

Блок: прямоугольник, содержащий имя и номер и используемый для описания функции.

Ветвление: разделение стрелки на два или большее число сегментов. Может означать «развязывание пучка».

Внутренняя стрелка: входная, управляющая или выходная стрелка, концы которой связывают источник и потребителя, являющиеся блоками одной диаграммы. Отличается от граничной стрелки.

Входная стрелка: класс стрелок, которые отображают вход IDEF0–блока, то есть данные или материальные объекты, которые преобразуются функцией в выход. Входные стрелки связываются с левой стороной блока

Выходная стрелка: класс стрелок, которые отображают выход IDEF0–блока, то есть данные или материальные объекты, произведенные функцией. Выходные стрелки связываются с правой стороной блока IDEF0.

Глоссарий: список определений для ключевых слов, фраз и аббревиатур, связанных с узлами, блоками, стрелками или с моделью IDEF0 в целом.

Граничная стрелка: стрелка, один из концов которой связан с источником или потребителем, а другой не присоединен ни к какому блоку на диаграмме. Отображает связь диаграммы с другими блоками системы и отличается от внутренней стрелки.

Декомпозиция: разделение моделируемой функции на функции – компоненты.

Дерево узлов: представление отношений между родительскими и дочерними узлами модели IDEF0 в форме древовидного графа. Имеет то же значение и содержание, что и перечень узлов.

Диаграмма А–0: специальный вид (контекстной) диаграммы IDEF0, состоящей из одного блока, описывающего функцию верхнего уровня, ее входы, выходы, управления, и механизмы, вместе с формулировками цели модели и точки зрения, с которой строится модель.

Диаграмма: часть модели, описывающая декомпозицию блока.

Диаграмма–иллюстрация (FEO): графическое описание, используемое, для сообщения специфических фактов о диаграмме IDEF0. При построении диаграмм FEO можно не придерживаться правила IDEF0.

Дочерний блок: блок на дочерней (порожденной) диаграмме.

Дочерняя диаграмма: диаграмма, детализирующая родительский (порождающий) блок.

Имя блока: глагол или глагольный оборот, помещенный внутри блока и описывающий моделируемую функцию.

Интерфейс: разделяющая граница, через которую проходят данные или материальные объекты; соединение между двумя или большим числом компонентов модели, передающее данные или материальные объекты от одного компонента к другому.

Код ICOM: аббревиатура (Input – Вход, Control – Управление, Output – Выход, Mechanism – Механизм), код, обеспечивающий соответствие граничных стрелок дочерней диаграммы со стрелками родительского блока; используется для ссылок.

Контекст: окружающая среда, в которой действует функция (или комплект функций на диаграмме).

Контекстная диаграмма: диаграмма, имеющая узловой номер A–n (n ≥ 0), которая представляет контекст модели, Диаграмма A–0, состоящая из одного блока, является необходимой (обязательной) контекстной диаграммой; диаграммы с узловыми номерами A–1, A–2,... – дополнительные контекстные диаграммы.

Метка стрелки: существительное или оборот существительного, связанные со стрелкой или сегментом стрелки и определяющие их значение.

Модель IDEF0: графическое описание системы, разработанное с определенной целью и с выбранной точки зрения. Комплект одной или более диаграмм IDEF0, которые изображают функции системы с помощью графики, текста и глоссария.

Номер блока: число (0–6), помещаемое в правом нижнем углу блока и однозначно идентифицирующее блок на диаграмме.

Перечень узлов: список, часто ступенчатый, показывающий узлы модели IDEF0 в упорядоченном виде. Имеет то же значение и содержание, что и дерево узлов.

Примечание к модели: текстовый комментарий, являющийся частью диаграммы IDEF0 и используемый для записи факта, не нашедшего графического изображения.

Родительская диаграмма: диаграмма, которая содержит родительский блок.

Родительский блок: блок, который подробно описывается дочерней диаграммой.

Связывание/развязывание: объединение значений стрелок в составное значение (связывание в «пучок»), или разделение значений стрелок (развязывание «пучка»), выраженные синтаксисом слияния или ветвления стрелок.

Сегмент стрелки: сегмент линии, который начинается или заканчивается на стороне блока, в точке ветвления или слияния, или на границе (несвязанный конец стрелки).

Семантика: значение синтаксических компонентов языка.

Синтаксис: Структурные компоненты или характеристики языка и правила, которые определяют отношения между ними.

Слияние: объединение двух или большего числа сегментов стрелок в один сегмент. Может означать «развязывание пучка».

C–номер: номер, создаваемый в хронологическом порядке и используемый для идентификации диаграммы и прослеживания ее истории; может быть использован в качестве ссылочного выражения при определении конкретной версии диаграммы.

Стрелка: направленная линия, состоящая из одного или нескольких сегментов, которая моделирует открытый канал или канал, передающий данные или материальные объекты от источника (начальная точка стрелки), к потребителю

(конечная точка с «наконечником»). Имеется 4 класса стрелок: входная стрелка, выходная стрелка, управляющая стрелка, стрелка механизма (включает стрелку вызова).

Стрелка вызова: вид стрелки механизма, который обозначает обращение из блока данной модели (или части модели) к блоку другой модели (или другой части той же модели) и обеспечивает связь между моделями или между разными частями одной модели.

Стрелка механизма: класс стрелок, которые отображают механизмы IDEF0, то есть средства, используемые для выполнения функции; включает специальный случай стрелки вызова. Стрелки механизмов связываются с нижней стороной блока IDEF0.

Стрелка, помещенная в туннель (туннельная стрелка): стрелка (со специальной нотацией), не удовлетворяющая обычному требованию, согласно которому каждая стрелка на дочерней диаграмме должна соответствовать стрелкам на родительской диаграмме.

Текст: любой текстовый (не графический) комментарий к графической диаграмме IDEF0.

Тильда: небольшая ломаная (волнистая) линия, используемая для соединения метки с конкретным сегментом стрелки или примечания модели с компонентом диаграммы.

Точка зрения: указание на должностное лицо или подразделение организации, с позиции которого разрабатывается модель.

Узел: блок, порождающий дочерние блоки; родительский блок.

Узловая ссылка: код, присвоенный диаграмме, для ее идентификации и определения положения в иерархии модели; формируется из сокращенного имени модели и узлового номера диаграммы с дополнительными расширениями.

Узловой номер диаграммы: часть узловой ссылки диаграммы, которая соответствует номеру родительского блока.

Узловой номер: код, присвоенный блоку и определяющий его положение в иерархии модели; может быть использован в качестве подробного ссылочного выражения.

Управляющая стрелка: класс стрелок, которые в IDEF0 отображают управления, то есть условия, при выполнении которых выход блока будет правильным. Данные или объекты, моделируемые как управления, могут преобразовываться функцией, создающей соответствующий выход. Управляющие стрелки связываются с верхней стороной блока IDEF0.

Функция: деятельность, процесс или преобразование (моделируемые блоком IDEF0), идентифицируемое глаголом или глагольной формой, которая описывает, что должно быть выполнено.

Цель: краткая формулировка причины создания модели.

3.3. Синтаксис графического языка IDEF0

Набор структурных компонентов языка, их характеристики и правила, определяющие связи между компонентами, представляют собой синтаксис языка. Компоненты синтаксиса IDEF0 – блоки, стрелки, диаграммы и правила. Блоки представляют функции, определяемые как деятельность, процесс, операция, действие или преобразование. Стрелки представляют данные или материальные объекты, связанные с функциями. Правила определяют, как следует применять компоненты; диаграммы обеспечивают формат графического и словесного описания моделей. Формат образует основу для управления конфигурацией модели.

Блок. Блок описывает функцию. Типичный блок показан на рис. 3.1. Внутри каждого блока помещается его имя и номер. Имя должно быть активным глаголом или глагольным оборотом, описывающим функцию. Номер блока размещается в правом нижнем углу. Номера блоков используются для их идентификации на диаграмме и в соответствующем тексте. Размеры блоков должны быть достаточными для того, чтобы включить имя блока. Блоки должны быть прямоугольными, с прямыми углами. Блоки должны быть нарисованы сплошными линиями.

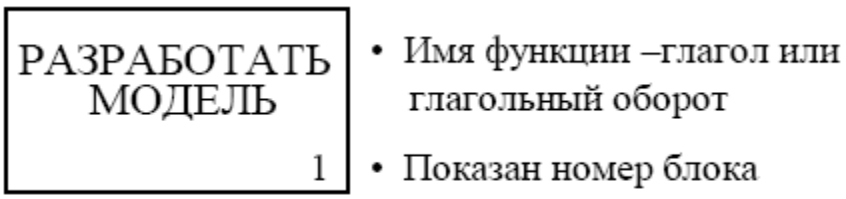


Рис. 3.1. Пример блока

Стрелка. Стрелка формируется из одного или более отрезков прямых и наконечника на одном конце. Как показано на рис. 3.2., сегменты стрелок могут быть прямыми или ломаными; в последнем случае горизонтальные и вертикальные отрезки стрелки сопрягаются дугами, имеющими угол 90 градусов. Стрелки не представляют поток или последовательность событий, как в традиционных блок–схемах потоков или процессов. Они лишь показывают, какие данные или материальные объекты должны поступить на вход функции для того, чтобы эта функция могла выполняться.

Стрелки должны быть нарисованы сплошными линиями различной толщины. Стрелки могут состоять только из вертикальных или горизонтальных отрезков; отрезки, направленные по диагонали, не допускаются. Концы стрелок должны касаться внешней границы функционального блока, но не должны пересекать ее. Стрелки должны присоединяться к блоку на его сторонах. Присоединение в углах не допускается.

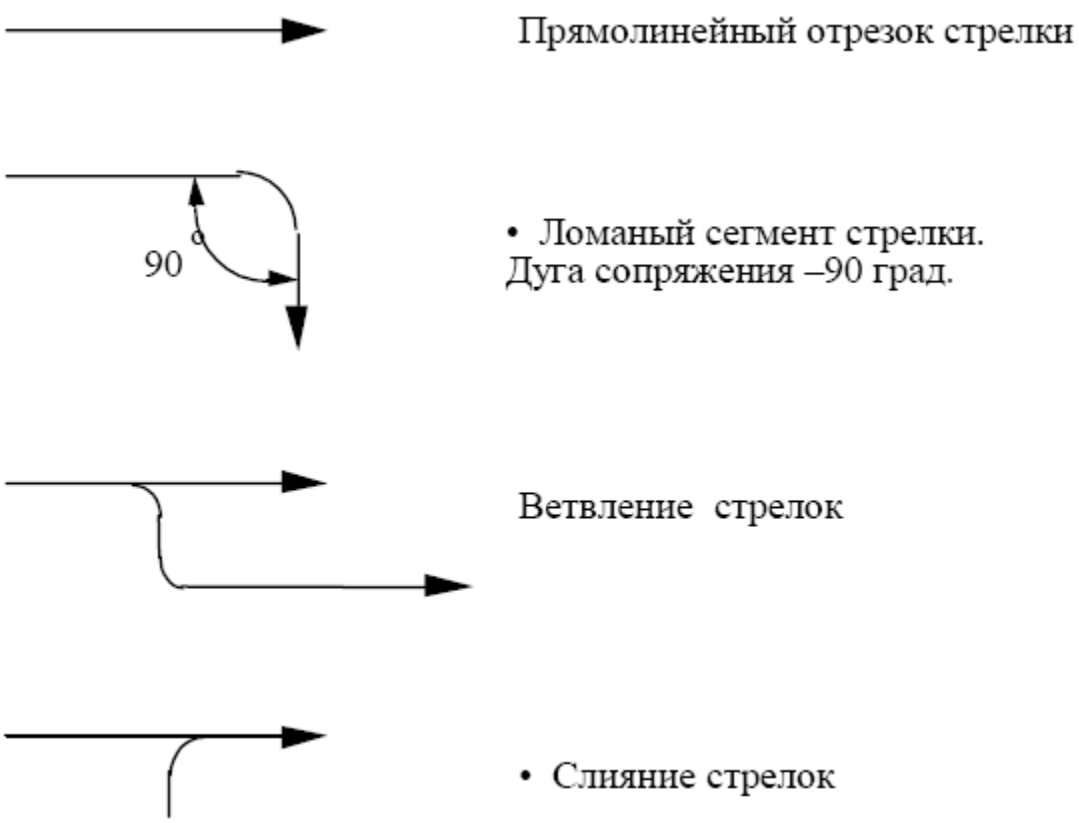


Рис. 3.2. Синтаксис стрелок

3.4. Семантика языка IDEF0

Семантика определяет содержание (значение) синтаксических компонентов языка и способствует правильности их интерпретации. Интерпретация устанавливает соответствие между блоками и стрелками с одной стороны и функциями и их интерфейсами – с другой.

Поскольку IDEF0 есть методология функционального моделирования, имя блока, описывающее функцию, должно быть глаголом или глагольным оборотом; например, имя блока "Выполнить проверку", означает, что блок с таким именем превращает непроверенные детали в проверенные. После присваивания блоку имени, к соответствующим его сторонам присоединяются входные, выходные и управляющие стрелки, а также стрелки механизма, что и определяет наглядность и выразительность изображения блока IDEF0.

Чтобы гарантировать точность модели, следует использовать стандартную терминологию. Блоки именуются глаголами или глагольными оборотами и эти имена сохраняются при декомпозиции Стрелки и их сегменты, как отдельные, так и связанные в «пучок», помечаются существительными или оборотами существительного. Метки сегментов позволяют конкретизировать данные или материальные объекты, передаваемые этими сегментами, с соблюдением синтаксиса ветвлений и слияний.

Каждая сторона функционального блока имеет стандартное значение с точки зрения связи блок/стрелки. В свою очередь, сторона блока, к которой присоединена стрелка, однозначно определяет ее роль.

Стрелки, входящие в левую сторону блока – входы. Входы преобразуются или расходуются функцией, чтобы создать то, что появится на ее выходе.

Стрелки, входящие в блок сверху – управления. Управления определяют условия, необходимые функции, чтобы произвести правильный выход.

Стрелки, покидающие блок справа – выходы, т.е. данные или материальные объекты, произведенные функцией.

Стрелки, подключенные к нижней стороне блока, представляют механизмы.

Стрелки, направленные вверх, идентифицируют средства, поддерживающие выполнение функции.

Другие средства могут наследоваться из родительского блока. Стрелки механизма, направленные вниз, являются стрелками вызова. Стрелки вызова обозначают обращение из данной модели или из данной части модели к блоку, входящему в состав другой модели или другой части модели, обеспечивая их связь, т.е. разные модели или разные части одной и той же модели могут совместно использовать один и тот же элемент (блок).

Стандартное расположение стрелок показано на рис. 3.3.

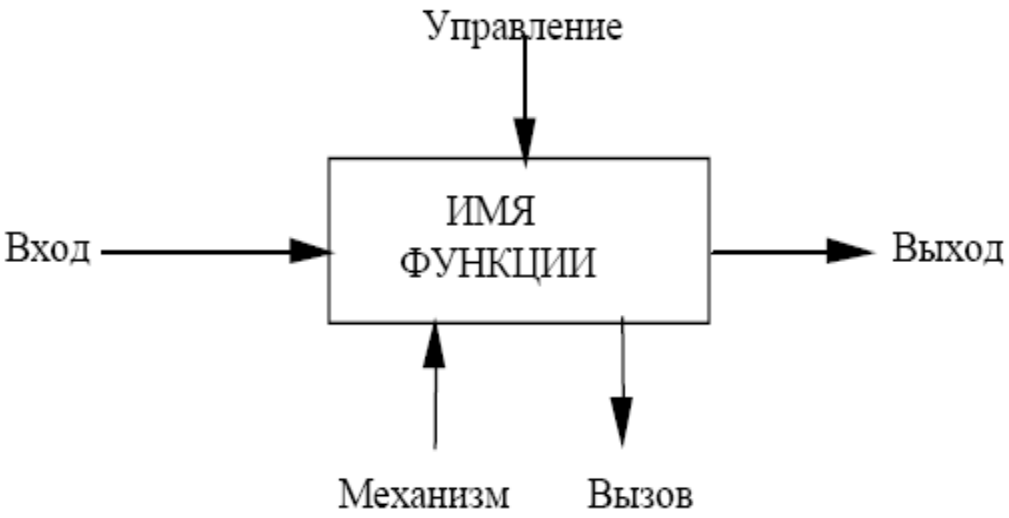


Рис. 3.3. Расположение стрелок

Пример функциональной диаграммы этапа "Выполнить проектирование" приведена на рисунке 3.3.1. Этап «Выполнить проектирование» (Design product)

может состоять из двух фаз: «Предварительное проектирование» (разрабатывается эскизный проект) и «Детальное проектирование» (разрабатывается технический проект). «Эскизный проект» разрабатывается только в том случае, когда его необходимость оговорена в ТЗ или документе, его заменяющем. Цветными стрелками выделены: вход, выход, управление, механизм)

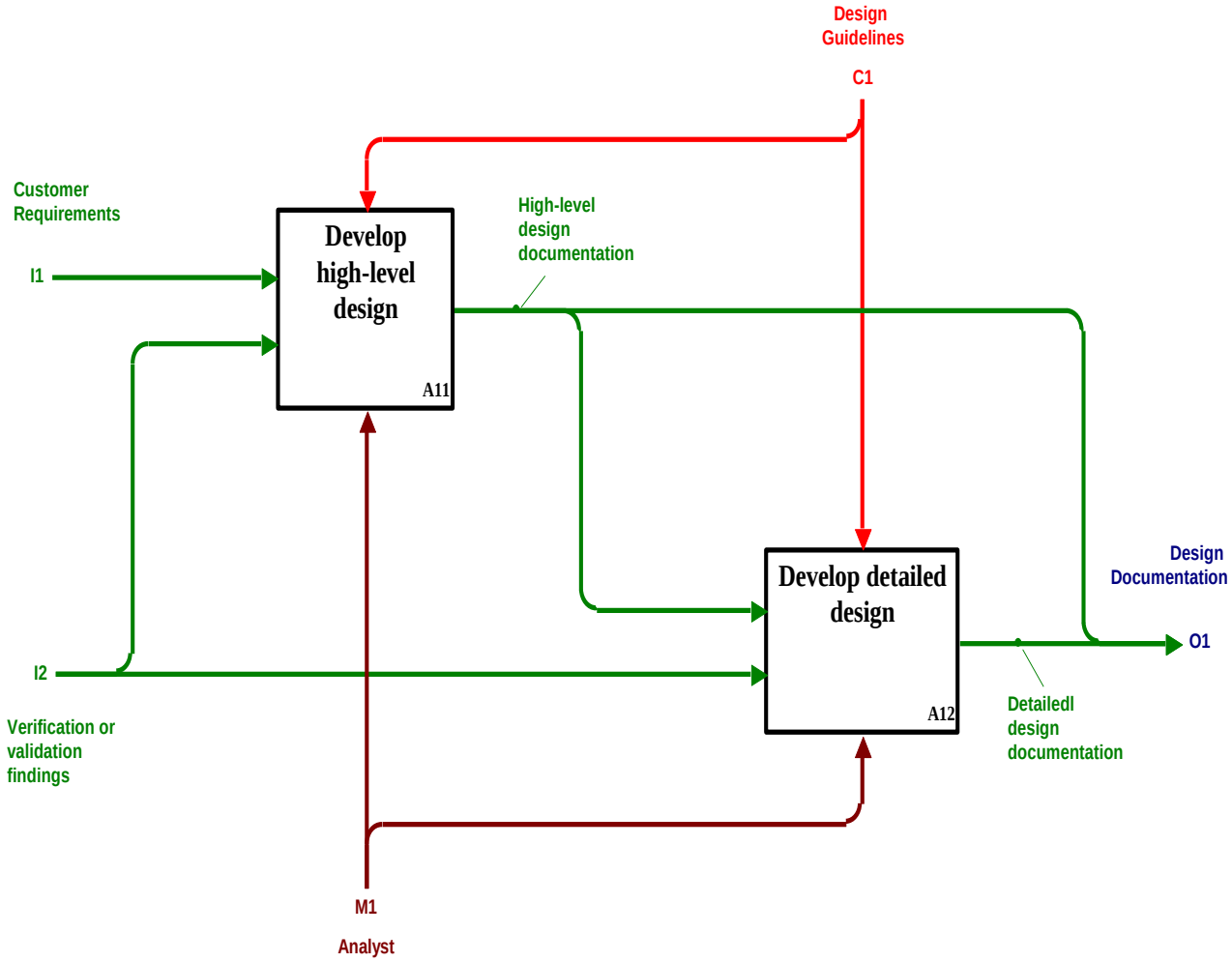


Рис.3.3.1. Фазы этапа «Выполнить проектирование»

3.5. Имена и метки

Имя блока должно быть активным глаголом или глагольным оборотом. Каждая сторона функционального блока должна иметь стандартное отношение блок/стрелки:

а) входные стрелки должны связываться с левой стороной блока;

б) управляющие стрелки должны связываться с верхней стороной блока;

в) выходные стрелки должны связываться с правой стороной блока;

г) стрелки механизма (кроме стрелок вызова) должны указывать вверх и подключаться к нижней стороне блока.

д) стрелки вызова механизма должны указывать вниз, подключаться к нижней стороне блока, и помечаться ссылкой на вызываемый блок.

Сегменты стрелок, за исключением стрелок вызова, должны помечаться существительным или оборотом существительного, если только единственная метка стрелки, несомненно, не относится к стрелке в целом. В метках стрелок не должны использоваться следующие термины: функция, вход, управление, выход, механизм, вызов.

Чтобы связать стрелку с меткой, следует использовать "тильду".

Как указывалось, имена функций – глаголы или глагольные обороты. Примеры таких имен: производить детали, планировать ресурсы, наблюдать за выполнением, проектировать систему, эксплуатировать, разработать детальные чертежи, изготовить компонент, проверять деталь.

Стрелки идентифицируют данные или материальные объекты, необходимые для выполнения функции или производимые ею. Каждая стрелка должна быть помечена существительным или оборотом существительного, например: спецификации, отчет об испытаниях, бюджет, конструкторские требования, конструкция, детали, директива, инженер–конструктор, плата в сборе, требования.

Пример размещения меток стрелок и имени блока показан на рис. 3.4.

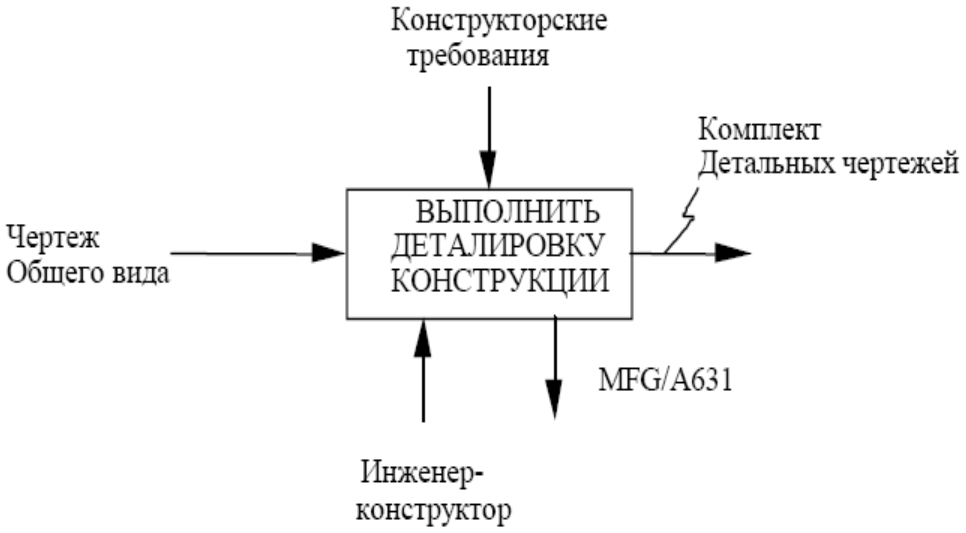


Рис. 3.4. Пример размещения меток стрелок и имени блока

3.6. Отношения блоков на диаграммах

В методологии IDEF0 существует 6 (шесть) типов отношений между блоками в пределах одной диаграммы:

– доминирование;
– управление;
– выход – вход;
– обратная связь по управлению;
– обратная связь по входу;
– выход – механизм.

Первое из перечисленных отношений определяется взаимным расположением блоков на диаграмме. Предполагается, что блоки, расположенные на диаграмме выше и левее, «доминируют» над блоками, расположенными ниже и правее. «Доминирование» понимается как влияние, которое один блок оказывает на другие блоки диаграммы.

Остальные пять отношений описывают связи между блоками и изображаются соответствующими стрелками.

Отношения управления и выход – вход являются простейшими, поскольку отражают прямые взаимодействия, которые понятны и очевидны.

Отношение управления (рис. 3.5.) возникает тогда, когда выход одного блока служит управляющим воздействием на блок с меньшим доминированием.

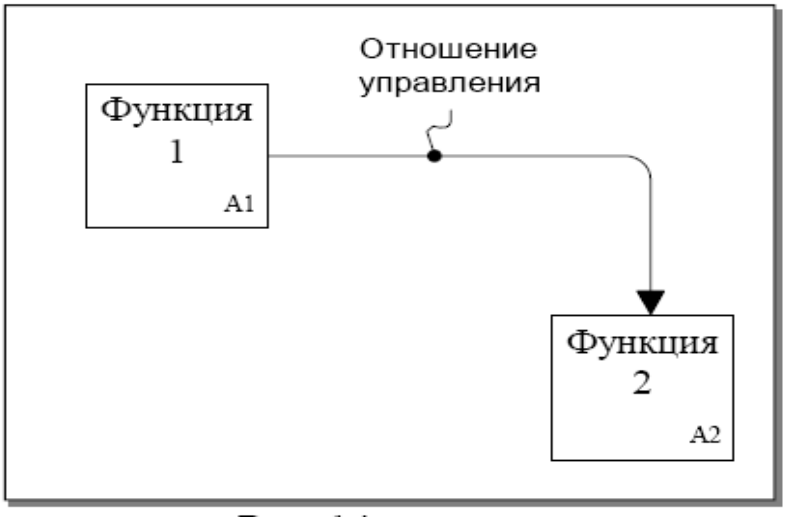


Рис. 3.5. Отношение управления

Отношение выход – вход (рис. 3.6.) возникает при соединении выхода одного блока с входом другого блока с меньшим доминированием.

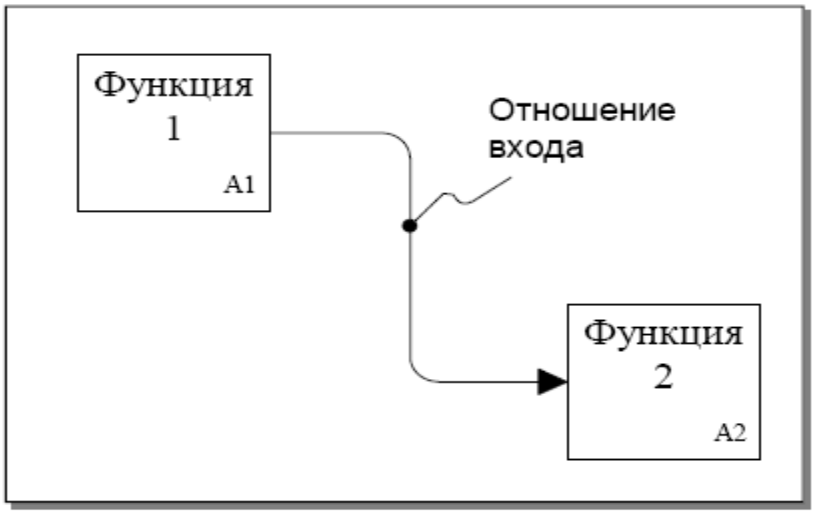


Рис. 3.6. Отношение входа

Обратная связь по управлению и обратная связь по входу являются более сложными типами отношений, поскольку они представляют итерацию (выход функции влияет на будущее выполнение других функций с большим доминированием, что впоследствии влияет на исходную функцию). Обратная связь по управлению (рис. 3.7.) возникает тогда, когда выход некоторого блока создает управляющее воздействие на блок с большим доминированием.

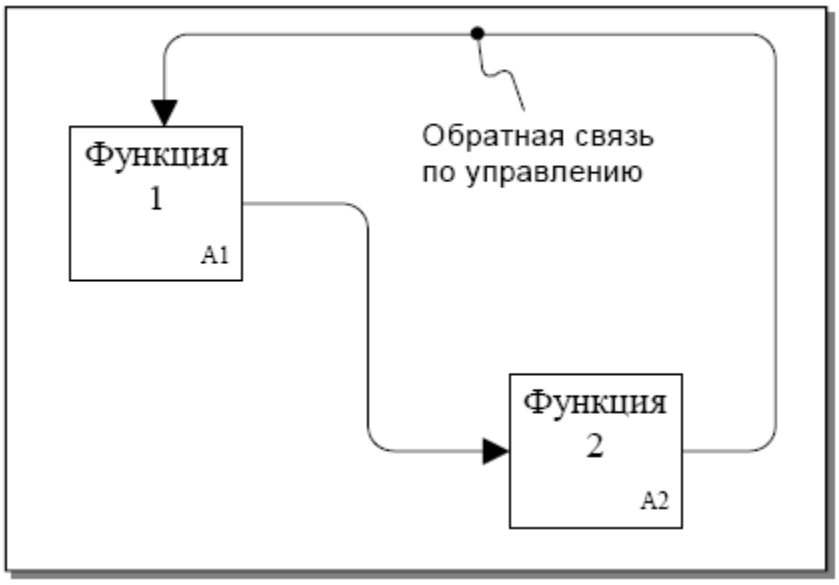


Рис. 3.7. Обратная связь по управлению

Отношение обратной связи по входу (рис. 3.8.) имеет место тогда, когда выход блока становиться входом другого блока с большим доминированием.

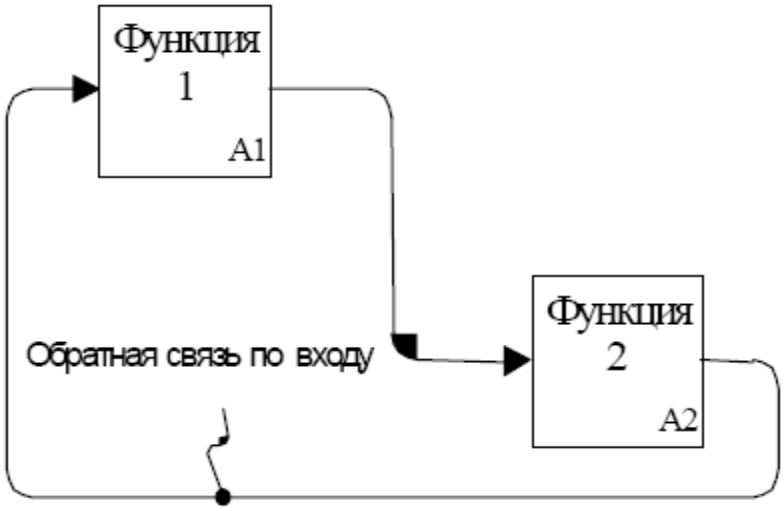


Рис. 3.8. Обратная связь по входу

Связи «выход – механизм» (рис. 3.9.) отражают ситуацию, при которой выход одной функции становиться средством достижения цели для другой. Связи «выход – механизм» возникают при отображении в модели процедур пополнения и распределения ресурсов, создания или подготовки средств для выполнения функций системы (например, приобретение или изготовление требуемых инструментов и оборудования, обучение персонала, организация физического пространства, финансирование, закупка материалов).

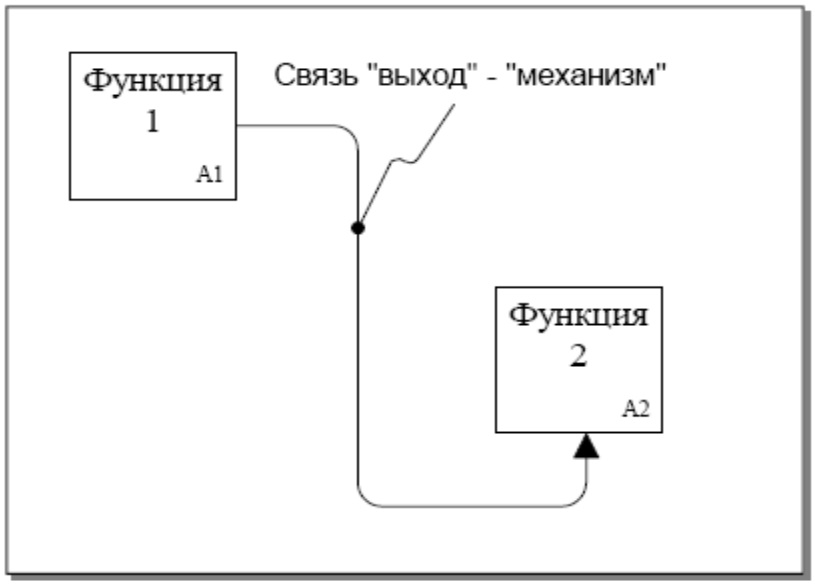


Рис. 3.9. Связь «выход – механизм»

3.7. Диаграммы IDEF0

IDEF0–модели состоят из трех типов документов: графических диаграмм, текста и глоссария. Эти документы имеют перекрестные ссылки друг на друга.

Диаграмме может быть поставлен в соответствие структурированный текст, представляющий собой краткий комментарий к содержанию диаграммы. Текст используется для объяснений и уточнений характеристик, потоков, внутриблочных соединений и т.д. Текст не должен использоваться для описания и без того понятных блоков и стрелок на диаграммах.

Глоссарий предназначен для определения аббревиатур (акронимов), ключевых слов и фраз, используемых в качестве имен и меток на диаграммах. Глоссарий определяет понятия и термины, которые должны быть одинаково понимаемы всеми участниками разработки и пользователями модели, чтобы правильно интерпретировать ее содержание.

Графическая диаграмма – главный компонент IDEF0–модели, содержащий блоки, стрелки, соединения блоков и стрелок и ассоциированные с ними отношения. Блоки представляют основные функции моделируемого объекта. Эти функции могут быть разбиты (декомпозированы) на составные части и представлены в виде более подробных диаграмм; процесс декомпозиции продолжается до тех пор, пока объект не будет описан на уровне детализации, необходимом для достижения целей конкретного проекта. Диаграмма верхнего уровня обеспечивает наиболее общее или абстрактное описание объекта моделирования. За этой диаграммой следует серия дочерних диаграмм, дающих более детальное представление об объекте.

Каждая модель должна иметь контекстную диаграмму верхнего уровня, на которой объект моделирования представлен единственным блоком с граничными стрелками. Эта диаграмма называется А–0 (А минус нуль). Стрелки на этой диаграмме отображают связи объекта моделирования с окружающей средой. Поскольку единственный блок представляет весь объект, его имя общее для всего проекта. Это же справедливо и для всех стрелок диаграммы, поскольку они представляют полный комплект внешних интерфейсов объекта. Диаграмма А–0 устанавливает область моделирования и ее границу. Пример диаграммы А–0 показан на рис. 3.10., рис. 3.11., рис. 3.12.

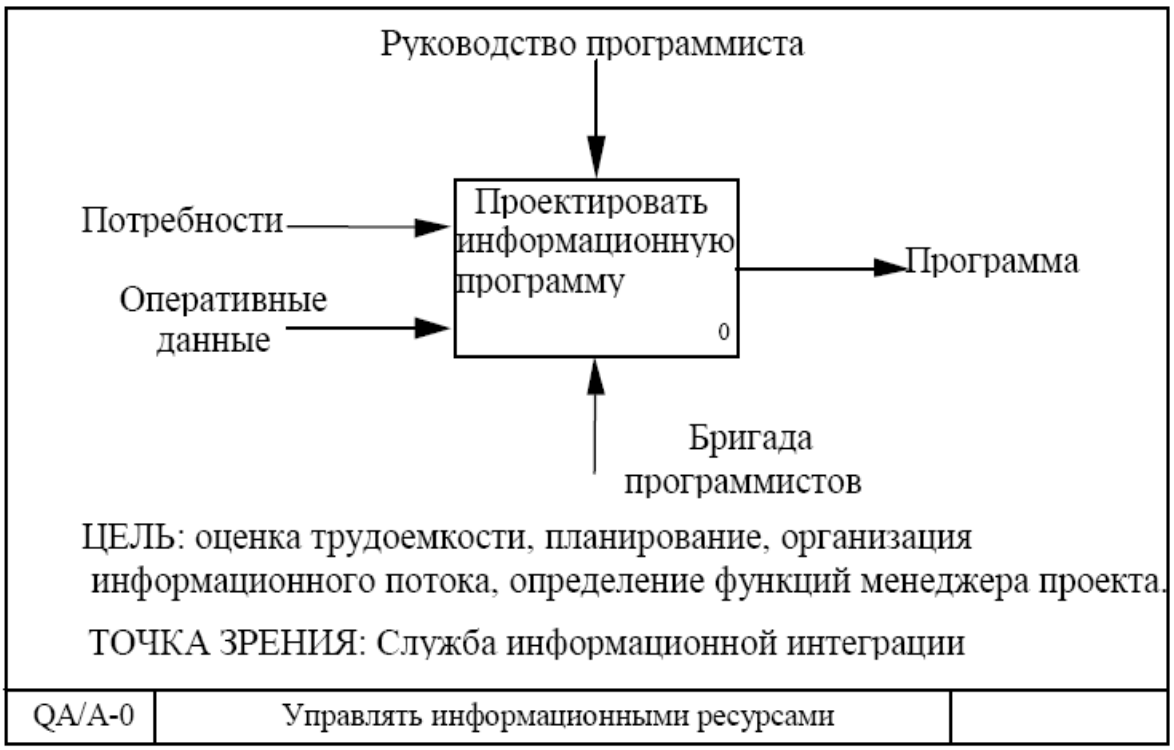


Рис. 3.10. Пример контекстной диаграммы

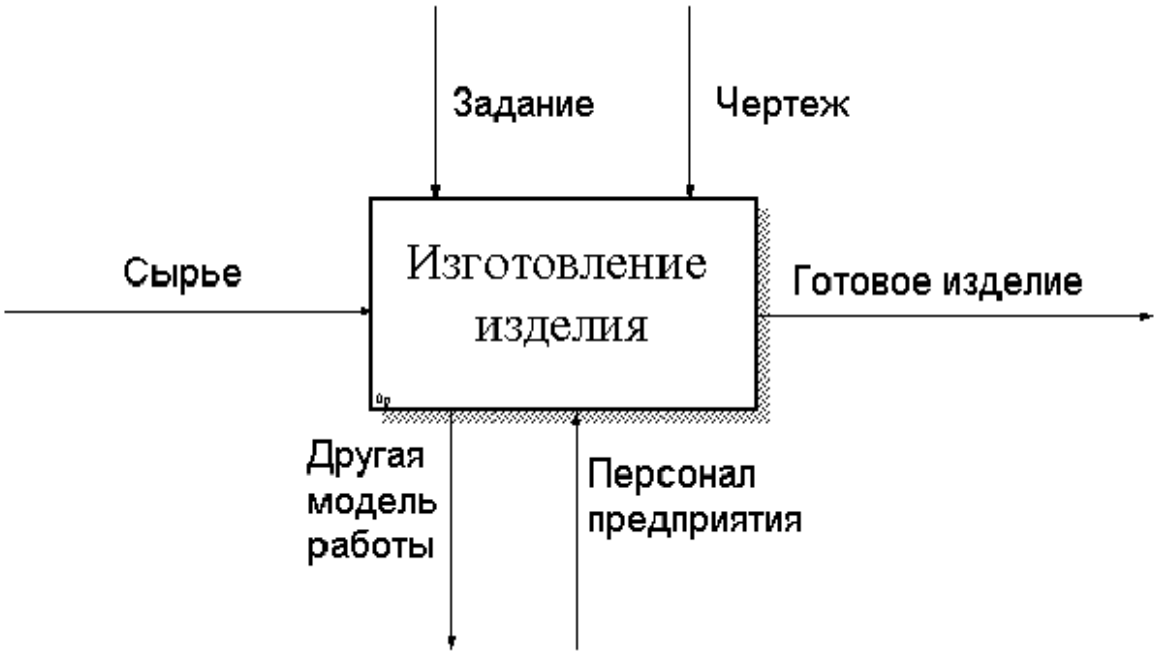


Рис. 3.11. Пример контекстной диаграммы

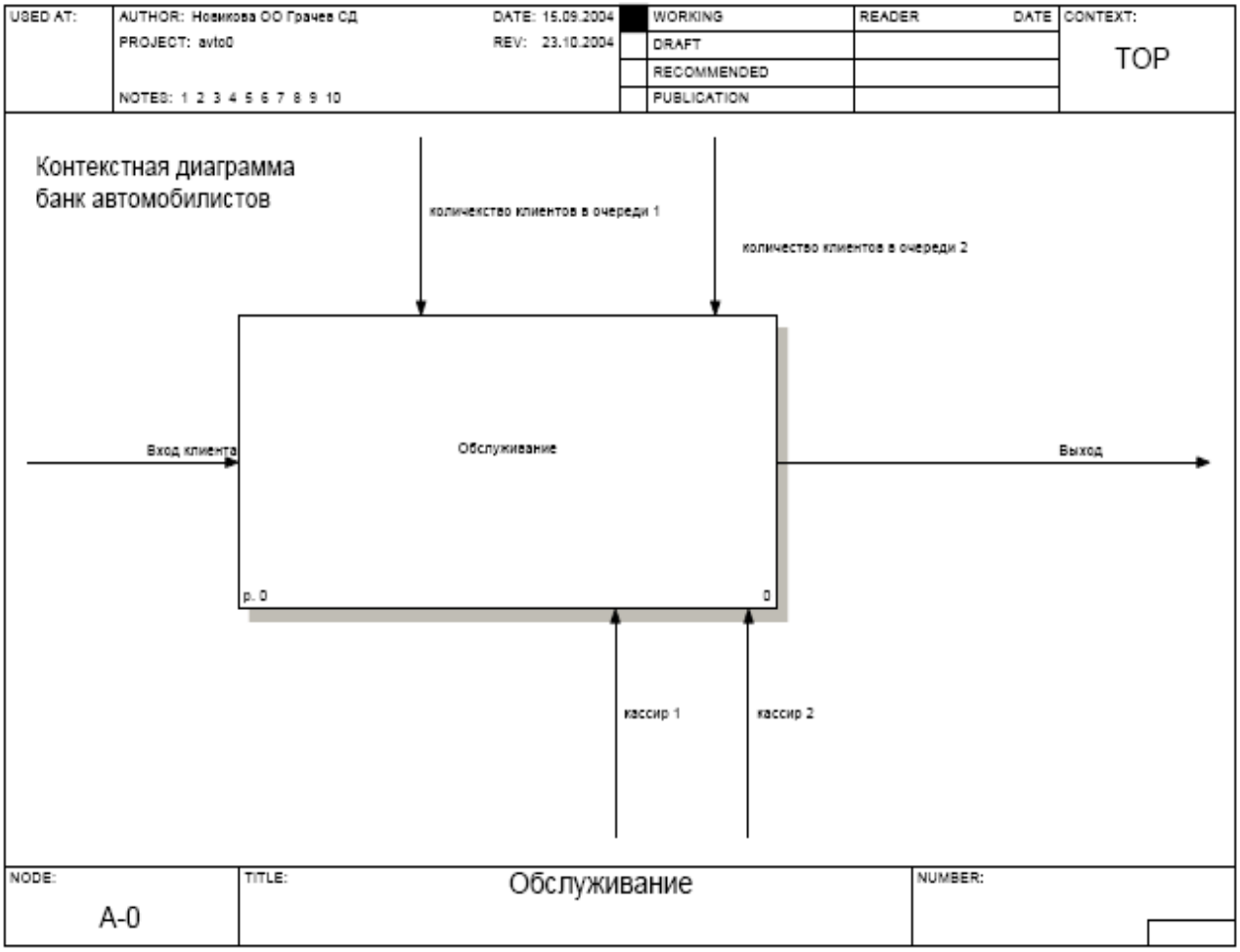


Рис. 3.12. Пример контекстной диаграммы

Контекстная диаграмма А–0 также должна содержать краткие утверждения, определяющие точку зрения должностного лица или подразделения, с позиций которого создается модель, и цель, для достижения которой ее разрабатывают. Эти утверждения помогают руководить разработкой модели и ввести этот процесс в определенные рамки.

Точка зрения определяет, что и в каком разрезе можно увидеть в пределах контекста модели. Изменение точки зрения, приводит к рассмотрению других аспектов объекта. Аспекты, важные с одной точки зрения, могут не появиться в модели, разрабатываемой с другой точки зрения на тот же самый объект.

Формулировка цели выражает причину создания модели, т.е. содержит перечень вопросов, на которые должна отвечать модель, что в значительной мере определяет ее структуру.

Наиболее важные свойства объекта обычно выявляются на верхних уровнях иерархии; по мере декомпозиции функции верхнего уровня и разбиения ее на подфункции, эти свойства уточняются. Каждая подфункция, в свою очередь, декомпозируется на элементы следующего уровня, и так происходит до тех пор, пока не будет получена релевантная структура, позволяющая ответить на вопросы, сформулированные в цели моделирования. Каждая подфункция моделируется отдельным блоком. Каждый родительский блок подробно описывается дочерней

диаграммой на более низком уровне. Все дочерние диаграммы должны быть в пределах области контекстной диаграммы верхнего уровня.

3.8. Дочерняя диаграмма

После создания контекстной диаграммы можно приступить к декомпозиции.

Единственная функция, представленная на контекстной диаграмме верхнего уровня, может быть разложена на основные подфункции посредством создания дочерней диаграммы. В свою очередь, каждая из этих подфункций может быть разложена на составные части посредством создания дочерней диаграммы следующего, более низкого уровня, на которой некоторые или все функции также могут быть разложены на составные части. Каждая дочерняя диаграмма содержит дочерние блоки и стрелки, обеспечивающие дополнительную детализацию родительского блока.

Дочерняя диаграмма, создаваемая при декомпозиции, охватывает ту же область, что и родительский блок, но описывает ее более подробно. Таким образом, дочерняя диаграмма как бы вложена в свой родительский блок. Эта структура иллюстрируется рис. 3.13., рис. 3.14., рис. 3.15.

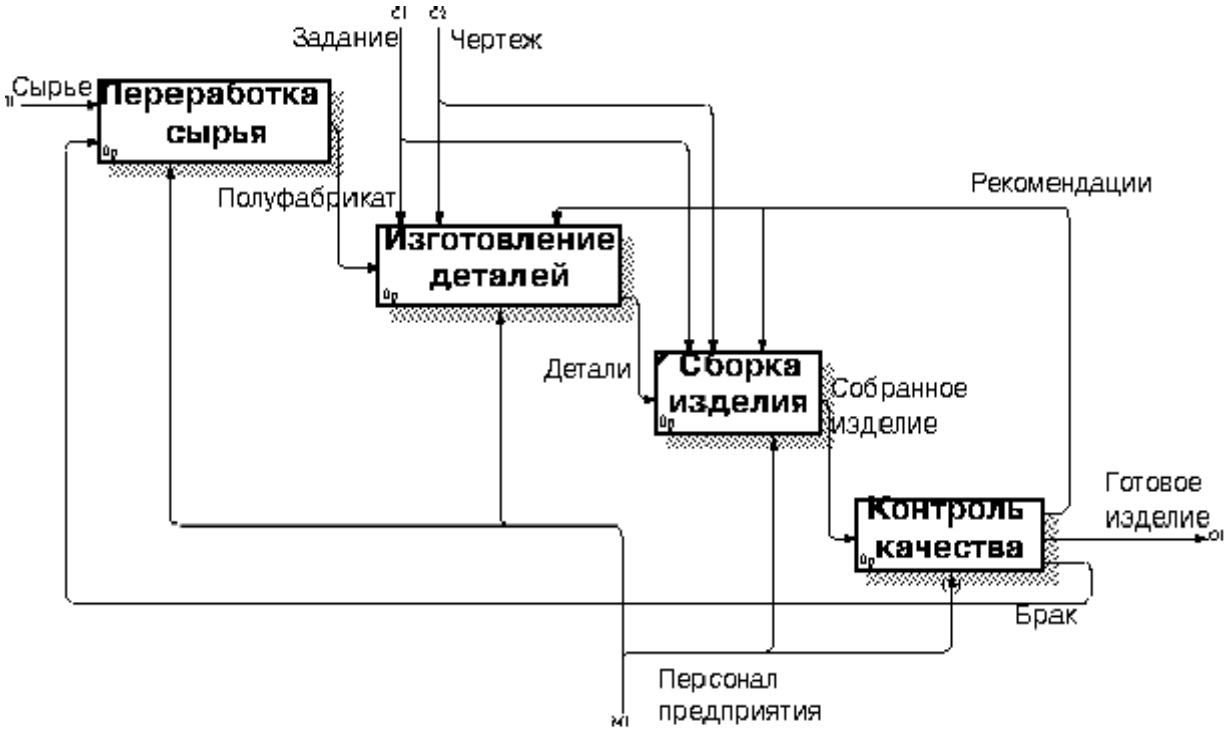


Рис. 3.13. Пример декомпозиции контекстной диаграммы рис. 3.11.

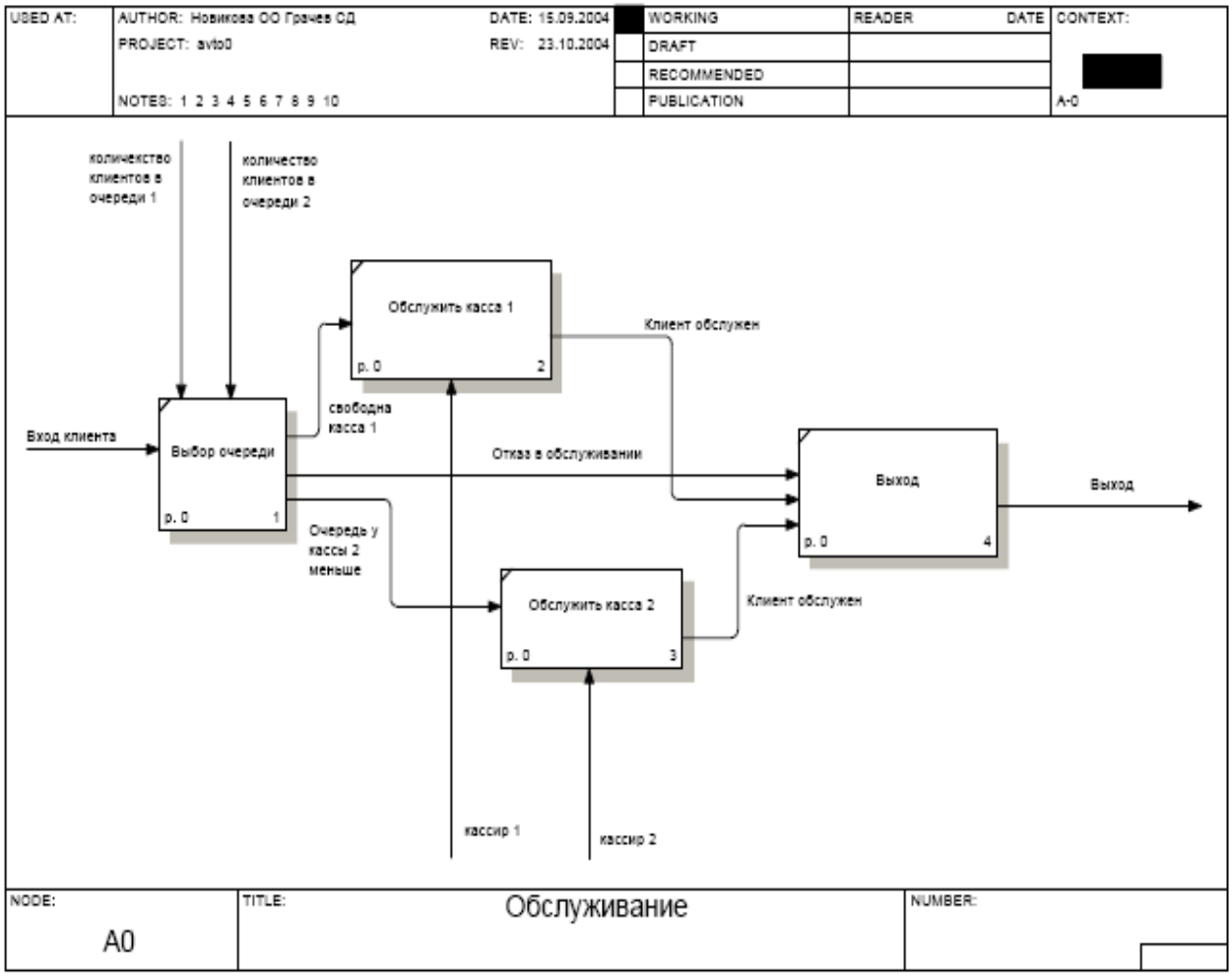


Рис. 3.14. Пример декомпозиции контекстной диаграммы рис. 3.12.

3.9. Родительская диаграмма

Родительская диаграмма – та, которая содержит один или более родительских блоков. Каждая обычная (не–контекстная) диаграмма является также дочерней диаграммой, поскольку, по определению, она подробно описывает некоторый родительский блок. Таким образом, любая диаграмма может быть как родительской диаграммой (содержать родительские блоки), так и дочерней (подробно описывать собственный родительский блок). Аналогично, блок может быть как родительским (подробно описываться дочерней диаграммой) так и дочерним (появляющимся на дочерней диаграмме). Основное иерархическое отношение существует между родительским блоком и дочерней диаграммой, которая его подробно описывает (рис. 3.15.).

То, что блок является дочерним и раскрывает содержание родительского блока на диаграмме предшествующего уровня, указывается специальным ссылочным кодом, написанным ниже правого нижнего угла блока. Этот ссылочный код может формироваться несколькими способами, из которых самый простой заключается в том, что код, начинающийся с буквы А (по имени диаграммы А–0), содержит цифры, определяемые номерами родительских блоков. Например, показанные на рис. 3.16. коды означают, что диаграмма является декомпозицией 1–го блока

диаграммы, которая, в свою очередь является декомпозицией 6–го блока диаграммы A0, а сами коды образуются присоединением номера блока.

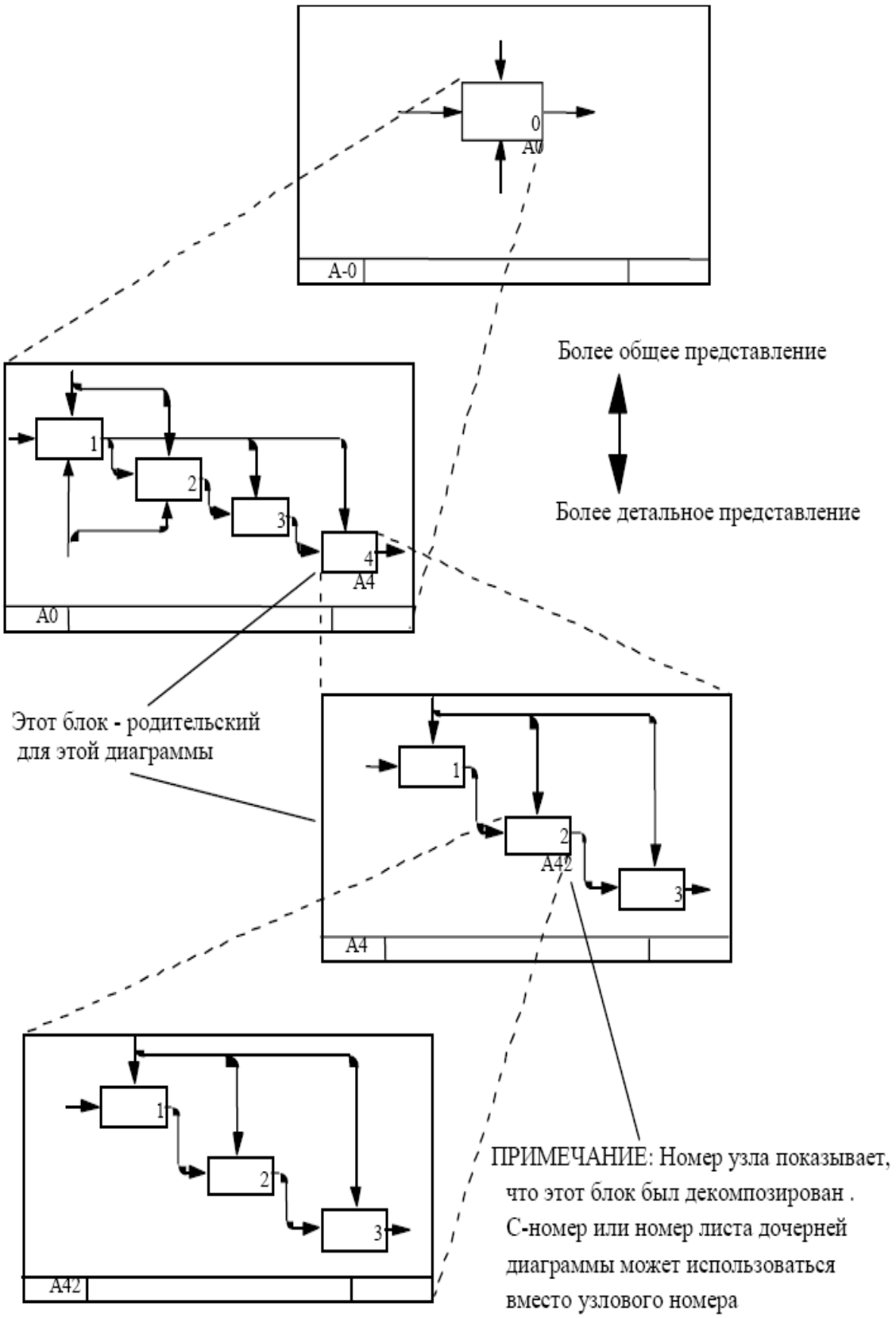


Рис. 3.15. Иерархия отношений

Узлы дерева проектирования отражают функциональность проектируемой системы (см. рис. 3.15.1.).

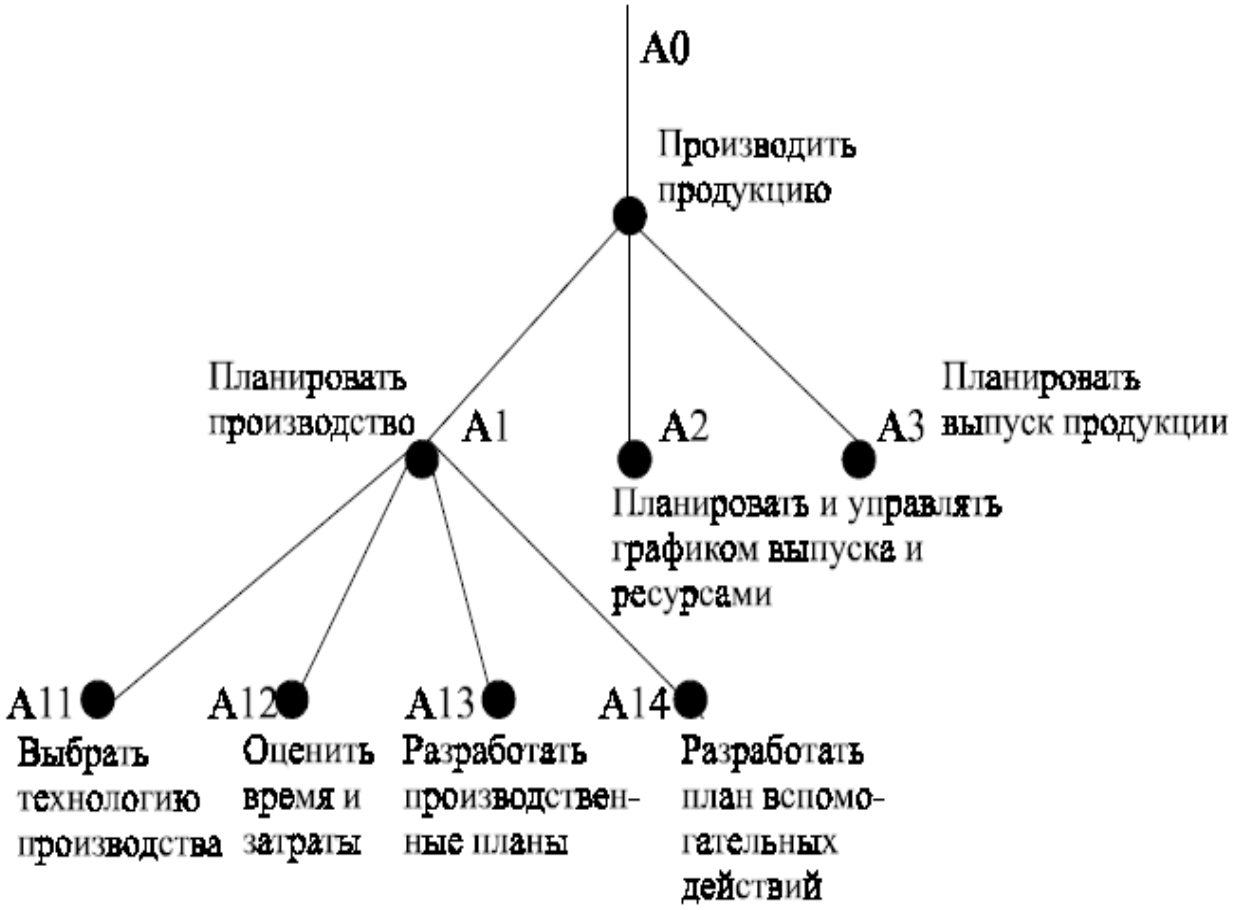


Рис. 3.15.1. Узлы дерева

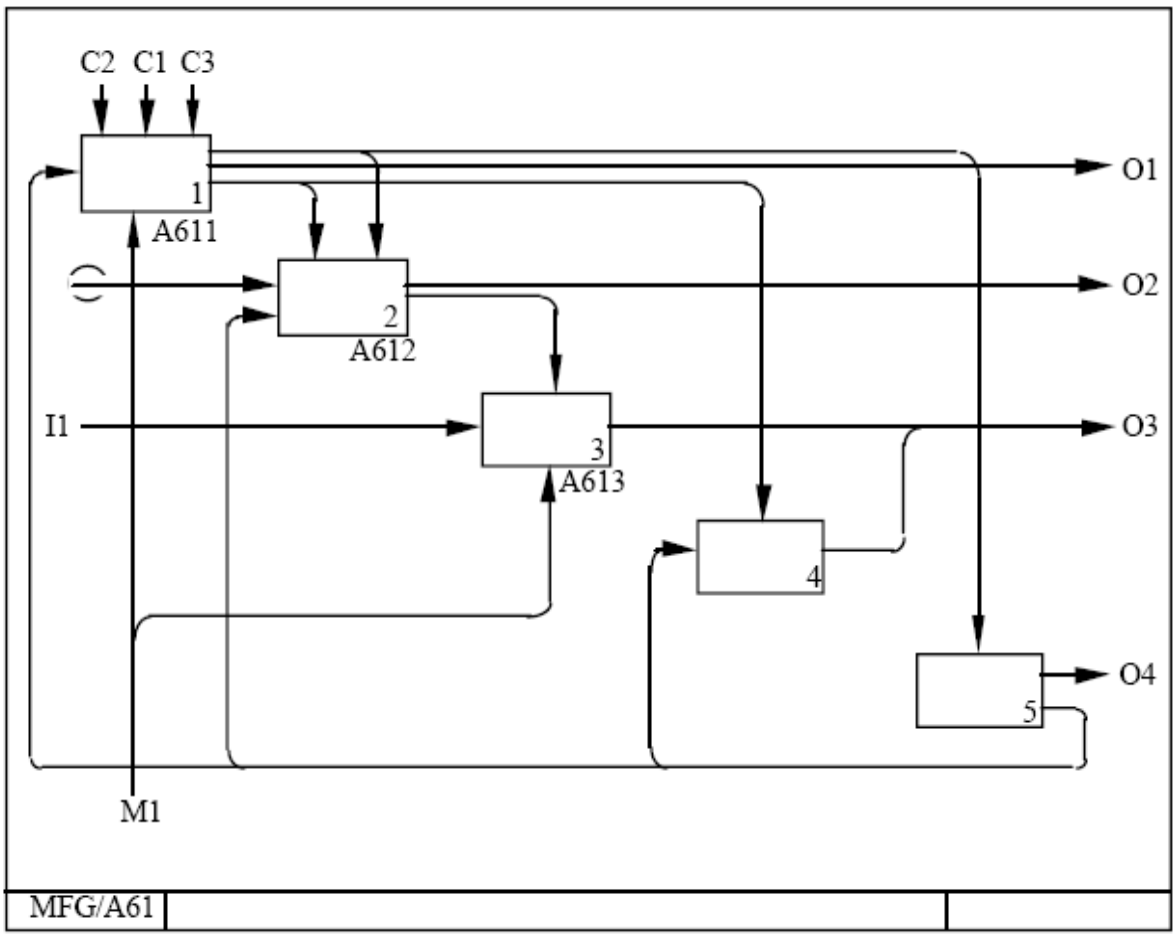


Рис. 3.16. Ссылочные коды

Таким образом, код формируется так:

```
A 6 1 * * * *
| | | | |_____________ и т.д.
| | | |_______________ Номер блока на диаграмме A61
| | |_________________Номер блока на диаграмме A6
| |___________________ Номер блока на диаграмме A0
|_____________________ Имя блока A0
```

3.10. Свойства диаграмм

3.10.1. Стрелки как ограничения

Стрелки на диаграмме IDEF0, представляя данные или материальные объекты, одновременно задают своего рода ограничения (условия). Входные и управляющие стрелки блока, соединяющие его с другими блоками или с внешней средой, по сути, описывают условия, которые должны быть выполнены для того, чтобы реализовалась функция, записанная в качестве имени блока.

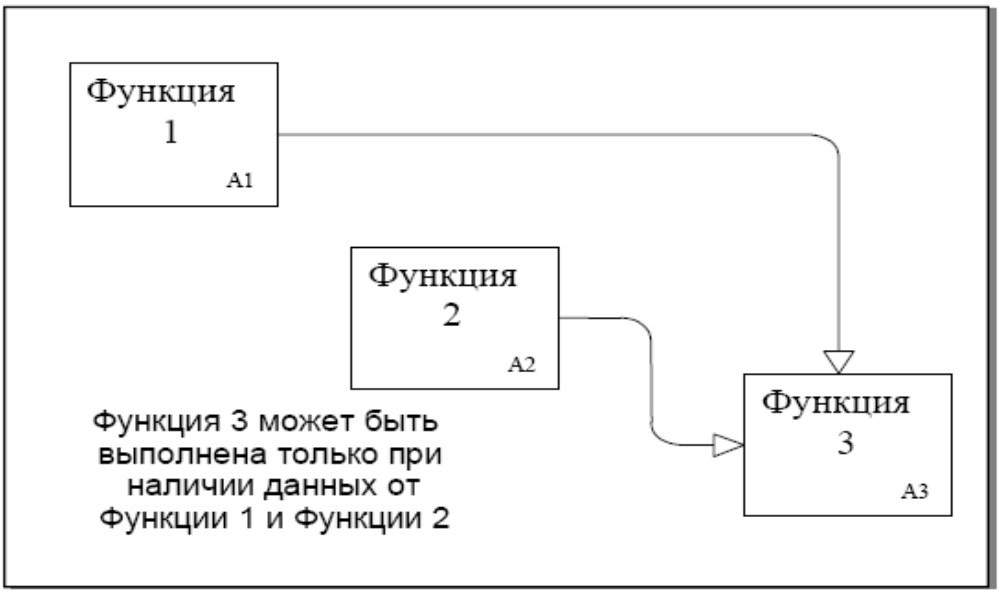
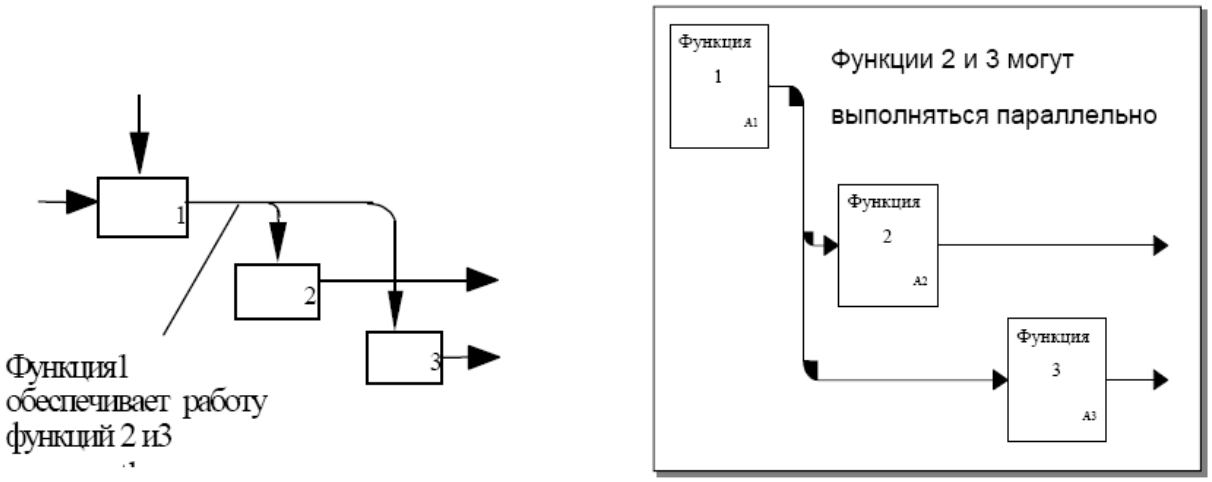


Рис. 3.17. Пример ограничения

Рис. 3.17. иллюстрирует случай, при котором "функция 3" может быть выполнена только после получения данных, выработанных "функцией 1" и "функцией 2".

3.10.2. Параллельное функционирование

Различные функции в модели могут быть выполнены параллельно, если удовлетворяются необходимые ограничения (условия). Как показано на рис. 3.18., один блок может создать данные или материальные объекты, необходимые для параллельной работы нескольких блоков.



а) б)

Рис. 3.18. Пример параллельного выполнения

3.10.3. Ветвление и слияние сегментов стрелок

Ветвление и слияние стрелок призвано уменьшить загруженность диаграмм графическими элементами (линиями). Чтобы стрелки и их сегменты правильно описывали связи между блоками – источниками и блоками – потребителями, используется аппарат меток. Метки связываются с сегментами посредством тильд. При этом между сегментами возникают определенные отношения, описанные ниже:

– непомеченные сегменты (рис. 3.19.) содержат все объекты, указанные в метке стрелки перед ветвлением (т.е. все объекты принадлежат каждому из сегментов);

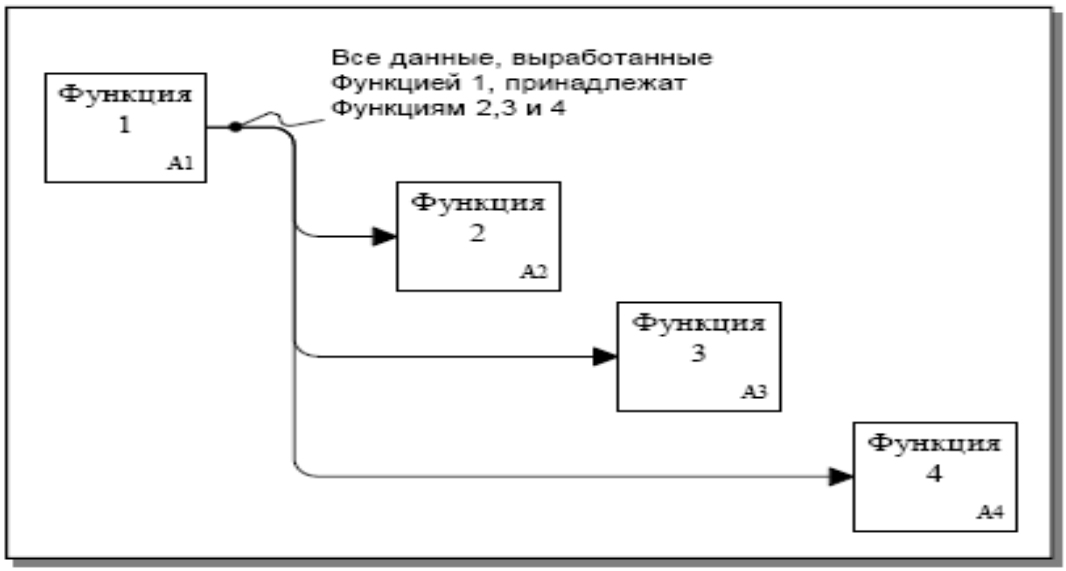


Рис. 3.19. Пример ветвления стрелок

– сегменты, помеченные после точки ветвления (рис. 3.20.), содержат все объекты, указанные в метке стрелки перед ветвлением, или их часть, описываемую меткой каждого конкретного сегмента;

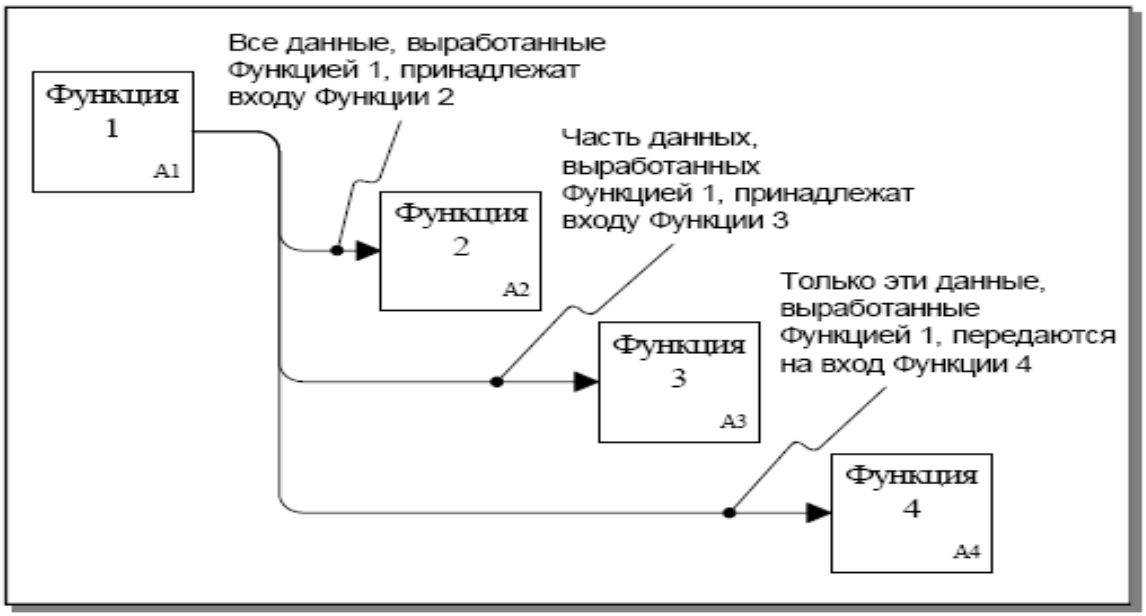


Рис. 3.20. Пример ветвления стрелок

– при слиянии непомеченных сегментов объединенный сегмент стрелки содержит все объекты, принадлежащие сливаемым сегментам и указанные в общей метке стрелки после слияния (рис. 3.21.);

– при слиянии помеченных сегментов (рис. 3.22.) объединенный сегмент содержит все или некоторые объекты, принадлежащие сливаемым сегментам и перечисленные в общей метке после слияния; если общая метка после слияния отсутствует, это означает, что общий сегмент передает все объекты, принадлежащие сливаемым сегментам;

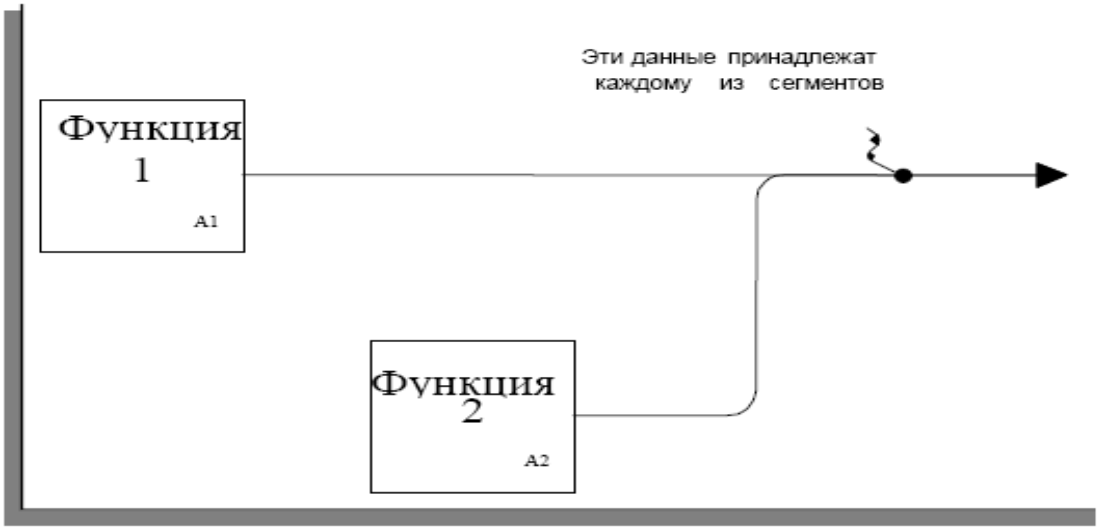


Рис. 3.21. Пример слияния непомеченных сегментов

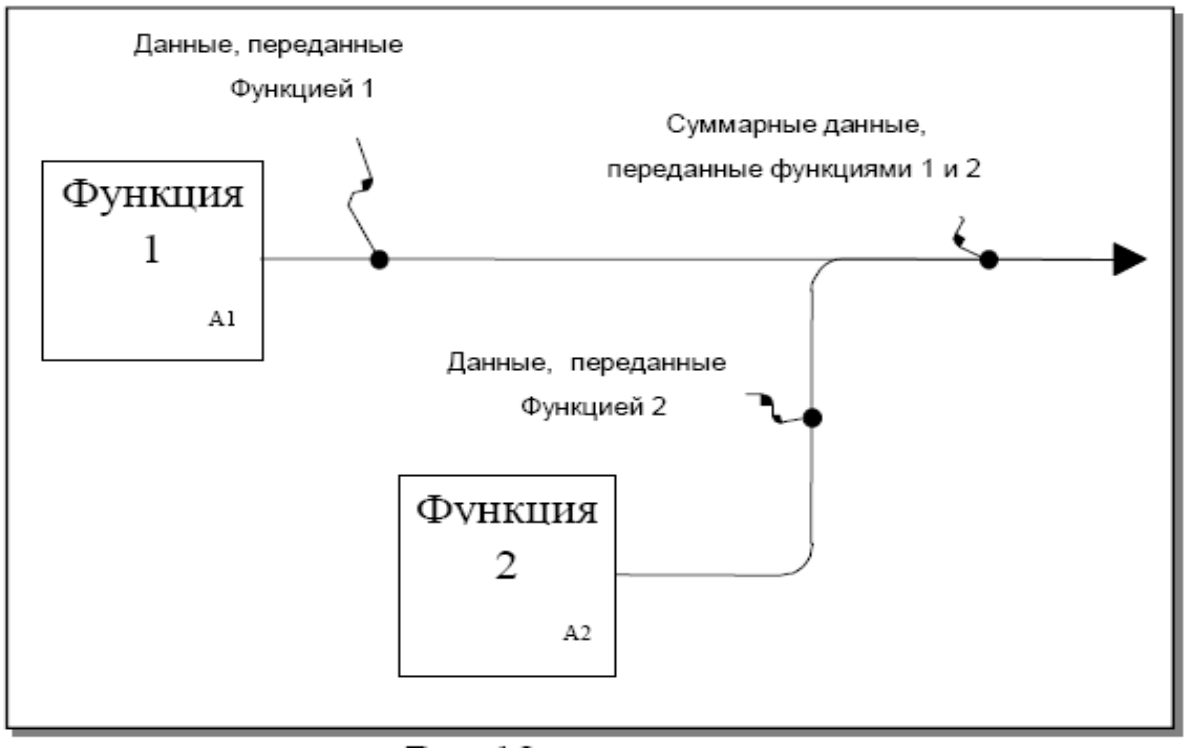


Рис. 3.22. Пример слияния помеченных сегментов

3.11. Создание диаграмм IDEF0 в среде AllFusion Process Modeler

После запуска программы на экране появиться диалоговое окно, в котором следует выбрать режим работы: либо создать новую модель (Create model), либо открыть существующую модель (Open model) (рис. 3.23.). При первом открытии программы (при создании новой модели) область построения содержит диаграмму IDEF0.

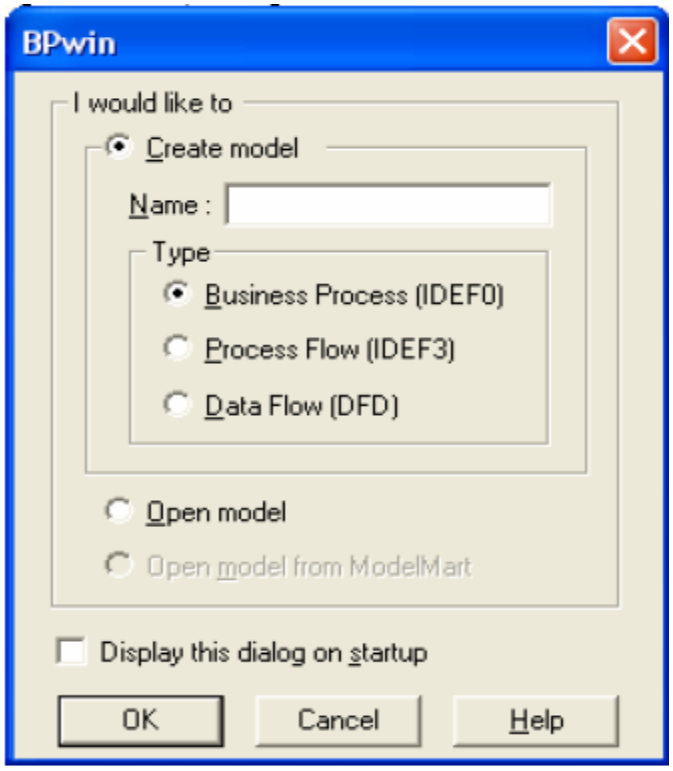


Рис. 3.23.

На основной панели инструментов расположены элементы управления, в основном знакомые по другим Windows–интерфейсам (рис. 3.24.).

Рис. 3.24.

На основной панели инструментов (либо в любом желаемом месте экрана) расположены инструменты редактора BPWin:

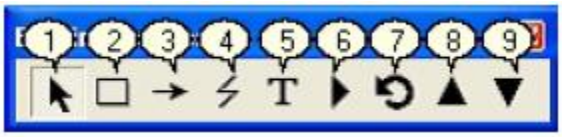


Рис. 3.25.

Любая диаграмма состоит из совокупности следующих объектов: блоков, дуг, текстовых блоков. Для работы с любым из этих объектов можно использовать либо основное меню (рис. 3.26.), либо контекстно–зависимое меню (меню, появляющееся при нажатии правой кнопке мыши). Принципы работы с меню являются стандартными для среды Windows. Объект сначала делается активным, затем над ним осуществляются необходимые действия.

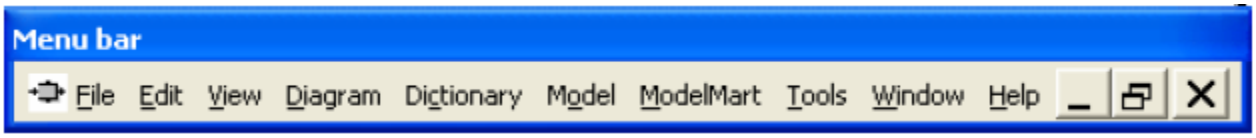


Рис. 3.26.

Каждая диаграмма располагается внутри бланка имеющего несколько информационных полей.

IDEF0–модель предполагает наличие четко сформулированной цели, единственного субъекта моделирования и одной точки зрения. Для внесения области, цели и точки зрения в модели IDEF0 в BPwin следует выбрать пункт меню Model/Model Properties, вызывающий диалог Model Properties (рис. 3.27.).

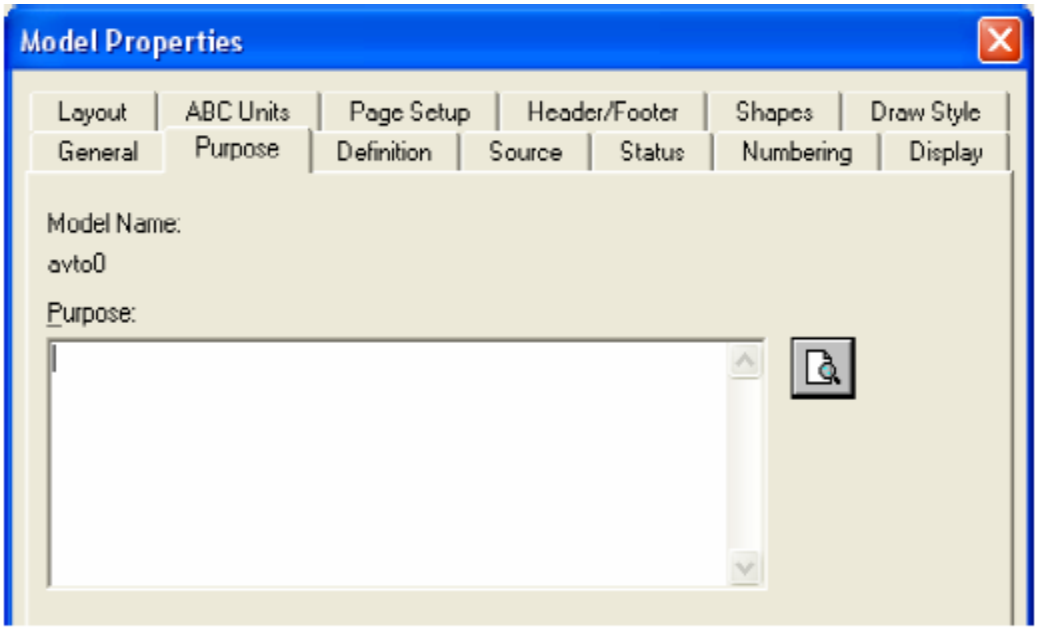


Рис. 3.27.

В закладке Purpose следует внести цель и точку зрения, а в закладку Definition –определение модели и описание области. В закладке Status того же диалога можно описать статус модели (черновой вариант, рабочий, окончательный и т.д.), время создания и последнего редактирования (отслеживается в дальнейшем автоматически по системной дате). В закладке Source описываются источники информации для построения модели (например, "Опрос экспертов предметной области и анализ документации"). Закладка General служит для внесения имени проекта и модели, имени и инициалов автора и временных рамок, модели – AS–IS и TO–BE.

Результат описания модели можно получить в отчете Model Report. Диалог настройки отчета по модели вызывается из пункта меню Tools/Reports/ModelReport. В диалоге настройки следует выбрать необходимые поля (при этом автоматически отображается очередность вывода информации в отчет) (рис. 3.28.).

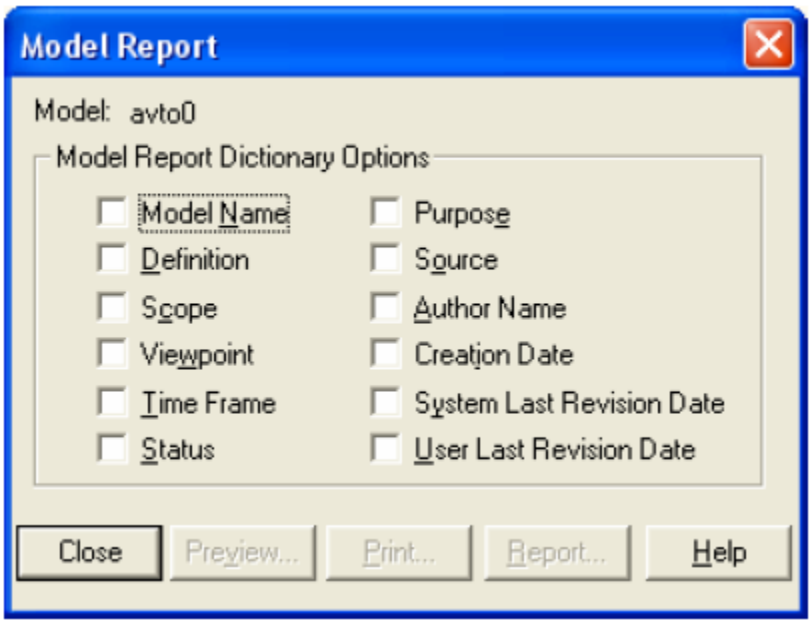


Рис. 3.28.

Принцип работы в пакете BPwin рассмотрим на примере задачи обслуживания клиентов в банке для автомобилистов.

Задача: В банке для автомобилистов имеется два окошечка, каждое из которых обслуживается одним кассиром и имеет отдельную подъездную полосу. Обе полосы расположены рядом. Из предыдущих наблюдений известно, что интервалы времени между прибытием клиентов в час пик распределены экспоненциально с математическим ожиданием равным 0,5 единицы времени. Так как банк перегружен только в часы пик, то анализируется только этот период. Продолжительность обслуживания у обоих кассиров одинакова и распределена экспоненциально с математическим ожиданием, равным 0,3 единицы времени. Известно также, что при равной длине очереди, а так же при отсутствии очередей клиенты отдают предпочтение первой полосе. Во всех других случаях клиенты выбирают более короткую очередь. После того как клиент въехал в банк, он не может покинуть его, пока не будет обслужен. Однако он может сменить очередь, если стоит последним и разница в длине очередей при этом составляет не менее двух автомобилей. Из–за ограниченного места на каждой полосе может находиться не более трех автомобилей. В банке, таким образом, не может находиться более восьми автомобилей, включая автомобили двух клиентов, обслуживаемых в текущий момент кассиром. Если место перед банком заполнено до отказа, прибывший клиент считается потерянным, так как сразу уезжает.

Начальные условия имитации:

1. Оба кассира заняты. Продолжительность обслуживания для каждого кассира нормально распределена с математическим ожиданием, равным 1 единице времени, и среднеквадратическим отклонением, равным 0,3 единицы времени.
2. Прибытие первого клиента запланировано на момент времени 0,1.
3. В каждой очереди ожидают по два автомобиля.

Необходимо оценить следующие характеристики:

1. загрузку по каждому кассиру

2. число обслуженных клиентов
3. среднее время пребывания клиента в банке
4. среднее число клиентов в каждой очереди
5. процент клиентов, которым отказано в обслуживании
6. число смен подъездных полос

Имитация системы проводиться в течение 1000 единиц времени.

Методология IDEF0 предписывает построение иерархической системы диаграмм – единичных описаний фрагментов системы. Сначала проводиться описание системы в целом (контекстная диаграмма), после чего проводиться декомпозиция – система разбивается на подсистемы, и каждая подсистема описывается отдельно.

После создания проекта мы видим окно с единственным блоком. Назовем данный блок «Банк автомобилистов». Для этого необходимо щелкнуть правой клавишей мыши по блоку и выбрать команду Name и в диалоговом окне ввести название (рис. 3.29.).

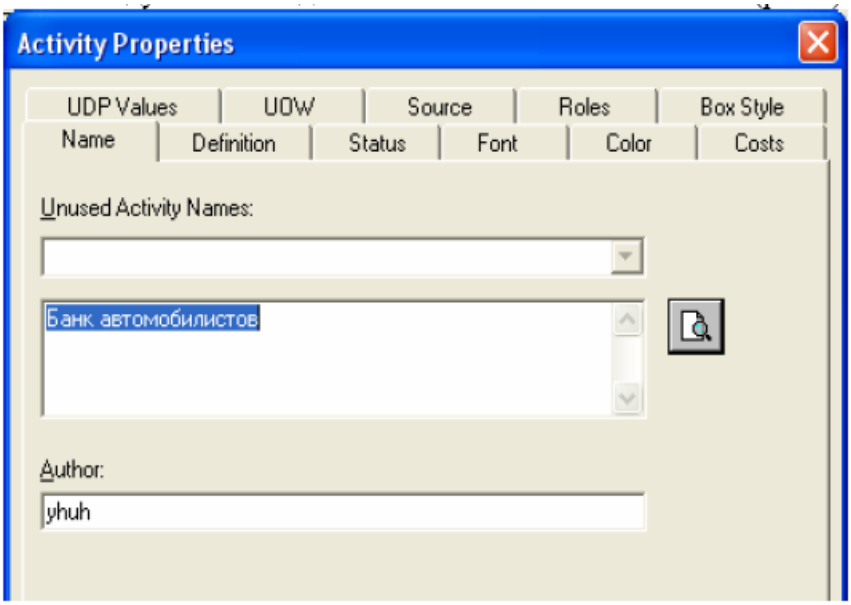


Рис. 3.29.

После создания объекта «Банк автомобилистов» необходимо обозначить его основные функции и элементы взаимодействия. В Bpwin этими элементами являются u1076 дуги. Для построения дуг управления, входа, выхода и механизмов необходимо выбрать инструмент (Arrow Tool), затем щелкнуть мышью со стороны периметра и второй щелчок с соответствующей стороны блока. Для построения дуги выхода щелкнуть первоначально справой стороны блока, затем со стороны периметра.

То с какой стороны дуга подходит к блоку является своего рода значением данной дуги.

Слева – вход в блок
Справа – выход в блок
Сверху – управляющая информация

Снизу – механизмы (средства производства)

Дугам, как и блокам можно придавать свои имена. Для этого необходимо: щелкнуть правой клавишей мыши по блоку и выбрать команду Name и в диалоговом окне ввести название дуги.

Определите наименования для созданных ранее дуг, соответственно типу дуги: «вход клиента», «выход клиента», «количество клиентов в очереди 1», «количество клиентов в очереди 2», «кассир 1», «кассир 2».

Название дуги является независимым объектом, который можно перемещать относительно дуги. Текст может располагаться по отношению к дуге в свободной форме, либо соединен с дугой символом тильды. Чтобы установить тильду следует нажать инструмент (Squiggle Tool), а затем выбрать дугу, либо использовать команду контекстно–зависимого меню Squiggle.

Дуга представляет собой совокупность отельных графических объектов: прямые участки, изогнутые участки, изображение наконечника. Отдельные элементы можно передвигать независимо друг от друга, меняя форму дуги, также дугу можно перемещать как единый неделимый элемент.

Для набора текста следует нажать инструмент (Text Block Tool), после чего щелкнуть мышью в позиции предполагаемого ввода текста. Затем в появившемся диалоговом окне набрать нужный текст и установить опцию значимости (обычный текст, цель, точка зрения).

Для удаления блока и дуги или текста необходимо их выделить щелчком левой кнопки мыши и нажать клавишу Delete, а затем подтвердить намерения по поводу удаления.

Пример контекстной диаграммы рассматриваемой задачи смотрите на рис. 3.12.

После создания контекстной диаграммы необходимо расписать работу отдельных участков банка автомобилистов. Для этого декомпозируем эту диаграмму (рис. 3.14.).

Для декомпозиции необходимо в браузере щелкнуть левой кнопкой мыши на имени диаграммы, а затем нажать кнопку  (Go to Child Diagram), затем в диалоговом окне (рис. 3.30.) ввести необходимое количество блоков и тип диаграммы.

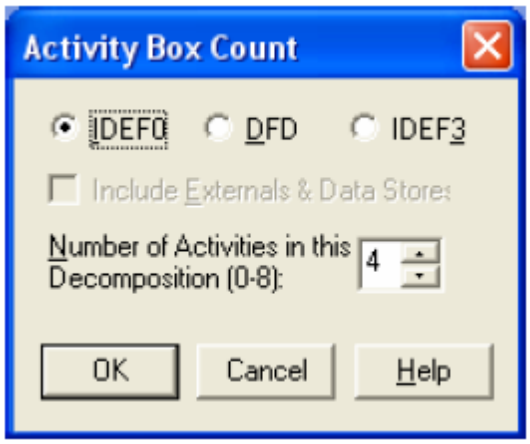


Рис. 3.30.

Выполним декомпозицию блока диаграммы A–0 (рис. 3.12.), создав диаграмму A0 (рис. 3.14.), состоящую из 4 блоков: «Выбор очереди», «Обслужить касса 1», «Обслужить касса 2», «Выход». Если в дальнейшем необходимо добавить блоки на диаграмме, то необходимо выбрать инструмент (Activity Box Tool) и щелкнуть мышью в нужном месте диаграммы.

После декомпозиции необходимо соединить получившиеся блоки дугами. Для этого необходимо выбрать инструмент, щелкнуть мышью по исходной стороне блока затем по конечной стороне следующего блока. Аналогично строятся разветвления и слияния дуг.

3.12. Диаграммы DFD

Диаграммы потоков данных (Data flow diagramming, DFD) используются для описания документооборота и обработки информации. Их можно использовать как дополнение к модели IDEF0 для более наглядного отображения текущих операций документооборота в корпоративных системах обработки информации. DFD описывают функции обработки информации (работы), документы (стрелки, arrow), объекты, сотрудников или отделы, которые участвуют в обработке информации (внешние ссылки, external references) и таблицы для хранения документов (хранилище данных, data store).

Диаграмма потоков данных (data flow diagram, DFD) — один из основных инструментов структурного анализа и проектирования информационных систем, существовавших в «доюмээльную» эпоху. Несмотря на имеющее место в современных условиях смещение акцентов от структурного к объектно-ориентированному подходу к анализу и проектированию систем, «старые» структурные нотации по-прежнему широко и эффективно используются как в бизнес-анализе, так и в анализе информационных систем.

Исторически сложилось так, что для описания диаграмм DFD используются две нотации — Йодана (Yourdon) и Гейна-Сарсона (Gane-Sarson), отличающиеся синтаксисом.

Информационная система принимает извне потоки данных. Для обозначения элементов среды функционирования системы используется понятие внешней сущности. Внутри системы существуют процессы преобразования информации, порождающие новые потоки данных. Потоки данных могут поступать на вход к другим процессам, помещаться (и извлекаться) в накопители данных, передаваться к внешним сущностям.

Модель DFD, как и большинство других структурных моделей — иерархическая модель. Каждый процесс может быть подвергнут декомпозиции, то есть разбиению на структурные составляющие, отношения между которыми в той же нотации могут быть показаны на отдельной диаграмме. Когда достигнута

требуемая глубина декомпозиции — процесс нижнего уровня сопровождается мини-спецификацией (текстовым описанием).

Кроме того, нотация DFD поддерживает понятие подсистемы — структурной компоненты разрабатываемой системы.

Нотация DFD — удобное средство для формирования контекстной диаграммы, то есть диаграммы, показывающей разрабатываемую АИС в коммуникации с внешней средой. Это — диаграмма верхнего уровня в иерархии диаграмм DFD. Ее назначение — ограничить рамки системы, определить, где заканчивается разрабатываемая система и начинается среда. Другие нотации, часто используемые при формировании контекстной диаграммы — диаграмма SADT, диаграмма вариантов использования.

Для решения задачи функционального моделирования на базе структурного анализа традиционно применяются два типа моделей: IDEF0-диаграммы и диаграммы потоков данных.

Методология разработки процессных диаграмм обычно применяется при проведении обследований предприятий в рамках проектов управленческого консалтинга, а также в проектах автоматизации крупных объектов при экспресс-обследовании (обычно для составления развернутого плана работ).

Нотация диаграмм потоков данных позволяет отображать на диаграмме как шаги бизнес-процесса, так и поток документов и управления (в основном, управления, поскольку на верхнем уровне описания процессных областей значение имеет передача управления). Также на диаграмме можно отображать средства автоматизации шагов бизнес-процессов. Обычно используется для отображения третьего и ниже уровня декомпозиции бизнес-процессов (первым уровнем считается идентифицированный перечень бизнес-процессов, а вторым - функции, выполняемые в рамках бизнес-процессов).

Диаграммы потоков данных (Data flow diagramming, DFD):

- являются основным средством моделирования функциональных требований к проектируемой системе;
- создаются для моделирования существующего процесса движения информации;
- используются для описания документооборота, обработки информации;
- применяются как дополнение к модели IDEF0 для более наглядного отображения текущих операций документооборота (обмена информацией);
- обеспечивают проведение анализа и определения основных направлений реинжиниринга ИС.

Диаграммы DFD могут дополнить то, что уже отражено в модели IDEF0, поскольку они описывают потоки данных, позволяя проследить, каким образом происходит обмен информацией как внутри системы между бизнес-функциями, так и системы в целом с внешней информационной средой.

В случае наличия в моделируемой системе программной/программируемой части (практически всегда) предпочтение, как правило, отдается DFD по следующим соображениям.

1. DFD-диаграммы создавались как средство проектирования программных систем, тогда как IDEF0 - как средство проектирования систем вообще, поэтому DFD имеют более богатый набор элементов, адекватно отражающих их специфику (например, хранилища данных являются прообразами файлов или баз данных).
2. Наличие мини-спецификаций DFD-процессов нижнего уровня позволяет преодолеть логическую незавершенность IDEF0, а именно обрыв модели на некотором достаточно низком уровне, когда дальнейшая ее детализация становится бессмысленной, и построить полную функциональную спецификацию разрабатываемой системы.
3. Существуют и поддерживаются рядом CASE-инструментов алгоритмы автоматического преобразования иерархии DFD в структурные карты, демонстрирующие межсистемные и внутрисистемные связи, а также иерархию систем, что в совокупности с мини-спецификациями является завершенным заданием для программиста.

С помощью DFD-диаграмм требования к проектируемой ИС разбиваются на функциональные компоненты (процессы) и представляются в виде сети, связанной потоками данных. Главная цель декомпозиции DFD-функций - продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами. На схемах бизнес-процесса отображаются:

- функции процесса;
- входящая и исходящая информация, при описании документов;
- внешние бизнес-процессы, описанные на других диаграммах;
- точки разрыва при переходе процесса на другие страницы.

Если при моделировании по методологии IDEF0 система рассматривается как сеть взаимосвязанных функций, то при создании DFD-диаграммы система рассматривается как сеть связанных между собой функций, т.е. как совокупность сущностей (предметов).

Структурный анализ - это системный пошаговый подход к анализу требований и проектированию спецификаций системы независимо от того, является ли она существующей или создается вновь. Методологии Гейна-Сарсона (Gane-Sarson) и Йордана/Де Марко (Yourdon/DeMarko) построения диаграмм потоков данных, основанные на идее нисходящей иерархической организации, наиболее ярко демонстрируют этот подход.

Целью этих двух методологий является преобразование общих, неясных знаний о требованиях к системе в точные (насколько это возможно) определения. Обе методологии фокусируют внимание на потоках данных, их главное назначение - создание базированных на графике документов по функциональным требованиям. Методологии поддерживаются традиционными нисходящими методами проектирования и обеспечивают один из лучших способов связи между

аналитиками, разработчиками и пользователями системы за счет интеграции следующих средств:

1. Диаграмм потоков данных.
2. Словарей данных, которые являются каталогами всех элементов данных, присутствующих в DFD, включая групповые и индивидуальные потоки данных, хранилища и процессы, а также все их атрибуты.
3. Миниспецификации обработки, описывающие DFD-процессы нижнего уровня и являющиеся базой для кодогенерации.

Миниспецификация - это алгоритм описания задач, выполняемых процессами, множество всех миниспецификации является полной спецификацией системы. **Миниспецификации содержат** номер и/или имя процесса, списки входных и выходных данных и тело (описание) процесса, являющееся спецификацией алгоритма или операции, трансформирующей входные потоки данных в выходные. Известно большое число разнообразных **методов,** позволяющих задать тело процесса, соответствующий язык может варьироваться от структурированного естественного языка или псевдокода до визуальных языков проектирования (типа FLOW-форм и диаграмм Насси-Шнейдермана) и формальных компьютерных языков.

Проектные спецификации строятся по DFD и их миниспецификациям автоматически. Наиболее часто для описания проектных спецификаций используется методика структурных карт Джексона, иллюстрирующая иерархию модулей, связи между ними и некоторую информацию об их исполнении (последовательность вызовов, итерацию). Существует ряд методов автоматического преобразования DFD в структурные карты.

Главной отличительной чертой методологии **Гейна-Сарсона** является наличие этапа моделирования данных, определяющего содержимое хранилищ данных (БД и файлов) в DFD в третьей нормальной форме. Этот этап включает построение списка элементов данных, располагающихся в каждом хранилище данных; анализ отношений между данными и построение соответствующей диаграммы связей между элементами данных; представление всей информации по модели в виде связанных **нормализованных таблиц**. Кроме того, методологии отличаются чисто синтаксическими аспектами, так, например различны графические символы, представляющие компоненты DFD.

Рассматриваемые методы представляют собой методы, помогающими от чистого листа бумаги или экрана перейти к хорошо организованной модели системы. Обе **методологии основаны** на простой концепции нисходящего поэтапного разбиения функций системы на подфункции:

На первом этапе формируется контекстная диаграмма верхнего уровня, идентифицирующая границы системы и определяющая интерфейсы между системой и окружением.

После интервьюирования эксперта предметной области, формируется список внешних событий, на которые система должна реагировать. Для каждого из таких событий строится пустой процесс (bubble) в предположении, что его функция обеспечивает требуемую реакцию на это событие, которая в большинстве случаев

включает генерацию выходных потоков и событий (но может также включать и занесение информации в хранилище данных для ее использования другими событиями и процессами).

На следующем уровне детализации аналогичная деятельность осуществляется для каждого из пустых процессов.

Для **усиления функциональности** в данной нотации диаграмм предусмотрены специфические элементы, предназначенные для описания информационных и документопотоков, такие как внешние сущности и хранилища данных.

Помимо нотации Йордона/Де Марко и Гейна - Сарсона для элементов **DFD-диаграм** могут использоваться и другие условные обозначения (OMT, SSADM, и т.д.). Все они обладают практически одинаковой функциональностью и различаются лишь в деталях.

Несмотря на то, что методология IDEF0 получила широкое распространение, по мнению многих аналитиков DFD гораздо больше подходит для проектирования **информационных систем** вообще и баз данных в частности. DFD позволяет уже на стадии функционального моделирования определить базовые требования к данным (этому способствует разделение потоков данных на материальные, информационные и управляющие). Кроме того интеграция DFD-моделей и ER-моделей (entity-relationship, "сущность-связь") не вызывает затруднений. Например, можно определить список атрибутов хранилищ данных, последние на стадии информационного моделирования однозначно отображаются в сущности модели "сущность- связь".

В свою очередь, как уже отмечалось, IDEF0 больше подходит для **решения задач**, связанных с **управленческим консультированием** (реинжинирингом процессов). Этому способствует также тесная связь IDEF0 с методом функционально - стоимостного анализа ABC (Activity Based Costing), позволяющим определить схему расчета стоимости выполнения той или иной деловой процедуры. Однако, существует ряд CASE - систем, предлагающих методологию IDEF0 на этапе функционального обследования предметной области. В таких системах на следующий этап передается просто список всех объектов IDEF0-модели (входы, выходы, механизмы, управление), которые затем рассматриваются на предмет включения в информационную модель.

3.12.1 Терминология DFD-нотации

DFD-БЛОКИ – графическое изображение операции (процесса, функции, работы) по обработке или преобразованию информации (данных). Смысл DFD-блока, отображающего функцию совпадает со смыслом блоков IDEFO и IDEF3, заключающиеся в преобразовании входов в выходы. DFD-блоки также имеют входы и выходы, но не поддерживают управление и механизмы, как IDEFO.

Назначение функции состоит в создании из входных потоков выходных в соответствии с действием, определяемым именем процесса. Поэтому имя функции должно содержать глагол в неопределенной форме с последующим дополнением.

Функции обычно именуются по названию системы, например "Разработка системы автоматизированного проектирования". Рекомендуется использовать глаголы, отображающие динамические отношения, например: «рассчитать», «получить», «заказать», «фрезеровать», «точить», «вычислить», «включить», «моделировать» и т.д. Если автор использует такие глаголы, как “обработать”, “модернизировать”, или “отредактировать”, то это означает, что он, вероятно, пока недостаточно глубоко понимает данную функцию процесса и требуется дальнейший анализ.

По нотации Гейн-Сарсона DFD-блок изображается прямоугольником со скругленными углами. Каждый блок должен иметь уникальный номер для ссылки на него внутри диаграммы. Номер каждого блока может включать префикс, номер родительского блока (A) и номер объекта, представляющий собой уникальный номер блока на диаграмме. Например, функция может иметь номер A.12.4.

Для того чтобы избежать пересечений линий потоков данных один и тот же элемент может на одной и той же диаграмме отображаться несколько раз; в таком случае два или более прямоугольника, обозначающих один и тот же элемент, могут идентифицироваться линией перечеркивающей нижний правый угол.

DATA FLOW (поток данных) – механизм, использующийся для моделирования передачи информации между участниками процесса информационного обмена (функциями, хранилищами данных, внешними ссылками). По нотации Гейн-Сарсона поток данных (документы, объекты, сотрудники, отделы или иные участники обработки информации) изображается стрелкой между двумя объектами DFD-диаграммы, предпочтительно горизонтальной и/или вертикальной, причем направление стрелки указывает направление потока. Каждая стрелка должна иметь источник и цель. В отличие от стрелок IDEF0-диаграммы (ICOM), стрелки DFD могут входить или выходить из любой стороны блока.

Стрелки описывают, как объекты (включая данные) двигаются из одной части системы в другую. Поскольку в DFD каждая сторона блока не имеет четкого назначения, в отличие от блоков IDEF0-диаграммы, стрелки могут подходить и выходить из любой грани. В DFD-диаграммах для описания диалогов типа команды-ответа между операциями, применяются двунаправленные стрелки между функцией и внешней сущностью и/или между внешними сущностями. Стрелки могут сливаться и разветвляться, что позволяет описать декомпозицию стрелок. Каждый новый сегмент сливающейся или разветвляющейся стрелки может иметь собственное имя.

Иногда информация может двигаться в одном направлении, обрабатываться и возвращаться обратно. Такая ситуация может моделироваться либо двумя различными потоками, либо одним двунаправленным. На поток данных можно ссылаться, указывая процессы, сущности или накопители данных, которые поток соединяет.

Каждый поток должен иметь имя, расположенное вдоль или над стрелкой, выбранное таким образом, чтобы в наибольшей степени передавать смысл содержания потока пользователям, которые будут рассматривать диаграмму потоков данных. Набрасывая диаграмму потоков данных, можно опустить

наименования, если оно является очевидным для пользователя, но автор диаграммы должен в любой момент представить описание потока.

DATA FLOW ДИАГРАММА (DFD-диаграмма) – диаграмма применяемая для графического представления (flowchart) движения и обработки информации в организации или в каком-либо процессе. Обычно диаграммы этого типа используются для проведения анализа организации информационных потоков и для разработки ИС. DFD-диаграммы являются ключевой частью документа спецификации требований - графическими иерархическими спецификациями, описывающими систему с позиций потоков данных. Каждый узел-процесс в DFD может развертываться в диаграмму нижнего уровня, что позволяет на любом уровне абстрагироваться от деталей.

Для диаграмм этого типа обычно применяется сокращенное обозначение DFD. DFD являются. В состав DFD могут входить четыре графических символа, представляющих потоки данных, процессы преобразования входных потоков данных в выходные, внешние источники и получатели данных, а также файлы и БД, требуемые процессами для своих операций.

DFD-диаграммы моделируют функции, которые система должна выполнять, но почти ничего не сообщают об отношениях между данными, а также о поведении системы в зависимости от времени - для этих целей используются диаграммы сущность-связь и диаграммы переходов состояний, соответственно.

DATA STORE (хранилище данных) – графическое представление потоков данных импортируемых/экспортируемых из соответствующих баз данных. Обычно это таблицы для хранения документов. В отличие от стрелок, описывающих объекты в движении, хранилища данных изображают объекты в покое. Накопители данных являются неким прообразом базы данных информационной системы организации. Хранилища данных включаются в модель системы в том случае, если имеются этапы технологического цикла, на которых появляются данные, которые необходимо сохранять в памяти. При отображении процесса сохранения данных стрелка потока данных направляется в хранилище данных, и, наоборот – из хранилища, если идет импорт данных.

Хранилища данных предназначены для изображения неких абстрактных устройств для хранения информации, которую можно туда в любой момент времени поместить или извлечь, безотносительно к их конкретной физической реализации. Хранилища данных используются:

- в материальных системах - там, где объекты ожидают обработки, например в очереди;
- в системах обработки информации для моделирования механизмов сохранения данных для дальнейших операций.

По нотации Гейн-Сарсона хранилище данных обозначается двумя горизонтальными линиями, замкнутыми с одного края. Каждое хранилище данных должно идентифицироваться для ссылки буквой D и произвольным числом в квадрате с левой стороны, например D5. Имя должно подбираться с учетом наибольшей информативности для пользователя.

В модели может быть создано множество вхождений хранилищ данных, каждое из которых может иметь одинаковое имя и ссылочный номер. Для того, чтобы не усложнять диаграмму потоков данных пересечениями линий, можно изображать дубликаты накопителя данных дополнительными вертикальными линиями с левой стороны квадрата.

EXTERNAL REFERENCE (внешняя ссылка, внешняя сущность, external entities) – объект диаграммы потоков данных, являющийся источником или приемником информации извне модели. Внешние ссылки/сущности изображают входы и/или выходы, т.е. обеспечивают интерфейс с внешними объектами, находящимися вне моделируемой системы. Внешними ссылками системы обычно являются логические классы предметов или людей, представляющие собой источник или приемник сообщений, например, заказчики, конструкторы, технологи, производственные службы, кладовщики и т.д. Это могут быть специфические источники, такие, как бухгалтерия, информационно-поисковая система, служба нормоконтроля, склад. Если рассматриваемая система принимает данные от другой системы или передает данные в другую систему, то эта другая система является элементом внешней системы. Без объекта «внешняя сущность» аналитику бывает иногда сложно определить, откуда пришла в компанию данные документы. Или какие документы еще приходят от, такой внешней сущности как, например, "клиент".

По нотации Гейн-Сарсона пиктограмма внешней ссылки представляет собой оттененный прямоугольник верхняя и левая сторона, которого имеет двойную толщину для того, чтобы можно было выделить этот символ среди других обозначений на диаграмме, и обычно располагается на границах диаграммы. Внешняя ссылка может идентифицироваться строчной буквой E в левом верхнем углу и уникальным номером, например E5. Кроме того, внешняя ссылка имеет имя.

Одна и та же внешняя ссылка может быть использована многократно на одной или нескольких диаграммах. Обычно такой прием используют, чтобы не рисовать слишком длинных и запутанных стрелок. Каждая внешняя сущность имеет префикс.

При рассмотрении системы как внешней функции, часто указывается, что она находится за пределами границ, моделируемой системы. После проведения анализа некоторые внешние ссылки могут быть перемещены внутрь диаграммы потоков данных рассматриваемой системы или, наоборот, какая-то часть функций системы может быть вынесена и рассмотрена как внешняя ссылка.

OFF-PAGE REFERENCE (межстраничные ссылки) – инструмент нотации DFD, описывающий передачу данных или объектов с одной диаграммы модели на другую. Стрелка межстраничной стрелки имеет идентифицирующее имя, номер и изображение окружности.

При интерпретации DFD-диаграммы используются следующие правила:

- функции преобразуют входящие потоки данных в выходящие;
- хранилища данных не изменяют потоки данных, а служат только для хранения поступающих объектов;
- преобразования потоков данных во внешних ссылках игнорируется.

Помимо этого, для каждого информационного потока и хранилища определяются связанные с ними элементы данных. Каждому элементу данных присваивается имя, также для него может быть указан тип данных и формат. Именно эта информация является исходной на следующем этапе проектирования - построении модели "сущность-связь". При этом, как правило, информационные хранилища преобразуются в сущности, проектировщику остается только решить вопрос с использованием элементов данных, не связанных с хранилищами.

Представление потоков в виде стрелок совместно с хранилищами данных и внешними сущностями делает модели DFD более похожими на физические характеристики системы - движение объектов, хранение объектов, поставка и распространение объектов.

3.12.2 Построение диаграмм

Диаграммы DFD могут быть построены с использованием традиционного структурного анализа, подобно тому, как строятся диаграммы IDEFO:

- строится физическая модель, отображающая текущее состояние дел;
- полученная модель преобразуется в логическую модель, которая отображает требования к существующей системе;
- строится модель, отображающая требования к будущей системе;
- строится физическая модель, на основе которой должна быть построена новая система.

Альтернативным подходом является подход, применяемый при создании программного обеспечения, называемый событийным разделением (event partitioning), в котором различные диаграммы DFD выстраивают модель системы:

логическая модель строится как совокупность процессов и документирования того, что эти процессы должны делать;

с помощью модели окружения система описывается как взаимодействующий с событиями из внешних сущностей объект. Модель окружения (environment model) обычно содержит описание цели системы, одну контекстную диаграмму и список событий. Контекстная диаграмма содержит один блок, изображающий систему в целом, внешние сущности, с которыми система взаимодействует, ссылки и некоторые стрелки, импортированные из диаграмм IDEF0 и DFD. Включение внешних ссылок в контекстную диаграмму не отменяет требования методологии четко определить цель, область и единую точку зрения на моделируемую систему;

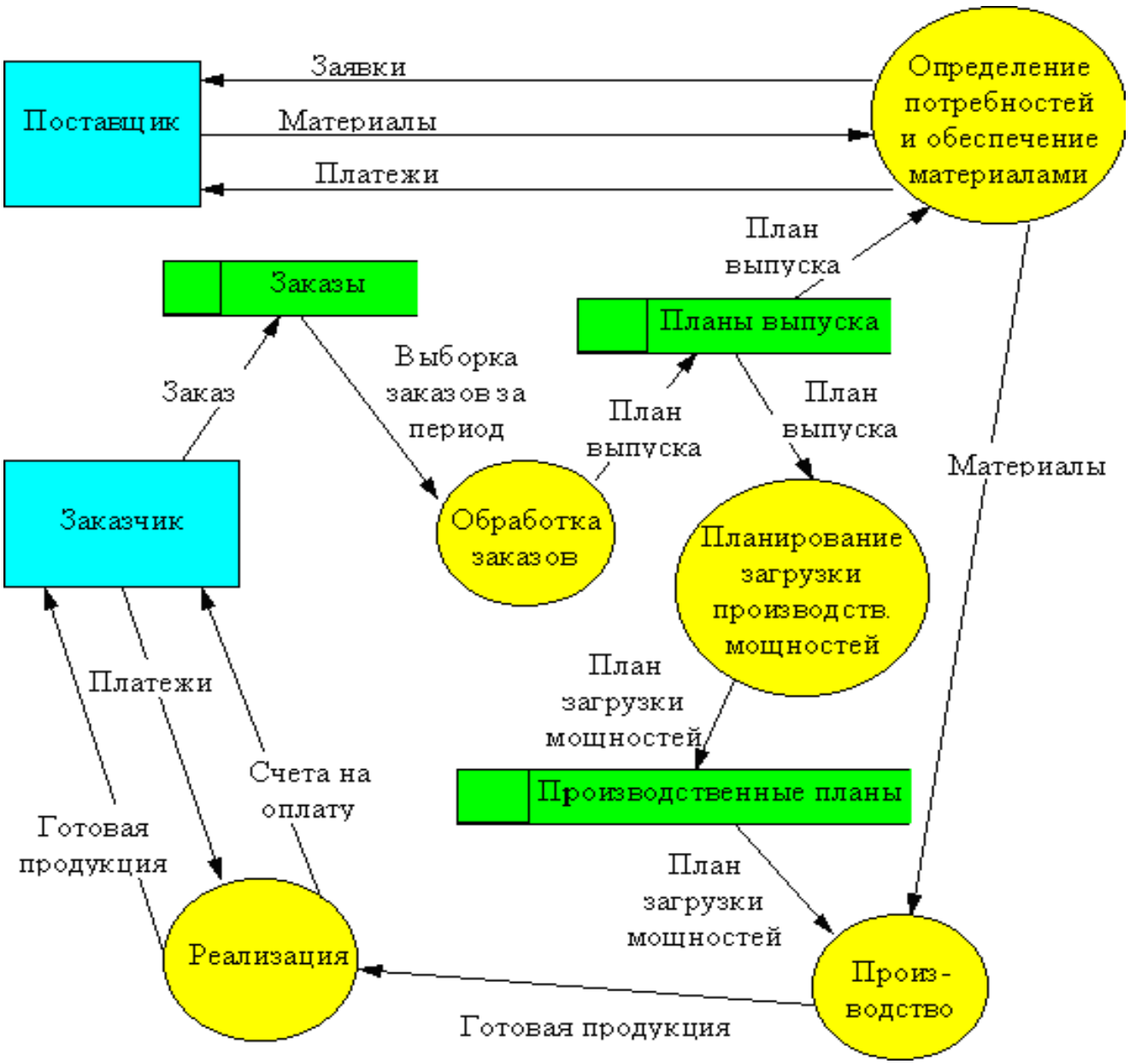
модель поведения (behavior model) показывает, как система обрабатывает события. Эта модель состоит из одной диаграммы, в которой каждый блок изображает каждое событие из модели окружения, могут быть добавлены хранилища для моделирования данных, которые необходимо запоминать между событиями. Потоки добавляются для связи с другими элементами, и диаграмма проверяется с точки зрения соответствия модели окружения.

Полученные диаграммы могут быть **преобразованы** с целью более наглядного представления системы, в частности могут быть декомпозированы функции.

Пример DFD-диаграмм по нотации **Гейна-Сарсона** для предприятия, строящего свою деятельность по принципу "изготовление на заказ" приведен на рисунке 3.31.

На основании полученных заказов формируется план выпуска продукции на определенный период. В соответствии с этим планом определяются потребность в комплектующих изделиях и материалах, а также график загрузки производственного оборудования. После изготоления продукции и проведения платежей, готовая продукция отправляется заказчику.

Заказы подвергаются входному контролю и сортировке. Если заказ не отвечает номенклатуре товаров или оформлен неправильно, то он аннулируется с соответствующим уведомлением заказчика. Если заказ не аннулирован, то определяется, имеется ли на складе соответствующий товар. В случае положительного ответа выписывается счет к оплате и предъявляется заказчику, при поступлении платежа товар отправляется заказчику. Если заказ не обеспечен складскими запасами, то отправляется заявка на товар производителю. После поступления требуемого товара на склад компании заказ становится обеспеченным и повторяет вышеописанный маршрут.



3.31. Пример DFD-диаграмм Гейна-Сарсона для выпуска продукции

Эта диаграмма представляет самый верхний уровень функциональной модели. Естественно, это весьма грубое описание предметной области. **Уточнение модели** производится путем детализации необходимых функций на DFD-диаграмме следующего уровня. Так мы можем разбить функцию "Определение потребностей и обеспечение материалами" на подфункции "Определение потребностей", "Поиск поставщиков", "Заключение и анализ договоров на поставку", "Контроль платежей", "Контроль поставок", связанные собственными потоками данных, которые будут представлены на отдельной диаграмме. Детализация модели должна производиться до тех пор, пока она не будет содержать всю информацию, необходимую для построения информационной системы.

Создание диаграмм DFD в среде BPwin аналогично созданию диаграмм IDEF0. Только после запуска среды следует выбрать область построения DFD.

Пример диаграммы DFD приведен на рис. 3.32.

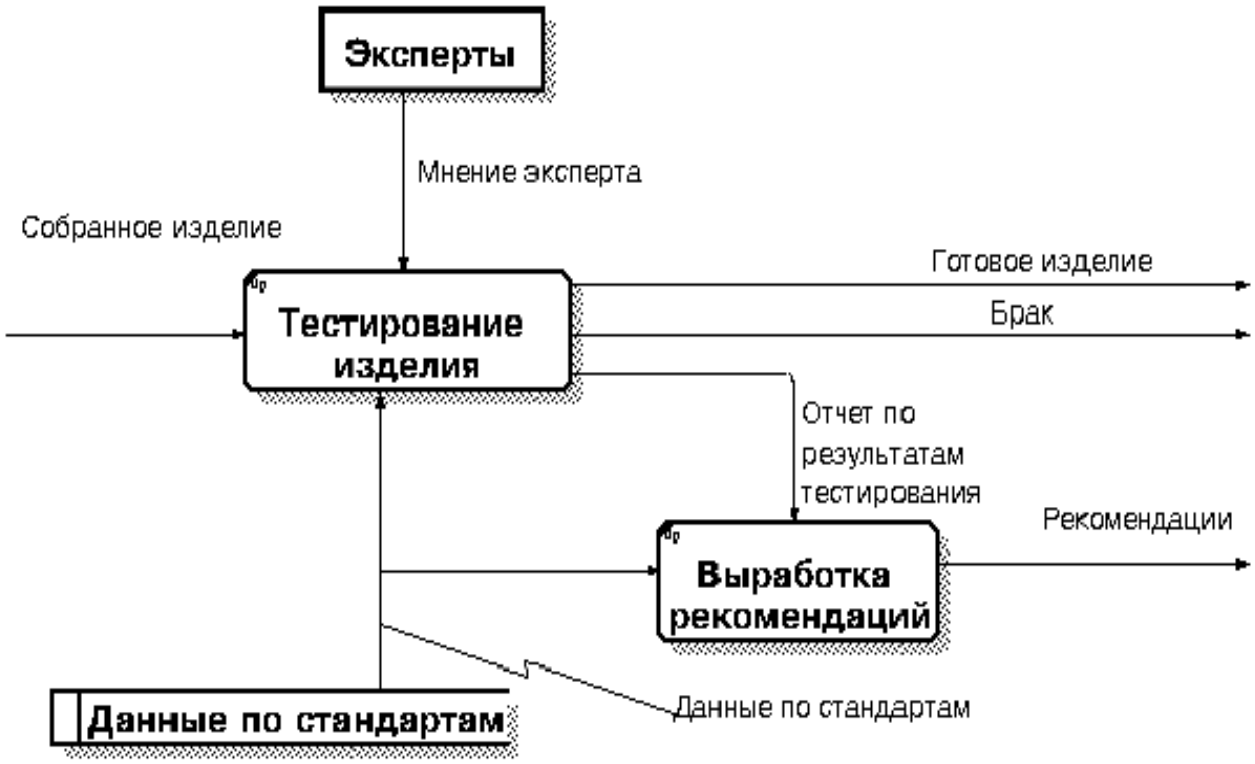


Рис. 3.32. Пример диаграммы DFD для проверки изделия

3.12.3 Примеры DFD диаграмм

В данном примере использован материал из документа по разработке требований.

Элементы DFD

На диаграммах потоков данных используются следующие элементы:

- потоки данных (обозначаются стрелками с названиями);
- элементы преобразования данных (обозначаются окружностями или овалами);
- хранилища данных (отрезки горизонтальных параллельных линий);
- внешние сущности (прямоугольники).

Для определения всех потоков и хранилищ данных используется словарь данных. Каждых овал (окружность) определяет базовую функциональность, которую обеспечивает компонент системы. Эта функциональность описывается с помощью П-спецификаций (P-spec) или мини-спецификаций (mini-spec), которые представляют собой текстовое описание, зачастую написанное с помощью псевдокода

Функциональная декомпозиция DFD

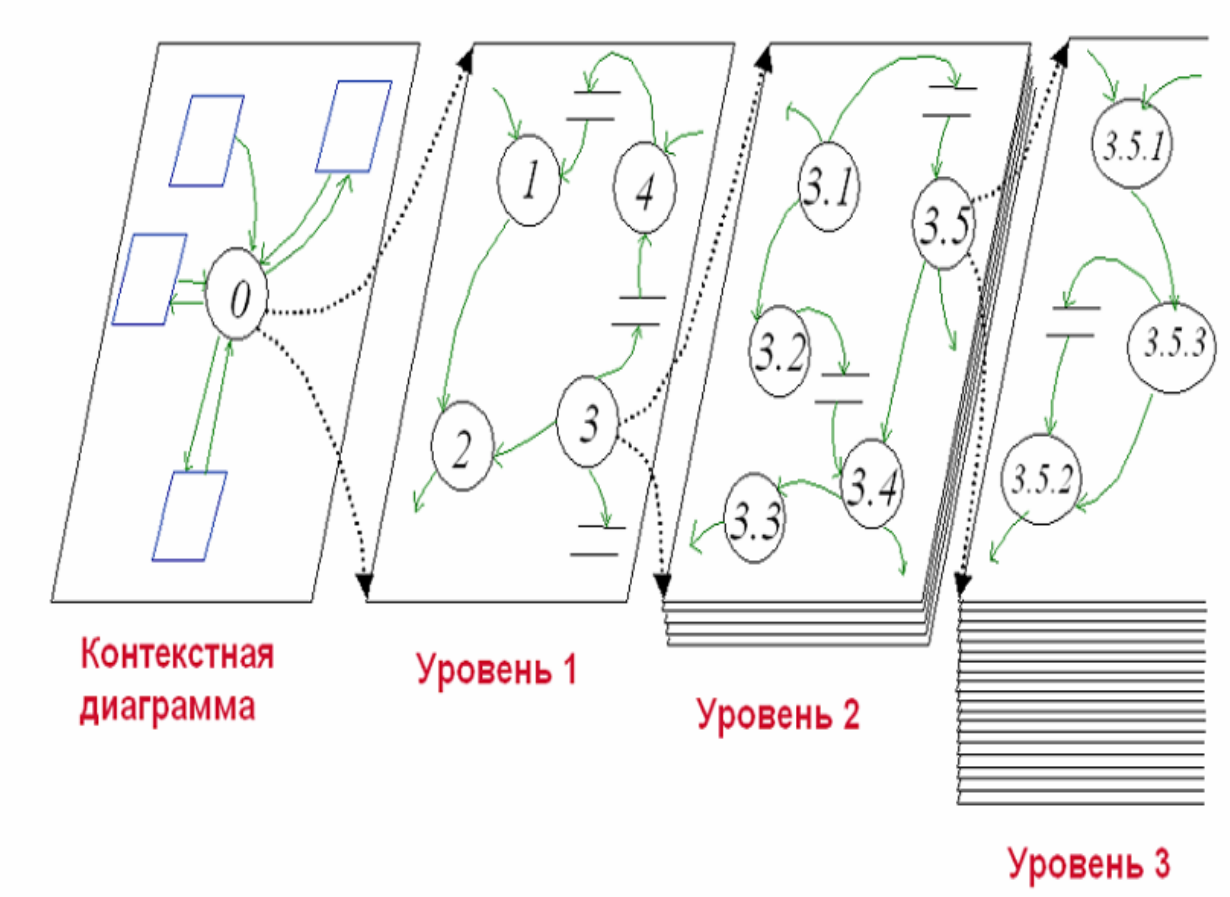


Рис. 3.33. Декомпозиция DFD диаграмм

Пример – модель системы управления скорой помощи

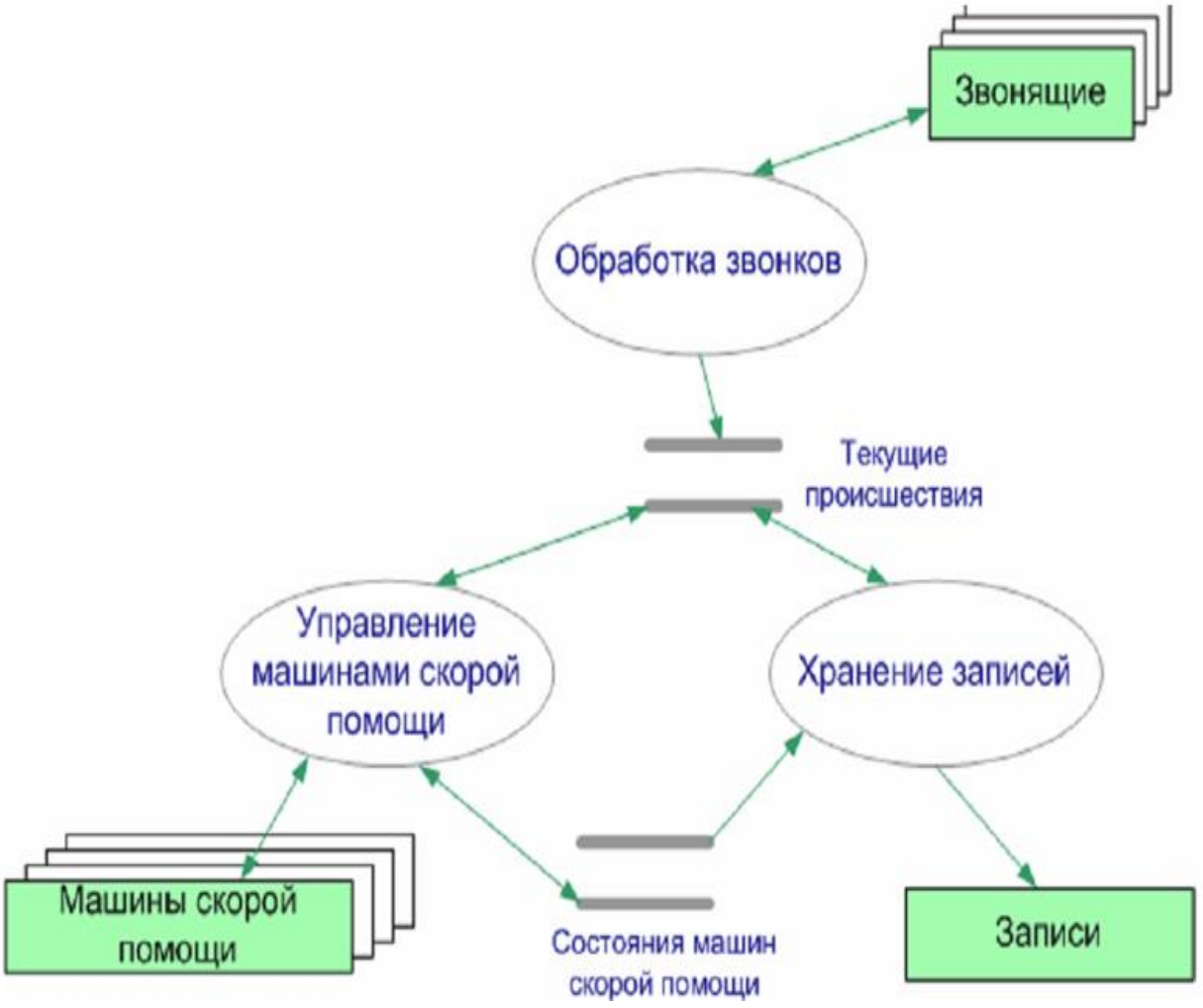


Рис. 3.34. Пример диаграммы DFD – управление скорой помощью

Пример: Детализация модели

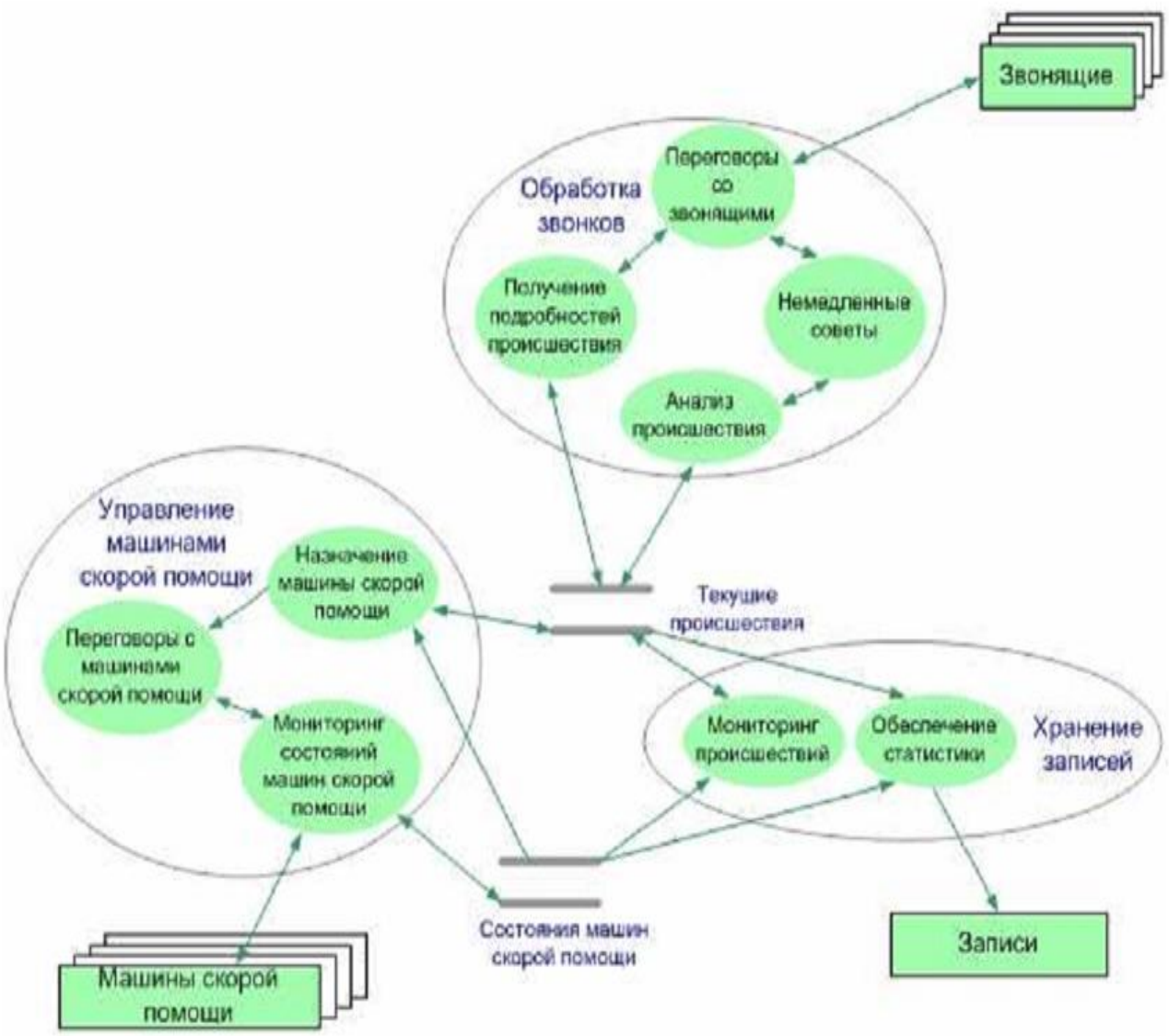


Рис. 3.35. Детализация управление скорой помощью

Пример: Функциональная структура

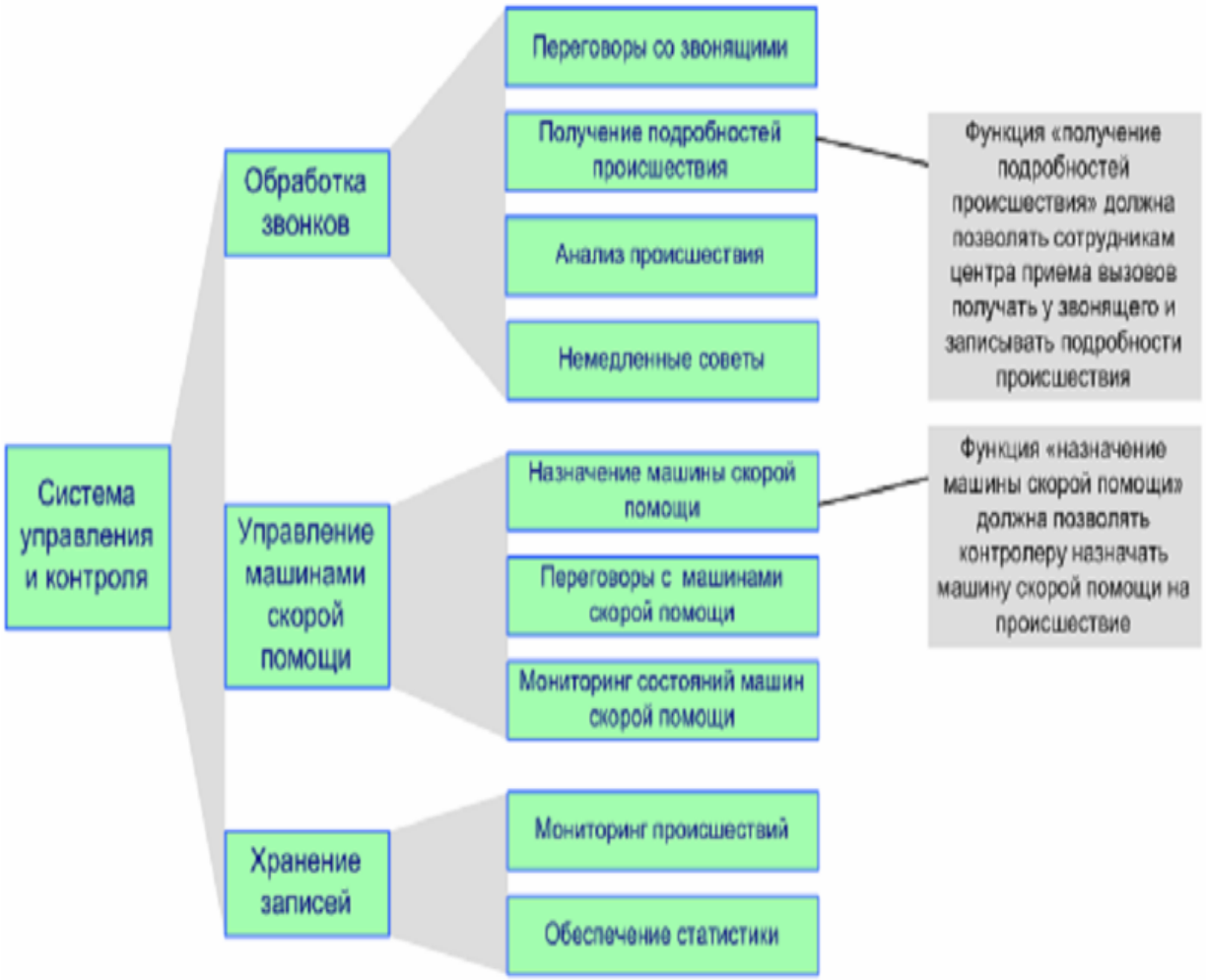


Рис. 3.36. Функции управление скорой помощью

Пример: Системные операции

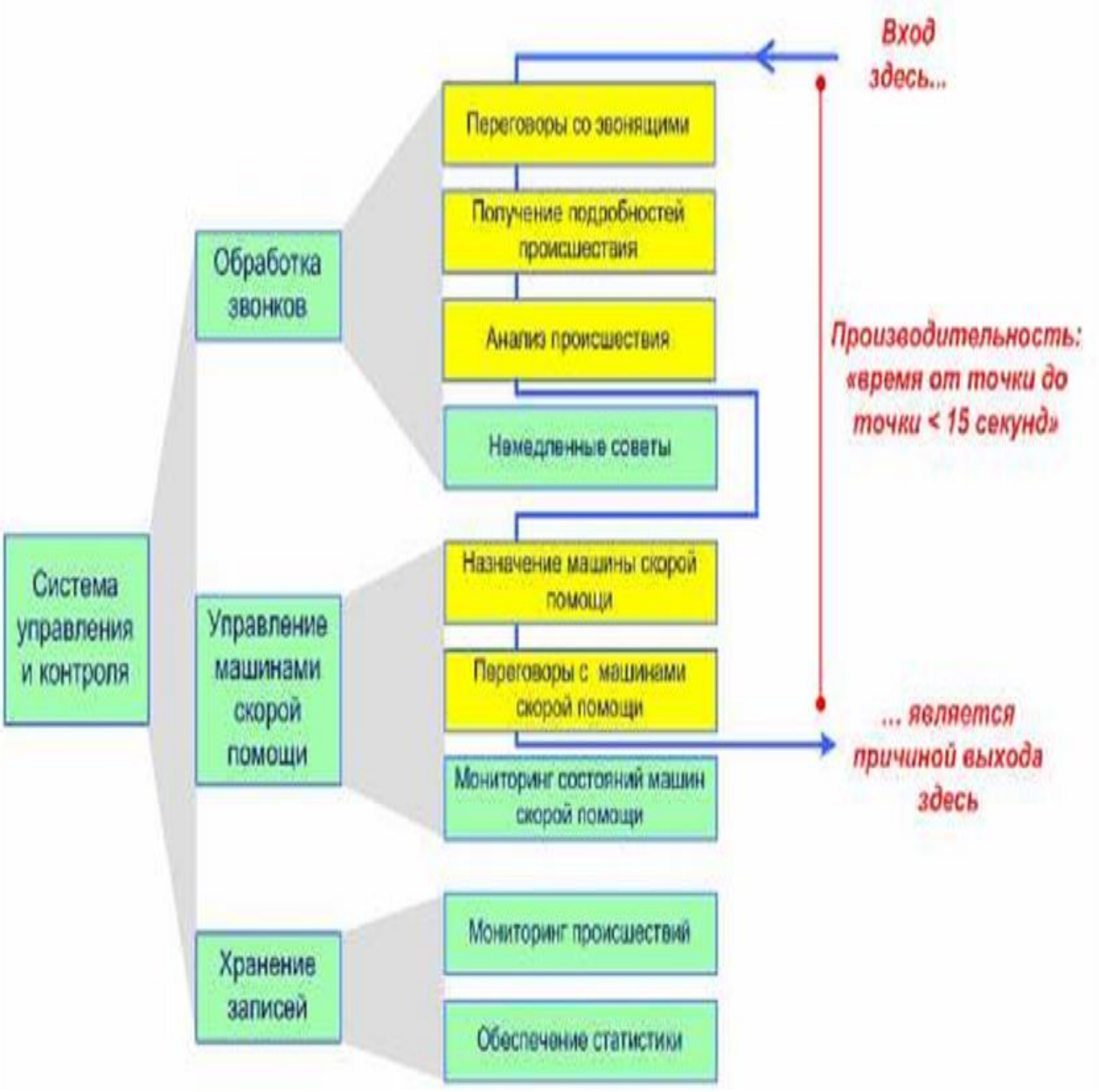


Рис. 3.37. Системные операции управления скорой помощью

Пример: Системные операции

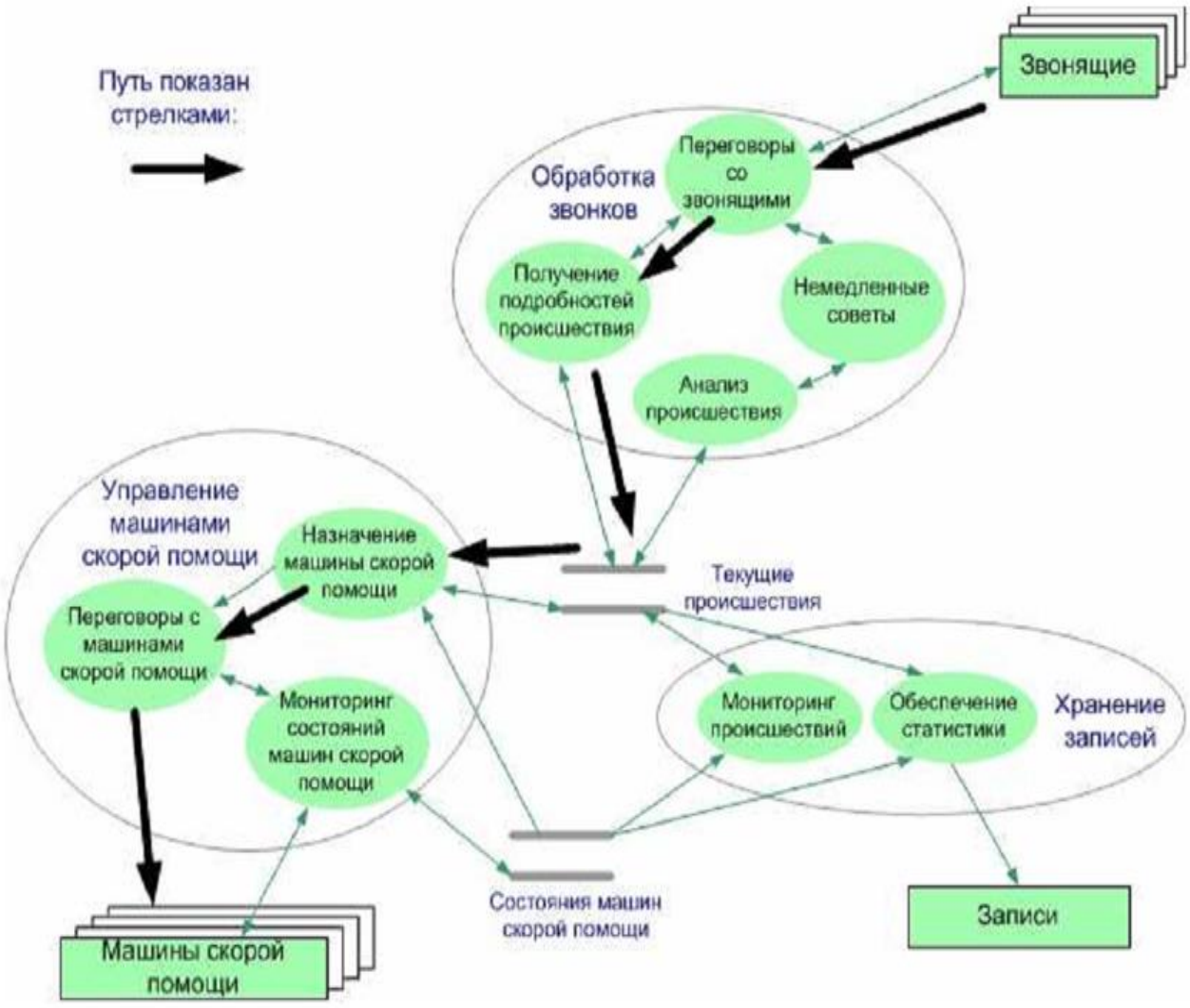


Рис. 3.38. Детализация системные операций управления скорой помощью

Заключение по DFD

- отображают в целом потоки данных и функциональность системы;
- определяют функции, потоки и хранилища данных;
- определяют интерфейсы между функциями;
- обеспечивают базу для получения системных требований;
- имеют инструментальные средства для работы с ними (специальное ПО);
- широко используются в разработке программного обеспечения;
- применимы для проектирования системам вообще (не только для ПО).

3.13. Пример проектирования функций подсистемы обработки и хранения данных

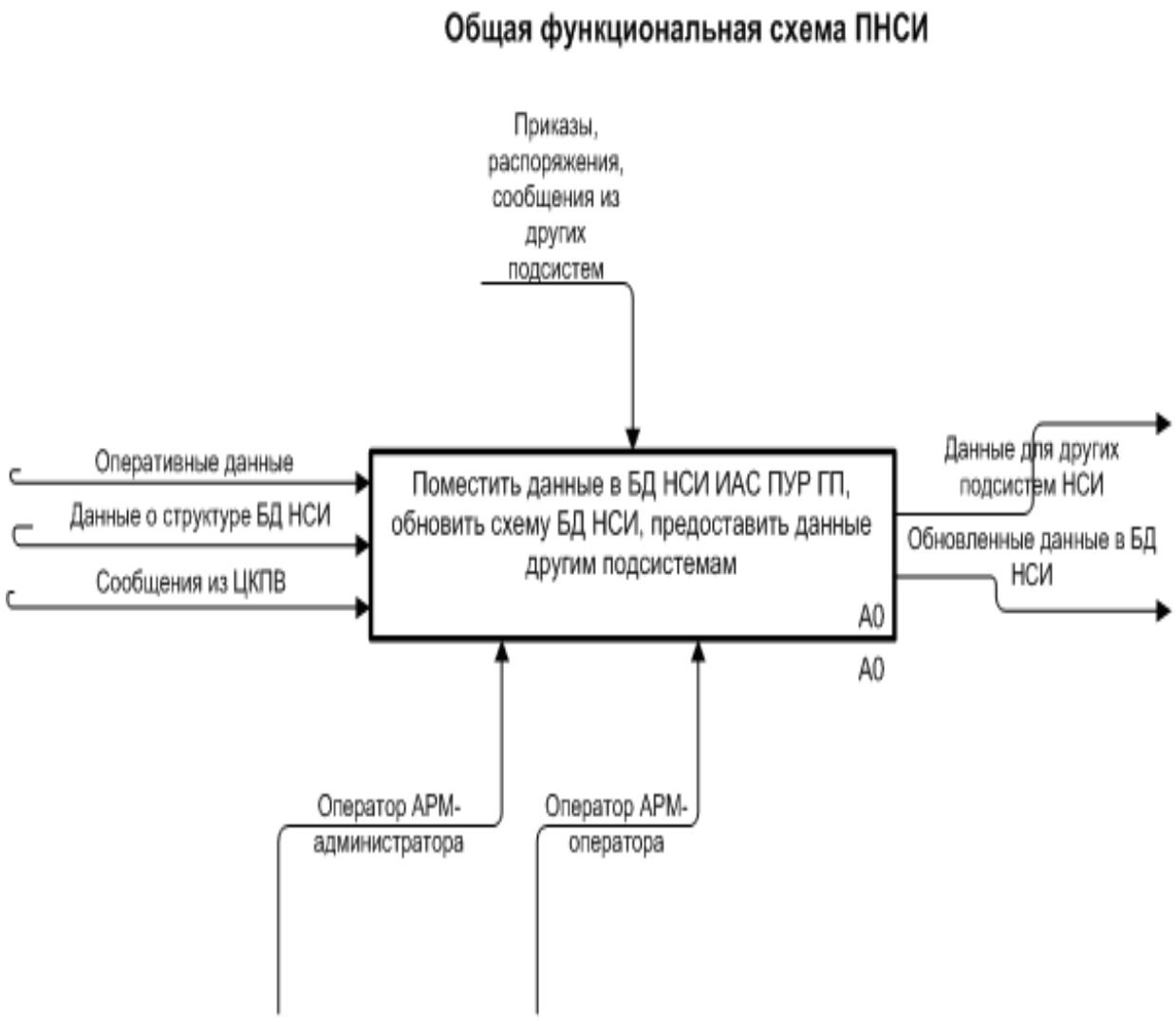


Рис. 3.39.

Общая функциональная схема компонент ПНСИ
АРМ-оператора, АРМ-администратора, обработки сообщений и репликации

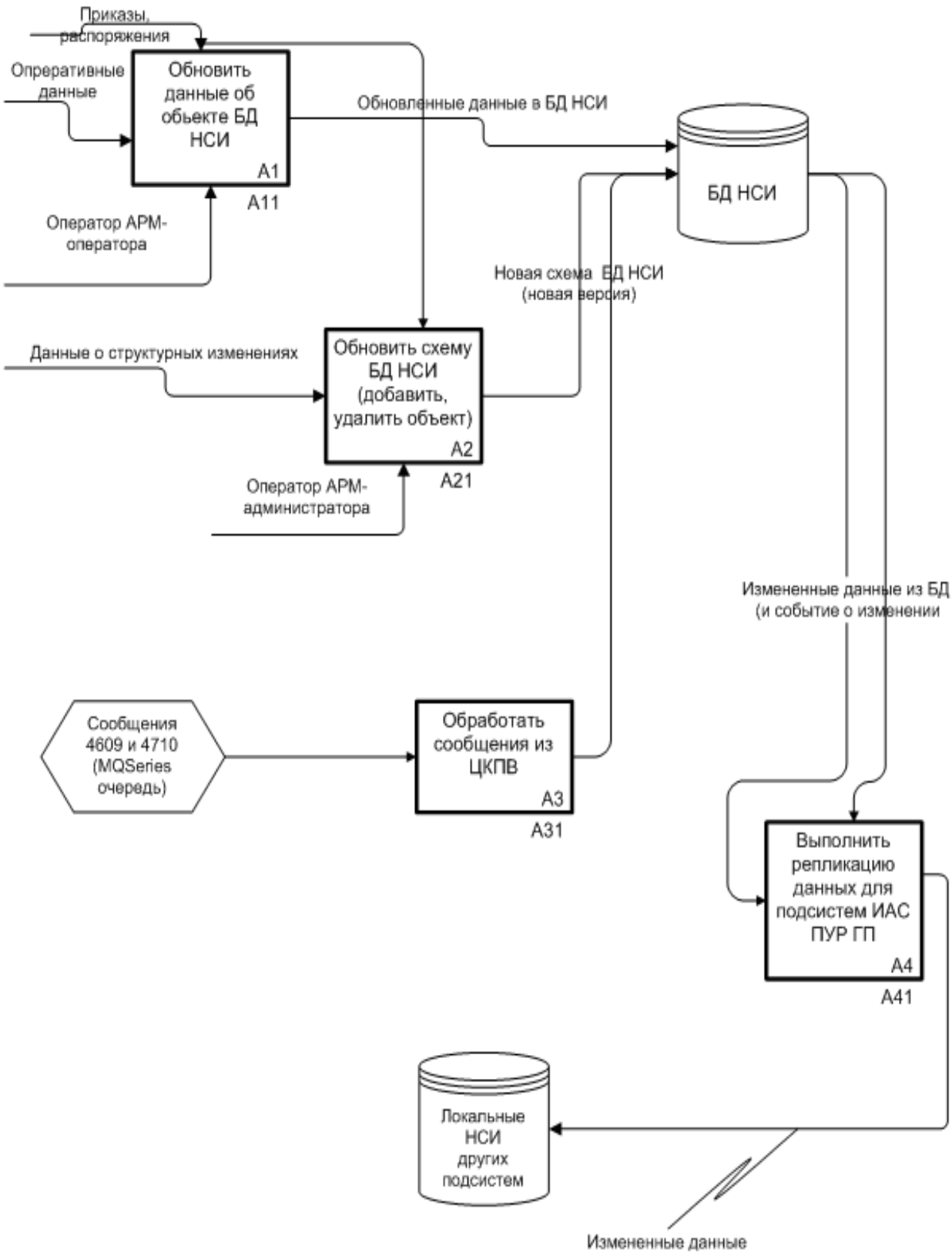


Рис. 3.40.

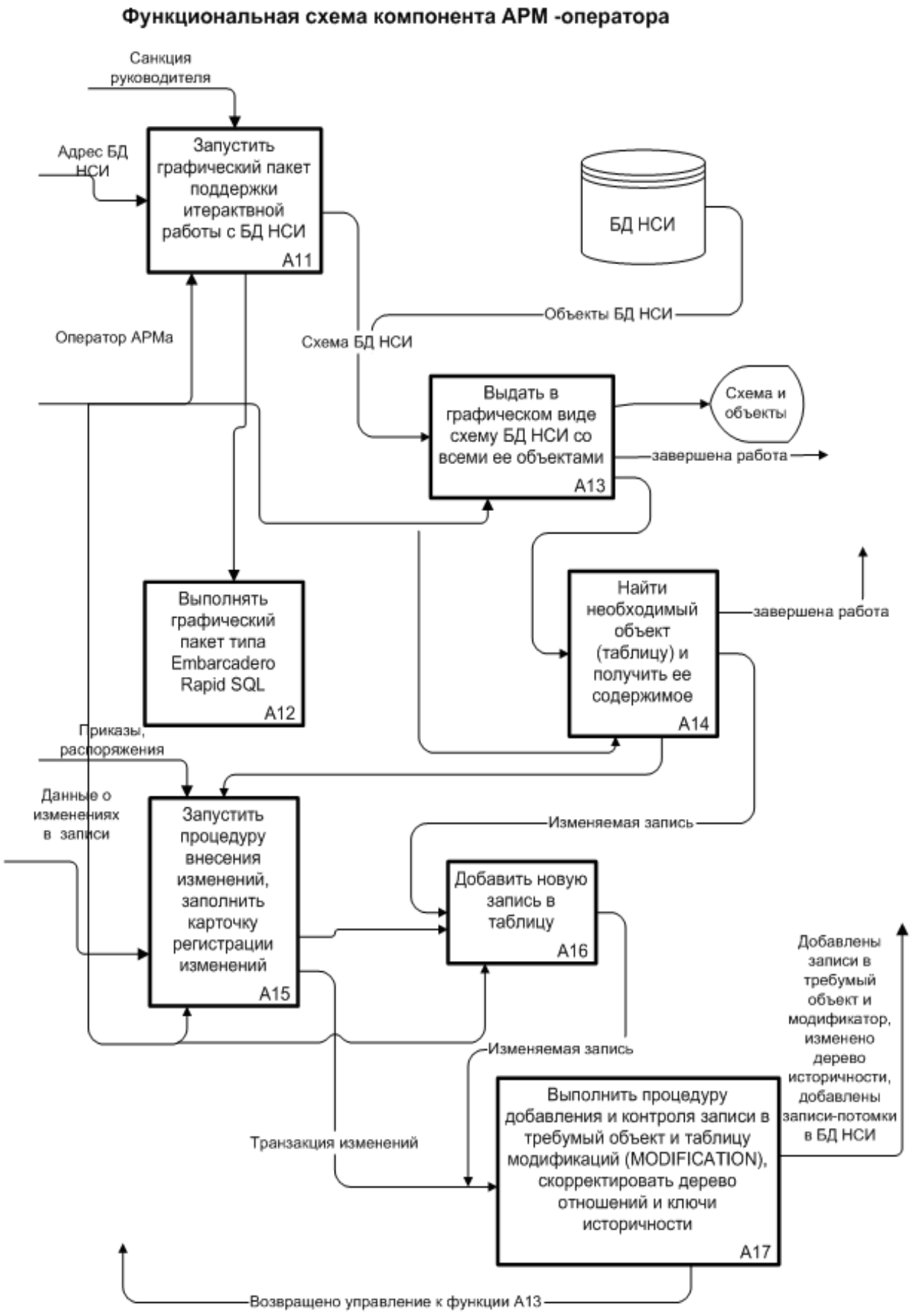

Функциональная схема компонента АРМ -оператора
Санкция руководителя
Адрес БД НСИ
Запустить графический пакет поддержки итерактвной работы с БД НСИ
A11
БД НСИ
Объекты БД НСИ
Оператор АРМа
Схема БД НСИ
Выдать в графическом виде схему БД НСИ со всеми ее объектами
A13
Схема и объекты
завершена работа
Выполнять графический пакет типа Embarcadero Rapid SQL
A12
Найти необходимый объект (таблицу) и получить ее содержимое
A14
завершена работа
Приказы, распоряжения
Данные о изменениях в записи
Запустить процедуру внесения изменений, заполнить карточку регистрации изменений
A15
Изменяемая запись
Добавить новую запись в таблицу
A16
Изменяемая запись
Добавлены записи в требумый объект и модификатор, изменено дерево историчности, добавлены записи-потомки в БД НСИ
Транзакция изменений
Выполнить процедуру добавления и контроля записи в требумый объект и таблицу модификаций (MODIFICATION), скорректировать дерево отношений и ключи историчности
A17
Возвращено управление к функции A13

Рис. 3.41.

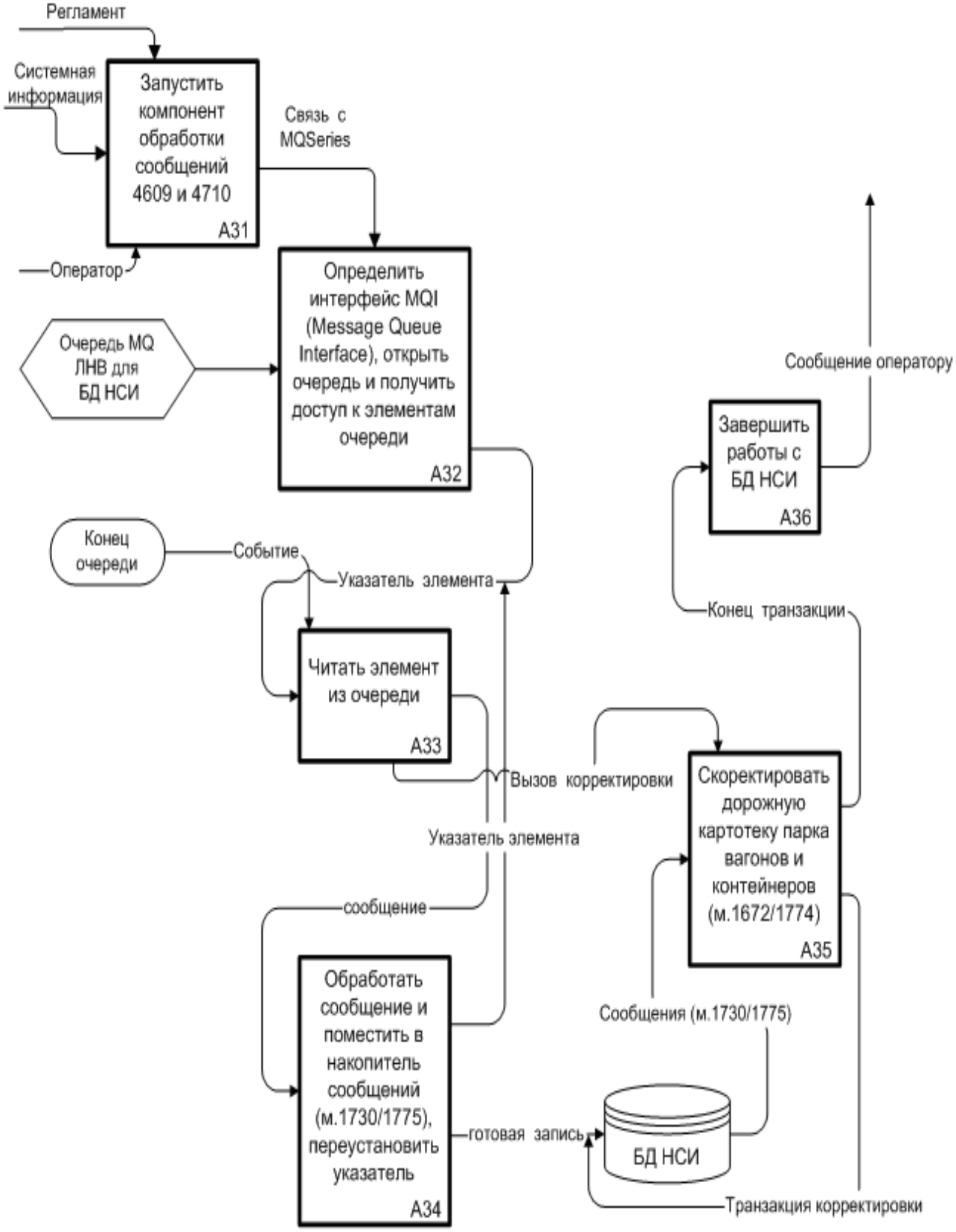

Регламент
Системная информация
Запустить компонент обработки сообщений 4609 и 4710
A31
Связь с MQSeries
Оператор
Определить интерфейс MQI (Message Queue Interface), открыть очередь и получить доступ к элементам очереди
A32
Очередь MQ ЛНВ для БД НСИ
Сообщение оператору
Завершить работы с БД НСИ
A36
Конец очереди
Событие
Указатель элемента
Конец транзакции
Читать элемент из очереди
A33
Вызов корректировки
Скоректировать дорожную картотеку парка вагонов и контейнеров (м.1672/1774)
A35
Указатель элемента
сообщение
Обработать сообщение и поместить в накопитель сообщений (м.1730/1775), переустановить указатель
A34
Сообщения (м.1730/1775)
готовая запись
БД НСИ
Транзакция корректировки

Рис. 3.42.

Функциональная схема репликации данных

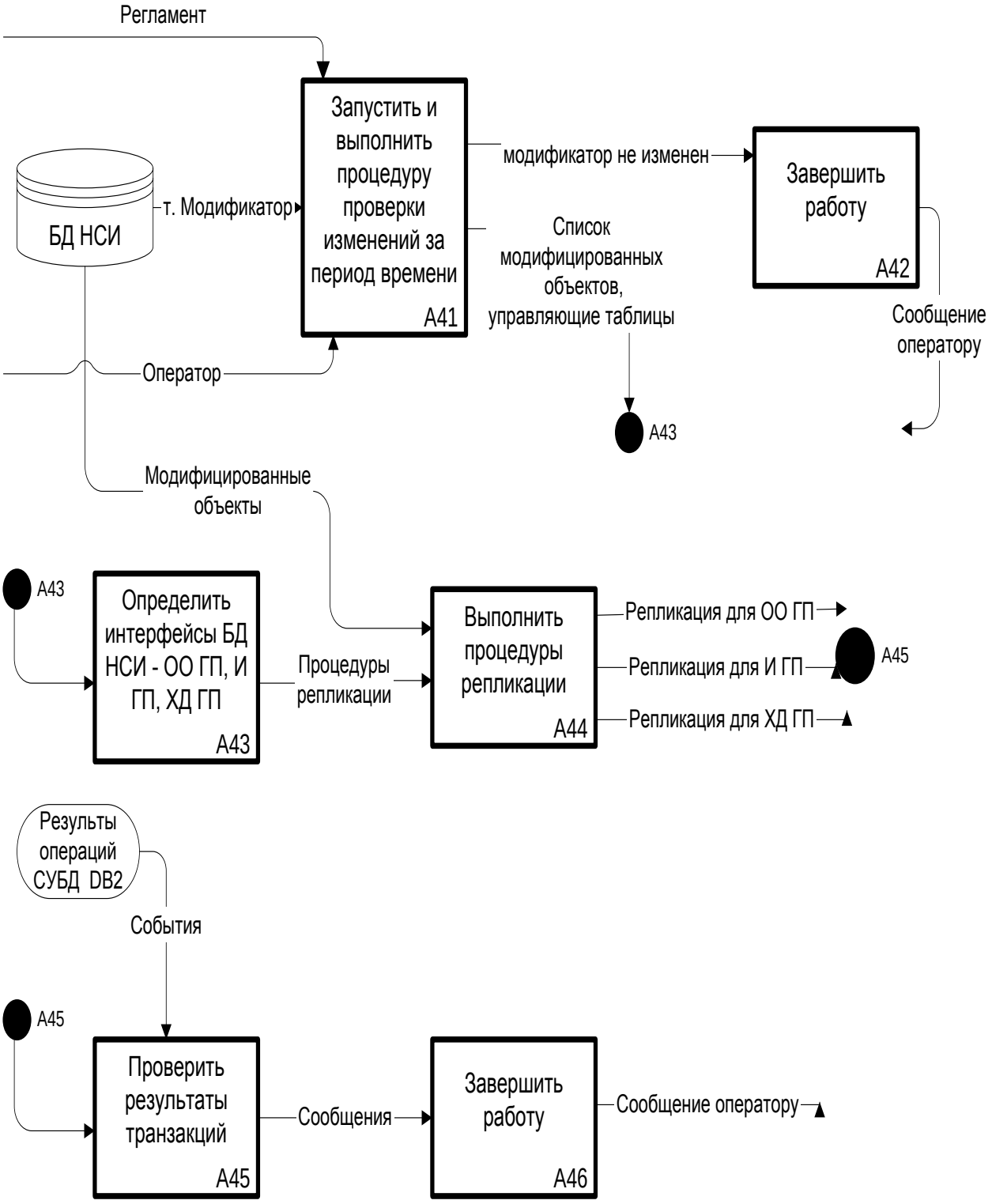


Рис. 3.43.

Пример описания логики работы системы управления данными:

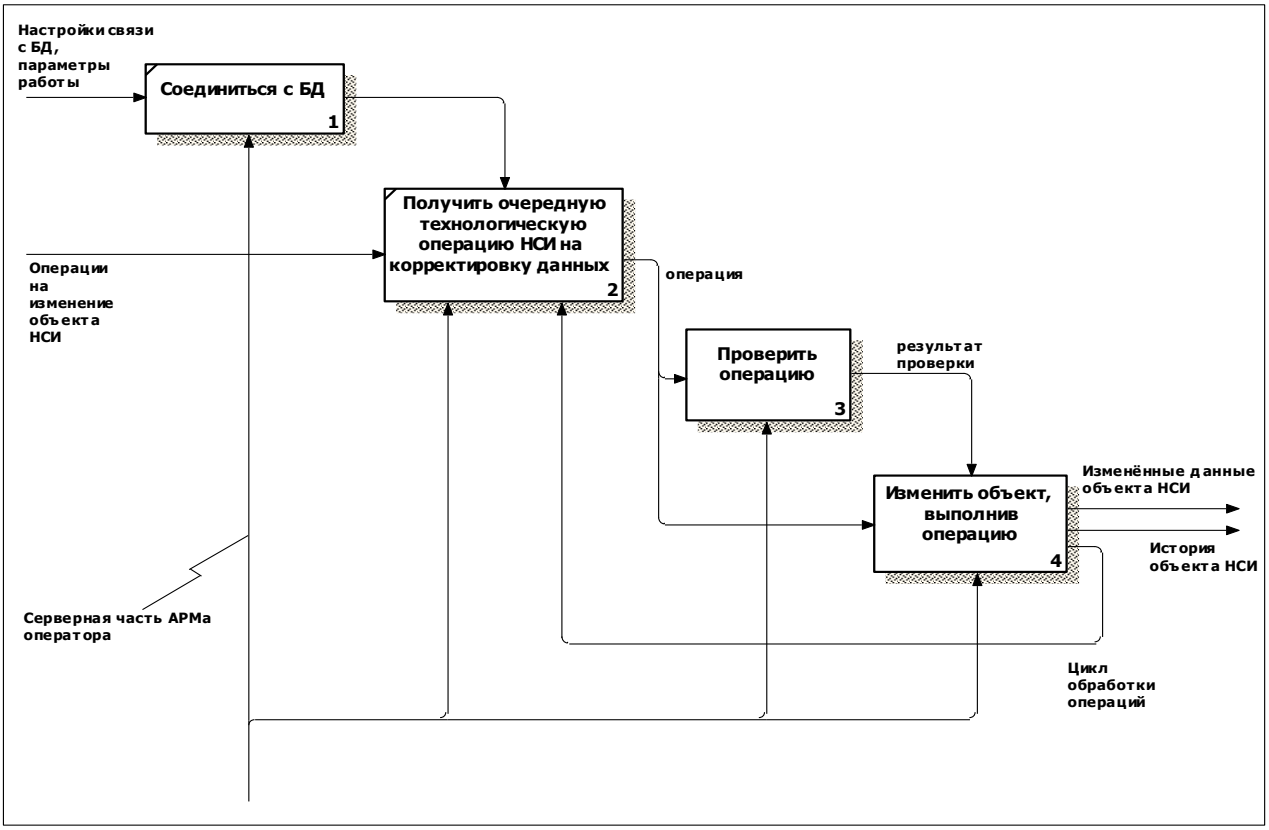


Рис. 3.44.

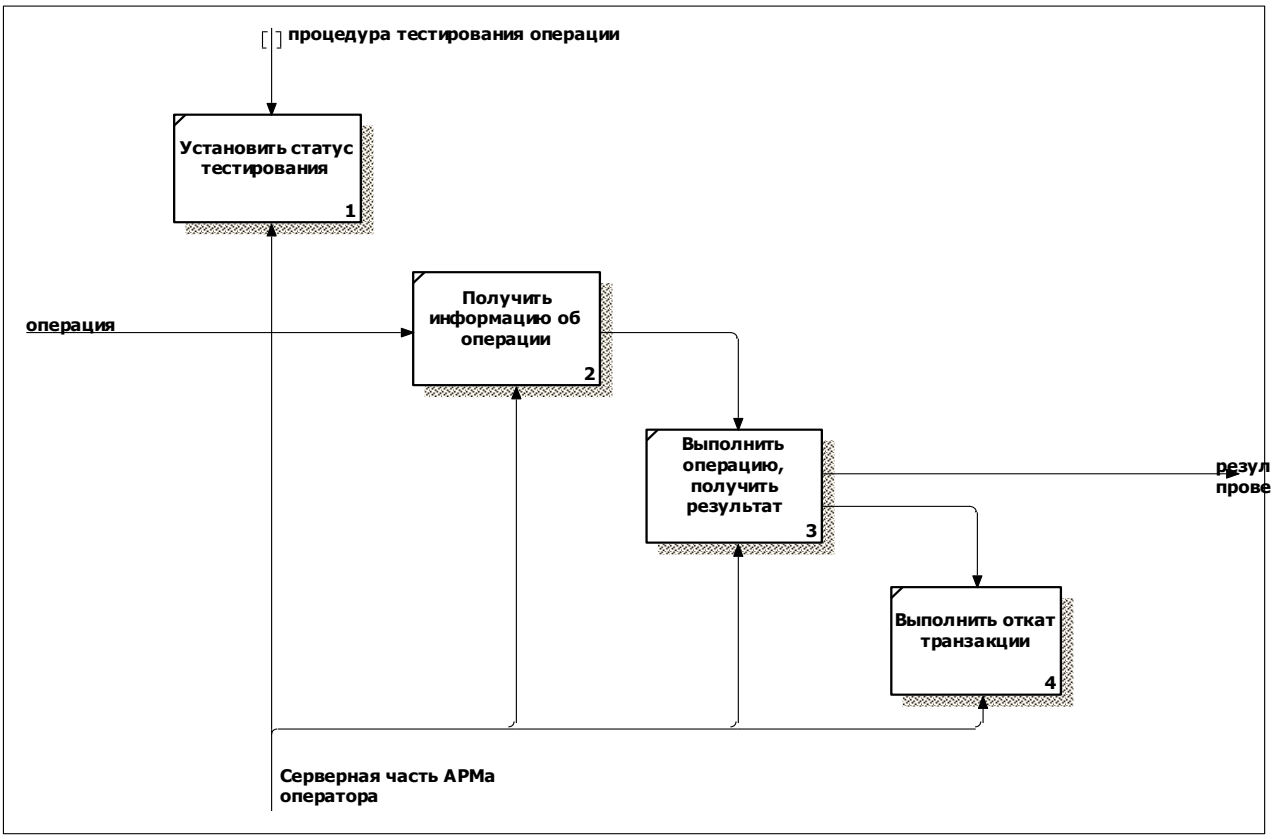


Рис. 3.45.

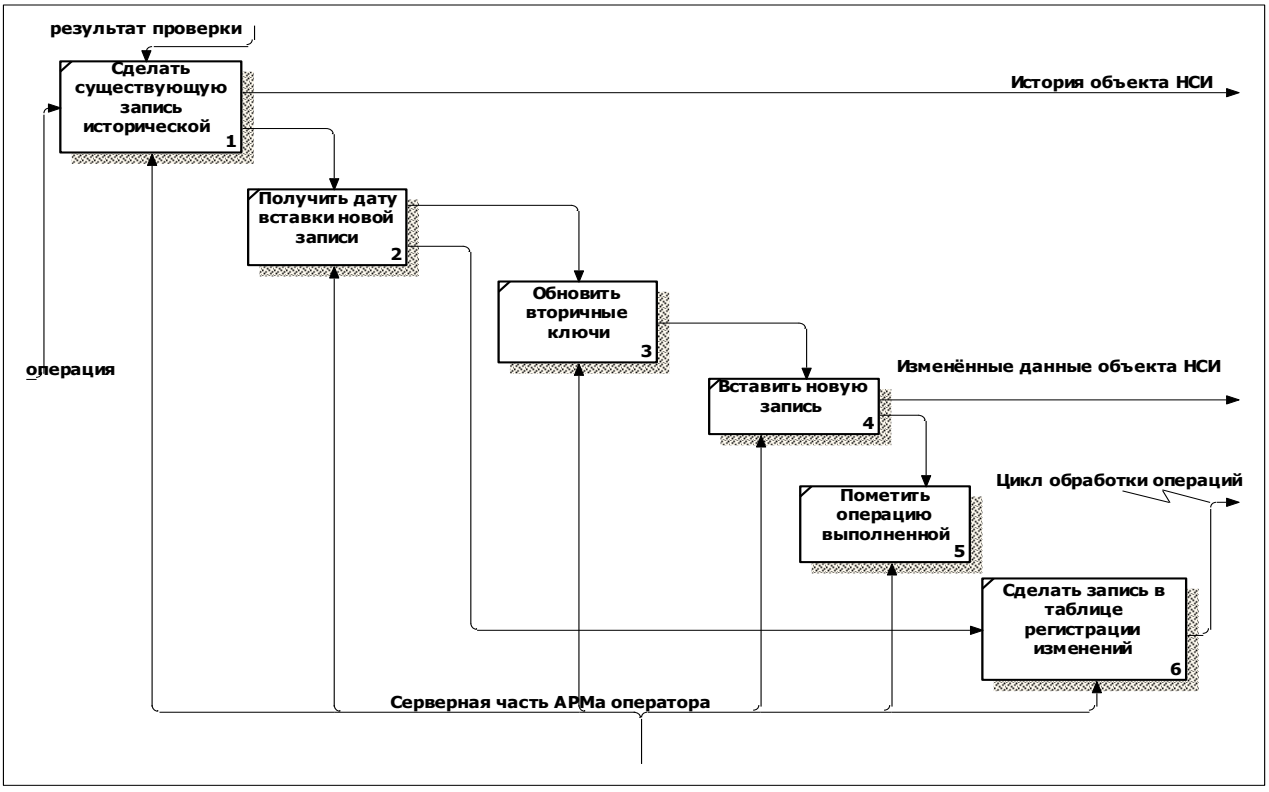


Рис. 3.46.

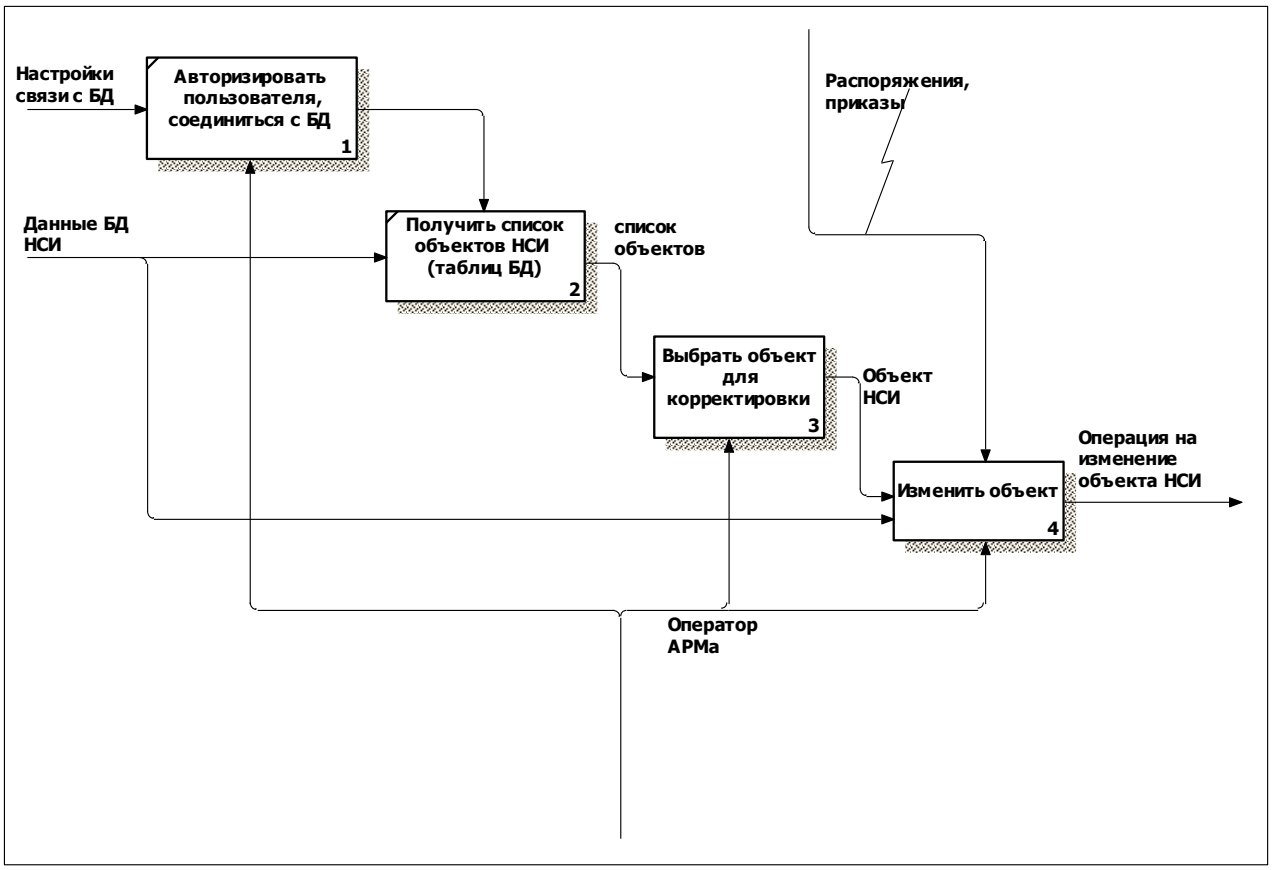


Рис. 3.47.

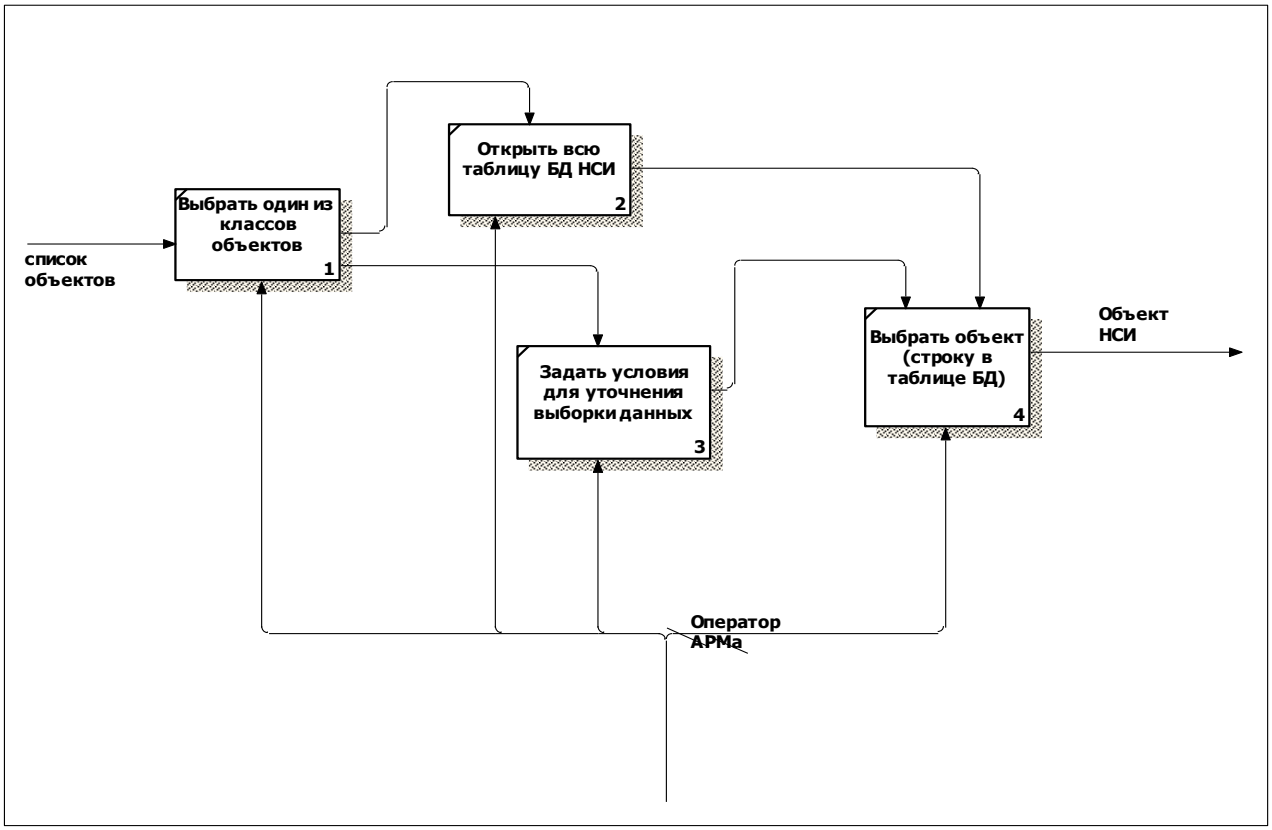


Рис. 3.48.

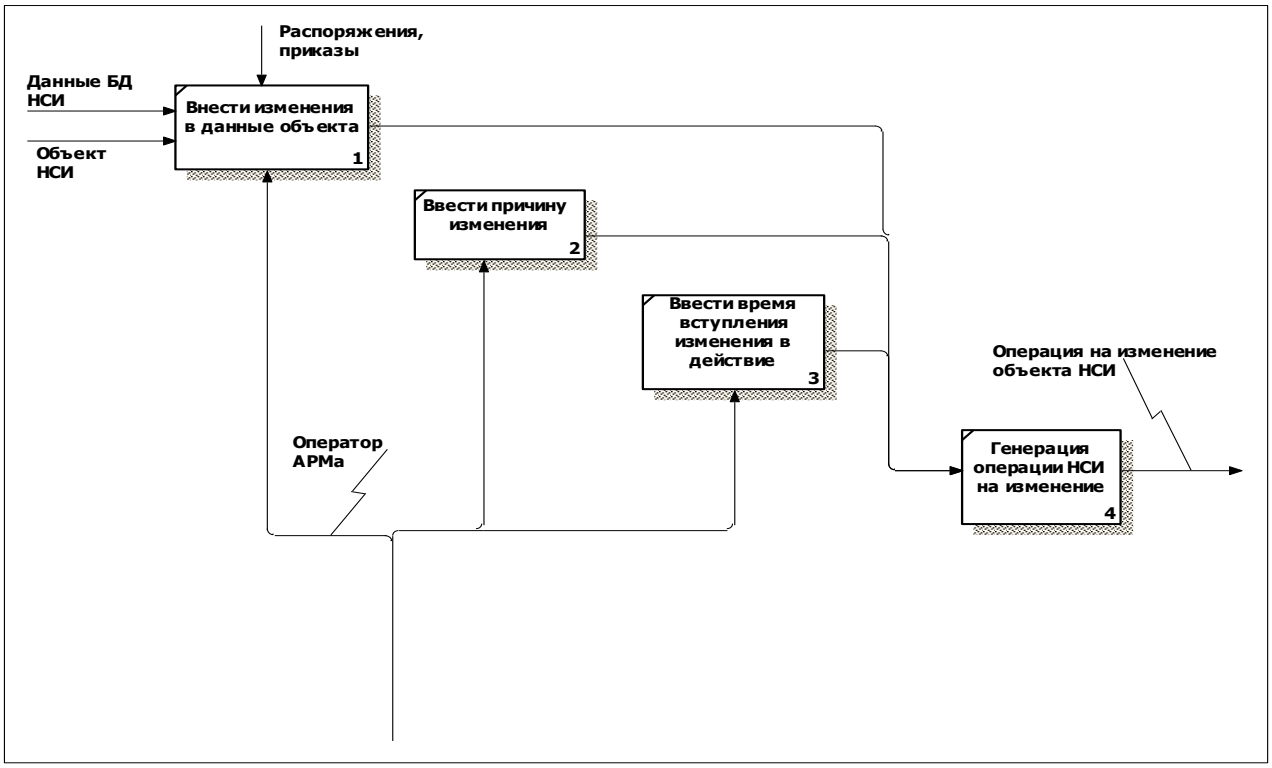


Рис. 3.49.

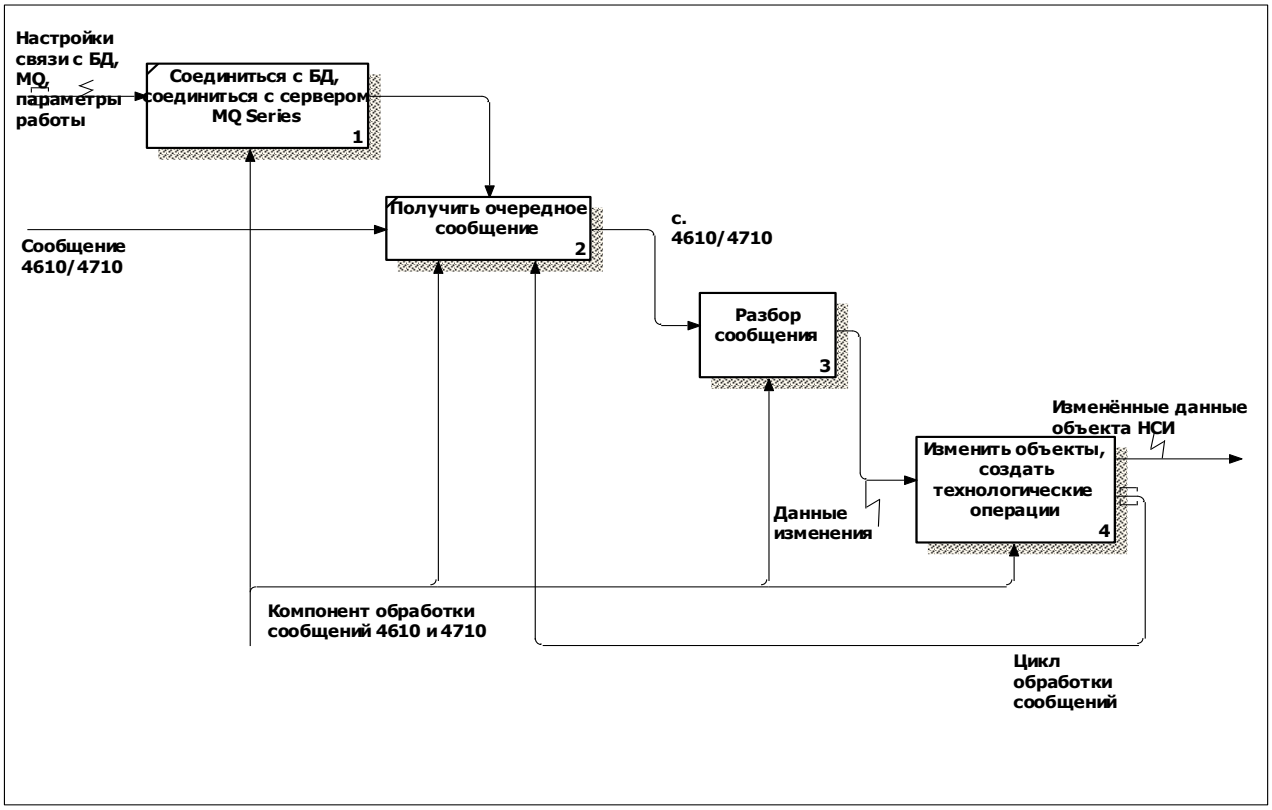


Рис. 3.50.

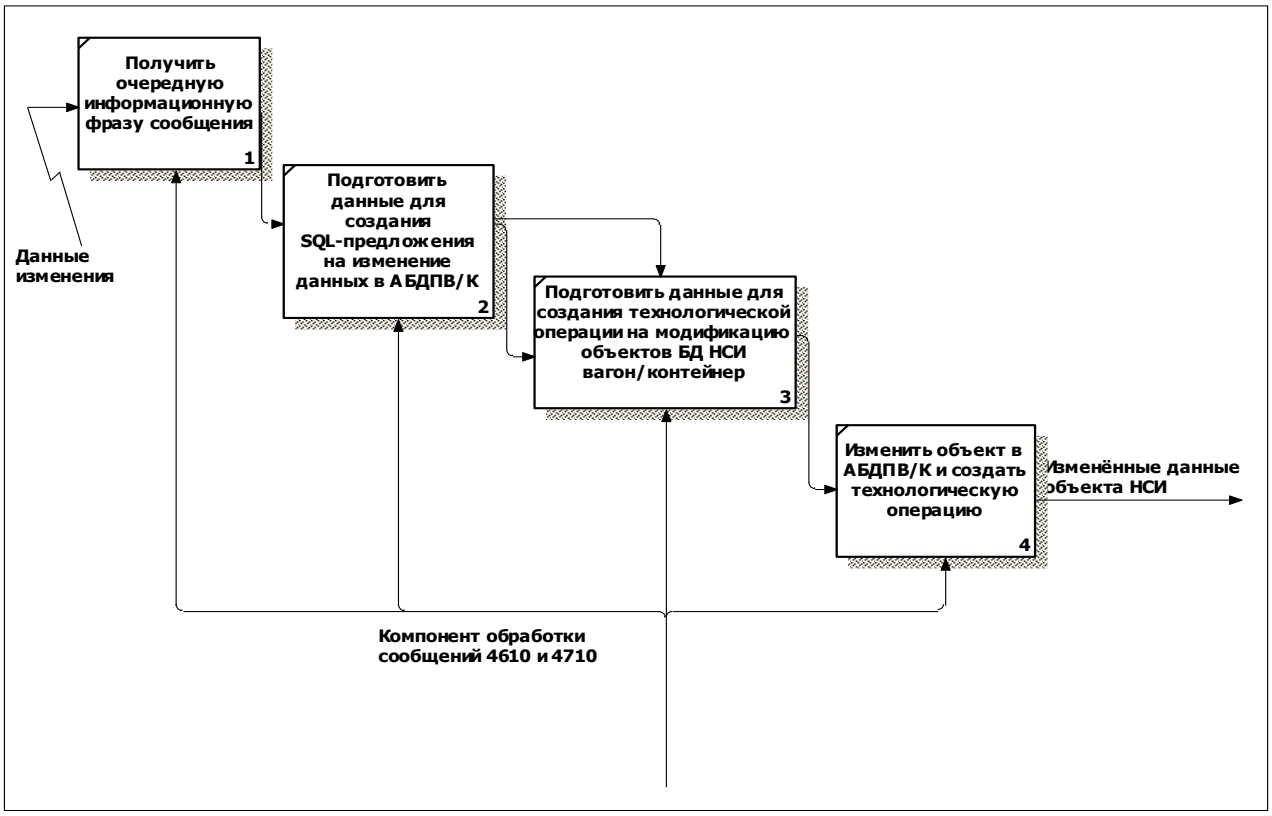

Получить очередную информационную фразу сообщения
1
Подготовить данные для создания SQL-предложения на изменение данных в АБДПВ/К
2
Подготовить данные для создания технологической операции на модификацию объектов БД НСИ вагон/контейнер
3
Изменить объект в АБДПВ/К и создать технологическую операцию
4
Данные изменения
Изменённые данные объекта НСИ
Компонент обработки сообщений 4610 и 4710

Рис. 3.51.

4. IDEF3 – методология описания и моделирования процессов

Введем понятие процесса. **Процесс** – форма представления и методы использования информации о производстве и используемых производственных ресурсах, их характеристиках и ограничениях с точки зрения управления производством. – Это упорядоченный набор функций, охватывающий различные сущности предприятия и завершающийся глобальной целью. Есть и другие определения процесса. Под “процессом” мы понимаем набор операций, которые, взятые вместе, создают результат, имеющий ценность для потребителя.

Бизнес процессы могут быть разного масштаба: масштаба предприятия – в него вовлечены работники нескольких подразделений, бизнес–процесс может не выходить за рамки отдела или производственного цеха и т.п. Внутри одного бизнес процесса часть составляющих его технологических и организационно–деловых процессов может быть организована в отдельный вложенный бизнес–процесс меньшего масштаба. Отдельные технологические и организационно–деловые процессы могут раскладываться на операции (законченные части процесса, выполняемые на одном рабочем месте, такие как выписать накладную, составить договор), которые в свою очередь делятся на функции (законченные части операции, выполняемые одними и теми же средствами – позвонить, записать, фрезеровать и т.п.).

Введем понятие модель. **Модель** – это идеализированное представление достаточно близко отражающее описываемую систему. Мощность модели заключается в ее способности упростить реальную систему, что дает возможность предсказывать факты в системе на основании соответствующих фактов представленных в модели. Основная цель моделирования процесса состоит в том, чтобы идентифицировать и документировать все аспекты работы системы.

Стандарт IDEF3 это методология описания процессов, рассматривающая последовательность выполнения и причинно–следственные связи между ситуациями и событиями для структурного представления знаний о системе, и описания изменения состояний объектов, являющихся составной частью описываемых процессов.

Моделирование в стандарте IDEF3 производится с использованием графического представления процесса, материальных и информационных потоков в этом процессе, взаимоотношений между операциями и объектами в процессе. При помощи IDEF3 описывают логику выполнения работ, очередность их запуска и завершения, т.е. IDEF3 предоставляет инструмент моделирования сценариев действий сотрудников организации, отделов, цехов и т.п., например порядок обработки заказа или события, на которые необходимо реагировать за конечное время, выполнение действий по производству товара им т.д.

Метод IDEF3 использует категорию **Сценариев** для упрощения структуры описаний сложного многоэтапного процесса. Сценарии определяют граничные условия описания. Здесь под сценарием (Scenario) будем понимать повторяющуюся последовательность ситуаций или действий, которые описывают типичный класс проблем, присутствующих в организации или системе, описание

последовательности изменений свойств объекта, в рамках рассматриваемого процесса (например, описание последовательности обработки менеджером заявки).

IDEF3 предоставляет инструментарий для наглядного исследования и моделирования сценариев выполнения процессов. Метод позволяет проводить описание с необходимой степенью подробности посредством использования декомпозиции. IDEF3 как инструмент моделирования фиксирует следующую информацию о процессе:

Объекты, которые участвуют при выполнении сценария,

Роли, которые выполняют эти объекты (например, агент, транспорт и т.д.),

Отношения между работами в ходе выполнения сценария процесса,

Состояния и изменения, которым подвергаются объекты,

Время выполнения и контрольные точки синхронизации работ,

Ресурсы, которые необходимы для выполнения работ.

Основная цель IDEF3 состоит в том, чтобы предоставить специалисту (эксперту) предметной области инструмент – методологию структурного анализа, при помощи, которой он сможет представлять знания о выполнении операций в системе или организации в целом. Цель описания может состоять как в документальном оформлении и распространении знаний о процессе, так и в идентификации противоречивости или несовместимости выполнения отдельных операций. Приобретение знаний допускается прямым сбором утверждений о практике выполнения процессов и возникновении различных ситуаций в процессе в форме, которая является наиболее естественной.

Итак, методология IDEF3 предназначена для обеспечения сбора данных о процессе. Описание процесса IDEF3 представляет собой структурированную базу знаний, которая состоит из набора диаграмм описания процесса, объектных диаграмм изменения состояний объектов и уточняющих форм. Средства документирования и моделирования IDEF3 позволяют выполнять следующие задачи:

Документировать имеющиеся данные о технологии выполнения процесса, выявленные, скажем, в процессе опроса специалистов предметной области, ответственных за организацию рассматриваемого процесса или участвующих в нем.

Анализировать существующие процессы и разрабатывать новые.

Определять и анализировать точки влияния потоков сопутствующего документооборота на сценарий технологических процессов.

Определять ситуации, в которых требуется принятие решения, влияющего на жизненный цикл процесса, например изменение конструктивных, технологических или эксплуатационных свойств конечного продукта.

Содействовать принятию оптимальных решений при реорганизации процессов.

Разрабатывать имитационные модели технологических процессов, по принципу "КАК БУДЕТ, ЕСЛИ..."

IDEF3 – уникальная методология способная фиксировать и структурировать описание работы системы. При этом сбор сведений может производится из многих

источников, что позволяет зафиксировать информацию от экспертов о поведении системы, а не наоборот – строить модель, чтобы приблизить поведение системы. Эта особенность IDEF3 как инструмента моделирования выделяется среди основных характеристик, отличающих IDEF3 от альтернативных предложений.

В стандарте IDEF3 существуют два типа диаграмм, представляющие описание одного и того же сценария технологического процесса в разных ракурсах. Диаграммы, относящиеся к первому типу, называются диаграммами Описания Последовательности выполнения Процесса (Process Flow Description Diagrams, PFDD). Второй тип диаграмм описывает Состояния Объекта и Трансформаций в Процессе и называется (Object State Transition Network, OSTN) Сеть изменений объектных состояний.

Если диаграммы PFDD описывают процесс, то другой класс диаграмм IDEF3 OSTN используются для иллюстрации трансформаций объекта, которые происходят на каждой стадии выполнения соответствующих работ ("С точки зрения объекта"). Состояния объекта и Изменение состояния являются ключевыми понятиями OSTN диаграммы.

Диаграммы Процесса имеют тенденцию быть наиболее знакомыми и широко использовали компонент метода IDEF3. Эти диаграммы обеспечивают механизм визуализации для центрированных процессом описаний сценария. Графические элементы, которые включают диаграммы процесса, включают Модуль полей (UOB) Поведения, связей старшинства, перекрестков, объектов ссылки, и примечаний.

Объекты ссылки и примечания – конструкции, которые являются общими для диаграмм процесса и объектов.

Каждый из графических элементов, используемых для разработки диаграмм процесса, представлен ниже, вместе с обсуждениями того, как формулировать более комплексные инструкции, использующие эти графические элементы. Обсуждение начинается с наиболее фундаментального из этих стандартных блоков – функционального элемента (UOB).

Основы создания диаграмм IDEF3 в среде BPwin аналогичны созданию диаграмм IDEF0. Только после запуска среды следует выбрать область построения IDEF3.

4.1. Функциональный элемент

Описание процесса представляет всевозможные ситуации (процессы, функции, действия, акты, события, сценарии, процедуры, операции или решения), которые могут происходить в моделируемой системе в логических и временных отношениях. Каждый процесс представлен полем, отображающим название процесса (рис. 4.1.). Номер идентификатора процесса назначается по порядку. В правом нижнем углу UOB элемента располагается ссылка (IDEF0/USER или другие) и используется для указания ссылок либо на элементы из функциональной

модели IDEF0, либо для указания на отделы или конкретных исполнителей, которые будут выполнять указанную работу.

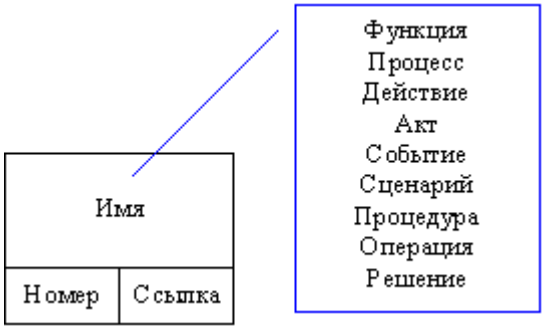


Рис. 4.1. Синтаксис UOB элемента

4.2. Элемент связи

Связь необходима для связи элементов диаграммы и описания динамики происходящих процессов. Связи используются, прежде всего, для обозначения отношений между функциональными элементами UOB. Для отображения временной последовательности выполнения сценариев в диаграммах описания процесса используются два основных типа связей: связи старшинства и относительные связи. Для описания специфических отношений между элементами предназначены четыре дополнительных типа связей – сдерживаемых связей старшинства. Использование в IDEF3 диаграммах описания процесса различных типов связей дает возможность пользователям метода фиксировать дополнительные информацию о специфике отношений между элементами диаграммы. Символы, которые представляют каждый тип связей, изображены на рис. 4.2.

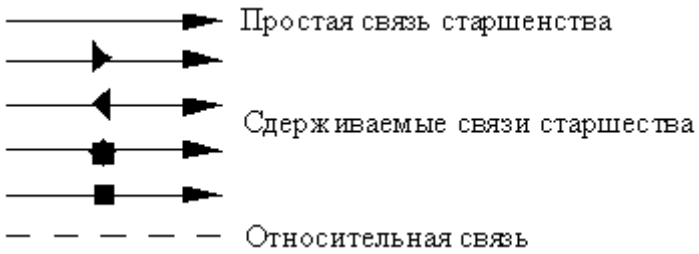


Рис. 4.2. Типы связей в диаграммах описания процесса

4.2.1. Связи старшинства

Связи Старшинства выражают временные отношения старшинства между элементами диаграммы. При этом первый элемент должен завершиться прежде,

чем начнет выполняться следующий. Графически стрелка предшествования (старшинства) отображается сплошной линией с одиночной стрелкой (рис. 4.3.).

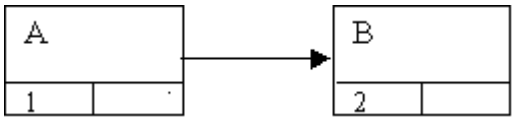


Рис. 4.3. Семантика использования связи старшинства

4.2.2. Сдерживаемые связи старшинства

Данные виды связей не представлены ни в одном из CASE–продуктов, поддерживающих методологию IDEF3. Сдерживаемые связи старшинства указывают (в дополнение к семантике запуска связей простого старшинства):

элементу A нужно предшествовать элементом B,

элементу B нужно предшествовать элементом A,

любой элемент должен сопроводиться элементом B, и что элементу B нужно предшествовать элементом A.

Эти связи добавляют дополнительные условия к системе. Эти дополнительные условия не только выражают то, как система работает, но и устанавливают требования к тому, как система должна себя вести.

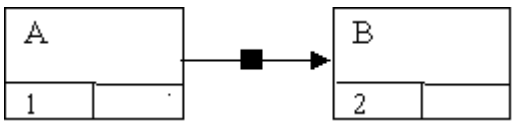


Рис. 4.4. Обобщенное представление сдерживаемых связей предшествования

4.2.3. Относительные связи

Использование относительной связи указывает на тот факт, что между взаимодействующими элементами диаграммы описания процесса существует отношения неопределенного типа. Относительные связи графически показываются как пунктирные линии.

4.2.4. Связь поток объектов

Тип связи поток объектов предложен разработчиками CASE–средств, поддерживающих моделирование в стандарте IDEF3. Графически эта связь показываются как сплошная линия с двойной стрелкой (см. рис. 4.5.). Этот тип

связи выражает перенос одного или нескольких объектов от одного функционального элемента к другому. Этот вид связи элементов IDEF3 наследует все свойства простой связи старшинства. Таким образом, значение связи поток объектов таково: между UOB элементами происходит передача объекта(ов), причем первый элемент UOB должен завершиться прежде, чем начнет выполняться следующий.

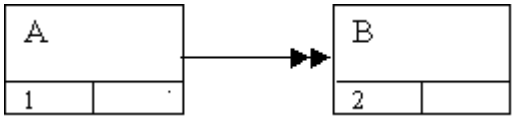


Рис. 4.5. Представление связи поток объектов

4.3. Перекресток

Перекрестки используются для отображения логики отношений между множеством событий и временной синхронизации активизации элементов диаграмм IDEF3. Различают перекрестки для слияния (Fan–in Junction) и разветвления (Fan–out Junction) стрелок.

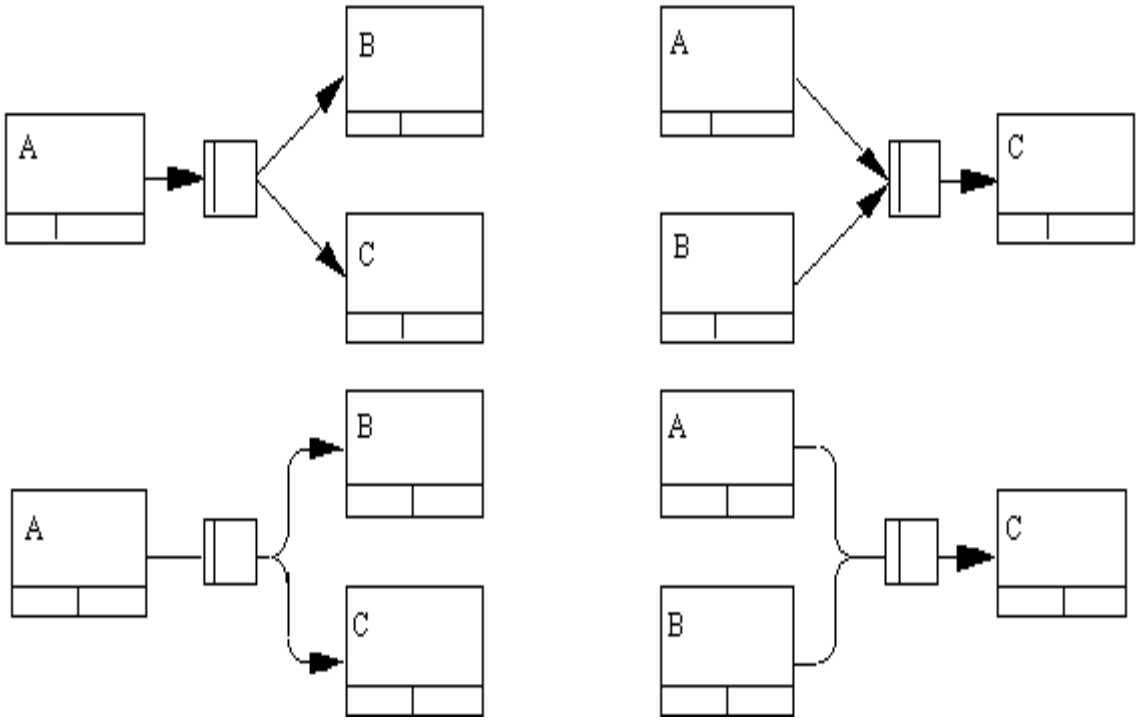


Рис. 4.6. Перекрестки разветвления и слияния

Перекресток не может использоваться одновременно для слияния и для разветвления. При внесении перекрестка в диаграмму необходимо указать тип перекрестка. Тип перекрестка определяет логику и временные параметры

отношений между элементами диаграммы. Все перекрестки в PFDD диаграмме нумеруются, каждый номер имеет префикс "J".

4.3.1. Типы перекрестков

Тип перекрестка обозначается на элементе как:
& — логический И
O – логический ИЛИ
X – логический перекресток НЕЭКВИВАЛЕНТНОСТИ.

Стандарт IDEF3 предусматривает разделение перекрестков типа & и O на синхронные и асинхронные. Это разделение позволяет учитывать в диаграммах описания процессов синхронизацию времени активизации. Более подробно этот вопрос будет рассмотрен далее на примерах.

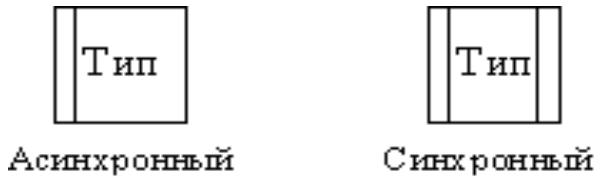


Рис. 4.7. Пример обозначения синхронности и асинхронности перекрестков

Для последующего изложения материала необходимо ввести понятие – График запуска. График запуска – это визуальное отображение временной последовательности выполнения UOB элементов. Пример графика запуска приведен на рис. 4.8. Визуальное отображение на графике запуска временной последовательности выполнения UOB элементов поможет правильно понять как перекрестки описывают логику отношений между элементами диаграммы описания процессов и каким образом перекрестки позволяют синхронизировать по времени выполнение UOB элементов.

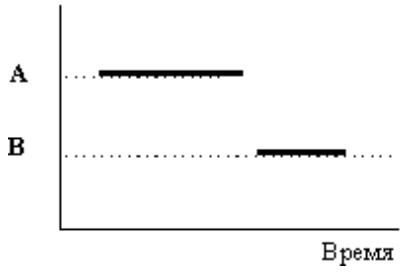


Рис. 4.8. Пример графика запуска

4.3.2. Значения комбинаций перекрестков

Методология IDEF3 использует пять логических типов для моделирования возможных последовательностей действий процесса в сценарии:

Наименование	**Смысл в случае слияния стрелок (Fan–in Junction)**	**Смысл в случае разветвления стрелок (Fan–out Junction)**
Asynchronous AND	Все предшествующие процессы должны быть завершены	Все следующие процессы должны быть запущены
Synchronous AND	Все предшествующие процессы завершены одновременно	Все следующие процессы запускаются одновременно
Asynchronous OR	Один или несколько предшествующих процессов должны быть завершены	Один или несколько следующих процессов должны быть запущены
Synchronous OR	Один или несколько предшествующих процессов завершаются одновременно	Один или несколько следующих процессов запускаются одновременно
XOR	Только один предшествующий процесс завершен	Только один следующий процесс

Примеры:

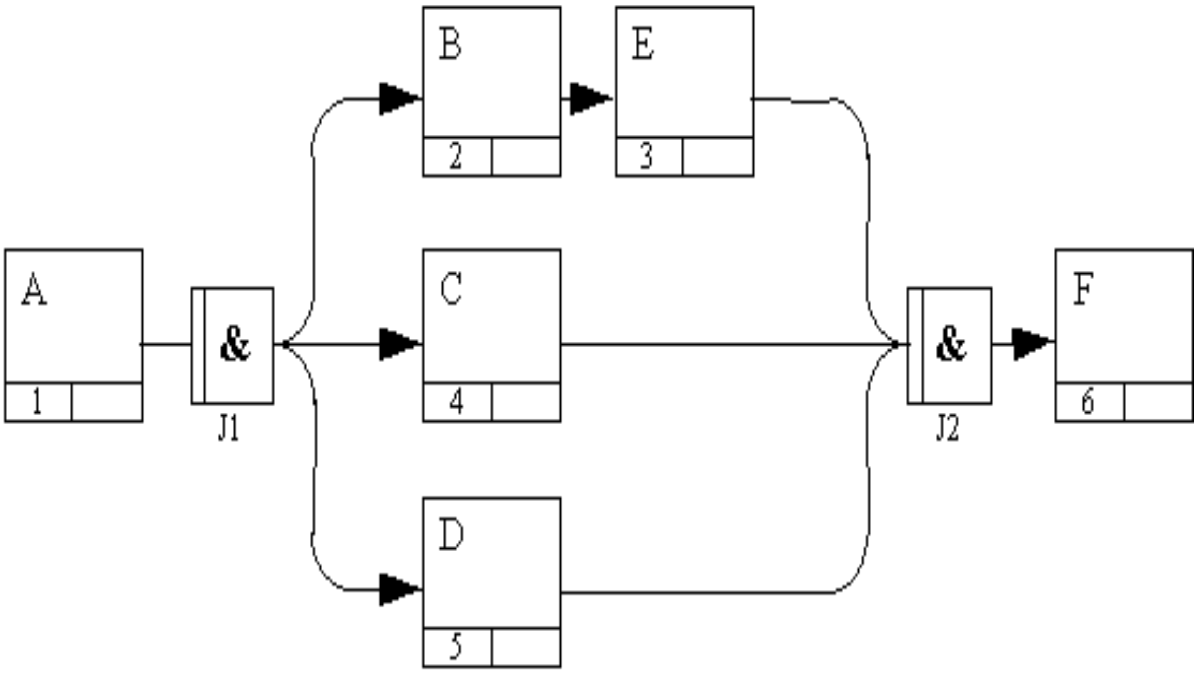


Рис. 4.9. Использование перекрестков асинхронный AND

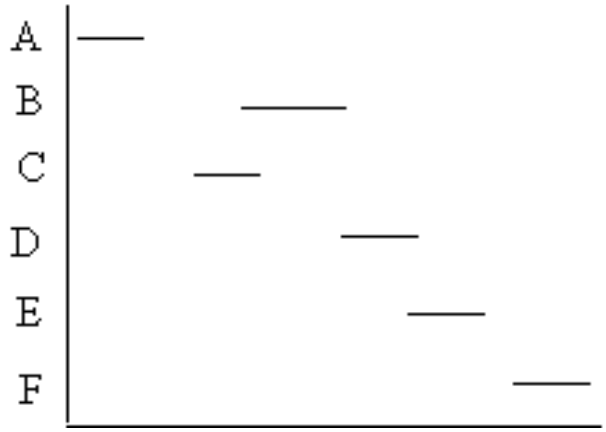


Рис. 4.10. Возможный график запуска для рис. 4.9

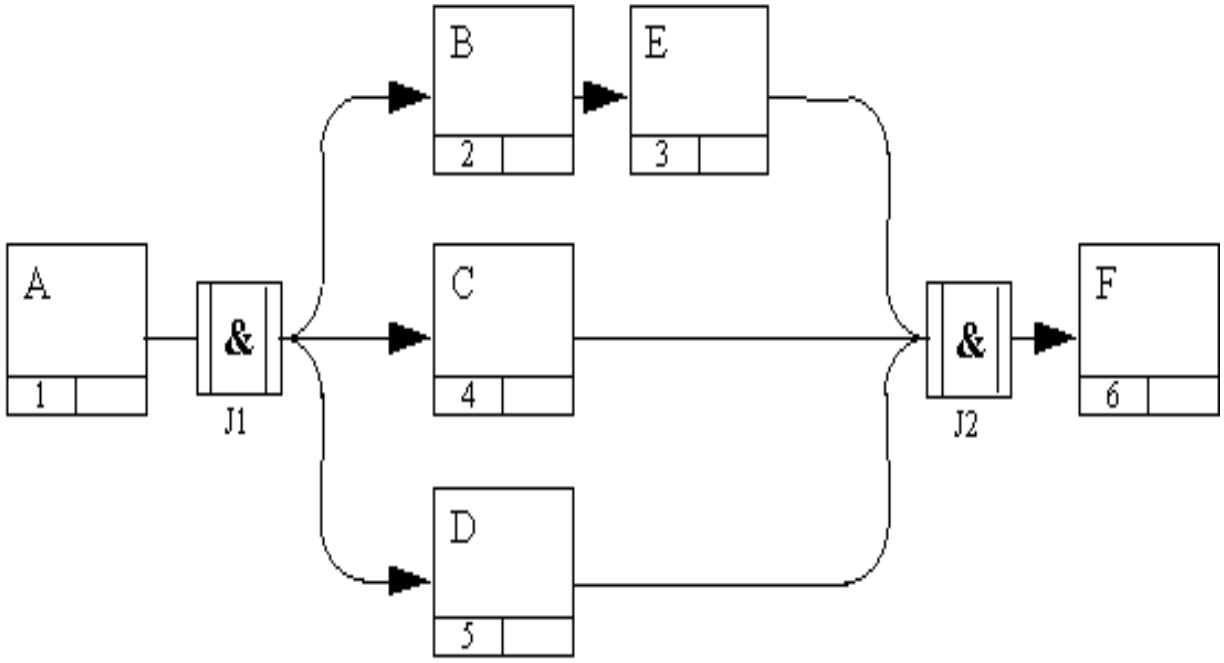


Рис. 4.11. Использование перекрестков синхронный AND

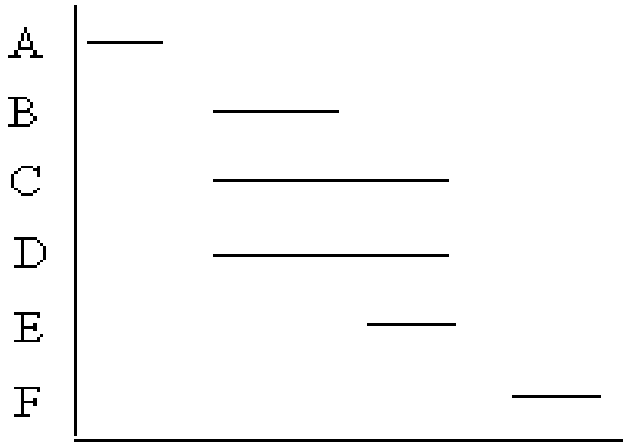


Рис. 4.12. Возможный график запуска для рис. 4.11

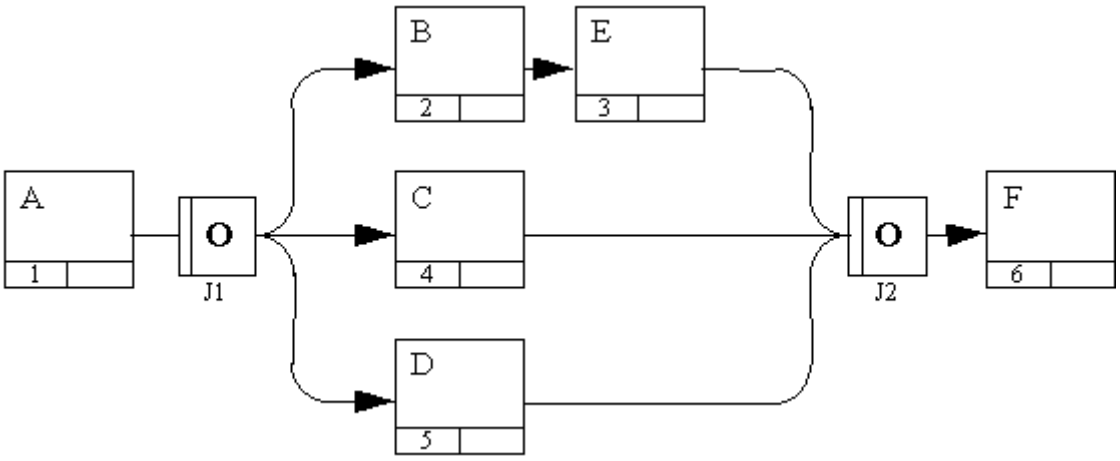


Рис. 4.13. Использование перекрестков асинхронный OR

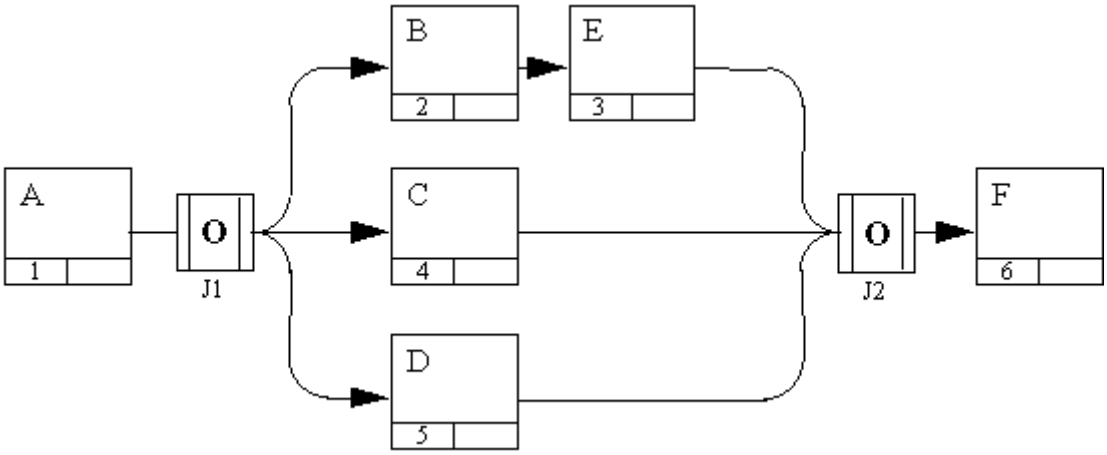


Рис. 4.14. Использование перекрестков синхронный OR

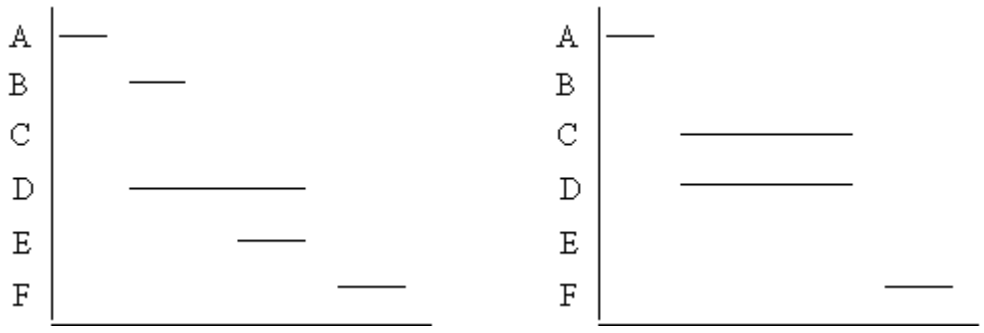


Рис. 4.15. Возможный график запуска для рис. 4.13 и 4.14.

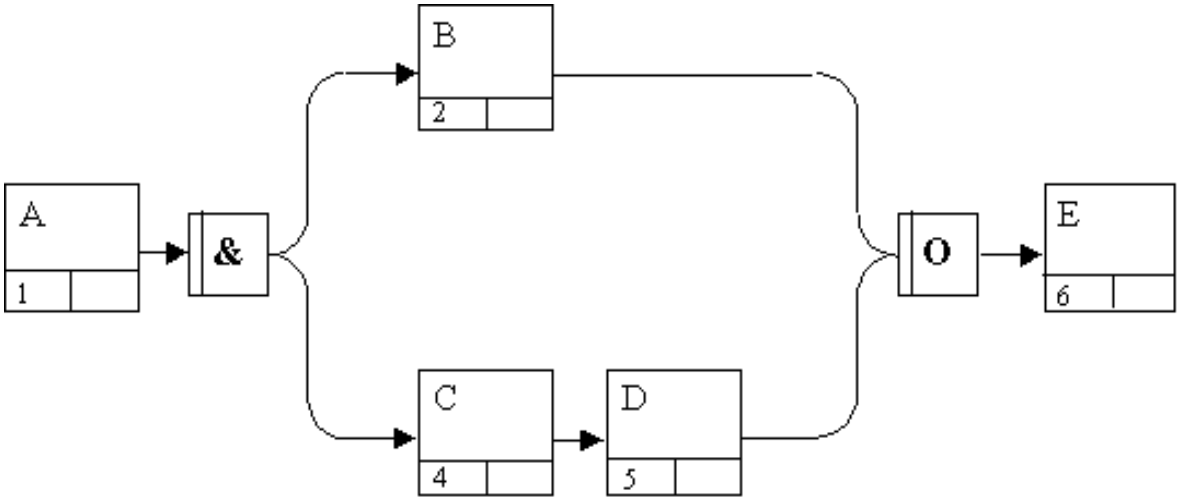


Рис. 4.16. Использование асинхронный AND перекрестка разветвления и асинхронного OR перекрестка слияния

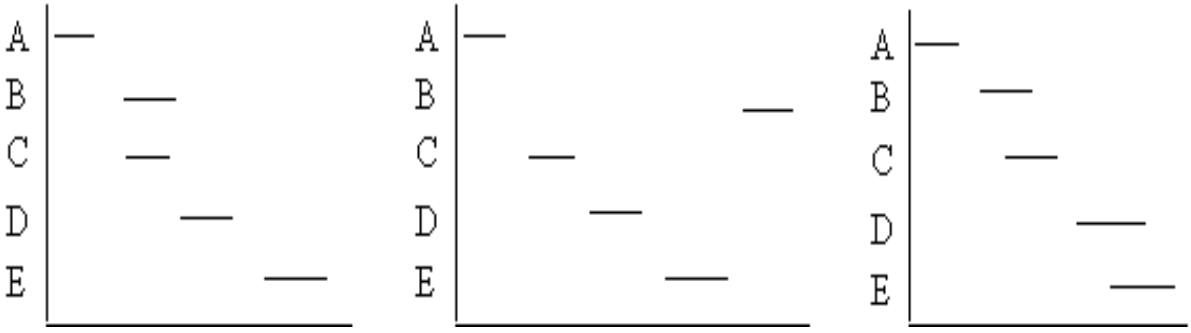


Рис. 4.17. Возможный график запуска для рис. 4.16.

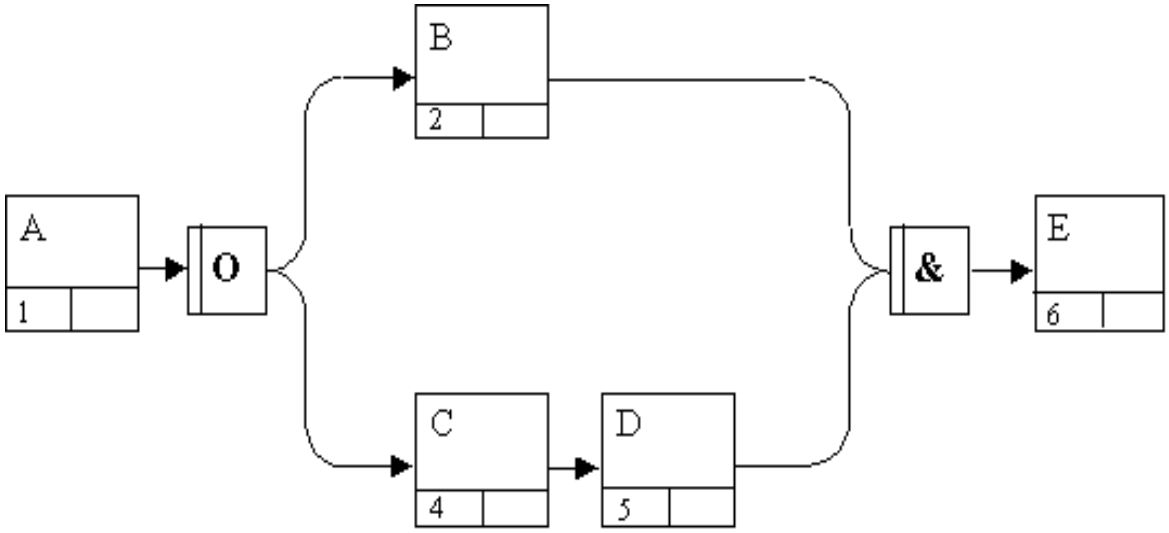


Рис. 4.18. Невозможное совместное использование перекрестков

4.4. Декомпозиция описания процесса

Методология IDEF3 дает возможность представлять процесс в виде иерархически организованной совокупности диаграмм. Диаграммы состоят из нескольких элементов описания процесса IDEF3, причем каждый функциональный элемент UOB потенциально может быть детализирован на другой диаграмме. Такое разделение сложных комплексных процессов на его структурные части называется декомпозицией. Декомпозиция формирует границы для описания процесса и каждый UOB элемент рассматривается как формальная граница некоторой части целой системы, которая описывает весь процесс. Декомпозированная диаграмма, называемая диаграммой потомком, более детально описывает процесс. Декомпозируемый UOB элемент называется родительским, а содержащая его диаграмма – соответственно родительской диаграммой.

Итак, декомпозиция – это процесс создания диаграммы, детализирующей определенный UOB элемент. Результатом ее является описание, которое представляет собой дробление родительского UOB элемента на меньшие и более частные операции или функции. Кроме того, само слово "анализ" означает разложение на составляющие. Но декомпозиция – это больше, чем разложение на части. Она включает также синтез. Подлинная декомпозиция заключается в начальном разделении объекта на более мелкие части и последующем соединении их в более детальное описание.

Применяя принцип декомпозиции неоднократно, возможно структурировать описание процесса до любого уровня подробности. Декомпозиция обеспечивает средства организации более детального описания UOB элементов. Каждый UOB элемент может иметь любое число различных декомпозиций на том же самом уровне детализации с целью представления различных точек зрения или обеспечения большей подробности при описании исходного процесса.

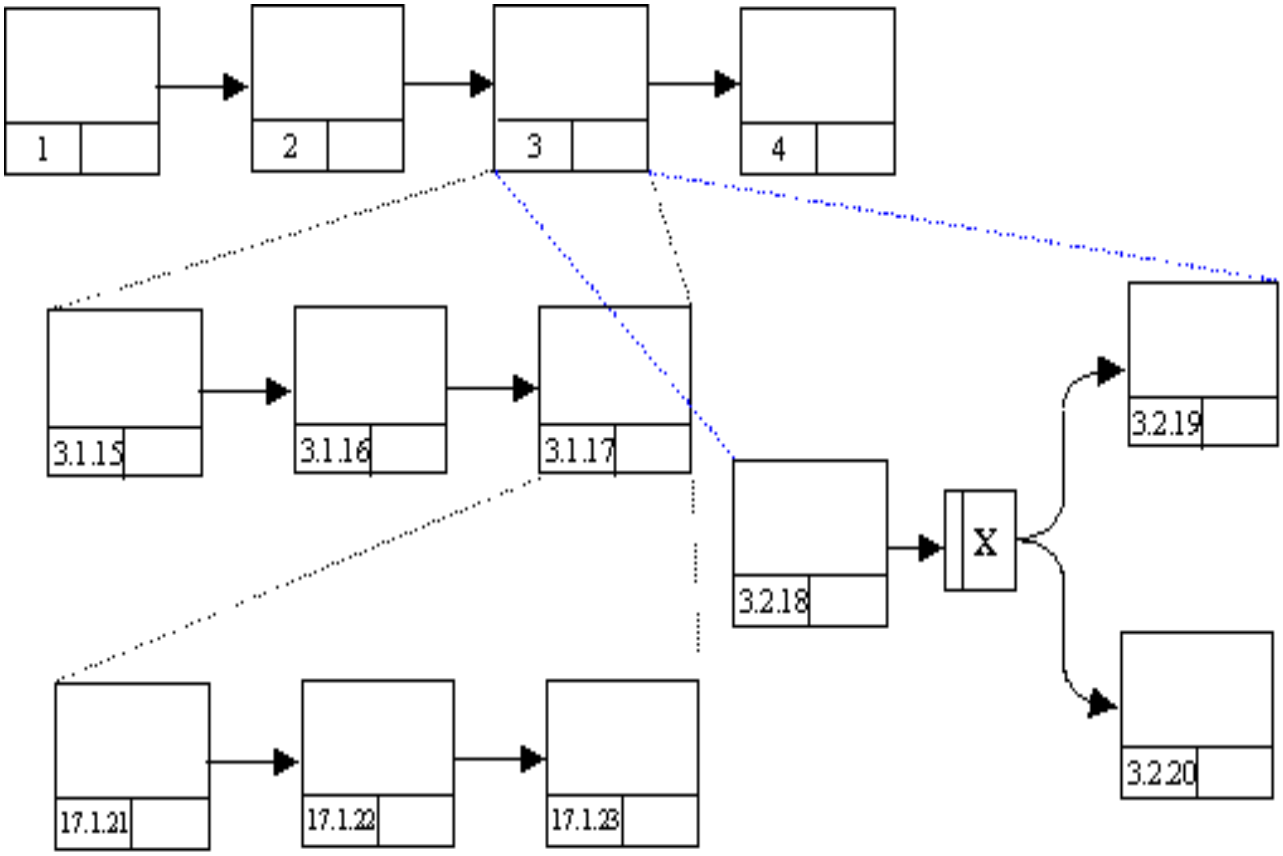


Рис. 4.19. Пример нумерации UOB элементов при использовании декомпозиции и описания различных точек зрения на выполнение процессов

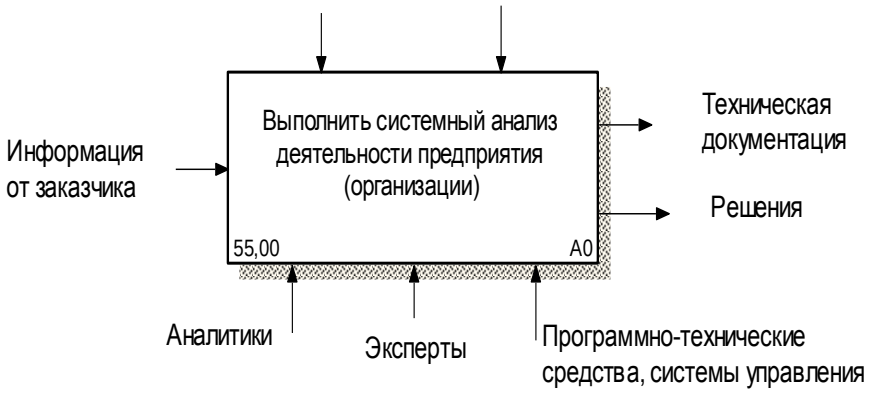
4.5. Примеры

Выполнить системный анализ деятельности предприятия (организации)

USED AT: Оценки затрат	AUTHOR: Чеботарев Валерий	DATE: 13.01.1999	WORKING	READER DATE	CONTEXT:
	PROJECT: Описание процесса выполнения консалтингового договора	REV: 26.01.1999	DRAFT		TOP
			RECOMMENDED		
	NOTES: 1 2 3 4 5 6 7 8 9 10		PUBLICATION		

Эта небольшая функциональная IDEF0/IDEF3 модель разработана в пакете BPwin.
Она представляет собой описание бизнес-процесса выполнения консалтингового договора.

Для оценки трудоемкости выполнения разработки и отдельных компонентов использована методология функционально-стоимостного анализа (ABC).
Информация о типичном времени выполнения (рабочих дней) приведена в левом нижнем углу блока A0 и в левых верхних углах других блоков.

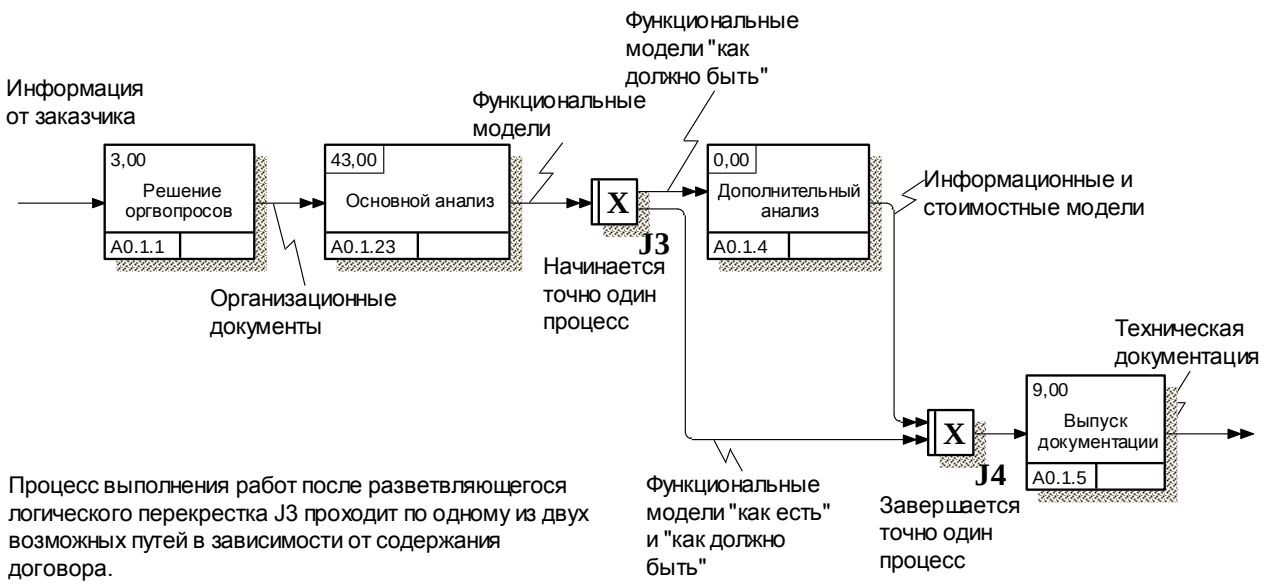


ЦЕЛЬ: Внедрение на предприятии интегрированной системы управления современного уровня
ТОЧКА ЗРЕНИЯ: Аналитической группы компании Interface ltd

Для достижения поставленной цели используется один из двух путей:
- внедрение готовой системы управления предприятием, например, таких, как ERP система Renaissance CS;
- разработка новой системы управления предприятием.

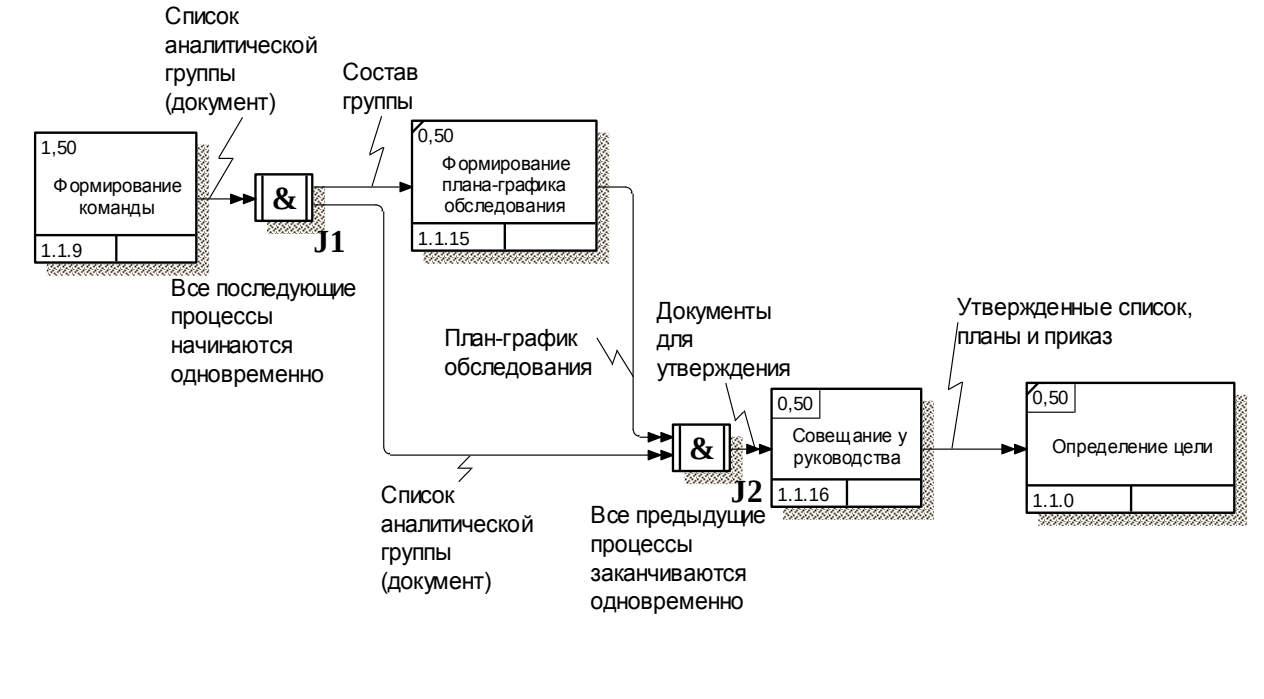
NODE: A-0	TITLE: Выполнить системный анализ деятельности предприятия (организации)	NUMBER: VChe 1

USED AT:	AUTHOR: Чеботарев Валерий PROJECT: Описание процесса выполнения консалтингового договора NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 13.01.1999 REV: 25.01.1999	WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT: A-0



NODE: A0.1	TITLE: Выполнить системный анализ деятельности предприятия (организации)	NUMBER: VChe 2

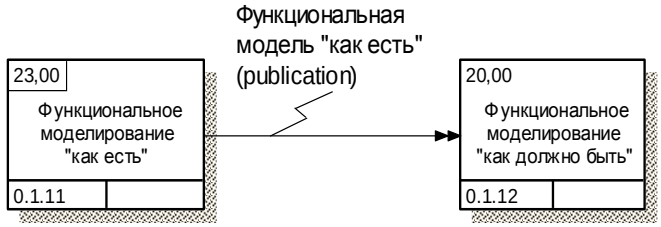
USED AT:	AUTHOR: Чеботарев Валерий PROJECT: Описание процесса выполнения консалтингового договора NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 13.01.1999 REV: 28.01.1999	WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT: A0.1



NODE: 1.1	TITLE: Решение оргвопросов	NUMBER: VChe 3

USED AT:	AUTHOR: Чеботарев Валерий	DATE: 13.01.1999	WORKING	READER DATE	CONTEXT:
	PROJECT: Описание процесса выполнения консалтингового договора	REV: 28.01.1999	DRAFT		
			RECOMMENDED		
	NOTES: 1 2 3 4 5 6 7 8 9 10		PUBLICATION		23.1

Функциональные модели состояний "как есть" и "как должно быть" позволяют выполнить анализ текущей деятельности предприятия, определить какие изменения на предприятии и в системе управления подлежат реализации, какие модули системы управления предприятием и в какой последовательности будут внедряться на предприятии (см., например, описание модулей системы Renaissance CS)

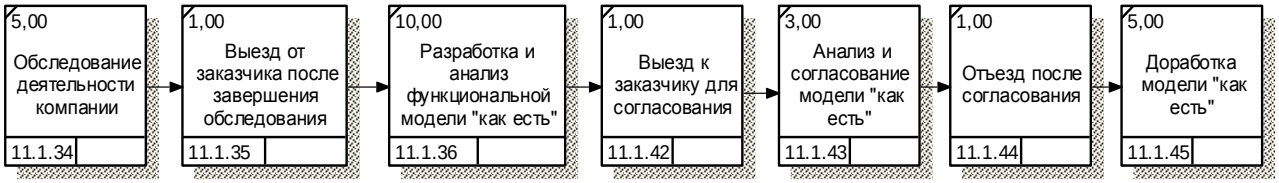


Моделирование выполняется последовательно

NODE: 0.1	TITLE: Функциональное моделирование	NUMBER: VChe 7

USED AT:	AUTHOR: Чеботарев Валерий	DATE: 14.01.1999	■	WORKING	READER DATE	CONTEXT:
	PROJECT: Описание процесса выполнения консалтингового договора	REV: 28.01.1999		DRAFT		
				RECOMMENDED		■ □
	NOTES: 1 2 3 4 5 6 7 8 9 10			PUBLICATION		0.1

Модель состояния "как есть" позволяет выполнить анализ текущей деятельности предприятия, а также определить какие бизнес-процессы на предприятии подлежат изменению

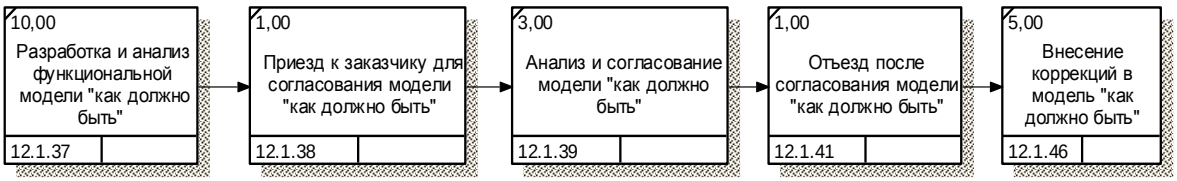


На этой диаграмме приведен процесс разработки функциональной модели состояния "как есть" с выездом к заказчику для сбора информации. В описание процесса включен также повторный выезд к заказчику для выполнения последней итерации по согласованию и доработки модели. Предполагается, что предварительное согласование модели выполняется с использованием средств коммуникации, например, по E-mail.

NODE: 11.1	TITLE: Функциональное моделирование "как есть"	NUMBER: VChe 8

USED AT:	AUTHOR: Чеботарев Валерий	DATE: 14.01.1999	WORKING	READER DATE	CONTEXT:
	PROJECT: Описание процесса выполнения консалтингового договора	REV: 28.01.1999	DRAFT		
			RECOMMENDED		
	NOTES: 1 2 3 4 5 6 7 8 9 10		PUBLICATION		0.1

Функциональная модель состояния "как должно быть" в общем случае является компромиссным вариантом, который реализует, как изменения бизнес-процессов на предприятии, так и изменения в автоматизированной системе управления.

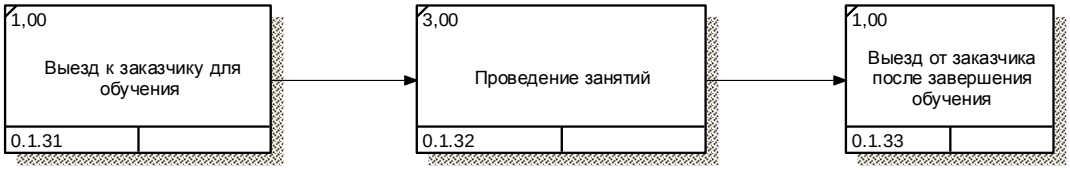


В процесс разработки функциональной модели "как должно быть" включен выезд к заказчику для выполнения последней итерации по согласованию и доработке модели (перевод модели в состояние "publication").
Предполагается, что предварительное согласование модели в процессе разработки выполняется с использованием средств коммуникации, например, по E-mail.

NODE: 12.1	TITLE: Функциональное моделирование "как должно быть"	NUMBER: VChe
		9

USED AT:	AUTHOR: Чеботарев Валерий	DATE: 14.01.1999	WORKING	READER DATE	CONTEXT:
	PROJECT: Описание процесса выполнения консалтингового договора	REV: 25.01.1999	DRAFT		
			RECOMMENDED		
	NOTES: 1 2 3 4 5 6 7 8 9 10		PUBLICATION		23.1

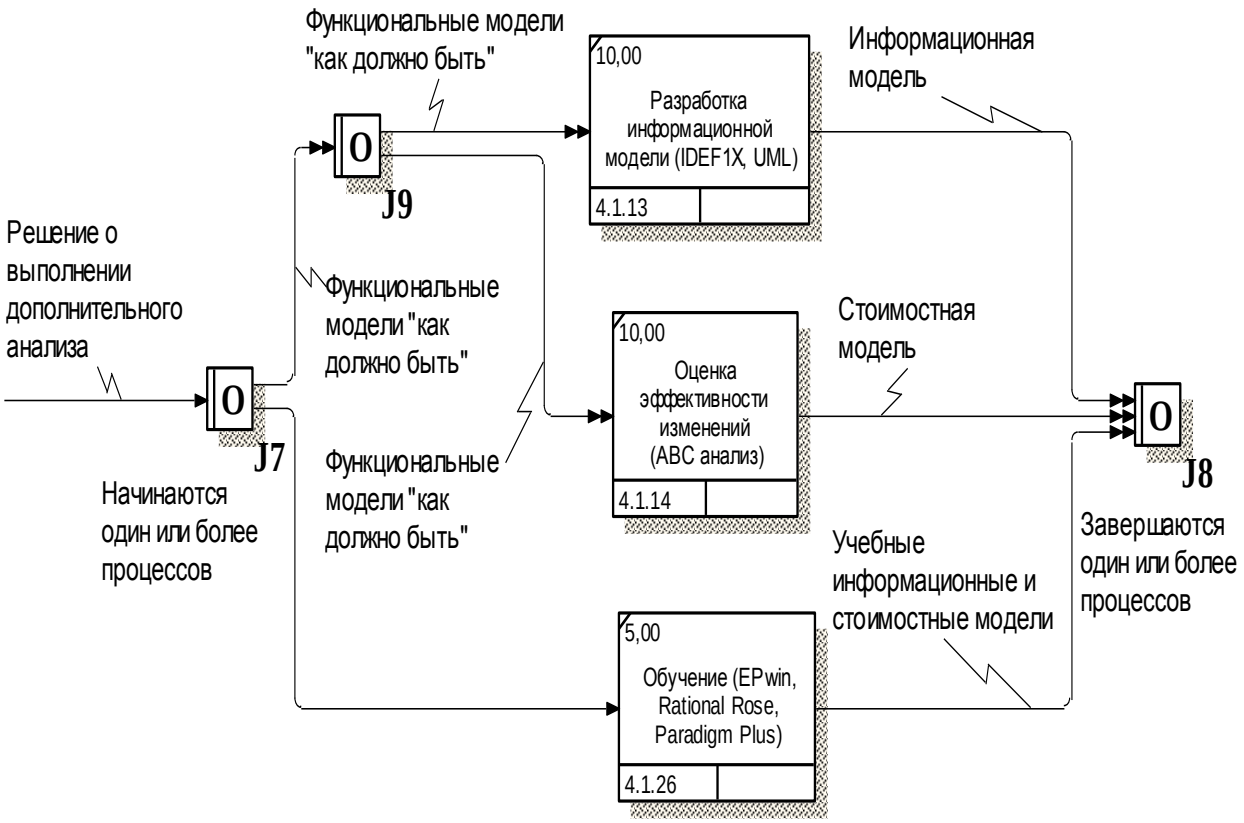
В процессе обучения у заказчика формируется аналитическая группа, которая будет в дальнейшем поддерживать функциональную модель в актуальном состоянии, активно участвовать в процессе внедрения системы управления предприятием, а также моделировать изменения бизнес-процессов при эксплуатации системы.



На этой диаграмме представлен процесс обучения с выездом к заказчику.

NODE: 0.1	TITLE: Обучение BPwin	NUMBER: VChe
		10

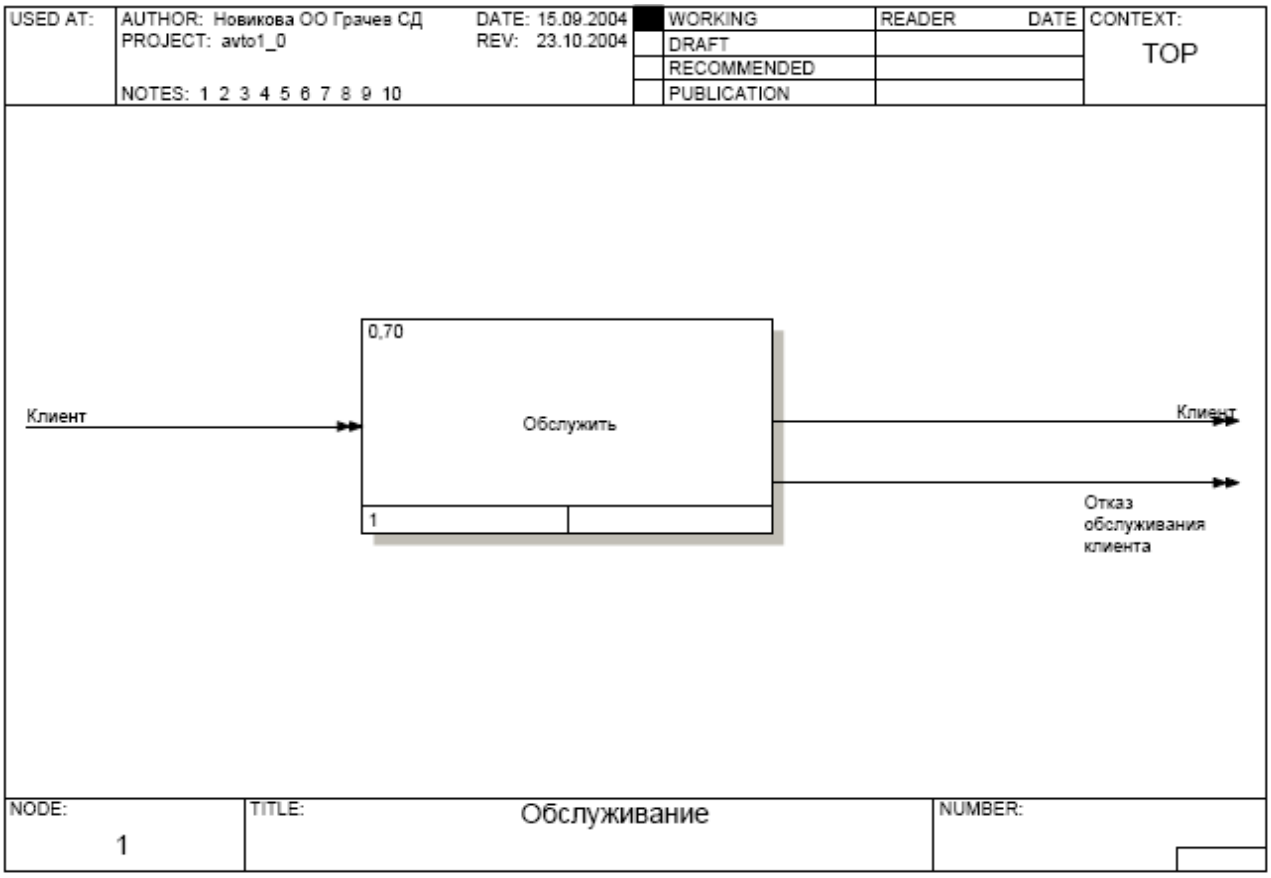
USED AT:	AUTHOR: Чеботарев Валерий	DATE: 13.01.1999	WORKING	READER DATE	CONTEXT:
	PROJECT: Описание процесса выполнения консалтингового договора	REV: 28.01.1999	DRAFT		
			RECOMMENDED		
	NOTES: 1 2 3 4 5 6 7 8 9 10		PUBLICATION		A0.1



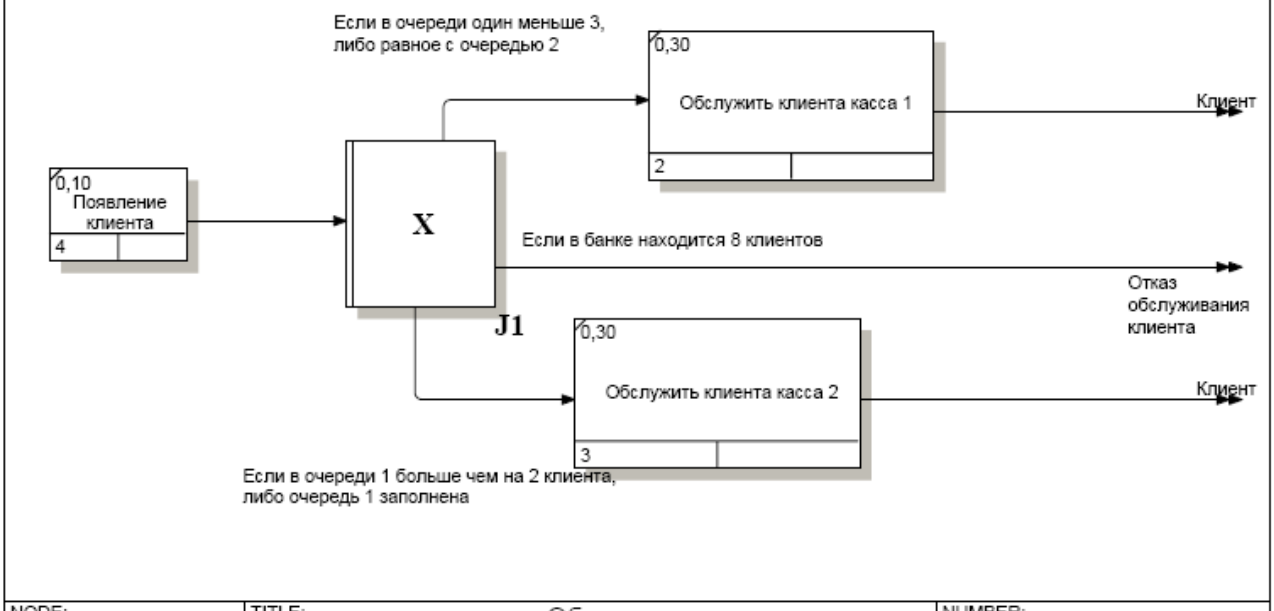
На этой диаграмме приведен процесс разработки дополнительных моделей и обучения без выезда к заказчику. Предполагается, что необходимая для моделирования информация получена на более ранних этапах разработки.

NODE: 4.1	TITLE: Дополнительный анализ	NUMBER: VChe 11

Деятельность банка



USED AT:	AUTHOR: Новикова ОО Грачев СД	DATE: 15.09.2004		WORKING	READER DATE	CONTEXT:
	PROJECT: avto1_0	REV: 23.10.2004		DRAFT		
				RECOMMENDED		
	NOTES: 1 2 3 4 5 6 7 8 9 10			PUBLICATION		1.1



NODE:	TITLE:	NUMBER:
1.1	Обслужить	

5. Язык моделирования баз данных IDEF1x

Методология IDEF1X – один из подходов к семантическому моделированию баз данных, основанный на концепции "сущность–связь" (Entity–Relationship). Это инструмент для анализа информационной структуры систем различной природы. Информационная модель, построенная с помощью IDEF1X–методологии, отображает логическую структуру информации об объектах системы.

Методология IDEF1X является стандартом разработки реляционных баз данных и использует условный синтаксис, специально разработанный для удобного построения концептуальной схемы.

Концептуальная схема – универсальное представление структуры данных, независимое от конечной реализации базы данных и аппаратной платформы.

Понятие концептуальной модели данных связано с методологией семантического моделирования данных, т.е. с представлением данных в контексте их взаимосвязей с другими данными. Концептуальная модель, представленная в соответствии со стандартом IDEF1X, является логической схемой базы данных для проектируемой системы.

Основным преимуществом IDEF1X, по сравнению с другими многочисленными методами разработки реляционных баз данных, является жесткая и строгая стандартизация моделирования. Установленные стандарты позволяют избежать различной трактовки построенной модели, что позволяет избежать многих ошибок при проектировании БД и хранилищ, а также организовать взаимодействие разработчиков.

5.1. Сущности

При моделировании на языке IDEF1x на логическом уровне, объекты реального мира представляются сущностями (одной или более).

Сущность отражает набор экземпляров объекта (объектов) реального мира, представляет совокупность или набор экземпляров похожих по свойствам, но однозначно отличаемых друг от друга. Каждая сущность имеет имя и некоторый набор атрибутов (моделируемых характеристик реального мира). Кроме имени и атрибутов, каждая сущность должна иметь текстовое определение (definition), что собой представляет данная сущность.

В дальнейшем данное определение используется как документация к проекту и как комментарий к таблице физически представляющей данную сущность в конкретной СУБД.

Ниже в таблицах приведены определения сущности “Страны”, их атрибутов и индексов для работы с данными:

Столбцы таблицы(s) of "COUNTRYS" Table – Страны

Name	Definition/Comment
COUNTRY_NO	Код страны
COU_DATE_INS	Дата вставки записи в таблицу COUNTRYS
REGION	Код региона
REG_DATE_INS	Дата вставки записи в таблицу REGION
COUNTRY_ID	Мнемокод страны
COUNTRY_NAME	Наименование страны
COUNTRY_FULLNAME	Полное наименование страны

Индексы(s) of "COUNTRYS" Table

Name	Definition/Comment	Type
P_K_COU_	Код страны	PK
H_COU_NO	Индекс историчности объекта	AK1
I_COU_NO	Индекс поколения объекта	IE1
F_REGION	Ссылка на "Регионы"	FK

Первичный ключ(s) of "COUNTRYS" Table

Name	Datatype	Definition/Comment
COUNTRY_NO	CHAR(3)	Код страны
COU_DATE_INS	TIMESTAMP	Дата вставки записи в таблицу COUNTRYS

Конкретное значение объекта представляется конкретным значением набора каждого из атрибутов сущности (или сущностей) и называется экземпляром сущности (instance), другими словами значением сущности. Каждый экземпляр сущности однозначно идентифицируется с помощью значений одного или более атрибутов. Данные атрибуты образуют первичный ключ. При реализации логической модели средствами конкретной СУБД каждая сущность на физическом уровне, как правило, отображается в таблицу, а набор атрибутов сущности в колонки таблицы с указанием типа данных каждой колонки. Конкретный экземпляр значений всех колонок представляется записью в данной таблице, т.е. каждая запись в таблице (или таблицах) отражает некоторое значение объекта реального мира. Первичный ключ используется для поиска конкретной записи в таблице. На примере ниже приведена таблица Countries, которая отражает сущность «Страна» и атрибуты – колонки, которые отражают характеристики каждой реальной страны (Country_No – Код страны, Country_Name – Наименование страны и т.д.), а первичным ключом является колонка

COUNTRY_NO – атрибут «Код страны» и скрытый атрибут «дата вставки записи в таблицу» (COU_DATE_INS).

Таблица. "COUNTRYS" – Страны

Код страны COUNTRY_NO	Код региона REGION	Мнемокод страны COUNTRY_ID	Наименование страны COUNTRY_NAME	Полное наименование страны COUNTRY_FULLNAME
000	N	NOT	НЕОПРЕДЕЛЕН–НОЕ ЗНАЧЕНИЕ	НЕОПРЕДЕЛЕННОЕ ЗНАЧЕНИЕ
004	O	AF	АФГАНИСТАН	РЕСПУБЛИКА АФГАНИСТАН
008	O	AL	АЛБАНИЯ	НАРОДНАЯ СОЦИАЛИСТИЧЕСКАЯ РЕСПУБЛИКА
010	O	AQ	АНТАРКТИКА	АНТАРКТИКА
012	O	DZ	АЛЖИР	АЛЖИРСКАЯ НАРОДНАЯ ДЕМОКРАТИЧЕСКАЯ РЕСПУБЛИКА
016	O	AS	ВОСТОЧ.САМОА (США)	АМЕРИКАНСКОЕ САМОА
020	O	AD	АНДОРРА	КНЯЖЕСТВО АНДОРРА
024	O	AO	АНГОЛА	НАРОДНАЯ РЕСПУБЛИКА АНГОЛА (НРА)
028	O	AG	АНТИГУА И БАРБУДА	АНТИГУА И БАРБУДА
031	S	AZ	АЗЕРБАЙДЖАН	АЗЕРБАЙДЖАН
032	O	AR	АРГЕНТИНА	АРГЕНТИНСКАЯ РЕСПУБЛИКА
036	O	AU	АВСТРАЛИЯ	АВСТРАЛИЯ

5.2. Связи и отношения

В процессе моделирования сущности связываются отношениями, например, отдел А «состоит из» из сотрудников В, сотрудник из В «пишет» программу С и Д.

В данном случаи, связь является логическим отношением между сущностями. При таком связывании сущность А является родительской для сущности В, а сущность В является потомком (дочерней) для сущности А.

Сами сущности могут быть «зависимые» и «независимые», а связи «идентифицирующими» – между «независимой» и «зависимой» сущностями или «не идентифицирующими», дочерняя сущность остается независимой. В случае если потомок связан «идентифицирующей» связью, дочерняя сущность «зависит» от родительской, то каждое значение родительской сущности связано с одним или более значениями дочерней сущности, и отображает связи, которые не могут быть без родительской сущности. В этом случае, любое значение дочерней сущности идентифицируется с помощью некоторого значения родительской сущности и атрибуты первичного ключа родительской сущности мигрируют в атрибуты первичного ключа дочерней сущности.

Если же связи между двумя сущностями не могут быть идентифицированы, но существуют, сущности «независимые», то любое значение дочерней сущности однозначно определяется без значения родительской сущности, и атрибуты первичного ключа родительской сущности мигрируют в состав не ключевых атрибутов дочерней сущности. В языке IDEF1x, «независимые» сущности изображаются прямоугольником с острыми углами, а «зависимые» прямоугольником с овальными углами, сами связи изображаются прямой линией с точкой, смотрите рис. 5.1. (прерывистой – «не идентифицирующие» или сплошной – «идентифицирующие»).

5.2.1. Мощность связей

Спецификация мощности связей (отношений) служит для обозначения количества экземпляров (значений) дочерних сущностей связанных с соответствующим значением родительской сущности. Здесь приняты следующие обозначения: Ра – родительская сущность, Ch – дочерняя:

Ра ●——— Ch 0, 1 или много

Ра ●——— Ch 1 или много
P

Ра ●——— Ch 0 или 1
z

Ра ●——— Ch точно N (7)
7

Ра ●——— Ch от n до m
N – m

Ра ●——— ● Ch многие – ко многим, отношения не установлены, должны быть разрешены позже.

На рис. 5.2. приведен диалог установления мощности отношений.

Рис. 5.1. Пример идентифицирующей связи

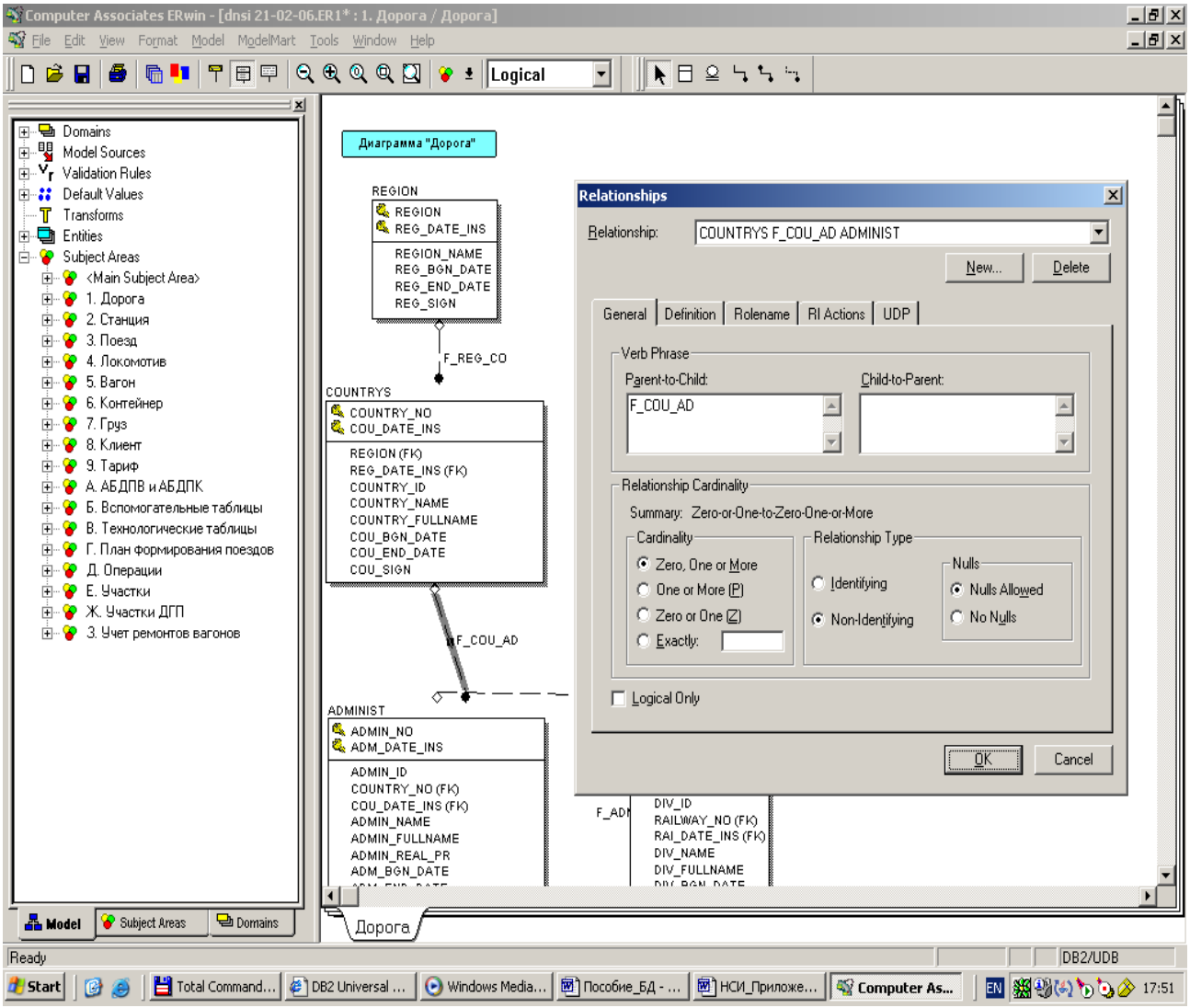


Рис. 5.2. Мощности отношений

5.3. Ключи

Как уже отмечалось выше, каждый экземпляр сущности должен быть уникален, и каждое значение атрибутов должно отличаться одно от другого. Для идентификации каждого экземпляра сущности используется первичный ключ – PK (Primary Key).

Рассмотрение атрибутов – кандидатов для формирования первичного ключа и его выбор (а он может быть только один для таблицы) требуют знания данных предметной области. Существуют строгие правила, использующиеся при выборе первичного ключа. Атрибут (атрибуты) первичного ключа должны:

уникально идентифицировать значение сущности;

не должны включать значение NULL;

не должны изменяться через промежуток времени. Всякий элемент сущности берет свою идентичность от ключа. Если ключ поменяется, то это уже другой элемент;

должен быть как можно более коротким, что облегчать индексацию и восстановление данных. Если нужно использовать ключ, который является комбинацией ключей из других сущностей, то каждая часть ключа должна придерживаться этих правил.

Связи между сущностями реализуются с помощью внешних ключей – FK (Foreign Key), которые образуются из PK путем их миграции из родительской сущности в дочернюю сущность. Сущности и их связи, ключи PK и FK – это основные элементы модели данных. Кроме ключей PK и FK – при построении модели данных, используются альтернативные и инверсные ключи – AK (Alternate Key) и IK (Inversion Key). Альтернативные ключи, так же как и PK, обеспечивают уникальный доступ к значениям сущности, но по другим наборам атрибутов, а инверсные ключи, обеспечивают доступ к некоторому набору значений сущности, имеющего общие характеристики.

5.3.1 Внутренние и внешние ключи

Одно из важнейших понятий, используемых при работе с базами данных, – это первичные ключи, с их помощью идентифицируются значения сущности (записи в таблице). В качестве первичного ключа может использоваться суррогатный или естественный ключ.

Естественный ключ – это атрибут или набор атрибутов, которые однозначно идентифицируют значение сущности (записи таблицы) и имеют некий физический смысл вне БД (номер вагона, номер станции, номер контейнера и т.д.).

Данный метод построения модели БД основан на естественной идентификации сущностей. Уникальность записи может достигаться не только значением уникального кода внешнего объекта, но, например, и датой вставки записи в таблицу (первичный ключ PK – состоит из значения кода объекта и даты вставки записи в БД), см. примеры диаграмм приведенные выше. Конечно естественная идентификация значений сущностей более понятная и предпочтительная, т.к. нет необходимости вводить дополнительные атрибуты не несущие ни какой внешней смысловой нагрузки. Да, и при эксплуатации БД в сложных ситуациях можно восстановить их значения. Но, не всегда возможно выбрать атрибут или короткий набор атрибутов для идентификации значений. В этом случаи без использования внутренних, суррогатных ключей трудно обойтись.

Суррогатный ключ – автоматически сгенерированное значение, никак не связанное с информационным содержанием записи, имеющий некий физический смысл только внутри БД; обычно в роли суррогатного ключа могут использоваться данные типа integer или datestamp.

В данном методе построения модели БД суррогатная идентификация значений сущностей позволяет решить проблему идентификации за счет введения

внутреннего ID – сущности и выработки внутреннего суррогатного ключа (PK) для каждого значения в БД. Такой подход предполагает, что для каждой сущности существует набор определенных характеристик, которые в процессе жизни могут меняться или уточняться. Поэтому в БД каждой сущности на предприятии присваивается своя внутренняя идентификация, уникальная на всей протяжении жизни сущности и все характеристики (текущие или измененные) уникально идентифицируются для каждой сущности (создается уникальный внутренний суррогатный ID – сущности). Данный уникальный идентификатор каждого значения в БД является первичным ключом записи (PK) в сущностях (таблицах), содержащих значения характеристик сущностей. На примере рис. 5.3. приведена таблица для выработки и хранения суррогатного ID сущности, на основе которого создаются суррогатные PK ключи значений сущностей (таблиц).

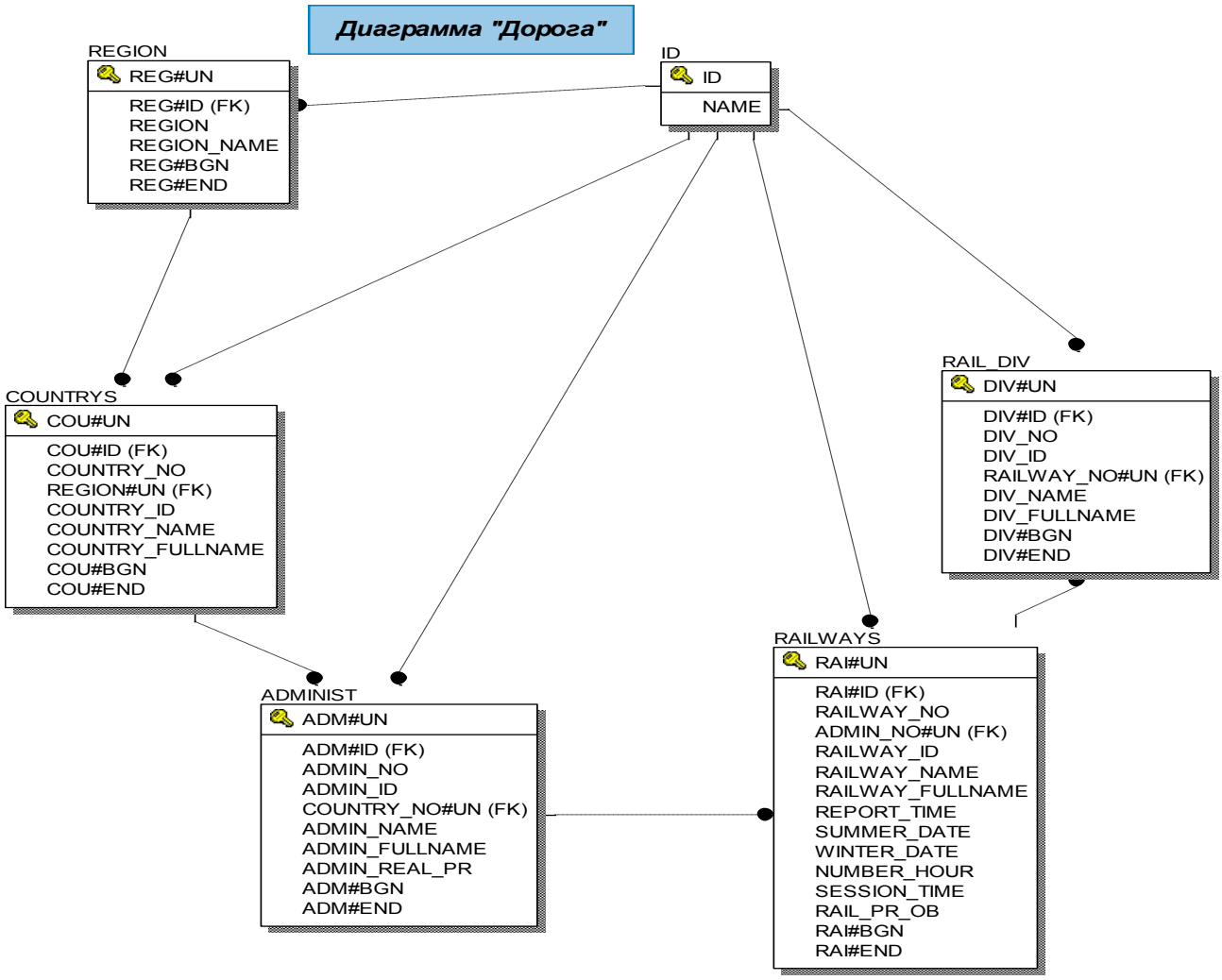


Рис. 5.3. Пример суррогатного ID сущности

5.3.2. Ссылочная целостность

Одной из главных задач при эксплуатации БД, является задача поддержания правил ссылочной целостности, которые были определены при проектировании. Эти правила отражают заданные связи между сущностями и порядок выполнения операций вставка, замена и удаление. Если эти правила не будут соблюдены, то БД быстро деградирует, и не будет отражать те бизнес – правила, которые были заложены при ее проектировании.

Правила контроля и выполнения этих операций может быть возложена на специально разработанные пользователем программы, но данное решение крайне не надежное и трудоемкое для его реализации. Задача поддержания правил ссылочной целостности обычно реализуется автоматически при генерации схемы БД на основе режимов реализации связей (отношений) с помощью триггеров, обеспечивающих ссылочную целостность. Данные триггера представляют собой программы, выполняемые всякий раз при выполнении операций вставка, замена и удаление. Таким образом, при проектировании автоматически обеспечивается правильная эксплуатация БД.

На рис. 5.4. приведен пример определения правил ссылочной целостности для реализации связи F_COU_AD в среде Erwin.

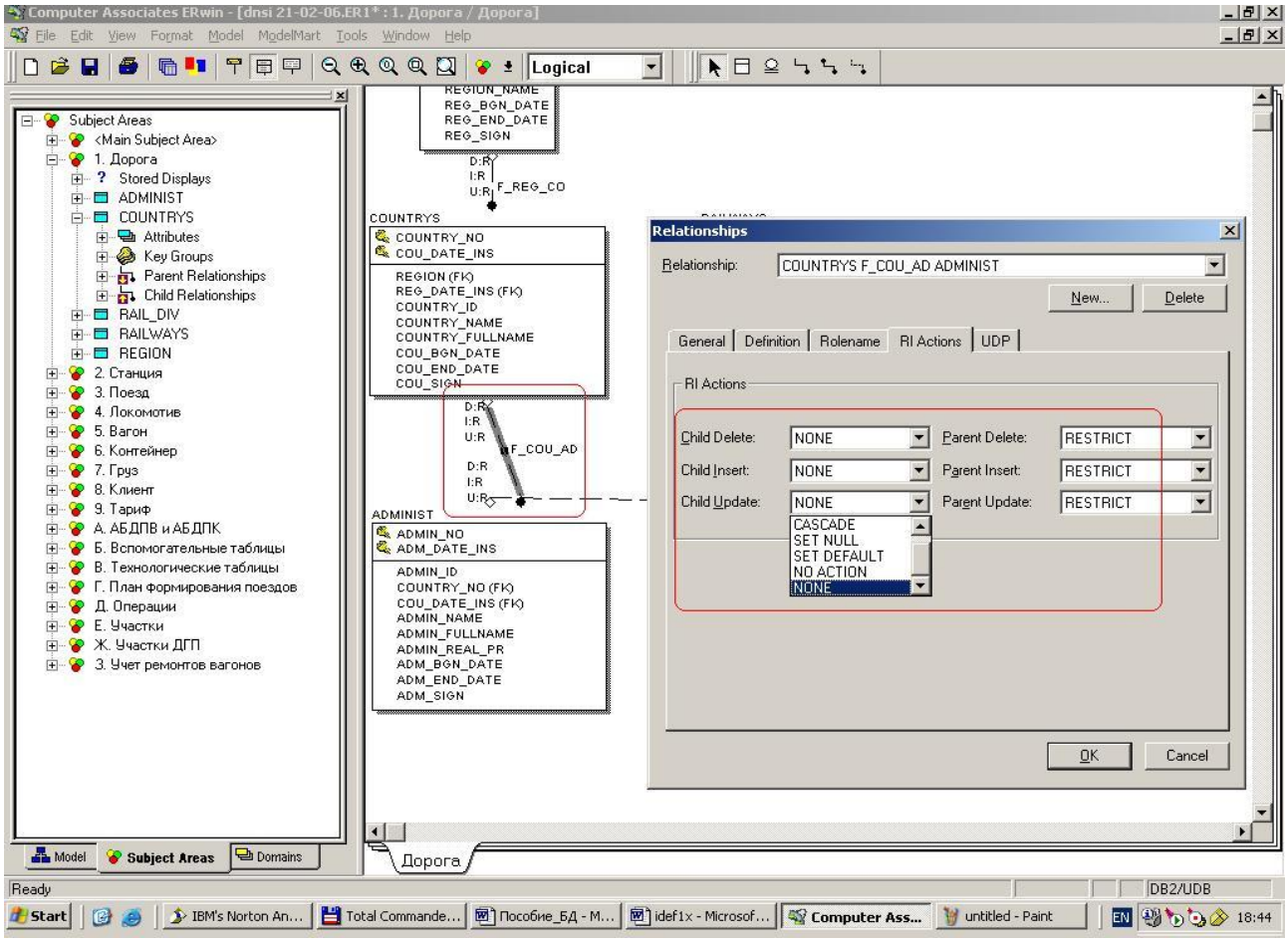


Рис. 5.4. Пример определения правил ссылочной целостности

Cascade – задает каскадное действие (например, удаление родительской записи влечет удаление всех записей у потомков), Restrict – задает ограничение на выполнение действия, (на пример, нельзя удалить родительскую запись, пока есть записи у потомков и наоборот), Set Null – задает действие по установлению FK равным NULL, при удалении родительской записи, Set Default – задает действие по установлению FK равным Default, при удалении родительской записи, None Action – ни каких действий не требуется, если удалены потомки (например, D:R – для родителя, D:S – для потомка или I:R – для потомка, если нет родителя).

5.4. Домены

Стандарт IDEF1x определяет домен как – поименованный и определенный набор значений, такой что один или более атрибутов получают значения из этого домена.

Домен, как и класс, который фиксирует и определяет допустимое множество значений. Например, домен «код–страны» (см. рис. 5.5.) – определяет код страны, который может состоять из двух специальных букв. Домен – это неизменный класс значения, которого не изменяются во времени. В противоположность этому значения сущности меняются во времени. Каждое значение домена уникальное в этом домене.

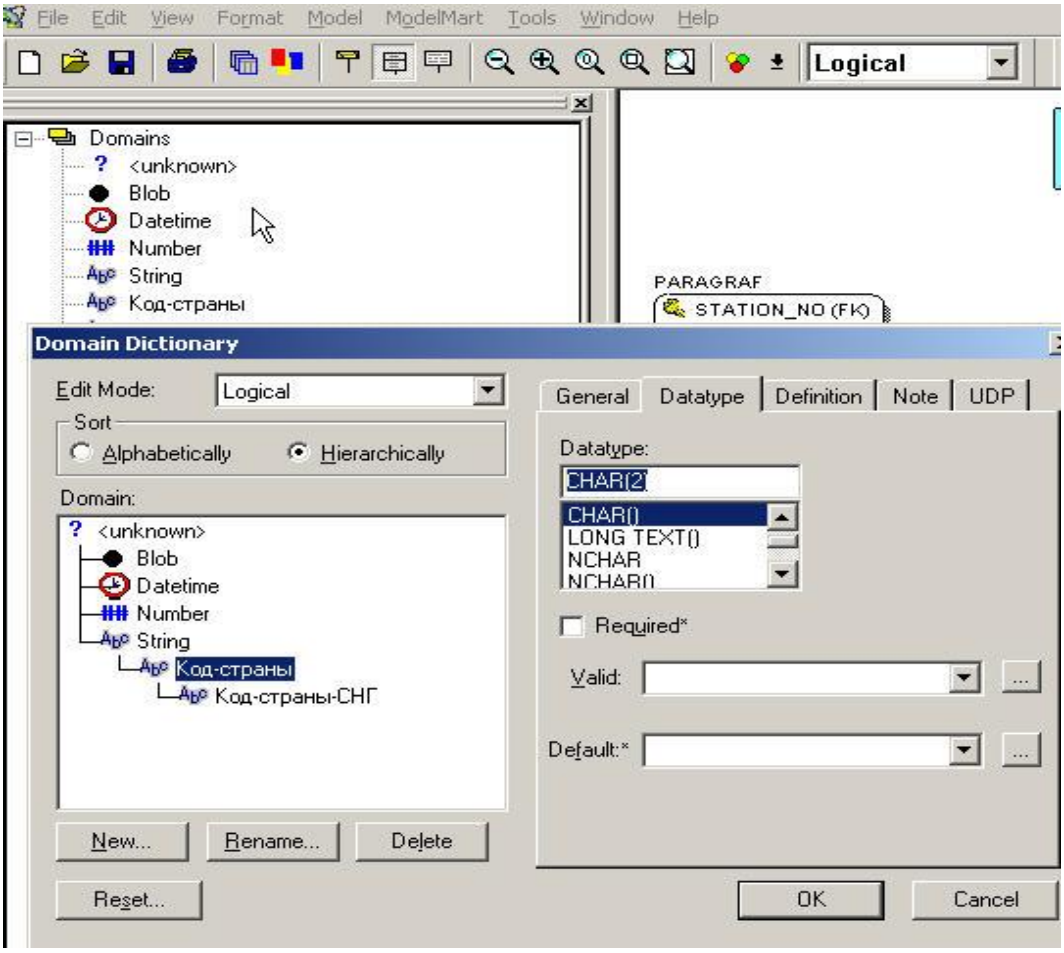


Рис. 5.5. Пример домена

Существуют два типа доменов: базовые домены (Character, Numeric, Boolean в IDEF1x, а в Erwin: String, Number, Datetime, Blob) и типизированные домены (производные), например, «код–страны–СНГ» (см. рис. 5.5.1). Все базовые домены имеют правила использования значений данных из домена. Наиболее используемые это два правила: список значений и диапазоны. Для каждого домена могут быть установлены свои правила определения значений.

Типизированные домены – это подтипы базовых типов или других типизированных доменов.

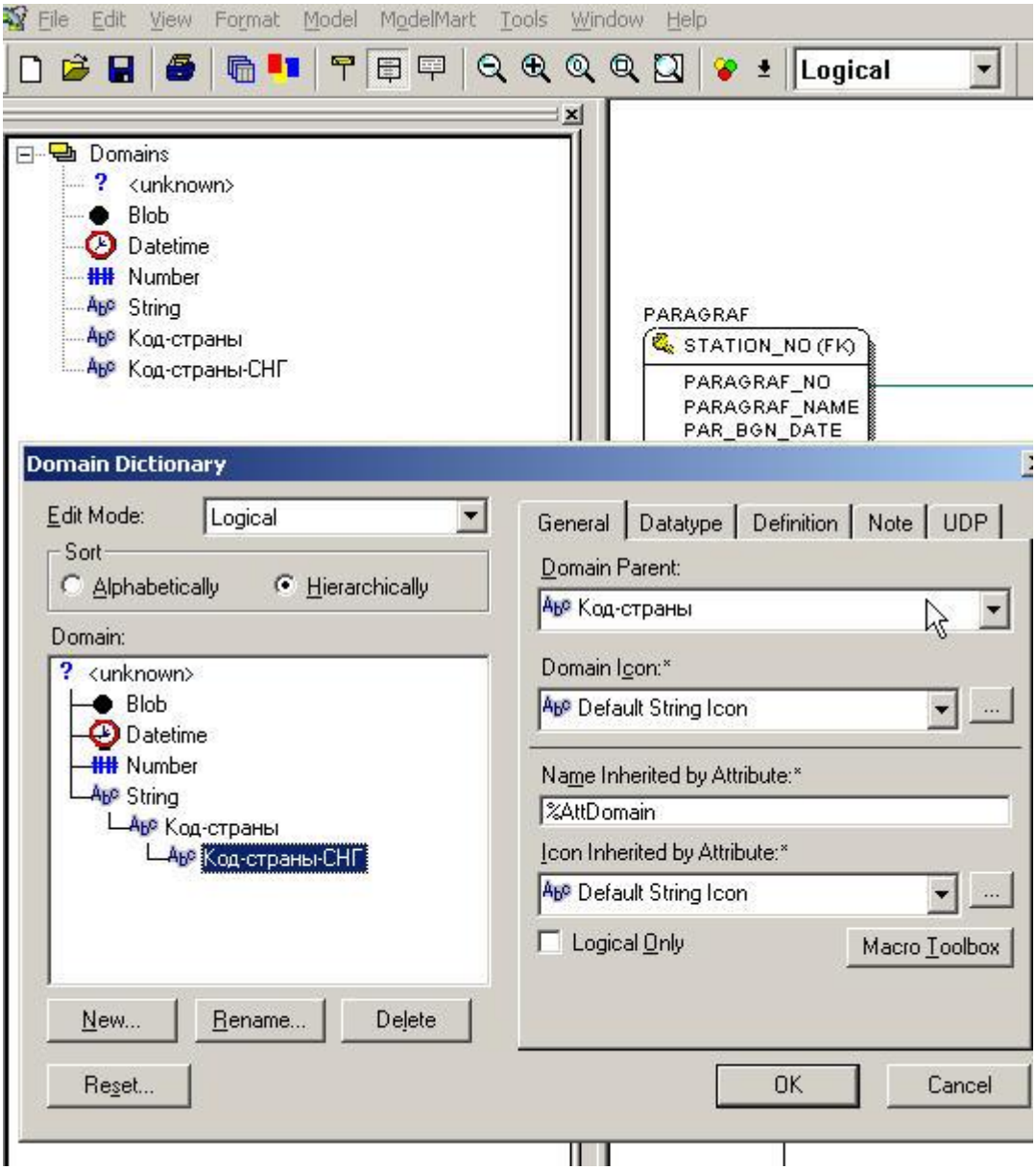


Рис. 5.5.1.

На логическом уровне домены можно описать без конкретных физических свойств, а уже на физическом уровне они получают конкретные специфические свойства. Например, домен «Код–страны–СНГ» (см. рис. 5.5.2.) на логическом уровне может иметь тип String, а на физическом уровне (DB2) будет присвоен тип Char(2) и указано конкретное множество значений домена и правила валидации. Для другой БД физический уровень значений может быть задан свой.

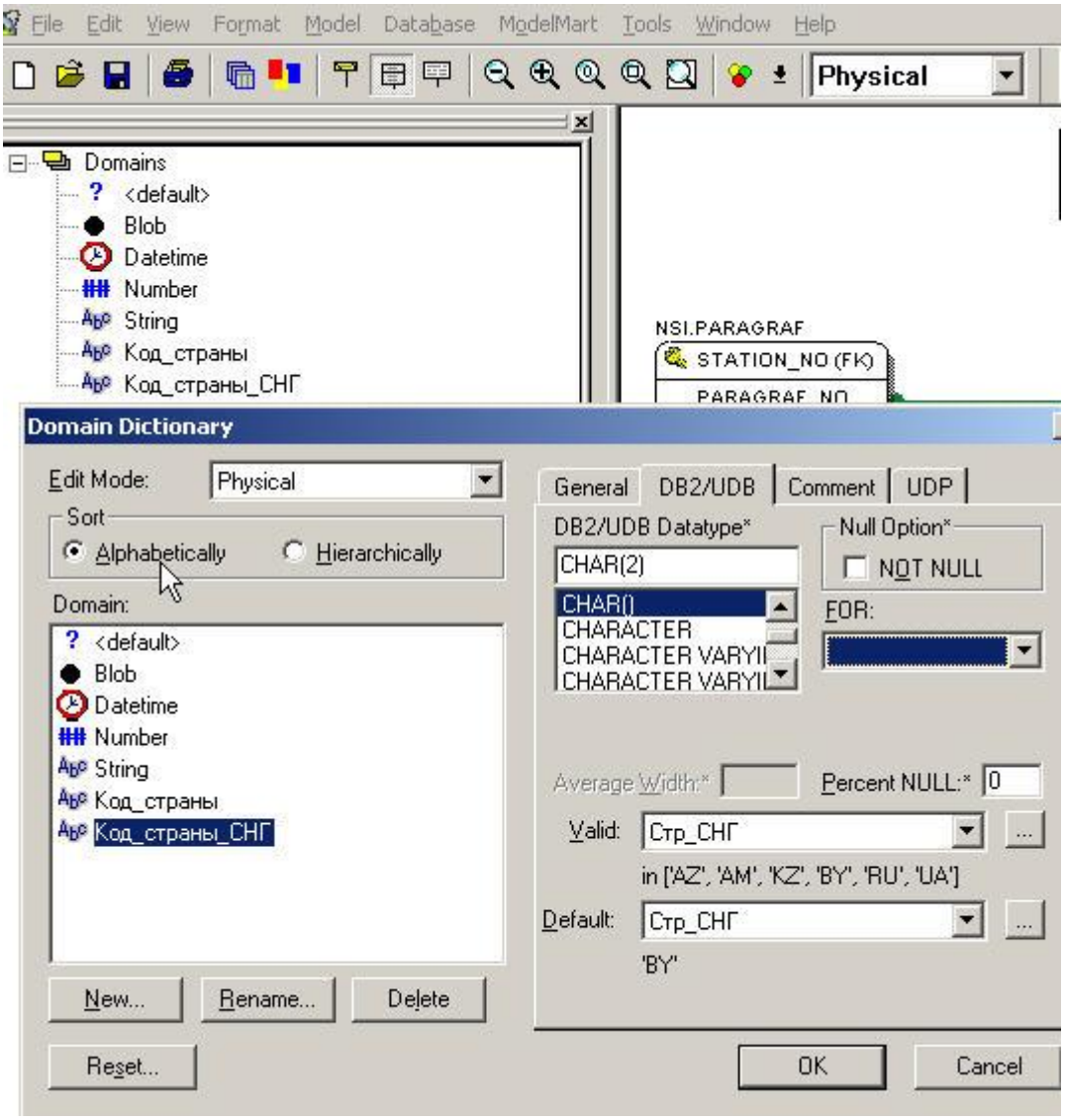


Рис. 5.5.2.

Теперь при определении атрибута «COUNTRY_ID» сущности «COUNTRYS» можно указать его значение из домена «Код_страны_СНГ» (рис. 5.5.3.).

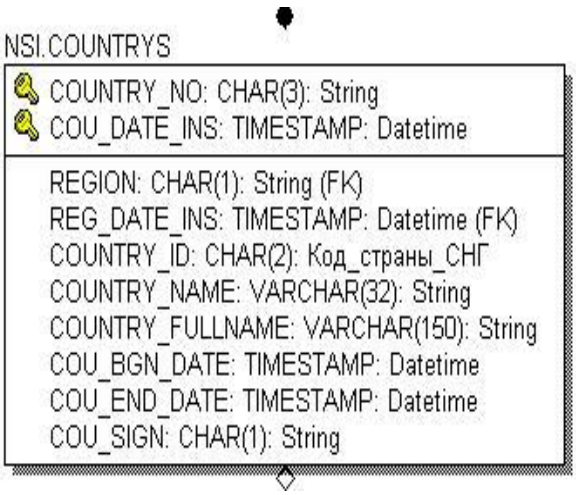


Рис. 2.22

5.5. Представления

Представления (View) представляют собой объекты СУБД данные, в которых формируются динамически при обращении к представлению, т.е. данные не хранятся постоянно как в таблицах БД. Представление определяется в терминах тех таблиц БД и их атрибутов, которые уже есть БД. Для создания представления может использоваться одна или более реальных таблиц БД.

Использование представлений (View) крайне удобный механизм:

– для обеспечения безопасности и секретности доступа к данным в БД, для каждого пользователя (разработчика), может быть разработано и предоставлено свое представление (видение) данных;

– для разработки интерфейса взаимодействия между подсистемами или системами;

– для разработки пользовательского интерфейса WEB – сайта;

– для замены сложной системы репликацией данных в другие системы, в которых модели данных существенно отличаются от модели данных исходной системы.

Разработка представлений разрабатывается на физическом уровне и AllFusion Erwin Data Modeler содержит специальные средства для создания таблиц представления и операторов SQL для заполнения их данными.

Так на рис. 5.6. приведена диаграмма представления для таблиц БД. Эти таблицы содержат внутренние суррогатные PK ключи идентификации данных, которые для пользователя БД ни о чем не говорят. Пользователю БД необходимы реальные общепринятые данные: код дороги, код отделения дороги, код станции на дороге – поэтому для отображения реальных данных используются представления. Таблицы представлений (прямоугольники с овальными углами) связаны отношениями с таблицами БД (прямоугольники) и содержат те атрибуты, которые будут заполнены из соответствующих атрибутов таблиц БД.

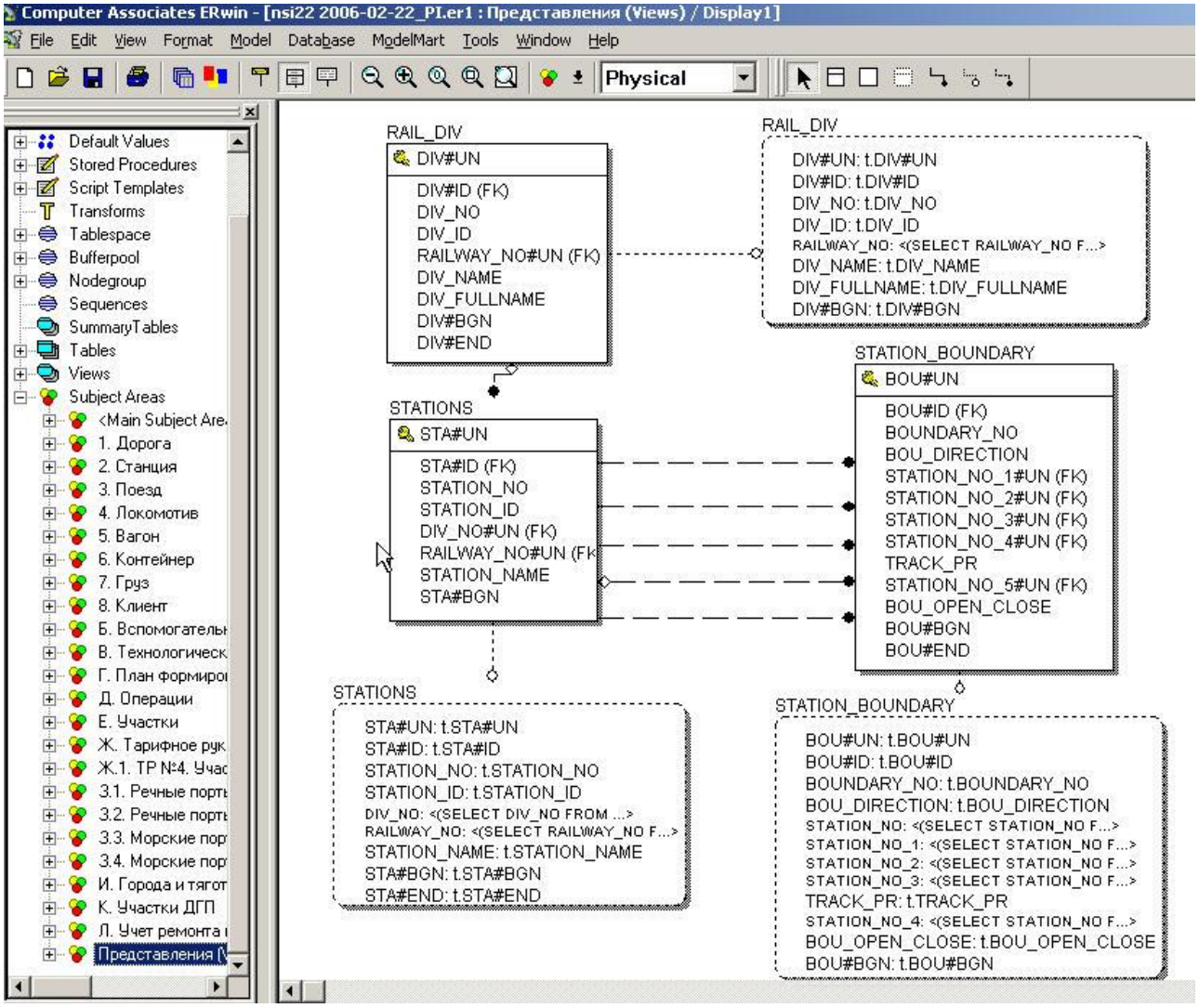


Рис. 5.6. Диаграмма представления для таблиц БД

На рис. 5.7. и 5.8. приведен диалог Views для определения представления и генерации оператора Create для его создания в среде ERwin.

На рис. 5.9. приведен диалог создания оператора Select для заполнения представления.

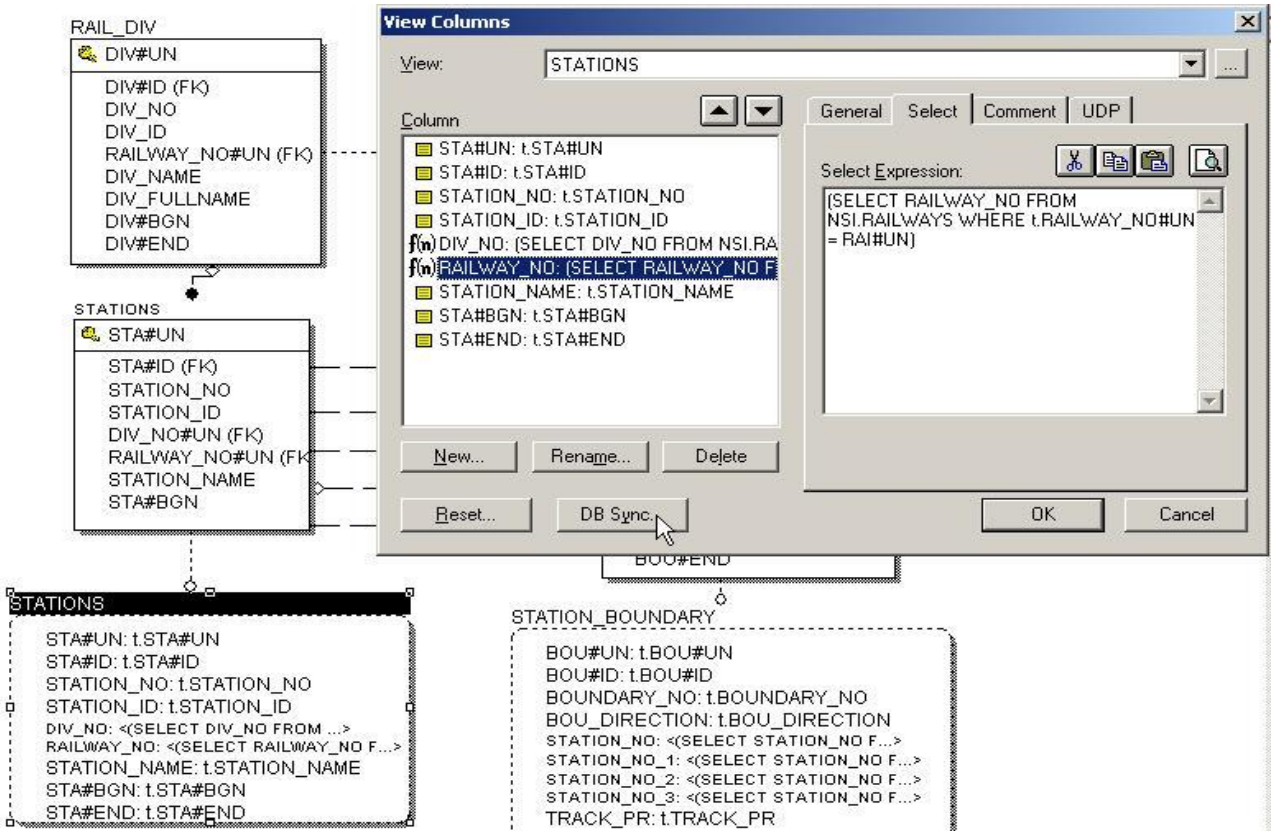


Рис. 5.7. Определение представления

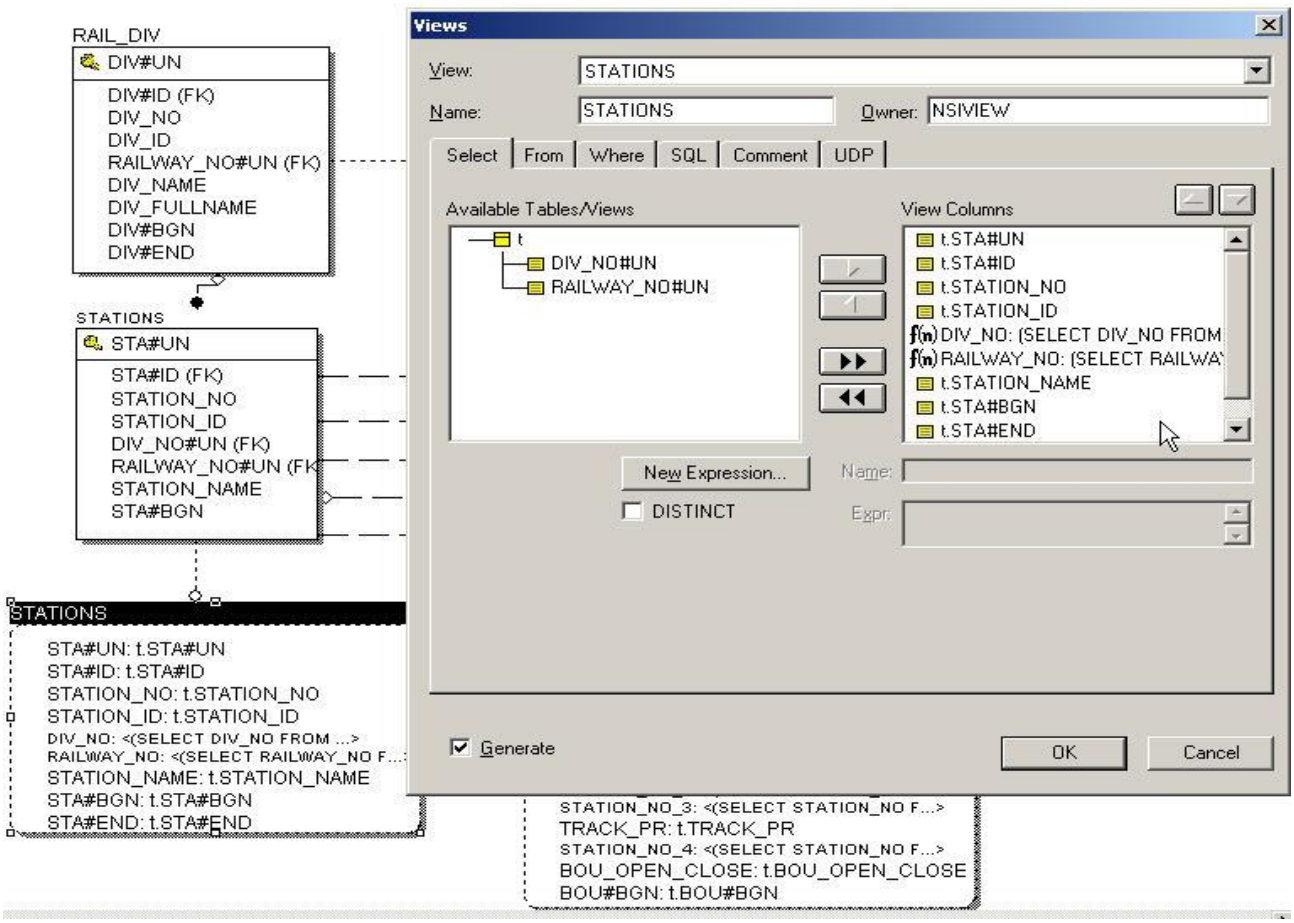


Рис. 5.8. Генерации оператора Create для создания представления

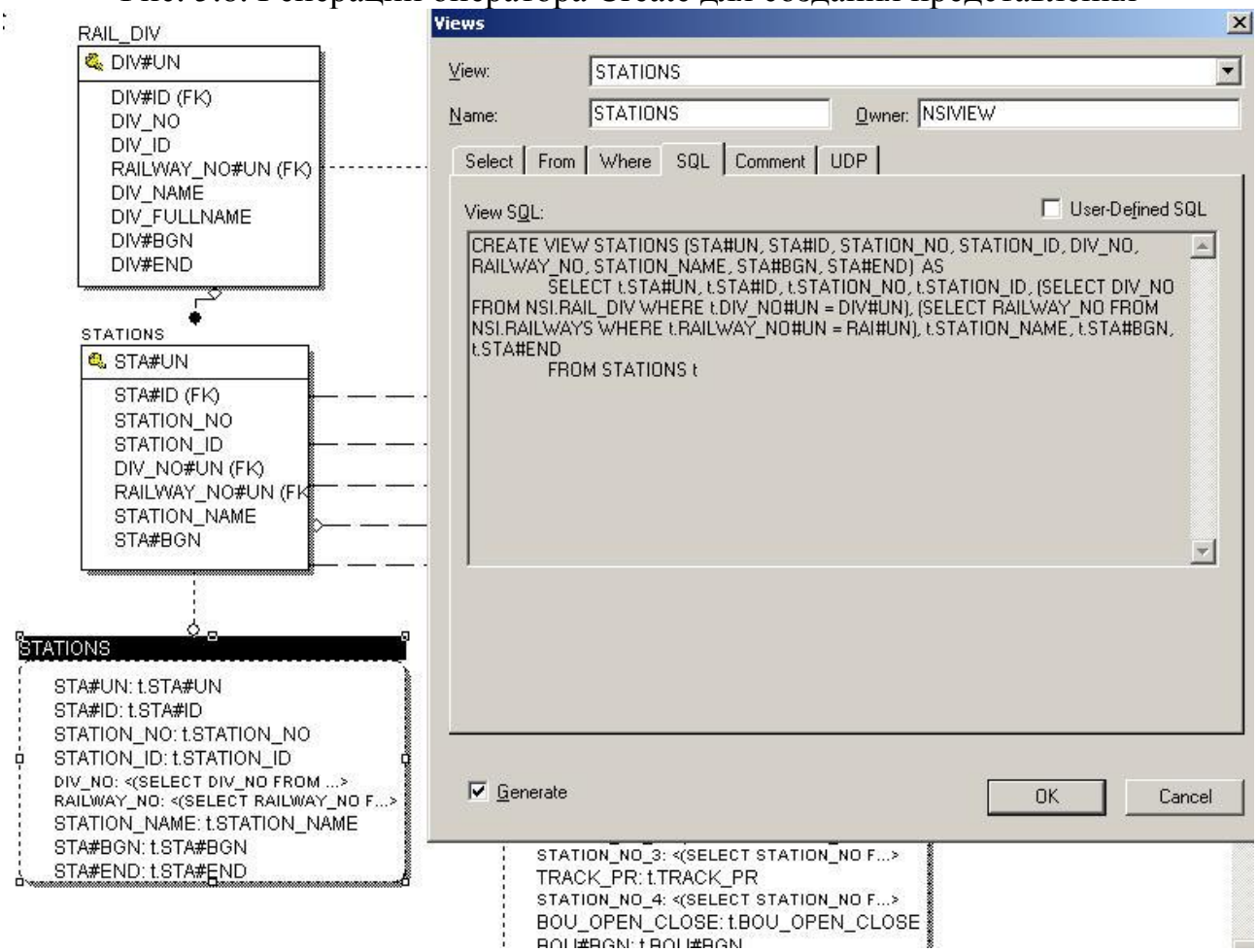


Рис. 5.9. Диалог создания оператора Select для заполнения представления

На рис. 5.10. приведена информация, выдаваемая пользователю прямо из таблицы БД, которая содержит внутренние ключи (PK и FK) и на рис. 5.11. приведена информация, выдаваемая пользователю через разработанное представление.

Таблица: STATIONS

Уникальный ключ записи **STA#UN**	Идентификатор объекта **STA#ID**	Код станции (ЕСР) **STATION_NO**	Мнемокод станции **STATION_ID**	Ссылка на таблицу RAIL_DIV (Код отделения) **DIV_NO#UN**	Ссылка на таблицу RAILWAYS (Код дороги) **RAILWAY_NO#UN**	Наименование станции **STATION_NAME**
192701	1927	140009	МИНС	30501	27001	МИНСК-СОРТ
192801	1928	140102	МИНСВ	30501	27001	МИНСК-СЕВ
192901	1929	140206	МИНП	30501	27001	МИНСК-ПАСС
193001	1930	140304	МИНВ	30501	27001	МИНСК-ВОСТ
193201	1932	140507	МИНЮ	30501	27001	МИНСК-ЮЖНЫЙ

Рис. 5.10. Информация, выдаваемая пользователю прямо из таблицы БД

Таблица: STATIONS

Уникальный ключ записи **STA#UN**	Идентификатор объекта **STA#ID**	Код станции (ЕСР) **STATION_NO**	Мнемокод станции **STATION_ID**	Ссылка на таблицу RAIL_DIV (Код отделения) **DIV_NO**	Ссылка на таблицу RAILWAYS (Код дороги) **RAILWAY_NO**	Наименование станции **STATION_NAME**
192700	1927	140009	МИНС	01	13	МИНСК-СОРТ
192800	1928	140102	МИНСВ	01	13	МИНСК-СЕВ
192900	1929	140206	МИНП	01	13	МИНСК-ПАСС
193000	1930	140304	МИНВ	01	13	МИНСК-ВОСТ
193100	1931	140403	СТЕП	01	13	СТЕПЯНКА
193200	1932	140507	МИНЮ	01	13	МИНСК-ЮЖНЫЙ
193300	1933	140600	РЗВ	01	13	РЕЗЕРВНАЯ СТАНЦИЯ
193400	1934	140704	ОЗЕР	01	13	ОЗЕРИЩЕ

Рис. 5.11. Информация, выдаваемая пользователю через разработанное представление

5.6. Нормализация данных

Здесь приводятся основные определения нормальных форм данных. Примеры нормализации данных можно найти в любом учебнике по БД.

Нормализация данных – это процесс проверки, тестирования и, если требуется, реорганизации сущностей (таблиц) и атрибутов (столбцов) в модели и сведения таблиц к набору столбцов, в котором все не ключевые столбцы зависят от столбцов первичного ключа. Она позволяет устранить недостатки в уже спроектированной модели данных, исключить дублирование данных, т.е. информация о некотором факте должна храниться только в одном месте – тем самым однозначно и окончательно определить принадлежность каждого столбца только определенной таблице. Нормализация делает БД более устойчивой к изменениям к новым требованиям задач, позволяет обеспечить целостность самой БД и сокращает память для хранения информации.

Все нормальные формы основаны на *понятии функциональной зависимости*.

Атрибут A1 сущности E *функционально* зависит от атрибута A2 сущности E тогда и только тогда, когда каждое значение A2 в сущности E связано точно с одним значением A1 в сущности E, т.е. для любого значения $x \leq A1$, существует только одно значение $y \leq A2$.

Атрибут A1 сущности E *полностью функционально* зависит от ряда атрибута A2 сущности E тогда и только тогда, когда A1: 1. функционально зависит от A2 и 2. нет зависимости от подмножества A2, т.е. нет транзитивной зависимости.

Первая нормальная форма (1NF) гласит – все атрибуты сущности содержат атомарные (неделимые) значения и среди атрибутов не должно встречаться повторяющихся групп. То есть, с одной стороны, атрибуты не должны содержать повторяющихся групп (нескольких значений атрибута) и, с другой, – атрибут не должен хранить разные по смыслу значения. Приведение к 1NF форме производится созданием новой таблицы для повторяющихся значений и установлением связи от прежней таблицы (PK) к новой (FK).

Сущность находится во второй нормальной форме (2NF), если она находиться в первой нормальной форме (1NF) и любой ее не ключевой атрибут полностью зависит от всего первичного ключа, а не от его части. 2NF имеет смысл для объектов со сложным ключом. Сущность приводится к 2NF выделением атрибутов, зависящих от части ключа в отдельную таблицу и установлением связи с основной таблицей.

Сущность находится в третьей нормальной форме (3NF), если она находится во второй нормальной форме (2NF) и любой ее не ключевой атрибут не зависит от другого не ключевого атрибута этой сущности. Не ключевые атрибуты не зависят от не ключевых атрибутов, в сущности нет атрибутов значения, которых получено из других атрибутов данной сущности. Сущность приводится к 3NF выделением атрибутов, зависящих от не ключевого атрибута в отдельную таблицу и установлением связи с основной таблицей.

Сущность находится в четвертой нормальной форме (4NF), если ни в одной строке нет нескольких многозначных фактов об одном объекте, т.е. нет

многозначных зависимостей между атрибутами. Сущность приводится к 4NF выделением атрибутов с многозначными фактами в отдельные таблицы.

Case система AllFusion Erwin Data Modeler помогает создание нормализованной модели БД, контролирует уникальность имени атрибута, поддерживает создания ключей и связей.

5.7. Примеры построения диаграмм

Диаграммы моделей строятся из сущностей (зависимых и независимых) и связей (идентифицирующих и не идентифицирующих) и отражают моделируемую предметную область. Ниже на рис. 5.12. приведена диаграмма модели «Участки диспетчерования», которая состоит сущностей: «диспетчерских участков», «укрупненных участков» и «линейных участков» и отношений связывающих эти сущности. В данном случае дочерние сущности зависят от родительских сущностей и в качестве отношений используются «идентифицирующие» связи. Родительские PK мигрировали в ключевую часть атрибутов потомков.

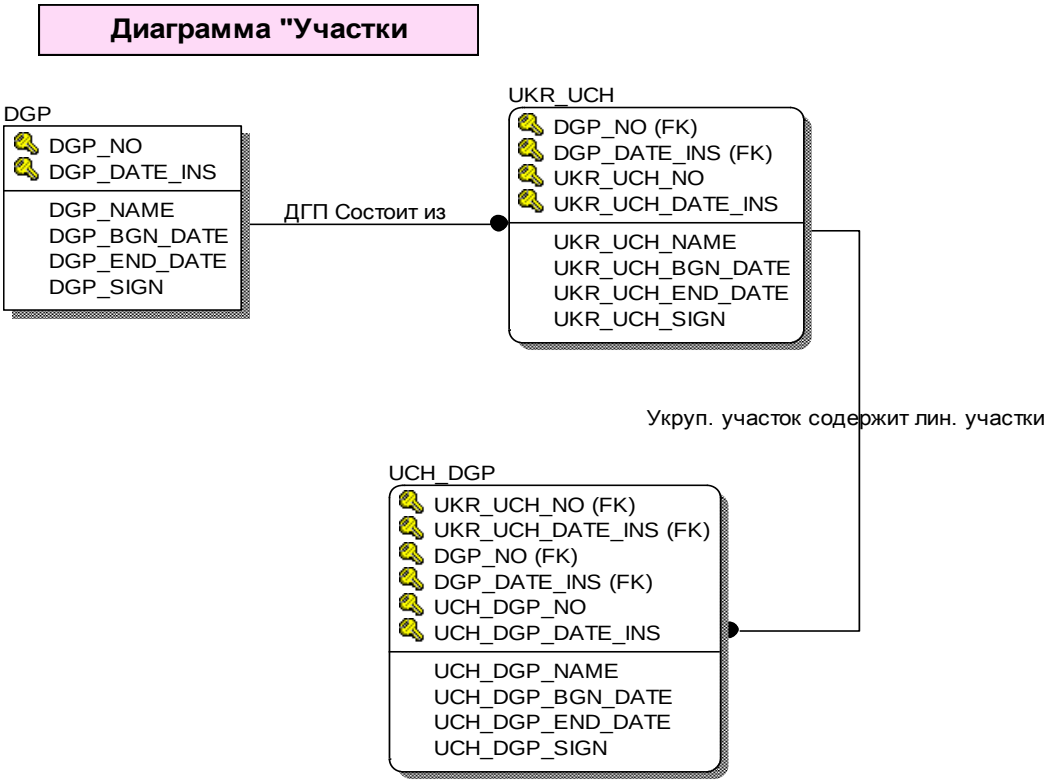


Рис. 5.12. Диаграмма модели «Участки диспетчерования»

На рис. 5.13. приведен фрагмент модели топологии железной дороги: в общем случае есть «Администрация дорог», которой подчиняются «Железные дороги» в данном государстве (Россия, Белоруссия и др.). «Железные дороги» делятся на «Отделения» и состоят из «Станций», полная административная подчиненность

известна, только для Белорусских станций. В данной диаграмме используются не идентифицирующие отношения, дочерние сущности не зависят от родительских сущностей, т.е. в некоторые FK могут иметь значения «по умолчанию» неравное значению родительского PK, «нет родительской записи».

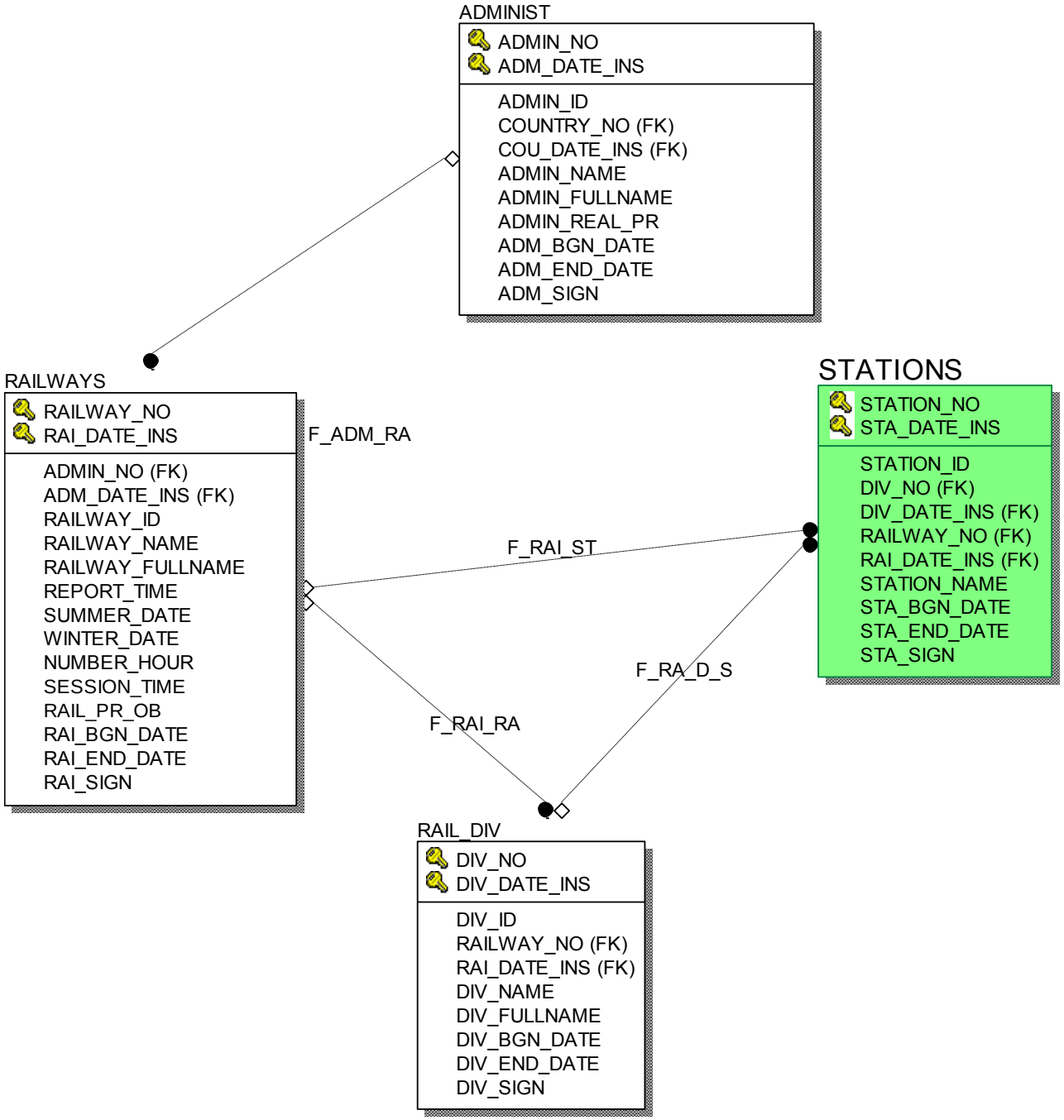


Рис. 5.13. Фрагмент модели топологии железной дороги

На рис. 5.14. сущность «Станция», содержащая все станции стран СНГ, используется для построения других сложных понятий таких как: «Внешний стык» и «Пограничный переход». Сущность «Внешний стык» определяется двумя станциями с одной и другой стороны «Железной дороги», а сущность «Пограничный переход» определяется пятью станциями (рис. 5.15.). Общее

множество по выделенным свойствам разбивается на другие подмножества, обладающие определенными признаками. Так как ключевой атрибут STATION_NO не может мигрировать больше одного раза как внешний ключ, то в дочерних сущностях данный атрибут переименован как STATION_NO_1, …., STATION_NO_5.

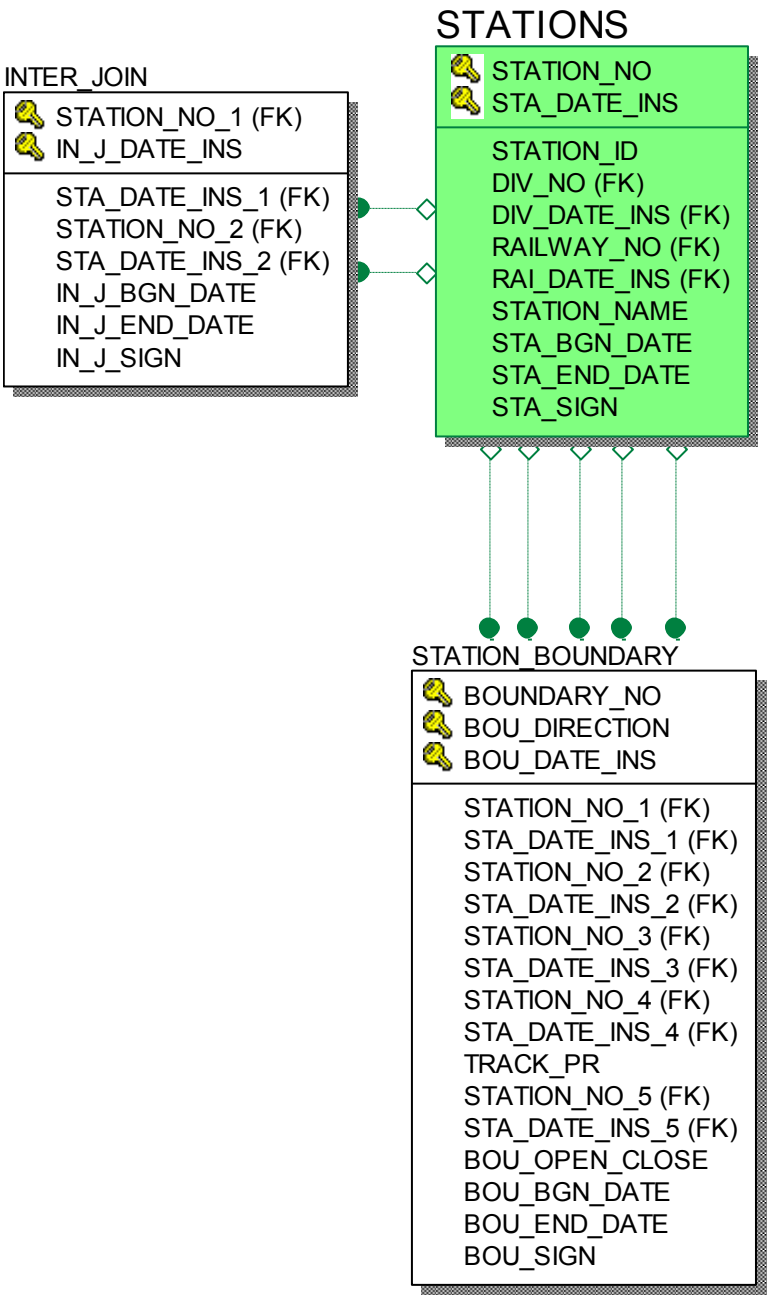


Рис. 5.14. Использование сущности «Станция»

Таблица.

Номер пограничного перехода BOUNDARY_NO	Номер направления в пограничном переходе BOU_DIRECTION	Код станции передачи вагонов Белорусской железной дороги (СПВ БЧ) STATION_NO_1	Код стыковой пограничной станции Белорусской железной дороги (СПС БЧ) STATION_NO_2	Код стыковой пограничной станции не Белорусской железной дороги (СПС не БЧ) STATION_NO_3	Код станции передачи вагонов не Белорусской железной дороги (СПВ не БЧ) STATION_NO_4	Код стыкового пункта учета поездов и вагонов (стык УПВ) STATION_NO_5
1	1	138507	136605	121101	121008	120518
1	2	137608	136605	121101	121008	120518
1	3	138507	136605	121101	120804	120518
1	4	137608	136605	121101	120804	120518
2	1	135208	135000	NULL	121008	134900
3	1	162900	164107	120202	121008	163000
3	2	162900	164107	120202	120804	163000
4	1	161306	161401	110200	110003	161700
5	1	161306	161202	066900	067000	161607
6	1	160002	161005	067405	067104	160801
7	1	160002	165805	172008	170004	165608
8	1	166704	169100	171401	170004	171346

Рис. 5.15. Определение сущности «Пограничный переход» пятью станциями

На рис. 5.16. и 5.17. приведены примеры разрешения связей многие – ко – многим. Данный тип отношений возникает в том случаи, если между сущностями не удается установить связь типа «родитель – потомок», эти сущности не зависят друг от друга. Так на рис. 5.16. приведены две сущности: классы операций и сами операции. С помощью дополнительной сущности «класс операции – операции» – установлены отношения между сущностями «класс операции» и «операции». Множество операций сущности «операция» и классифицировано по определенным признакам и разрешены отношения многие – ко – многим. Фактически новая сущность «класс операции – операция» может состоять только из всевозможных пар PK – ключей обеих сущностей, которые образуют PK новой сущности.

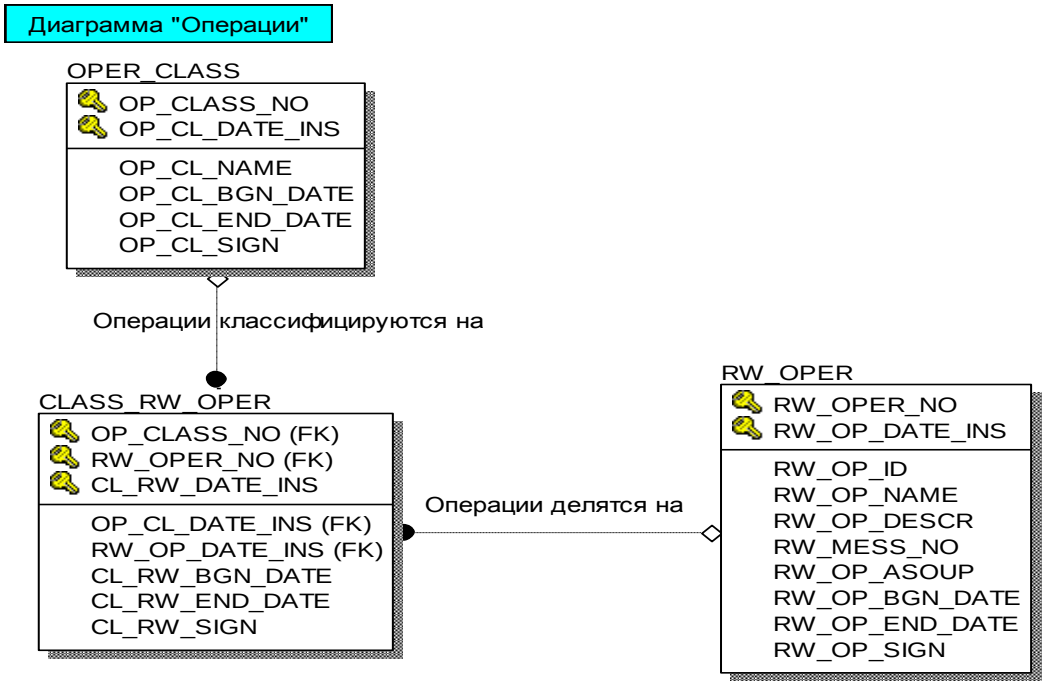


Рис. 5.16. Пример разрешения связей многие – ко – многим

Рис. 5.17. содержит аналогичное решение для разрешения проблемы связи между станциями и видами грузовых работ (параграфами станций) допустимых на них. На различных станциях могут выполняться различные виды грузовых работ, а на некоторых выполнение таких работ вообще не допускается (содержимое таблиц приведено на рис. 5.17.1 – 5.17.3).

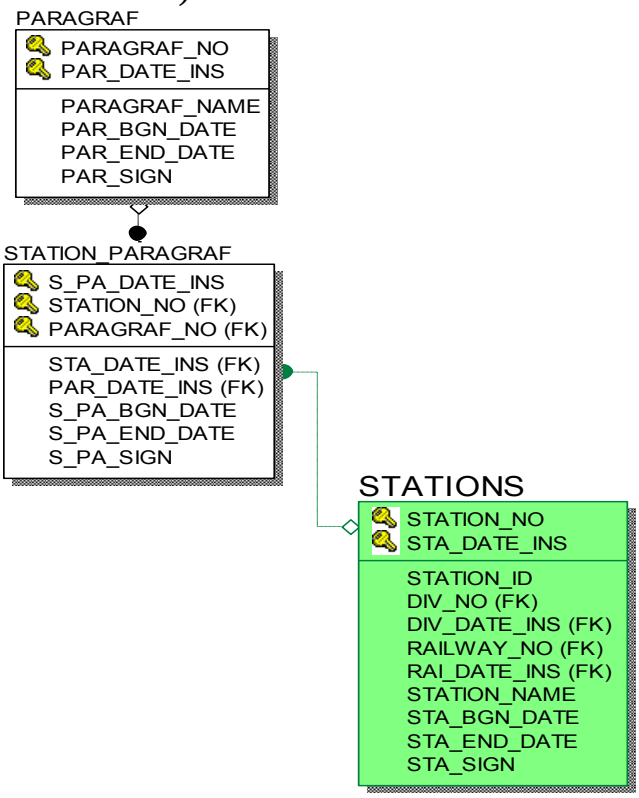


Рис. 5.17. Пример разрешения связей многие – ко – многим

NSI.PARAGRAF «Параграф станции»

Код параграфа PARAGRAF_NO	Наименование параграфа PARAGRAF_NAME
1	ПРИЕМ И ВЫДАЧА ПОВАГОННЫХ ОТПРАВОК ГРУЗОВ, ДОПУСКАЕМЫХ К ХРАНЕНИЮ НА ОТКРЫТЫХ ПЛОЩАДКАХ СТАНЦИЙ
10	ПРИЕМ И ВЫДАЧА ГРУЗОВ В УНИВЕРСАЛЬНЫХ КОНТЕЙНЕРАХ ТРАНСПОРТА МАССОЙ БРУТТО 30 Т НА СТАНЦИЯХ
10Н	ПРИЕМ И ВЫДАЧА ГРУЗОВ В УНИВЕРСАЛЬНЫХ КОНТЕЙНЕРАХ МАССОЙ БРУТТО 24 (30) И 30 Т НА ПОДЪЕЗДНЫХ ПУТЯХ
2	ПРИЕМ И ВЫДАЧА МЕЛКИХ ОТПРАВОК ГРУЗОВ,ТРЕБУЮЩИХ ХРАНЕНИЯ В КРЫТЫХ СКЛАДАХ СТАНЦИЙ
3	ПРИЕМ И ВЫДАЧА ГРУЗОВ ПОВАГОННЫМИ И МЕЛКИМИ ОТПРАВКАМИ, ЗАГРУЖАЕМЫМИ ЦЕЛЫМИ ВАГОНАМИ, ТОЛЬКО НА ПОДЪЕЗДНЫХ ПУТЯХ И МЕСТАХ НЕОБЩЕГО ПОЛЬЗОВАНИЯ
4	ПРИЕМ И ВЫДАЧА ПОВАГОННЫХ ОТПРАВОК ГРУЗОВ,ТРЕБУЮЩИХ ХРАНЕНИЯ В КРЫТЫХ СКЛАДАХ СТАНЦИЙ
5	ПРИЕМ И ВЫДАЧА ГРУЗОВ В УНИВЕРСАЛЬНЫХ КОНТЕЙНЕРАХ ТРАНСПОРТА МАССОЙ БРУТТО 3 И 5 Т НА СТАНЦИЯХ
6	ПРИЕМ И ВЫДАЧА ГРУЗОВ В УНИВЕРСАЛЬНЫХ КОНТЕЙНЕРАХ ТРАНСПОРТА МАССОЙ БРУТТО 3 И 5 Т НА ПОДЪЕЗДНЫХ ПУТЯХ
7	ЗАПРЕЩАЕТСЯ ПРИЕМ И ВЫДАЧА ЛЕГКОВОСПЛАМЕНЯЮЩИХСЯ ГРУЗОВ НА СТАНЦИЯХ
8	ПРИЕМ И ВЫДАЧА ГРУЗОВ В УНИВЕРСАЛЬНЫХ КОНТЕЙНЕРАХ ТРАНСПОРТА МАССОЙ БРУТТО 20 Т НА СТАНЦИЯХ
8Н	ПРИЕМ И ВЫДАЧА ГРУЗОВ В УНИВЕРСАЛЬНЫХ КОНТЕЙНЕРАХ МАССОЙ БРУТТО 20 И 24 Т НА ПОДЪЕЗДНЫХ ПУТЯХ

9	ПРИЕМ И ВЫДАЧА МЕЛКИХ ОТПРАВОК ГРУЗОВ, ДОПУСКАЕМЫХ К ХРАНЕНИЮ НА ОТКРЫТЫХ ПЛОЩАДКАХ СТАНЦИЙ

стр. 1 из 2 >> >|

Рис. 5.17.1

NSI.STATION_PARAGRAF «Таблица соответствия станций параграфам»

Код станции (ECP) STATION_NO	Код параграфа PARAGRAF_NO
010708	3
010801	3
010905	3
011005	3
011109	3
011202	3
011306	1
011306	2
011306	3
011306	4
011306	9
011400	1

Рис. 5.17.2

NSI.STATIONS «Станция»

Код станции (ЕСР) STATION_NO	Мнемокод станции STATION_ID	Код отделения DIV_NO	Код дороги RAILWAY_NO	Наименование станции STATION_NAME
137006	СЛОН	02	13	СЛОНИМ
137101	ПЛНК	02	13	ПОЛОНКА
137129	NULL	02	13	БОРОВЦЫ
137203	МОСТ	02	13	МОСТЫ
137307	РОСЬ	02	13	РОСЬ
137400	РОЖН	02	13	РОЖАНКА
137504	СКРБ	02	13	СКРИБОВЦЫ
137608	ЛИДА	02	13	ЛИДА
137612	NULL	02	13	МИНОЙТЫ
137701	ГУДЫ	02	13	ГУДЫ
137805	БАСТ	02	13	БАСТУНЫ
137909	БЕНК	02	13	БЕНЯКОНИ

Рис. 5.17.3.

На рис. 5.18. приведена не простая диаграмма, отражающая топологию участков железной дороги. Кроме этого на этой диаграмме отражены участки движения локомотивов. Опять же, основным строительным элементом является сущность «Станция» и ее подмножество «Выделенная станция». Логическая сущность «Участок» объединяет такие объекты, которые содержат информацию об участках железной дороги, принадлежности станций участкам, выделенных станциях, ближайших выделенных станциях, длины участков и время хода на них, участках обращения локомотивов, принадлежности участков железной дороге, участку обращения локомотивов, расстояния и времена хода от выделенных до невыделенных станций.

Даная диаграмма в терминах языка IDEF1x реализует географическую топологию железной дороги: «Станция» (все станции), «Выделенная станция» (подмножество Станций), «Участок» (участок дороги от одной выделенной станции до другой, на котором располагаются другие станции), «Участок обращения локомотивов» (состоит из нескольких участков дороги) и содержит все элементы данных этой топологии.

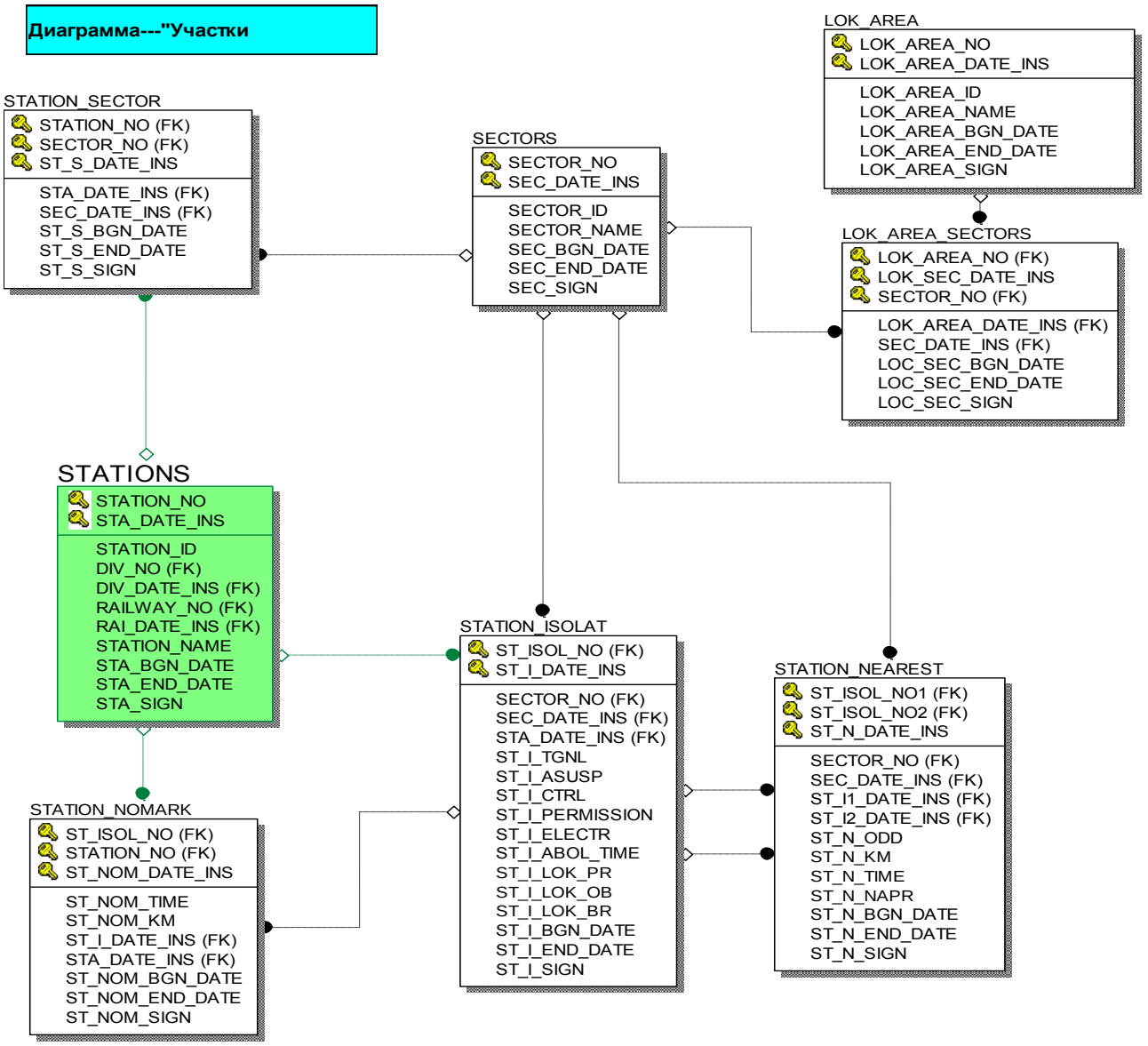


Рис. 5.18. Топология участков железной дороги

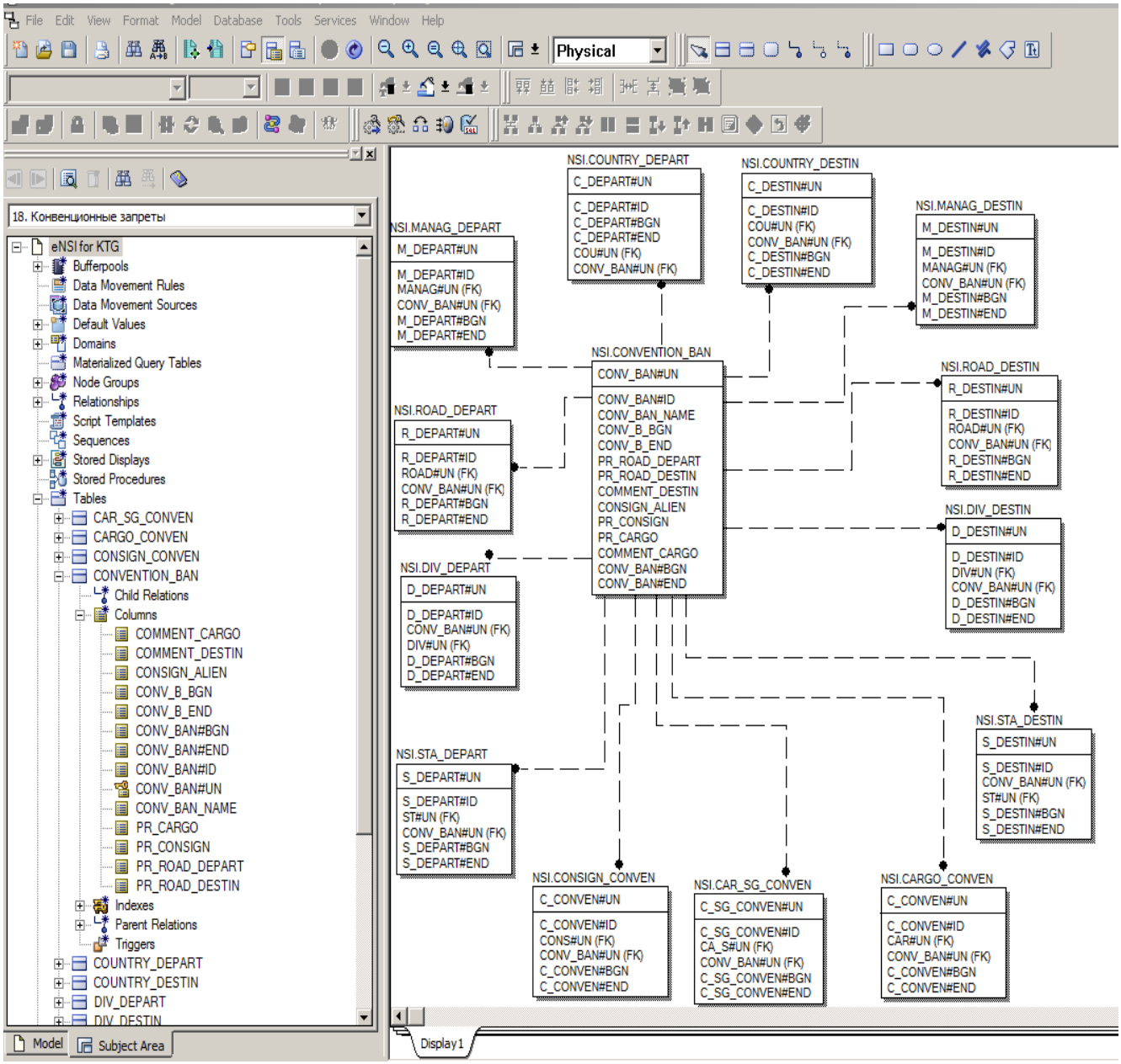


Рис. 5.18.1. Конвенционные запреты

Существует особый вид отношений – рекурсия. Это такой вид отношений (связей) когда одна и та же сущность является и родительской и дочерней. При реализации таких связей PK мигрирует в качестве FK в состав не ключевых атрибутов, один и тот же атрибут не может появиться дважды под одним и тем же именем. Поэтому нужно или переименовывать атрибуты FK (см. рис. 5.14.) или использовать имена ролей связей при именовании таких FK (см. рис. 5.19.).

Возможны и другие виды рекурсии: неявные (косвенные), сетевые. Есть сущности, которые находятся сами с собою в связи «многие – ко – многим». Примером такой сущности может быть сущность родственник, когда значения такой сущности связаны с другими значениями этой же сущности. В общем случае, можно говорить о сетевой рекурсии внутри самой сущности. Разрешить такие связи можно с помощью введения новой сущности отношения родственников, в которой будут заданы пары ключевых атрибутов и атрибут тип отношения «мать–сын», «дед–внук», «тесть–зять» и т.д. Данная сущность будет дважды связана с сущностью родственник и PK сущности родственник дважды мигрирует в новую сущность отношения родственников как FK. Но и как в предыдущем случае, правила именования атрибутов должны быть соблюдены: или должны быть введены разные ролевые отношения или имена атрибутов должны бать изменены.

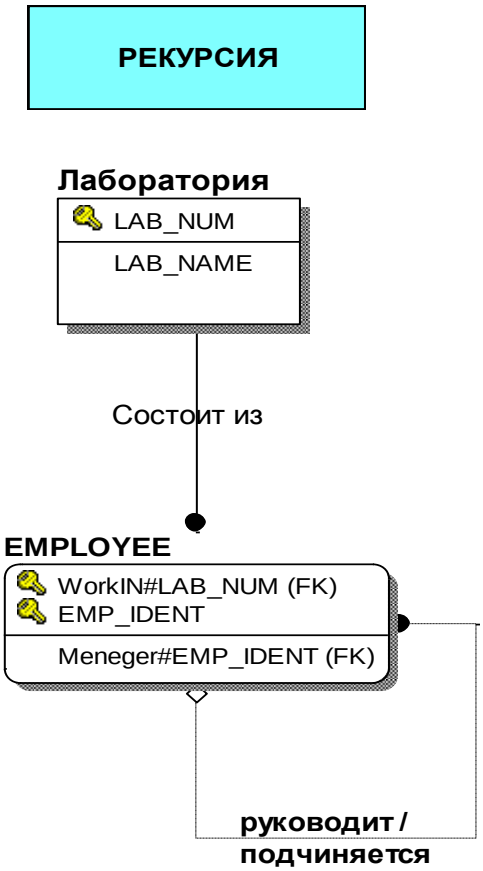


Рис. 5.19. Пример рекурсии

5.8. Общие сведения о среде проектирования AllFusion Erwin Data Modeler

В настоящее время быстрое и грамотное проектирование и реализация баз данных (БД) без средств графического моделирования данных практически невозможна. Ручные методы разработки и сопровождения БД давно уже исчерпали себя в силу низкого качества и эффективности получаемого результата и низкой производительности труда.

В данной работе, в качестве базового компонента проектирования и разработки БД используется готовое CASE средство AllFusion Erwin Data Modeler, опирающееся на стандарты разработки БД FIPS, ISO9001 и язык моделирования БД IDEF1X. CASE средство AllFusion Erwin Data Modeler реализует структурные подходы к развитию информационной системы и к дизайну данных этих систем. Данное CASE средство является лидером среди аналогичных систем, около 60% средств моделирования данных мирового рынка принадлежит AllFusion Erwin Data Modeler.

AllFusion Erwin Data Modeler не только помогает в дизайне (изображении) логической модели данных, он поддерживает дизайн соответствующей физической модели данных и автоматически генерирует структуру физической БД (“Forward engineering”).

AllFusion Erwin Data Modeler включает в себя средства для генерации из функционирующей физической БД соответствующей ей модели данных (“reverse engineering”), поддерживая при этом обе – физическую и логическую/физическую модели данных. Таким образом можно поддерживать функционирующие БД или осуществлять миграцию всей БД или ее части (подсхемы БД) на другие серверные платформы.

CASE средство также включает средства автоматического сравнения моделей и баз данных (“complete compare”), выдачи и анализа различий между ними, что позволяет в дальнейшем выборочно перемещать эти различия в модель или генерировать их в БД. Общая функциональная схема проектирования БД с помощью CASE средства AllFusion Erwin Data Modeler представлена на рис. 5.20.

Использование уровней моделирования позволяет реализовать хорошо структурированный нисходящий сверху вниз подход к разработке модели БД, при котором успех детализации объектов на нижестоящем уровне создается на каждом этапе проектирования.

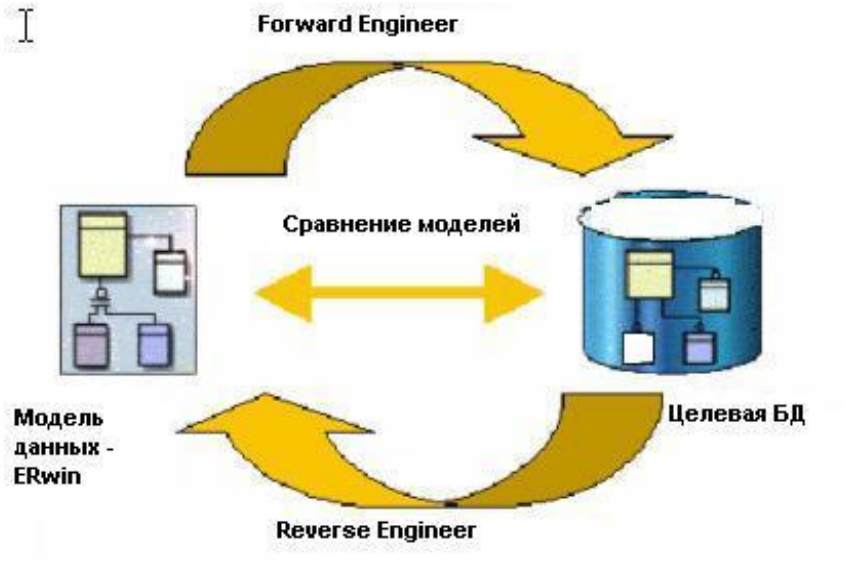


Рис. 5.20. Общая функциональная схема проектирования БД

На верхних уровнях моделирования информационная модель включает общие сведения об объектах, обеспечивая тем самым ее технологическую независимость от конкретной платформы СУБД. Она называется логической моделью. На более низких уровнях проектирования модель дополняется детализированными сведениями об объектах. Это связано с тем, что физическая организация данных, например, в СУБД DB2 существенно отличается от физической организации этих данных в СУБД Oracle. Такая детализированная модель называется физической моделью.

AllFusion Erwin Data Modeler обеспечивает два вышеуказанных уровня представления модели – логический и физический. Логический уровень включает наиболее общие сведения о предметной области. Объекты модели логического уровня называются сущностями и атрибутами. Логический уровень модели данных может быть построен на основе диаграмм модели бизнес–процессов предметной области. Физический уровень позволяет описать всю детальную информацию о конкретных физических объектах – таблицах, столбцах, связях между объектами, индексах, процедурах и др. При этом AllFusion Erwin Data Modeler позволяет создавать модели трех типов: модель, имеющую логический уровень представления данных, модель, имеющую физический уровень представления данных, и модель, имеющую как логический, так физический уровень.

Такое многоуровневое моделирование БД имеет ряд достоинств. Наиболее очевидное достоинство – это систематическое документирование, которое может использоваться постоянно для развития базы данных и прикладных применений, чтобы определить системные требования и связать их непосредственно с требованиями конечного пользователя. Второе достоинство – это обеспечение ясной картины ссылочных ограничений целостности БД. Поддержание ссылочной целостности существенно в реляционной модели, где отношения не кодируются явно. Третье достоинство – это независимая картина базы данных, которое следует из "логической" модели БД. Логическая модель БД может использоваться

автоматизированными средствами для генерации различных физических СУБД. Таким образом, можно использовать одну и ту же логическую модель на языке IDEF1x в Erwin, для получения из нее как схемы таблиц DB2, также как и схемы таблиц для других реляционных СУБД.

Ниже в таблице приводится перечень общих терминов логической и физической моделей, используемых в дальнейшем при проектировании и сопровождении БД:

Логическая модель	Физическая модель
Сущность /объект/ (Entity)	Таблица (Table)
Зависимая сущность	Внешний ключ FK является частью PK дочерней таблицы
Независимая сущность	Родительская или дочерняя таблица, у которой FK не является частью ее PK
Атрибут (Attribute)	Столбец (Column)
Логический тип данных (текст, число, дата)	Физический тип данных (зависит от выбранного сервера назначения)
Домен (логический)	Домен (физический)
Первичный ключ (Primary key)	Первичный ключ, PK индекс
Внешний ключ (Foreign key)	Внешний ключ, FK индекс
Альтернативный ключ (AK)	AK индекс— уникальный, но не первичный ключ
Инверсионный вход (IE)	IE индекс — не уникальный индекс, созданный для поиска информации в таблице по не уникальному значению
Ключевая группа (Key group)	Индекс (Index)
Бизнес–правило (Business rule)	Триггер или хранимая процедура
Правило проверки данных (Validation rule)	Принудительный контроль данных (Constraint)
Отношение (Relationship)	Отношение, реализованное с использованием FKs
Идентифицированные (определенные) отношения (Identifying)	FK как часть PK дочерней таблицы (выше линии)
Не идентифицированные (неопределенные) отношения (Non–Identifying)	FK не является частью PK дочерней таблицы (ниже линии)
Отношение подтипа (Subtype)	Денормализованные таблицы (Denormalized tables)
Отношение многие ко многим (Many–to–many)	Ассоциативная таблица (Associative table)

Ссылочная (относительная) целостность (cascade, restrict, set null, set default)	Триггеры для операций INSERT, UPDATE и DELETE
Мощность, количество элементов (Cardinality)	Триггеры для операций INSERT, UPDATE и DELETE

При проектировании смешанной модели, включающей логический и физический уровень, переключение между ними осуществляется с помощью списка выбора “Model type indicator” – «Указатель типа модели» в панели инструментов AllFusion Erwin Data Modeler (logical / physical). Состояние этого указателя очень важно, так как оно определяет приоритет терминов логического уровня перед физическим уровнем. Надо первоначально определять объекты и их характеристики на логическом уровне, затем при первоначальном переключении на физический уровень соответствующие термины для него будут созданы автоматически (рис. 5.21.).

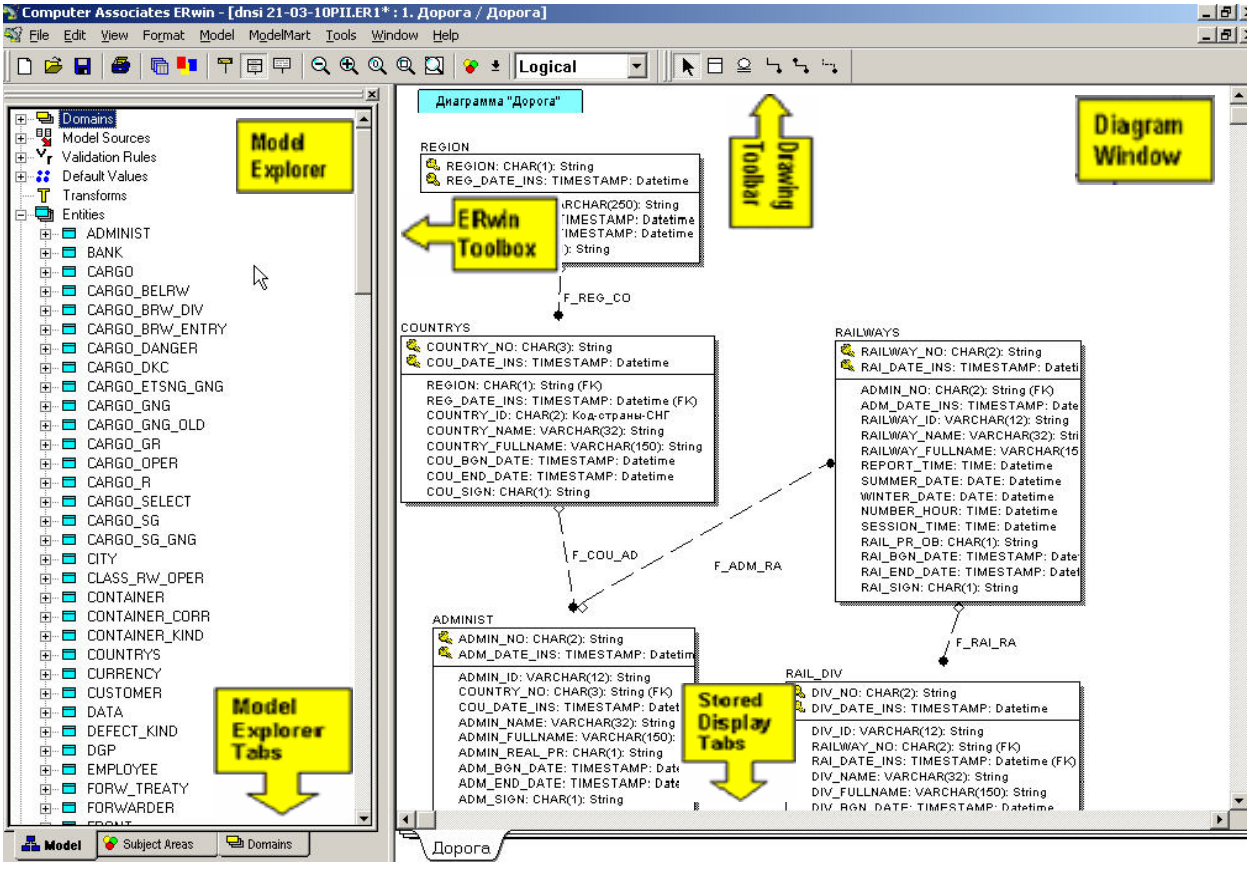


Рис. 5.21. Интерфейс пользователя AllFusion Erwin Data Modeler

Работа с компонентом в Erwin организуется с помощью Горизонтального меню “Menu bar” и двух рабочих пространств “Model Explorer” и “Diagram Window”. Горизонтальное меню AllFusion Erwin Data Modeler располагается в верхней части экрана и, хотя оно напоминает типовое для такого рода программ, полный перечень функций меню AllFusion Erwin Data Modeler достаточно большой и требует некоторого времени для поэтапного прохождения по всем операциям и

возможностям, причем некоторые из них выполняются достаточно редко. Рабочее пространство “Model Explorer” расположено в левой стороне экрана. Оно обеспечивает иерархическое текстовое изображение модели данных, которая синхронно в графическом виде представляется в правой части экрана в рабочем пространстве “Diagram Window”.

В любом из этих пространств по своему выбору разработчик БД может осуществлять вызов всех операций по обслуживанию элементов логической и физической схем БД: создание, удаление, переименование, изменение характеристик схем, подсхем, объектов, атрибутов, связей, таблиц, столбцов, ключей, шаблонов, триггеров, скриптов, табличных пространств, пулов буферов и других характеристик. AllFusion Erwin Data Modeler обеспечивает координацию внесения изменений в обоих рабочих пространствах – как в текстовом, так и в графическом виде.

Вся работа в рабочем пространстве “Model Explorer” выполняется в текстовом виде, т.е. вводом в выдаваемые AllFusion Erwin Data Modeler окна и поля соответствующих значений, либо путем выбора их из предоставленного AllFusion Erwin Data Modeler списка значений.

В “Diagram Window” инструментарий AllFusion Erwin Data Modeler синхронно создает или изменяет объекты схемы в графическом режиме. Разработчик БД может работать в этом окне, используя механизм перетягивания графических элементов схемы (объектов, связей) из AllFusion Erwin Data Modeler Toolbox и Drawing toolbar в окно “Diagram Window”. Здесь можно просто щелкнуть мышкой по любому объекту или связи и вызвать соответствующее им окно для выдачи сведений о нем и их корректировки.

Создание модели данных – длительный, итерационный процесс по идентификации и документированию предметной области в той части общесистемных требований, которая касается структур данных и взаимосвязей между ними.

В общем случае, разработка и сопровождения БД включает такие работы как: анализ предметной области, создание схемы БД, корректировку и поддержание схемы БД в актуальном состоянии, выпуск новой версии БД, выгрузку и загрузку данных. Жизненный цикл БД (по аналогии с жизненным циклом программного обеспечения) включает не только этап создания (проектирования) логической и физической моделей БД, этап генерации БД, этап загрузки данных и тестирования, но и этапы работ, которые выполняются при сопровождении БД:

получение исходных данных для внесения изменений и подготовка документации для внесения изменений;

выполнение подготовительных действий по анализу модели эксплуатируемой БД и планированию создания и запуска ее новой версии;

выполнение редактирования старой схемы и получение новой схемы, получение задания на языке DDL для создания новой версии БД;

запуск задания на создание новой пустой БД;

выполнение выгрузки (экспорта) данных из старой версии БД;

выполнение загрузки (импорта) данных из файлов–копий формата IXF в новую версию БД;

внесение или дополнение необходимых данных в новые объекты новой версии;

выполнение процедуры проверки и опытной эксплуатации новой версии БД;

ввод в промышленную эксплуатацию новой версии БД и объявление об ее выпуске.

5.8.1. Построение логической модели

Существует три вида логических моделей, которые используются, чтобы охватить информационные требования предметной области: “Entity Relationship Diagram” (ERD) – “Диаграмма сущность – связь”, “Key–Based Model” (KBM)– “Модель на основе ключа” и “Fully Attributed model” (FAM) – “Полностью атрибутная модель”.

ERD и KBM модели также называются "моделями данных области", потому что они часто охватывают широкую предметную область, которая является, как правило, большей чем бизнес правила, направленные на единственный отдельный проект автоматизации.

ERD модель используется на первоначальном этапе проектирования для отображения основных сущностей и возможных их связей для обсуждения с экспертами предметной области структур данных.

Модель данных, основанная на ключах (KBM), включает описание всех сущностей и ключей, которые соответствуют предметной области. Она дает более подробное описание предметной области.

Полностью атрибутная модель (FAM) – дает наиболее полное представление о структурах данных. Она состоит из полностью определенных: сущностей, их атрибутов и связей и должна представлять данные в третьей нормальной форме.

5.8.1.1. Диаграмма сущность – связь

Первый шаг в построении логической модели – конструирование “Диаграммы сущность – связь” (ERD), как самого высокого уровня моделирования данных рассматриваемой предметной области. ER–диаграмма показывает основные сущности и связи, которые определяют предметную область в целом.

Диаграмма связей (отношений) сущностей использует два главных строительных элемента: сущности и связи (отношения), и возможно некоторые атрибуты. Эти термины, проведя аналогию диаграммы ERD с разговорным языком, можно упрощенно интерпретировать следующим образом: сущности – это существительные, атрибуты – это прилагательные или характеристики и связи – это глаголы. Значениями сущностей являются экземпляры подобных объектов

реального мира, а атрибуты их характеристики. Построение модели данных в Erwin – просто вопрос обнаружения правильного собрания существительных, глаголов и прилагательных и размещения их всех вместе.

Цель ERD моделирования состоит в том, чтобы обеспечить наиболее общее представление о требованиях предметной области, достаточных для планирования дальнейшей разработки информационной системы. Эти модели не очень детализированы, включают только главные сущности, нет многих характеристик объектов (атрибутов), если таковые вообще имеются. Могут присутствовать отношения многие–ко–многим и неопределенные отношения, ключи обычно вообще отсутствуют. Это прежде всего первоначальное представление о бизнес–правилах или первоначальная модель для обсуждения.

ERD модель может быть разделена на подсхемы (subject areas), которые используются для определения части модели, удовлетворяющей интересы отдельного приложения или даже отдельных функций. Использование подсхем позволяет разделить большие модели на меньшие, более управляемые подмножества сущностей, что может быть легче для определения и дальнейшей поддержки (см. рис. 5.22.).

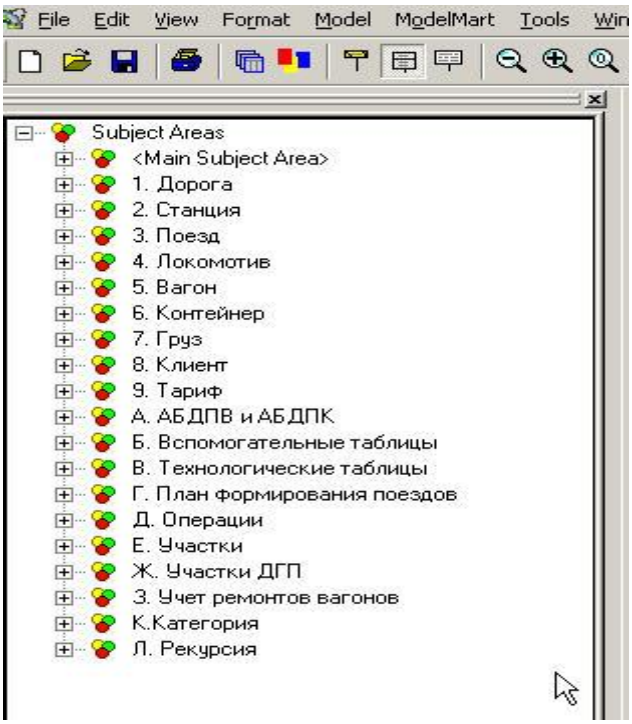


Рис. 5.22. Пример использования подсхем

В диаграмме ERD, сущность (объект) представляется в виде прямоугольника, который содержит название (имя) объекта. Названия сущностей (объектов) всегда желательно употреблять в единственном числе, что облегчает "чтение" диаграммы в дальнейшем. Связь между двумя объектами на диаграмме представляется в виде линии и она указывает, что данные одной таблицы находятся в некоторой связи

(отношении) с данными из другой таблицы. В этой связи всегда присутствует пара таблиц, родительская (Parent) и детская (Child).

На рис. 5.23. приведена ERD модель, на которой приведены имена сущностей, их определения и отношения между ними.

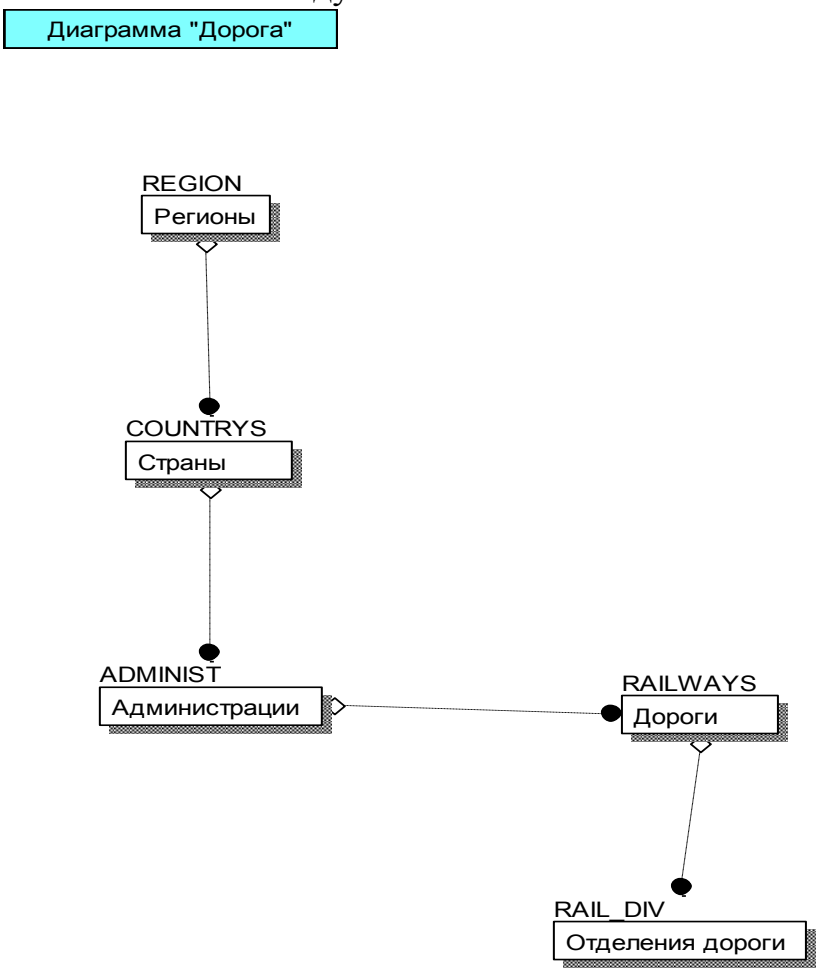


Рис. 5.23. Пример ERD модели

5.8.1.2. Модель данных на основе ключа

“Модель данных на основе ключа” – это модель данных, которая полностью описывает все основные структуры данных, которые удовлетворяют требованиям предметной области. Цель модели KBM – включение всех сущностей и атрибутов, которые представляют интерес для бизнеса. Модель на основе ключа, как и следует из ее названия, в основном содержит в себе ключи. В логической модели ключ однозначно идентифицирует экземпляр данных, т.е. строку внутри сущности (таблицы). В дальнейшем, когда соответствующая физическая модель будет использоваться, применение ключа обеспечит быстрый доступ к нужным данным.

В целом, модель KBM охватывает те же самые возможности, что и ERD, но раскрывает их более подробно и детально, насколько это возможно. В порядке развития ERD к правильной логической модели данных, необходимо определить

ключи, чтобы однозначно идентифицировать каждый элемент в сущности (объекте), т.е. его значение.

В каждом объекте в диаграмме модели данных горизонтальная линия разделяет атрибуты на две группы: ключевую область (выше линии) и не ключевую область или область данных (ниже линии).

Ключевая область содержит первичный ключ (Primary key PK) для сущности. Первичный ключ может включать один или более первичных ключевых атрибутов. Количество их должно быть достаточным, чтобы выбранные атрибуты сформировали уникальный идентификатор для каждого элемента в сущности. Сущность, кроме ключевых атрибутов, как правило, имеет не ключевые атрибуты.

Всем вышеописанным ключам Erwin автоматически присваивает имена индексов и порядковые номера путем добавления специальных кодов перед именем таблицы следующим образом, при желании имена ключе и индексов можно изменить вручную.

Замечание. Первичный ключ, выбранный для сущности при проектировании логической модели, может иногда и не быть первичным ключом в физической модели, как исключение, для более эффективного доступа к таблице. В этом случае ключ может быть изменен, чтобы удовлетворить потребности и требования физической модели базы данных в некотором исключительном случае.

Ниже на рис. 5.24. и 5.25. приведены KBM диаграммы некоторых моделей на основе PK ключей. В данных моделях присутствуют только имена атрибутов, которые могут быть использованы для идентификации значений сущности.

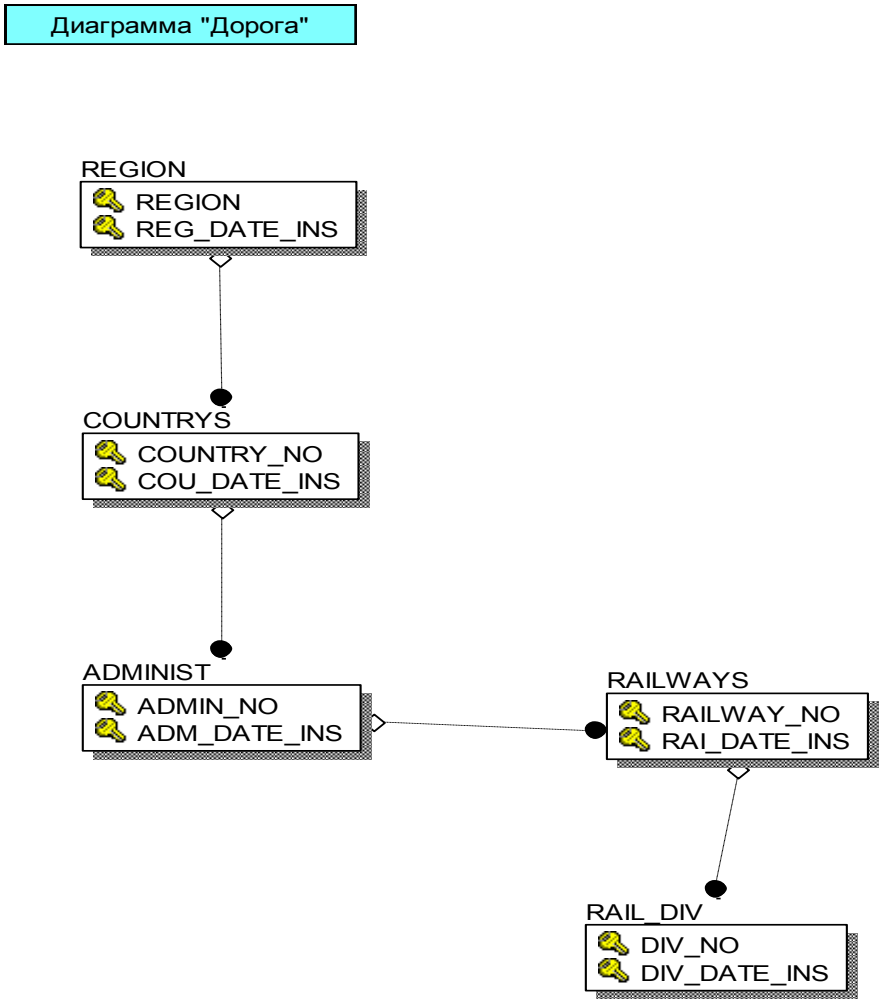


Рис. 5.24. Пример КВМ диаграммы

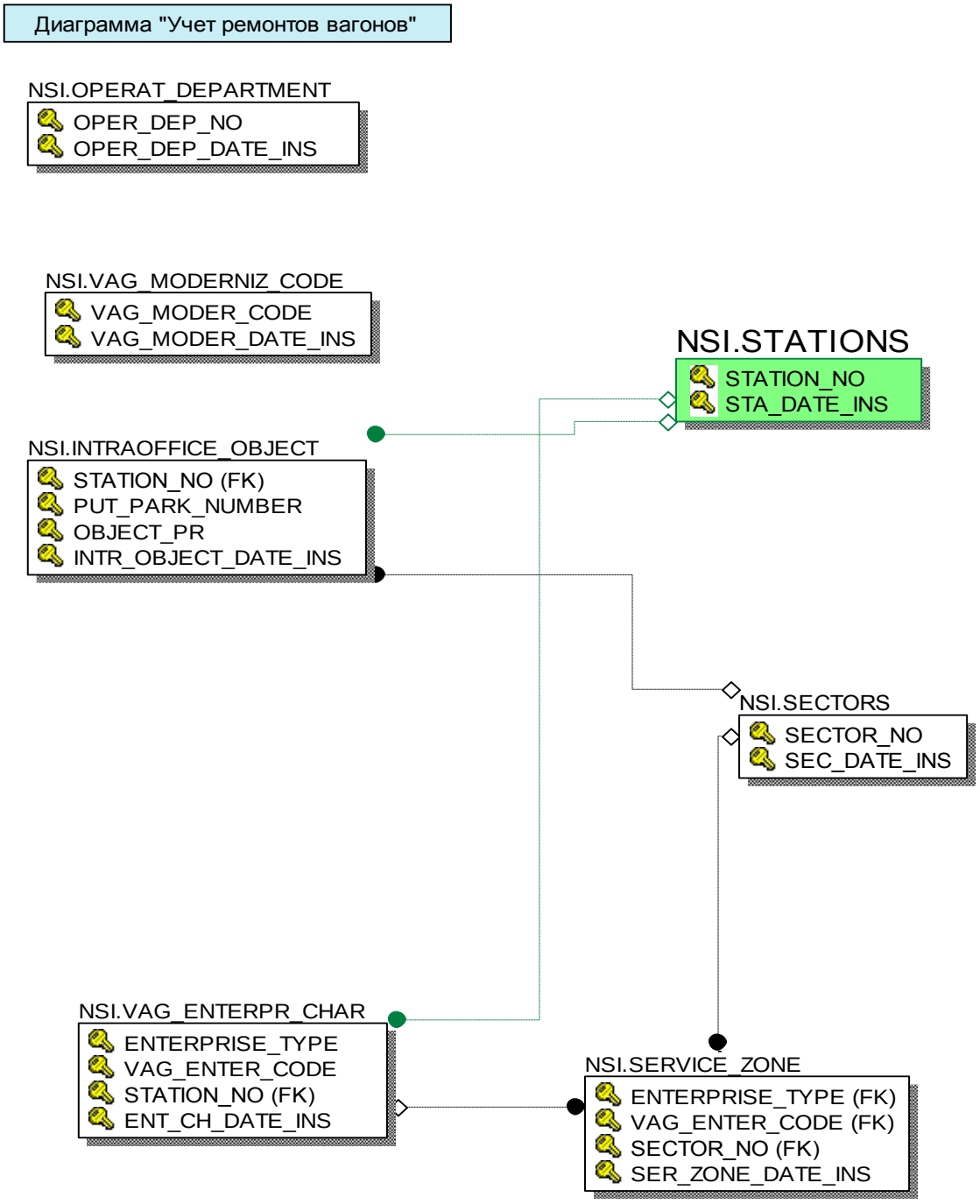


Рис. 5.25. Пример KBM диаграммы

5.8.1.3. Полная атрибутивная модель

“Полная атрибутивная модель” (FAM) получается из “Модели на основе ключа” добавлением в сущности недостающих атрибутов, уточнением связей между сущностями, установления ссылочной целостности и нормализацией самих сущностей.

В терминологии IDEF1X рассматривается концепция зависимых и независимых сущностей, как тип логических отношений, связывающих две сущности. Сущности, которые не зависят ни от какой другой сущности в модели, называются независимыми сущностями. В зависимых сущностях устанавливается логическая связь путем использования общих атрибутов в этих сущностях.

Связевые отношения выступают как “глаголы” между сущностями (“существительными”) на диаграмме. Правильный подбор этих “связевых глаголов” между зависимыми сущностями позволяет использовать их для чтения отношений от родительских таблиц к дочерним, и в обратную сторону, используя пассивную форму соответствующего глагола.

Зависимая сущность не может существовать без родителя и без идентификации этой зависимости. Это означает, что зависимая сущность не может быть идентифицирован без использования ключа родителя. Таким образом, в соответствии с концепцией отношений сущностей в IDEF1X, связь устанавливается миграцией атрибутов первичного ключа (PK) из родительской сущности в дочернюю, где эти атрибуты дочерней сущности называются уже внешним ключом (Foreing Key FK). В общем случае некоторая дочерняя сущность может быть одновременно связана с несколькими родительскими сущностями, и иметь несколько внешних ключей (FKn). На диаграмме модели данных соответствующие атрибуты связи в дочерней сущности сопровождаются меткой (FKn) (рис. 5.26.).

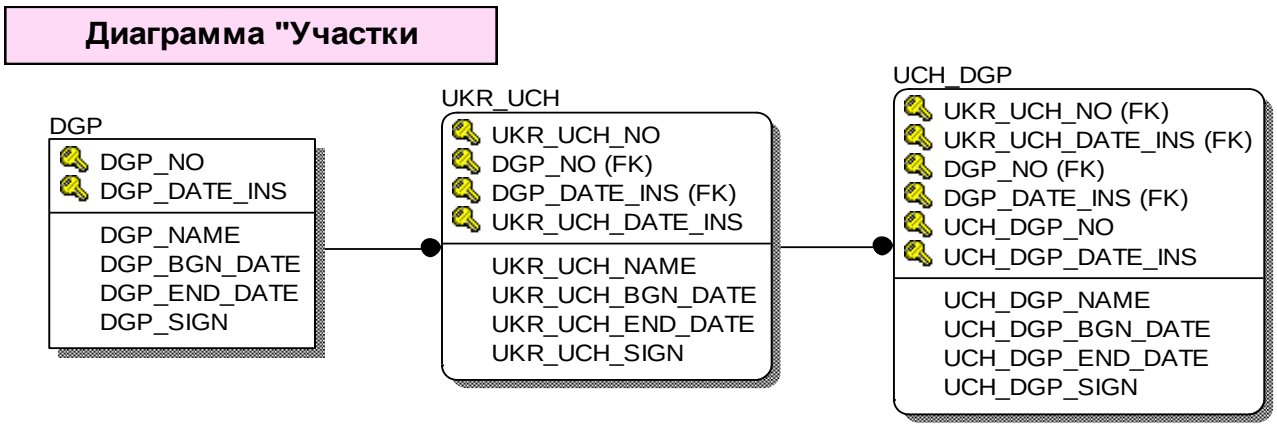


Рис. 5.26. Пример FAM диаграммы

Связь между двумя сущностями может быть определенная и неопределенная (идентифицирующая и не идентифицирующая). Идентифицирующие связи устанавливаются между сущности, если внешний ключ FKn мигрирует в ключевую область зависимой сущности. На диаграмме модели идентифицирующие связи между сущности изображаются сплошной линией, соединяющей эти сущности. Для таких типов связей, отношений СУБД самостоятельно контролирует ссылочную целостность между записями в родительской и дочерних таблицах на физическом уровне, например, могут выполняться каскадные операции удаления.

Не идентифицирующие связи (отношения) устанавливается, если FKn мигрирует в не ключевую область зависимой сущности. На диаграмме модели данных не идентифицирующие связи между сущностями изображаются прерывистой линией, соединяющей эти сущности. В обоих случаях на диаграмме помещается точка в конце соединительной линии у дочерней сущности (рис. 5.27.).

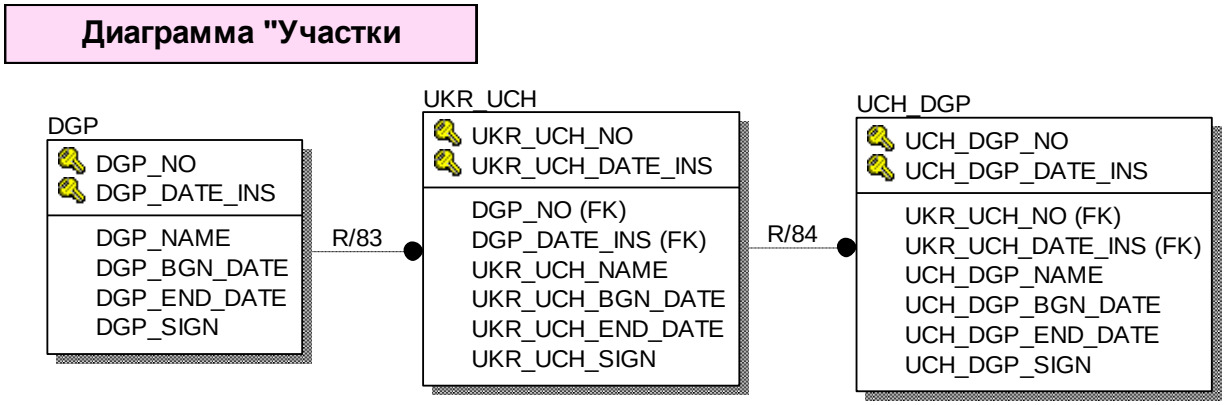


Рис. 5.27. Пример FAM диаграммы с не идентифицирующими связями

Для таких типов связей стратегия ссылочной целостность между записями в родительской и дочерних таблицах изменяется. Например, для задачи резервирования билетов (родительская сущность – «заказчик», дочерняя – «места»), при отказе «заказчика» от зарезервированных билетов выполняется операция удаления записи в родительской таблице (отказ от заказанных билетов), записи в дочерней таблице связанные с записью в родительской таблице могут и не удаляться, а их FK принимать значение по умолчанию (например, NULL, места то ведь остались).

Всем вышеописанным связям Erwin автоматически присваивает имена связи и порядковые номера связи.

После построения «Атрибутной модели» (FAM) выполняется нормализация данных.

5.8.2. Создание новой модели

Создание новой логической/физической модели, производится в Меню “File” выбором опции “New”. В появившемся окне необходимо выбрать из предложенного списка тип модели (логическая/физическая), указать используемую базу данных (DB2) и выбрать версию (рис. 5.28.). После нажатия кнопки “Ok” AllFusion Erwin Data Modeler открывает окна “Model Explorer” и “Diagram Window”.

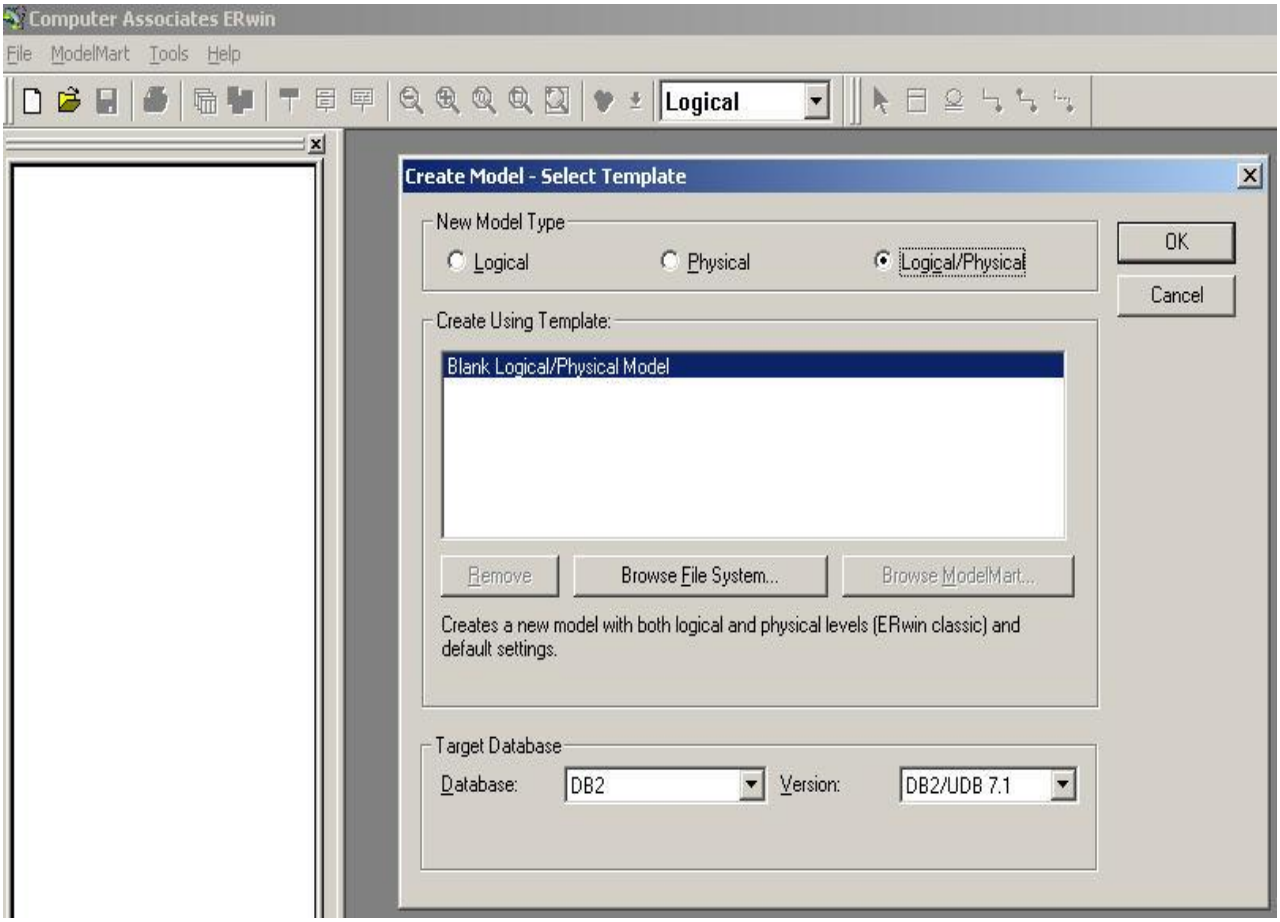


Рис. 5.28. Создание новой модели данных

5.8.3. Создание физического уровня базы данных на основе логического

Как уже было сказано выше, нужно с начало определить сущности и их характеристики на логическом уровне, затем при первоначальном переключении на физический уровень соответствующие термины для него будут созданы автоматически. Переключение между уровнями осуществляется с помощью списка выбора “Model type indicator” – «Указатель типа модели» в панели инструментов AllFusion Erwin Data Modeler (logical/physical) (рис. 5.21.).

5.8.4. Редактирование таблиц

Функция редактирования выполняется в графическом и текстовом режиме.

Выполнение функции “Создание/Добавление таблицы” в графическом режиме производится помещением курсора в поле “AllFusion Erwin Data Modeler Toolbox” (рис. 5.21.) и нажатием дважды на рисунок, изображающий таблицу “Entity”. Затем курсор перемещается в некоторое выбранное место для размещения таблицы в “Diagram Window” и делается щелчок мышкой. Таблица появляется с временным именем E/1. Такая операция называется перетягиванием и может повторяться для

нескольких таблиц. Поверх временного имени таблицы набирается ее действительное имя. Сразу после этого можно вводить имена атрибутов данной таблицы. Как указано выше, одновременно в окне “Model Explorer” в текстовом виде отражается появление созданной новой таблицы.

Выполнение этой функции можно производить и в текстовом режиме в окне “Model Explorer”: правый щелчок по “Entity”, во всплывающем меню выбрать “New” и щелкнуть по нему. В появившемся окне поверх имени E/1 набрать нужное имя таблицы. После появления таблицы в окне “Model Explorer” можно выполнять другие функции с таблицей схемы: переименование (Rename), удаление (Delete), переход к таблице на схеме (Go to), показать свойства (возможно для их корректировки или дополнения) таблицы (Property). Для этого необходимо сделать правый щелчок по выбранному “Entity”, во всплывающем меню выбрать требуемую функцию.

Выполнение функции “Переименование таблицы” легко выполняется в окне “Model Explorer”: правый щелчок по “Entity”, во всплывающем меню выбрать “Rename” и щелкнуть по нему. Поверх старого имени ввести новое имя таблицы.

Первоначальное определение объекта (Name, definition, note) производится в логической схеме, дальнейшее доопределение параметров объекта/таблицы при создании или добавлении таблицы можно производить в меню Property в окне “Model Explorer” по мере развития логической схемы и перехода ее к физической.

Параметры сущности для логической модели показаны на рис. 5.29. Параметры таблицы для физической модели показаны на рис. 5.30.

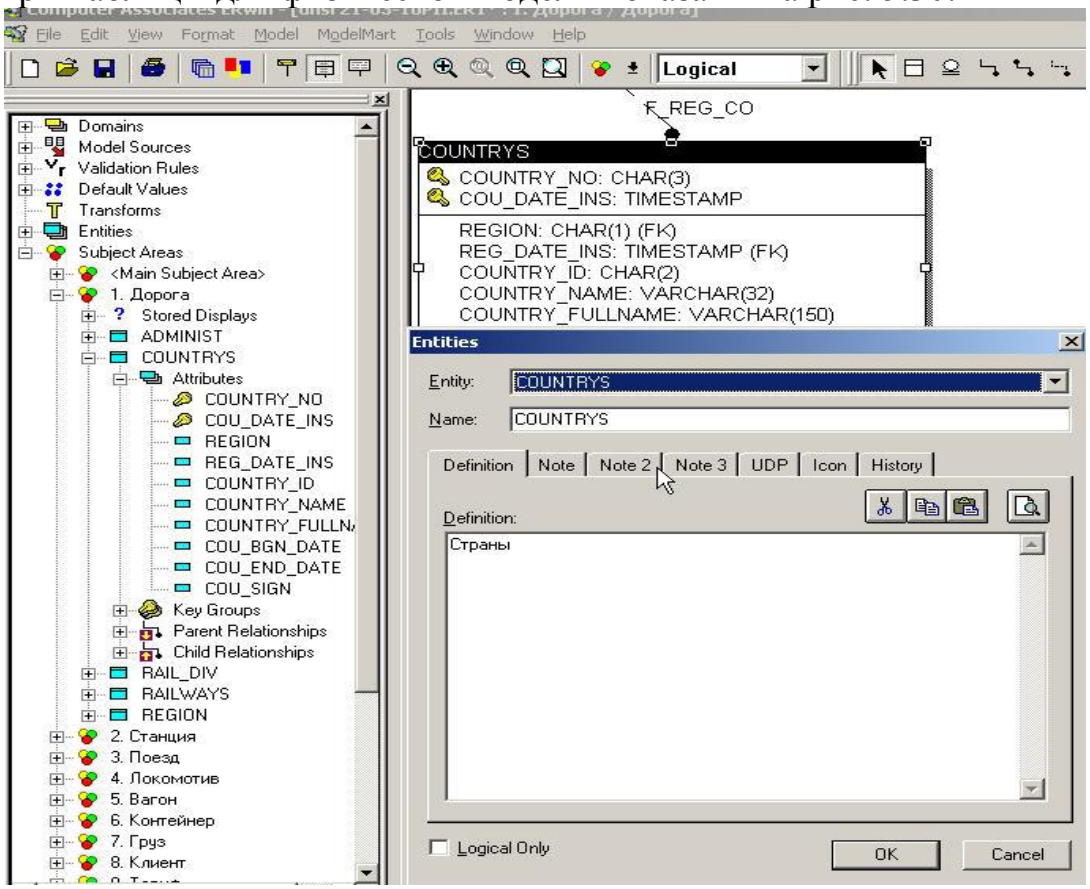


Рис. 5.29. Параметры сущности для логической модели

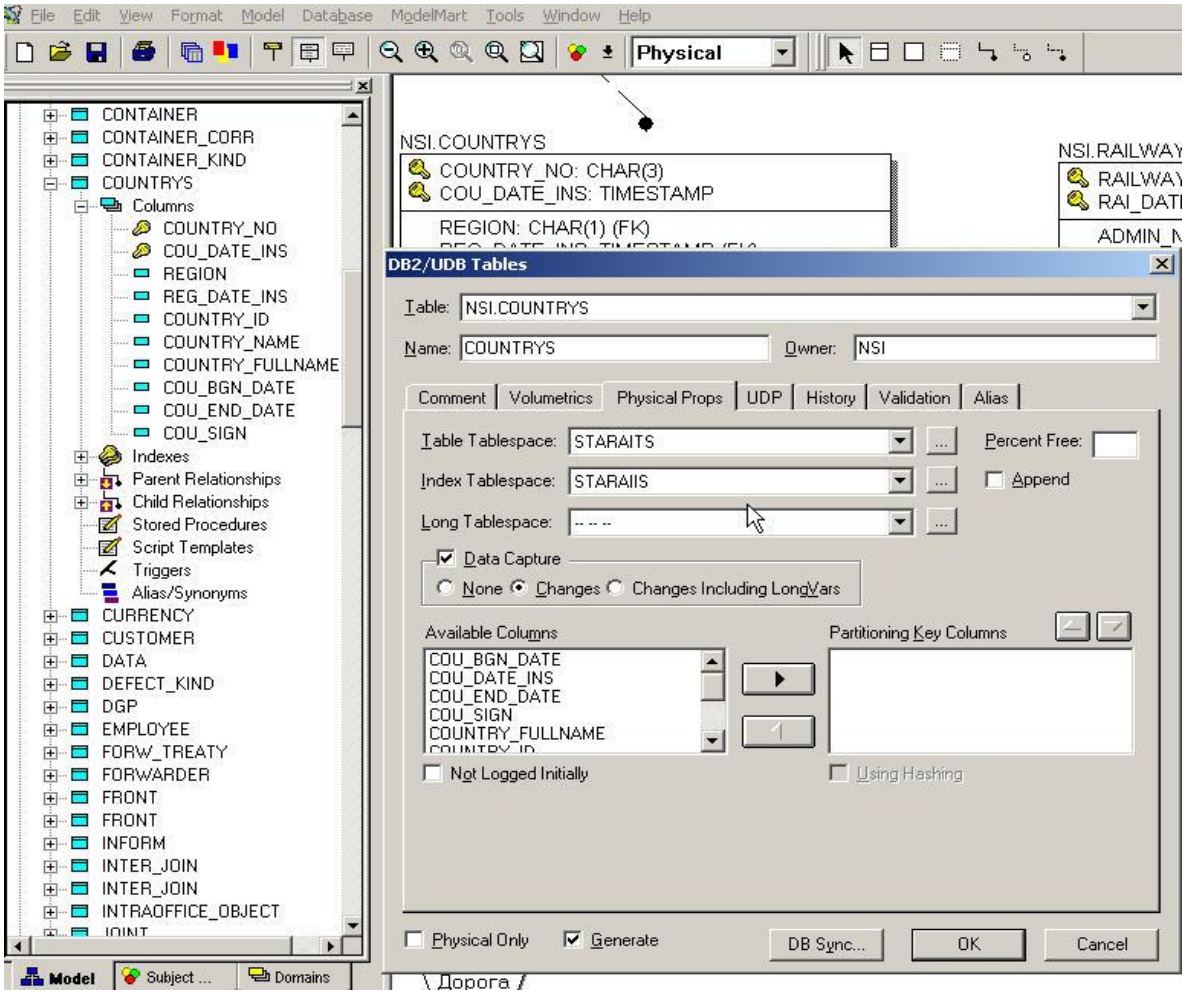


Рис. 5.30. Параметры таблицы для физической модели

5.8.5. Редактирование столбцов таблицы

Аналогично вышеизложенному производится выполнение функций по добавлению/переименованию/удалению атрибута таблицы. При создании таблицы и ее атрибутов в соответствии с требованиями реляционных баз данных необходимо обеспечить однозначную идентификацию каждой строки таблицы. Это достигается путем выбора одного или нескольких атрибутов таблицы в качестве первичного ключа (PK). Эти атрибуты образуют область ключа таблицы (Key area). Остальные неключевые атрибуты образуют область данных таблицы (Data area). Эти области разделяются в таблице в окне “Diagram Window” горизонтальной линией, поэтому при добавлении атрибутов во вновь создаваемую таблицу сначала выше горизонтальной линии вводятся ключевые атрибуты, а затем после нажатия клавиши “Tab” – неключевые.

AllFusion Erwin Data Modeler включает удобные средства перемещения атрибутов между вышеуказанными областями и внутри этих областей при корректировке модели.

Первоначальное определение атрибута (Name, General, Datatype, definition, note, Key Group) производится в логической схеме. Дальнейшее доопределение параметров атрибута/столбца таблицы при его добавлении можно производить в меню Property в окне “Model Explorer” по мере развития логической схемы и перехода ее к физической (рис. 5.31.).

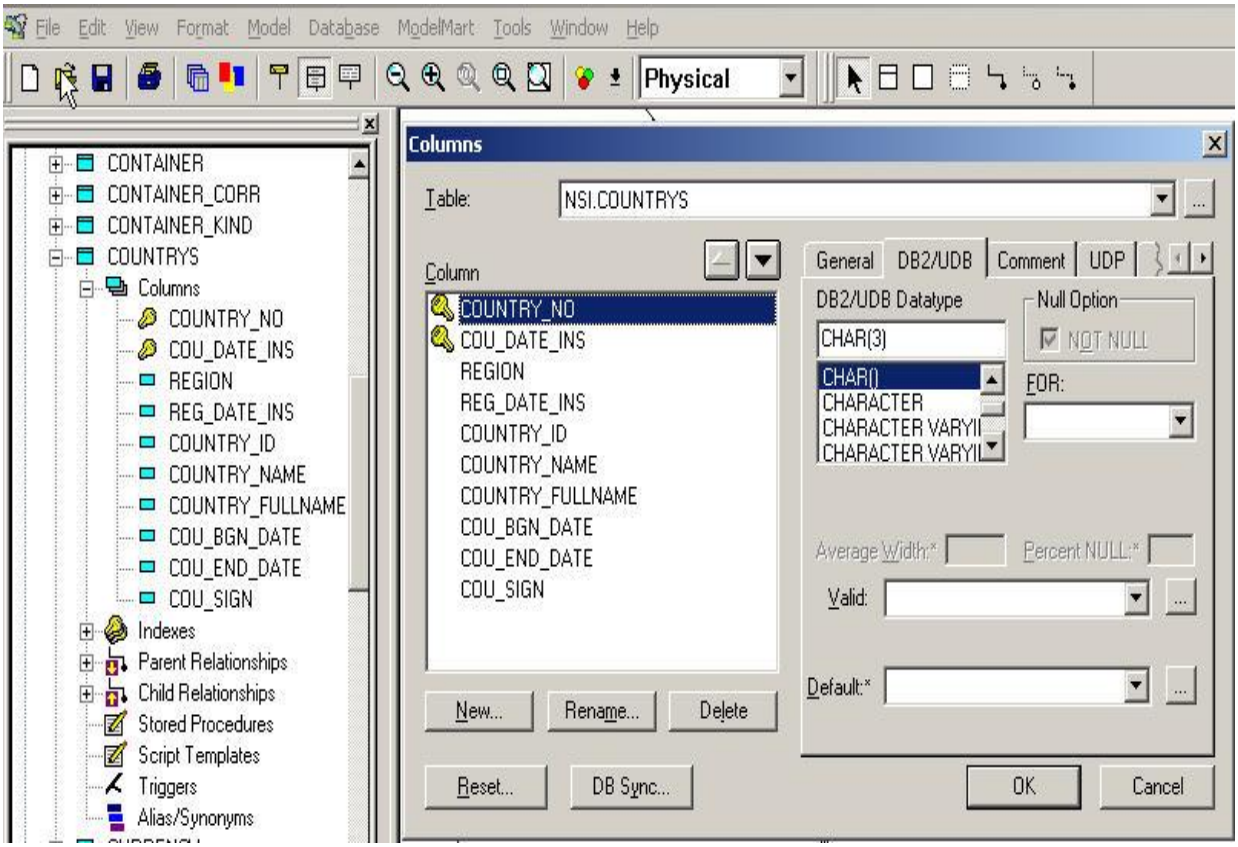


Рис. 5.31. Добавление/корректировка столбцов в физической модели данных

5.8.6. Редактирование ключей и индексов таблицы

При проектировании логической модели для некоторого объекта может создаваться “Ключевая группа”. Она автоматически включает в себя ключи PK, а также другие виды ключей – индексов. Первоначальное определение ключей (имя ключевой группы, тип индекса таблицы) производится в логической схеме. Детальное проектирование ключей и индексов (Index), как правило, производится на уровне физической модели.

Как уже излагалось выше, первичный ключ (PK) – это набор атрибутов, которые используются для однозначной идентификации экземпляров объекта. Первичный ключ может включать один или более первичных ключевых атрибутов. Ключ PK всегда единственный для таблицы и создается автоматически для каждой таблицы при вводе атрибутов в ключевую область.

Выполнения данной функции производится в меню “Key Group”/”Index” в окне “Model Explorer” выбором опции “Property”/”New” (рис. 5.32.).

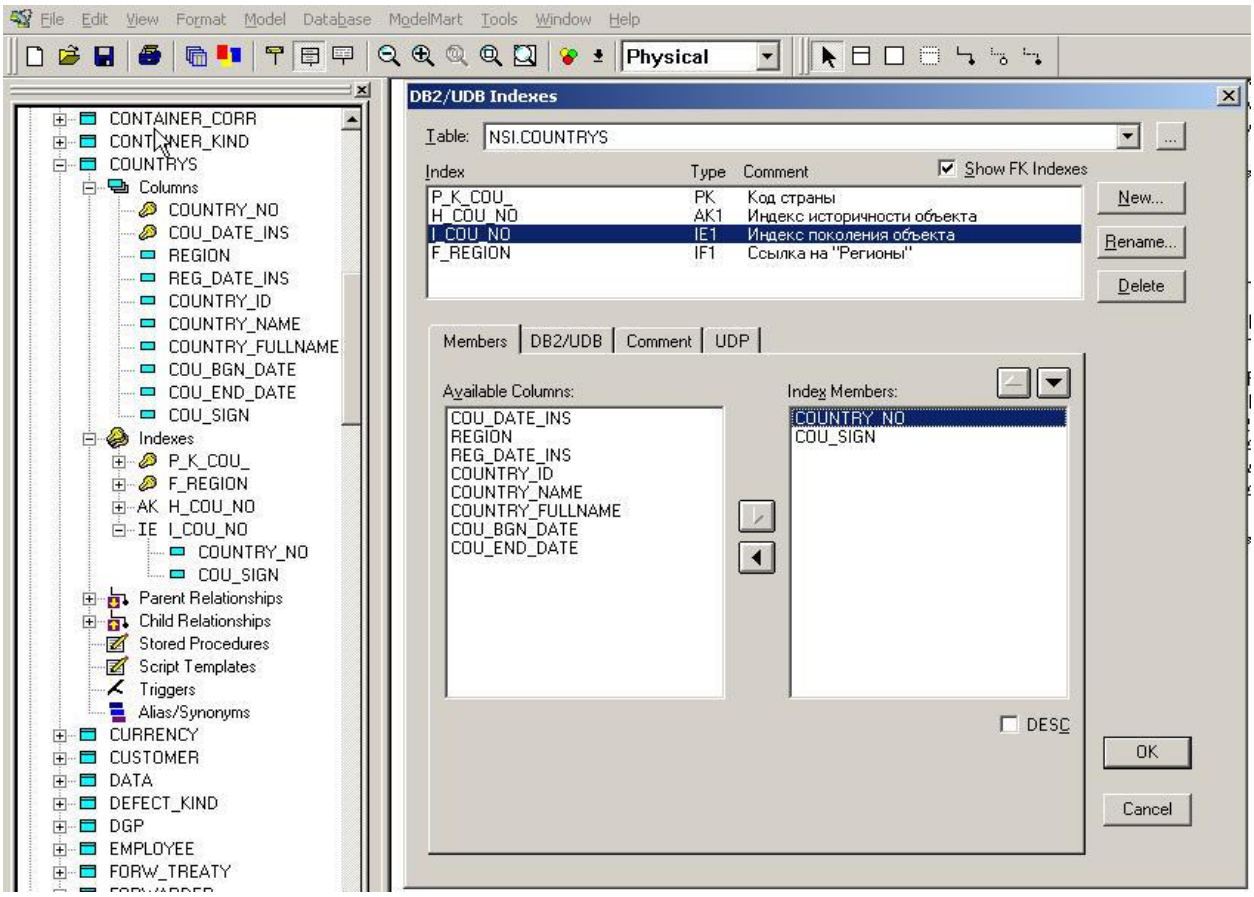


Рис. 5.32. Создание/корректировка индексов в физической модели данных

Название ключа можно переименовать, используя клавишу “Rename”, так же как и изменить любой из вышеуказанных параметров индекса на уровне логической или физической модели.

5.8.7. Редактирование связей таблиц

Модель БД имеет сложную древовидную (сетевую) структуру, в которой все сущности (таблицы) связаны между собой некоторыми связями или отношениями. Зависимая (дочерняя) сущность не может существовать без родительской и без идентификации этой зависимости. Это означает, что зависимая таблица не может быть идентифицирован без использования ключа родителя. Такая связь изображается сплошной или прерывистой линией между родительской и детской таблицами в окне “Diagram Window”. Каждая таблица может выступать в какой–то связи как родительская (“Parent relationships”) и, в то же время, в какой–то другой связи как дочерняя (“Child relationships”). В диаграмме сущностей схемы БД предусмотрено окно “Relationships” для регистрации всех параметров этой связи между сущностями (рис. 5.33.).

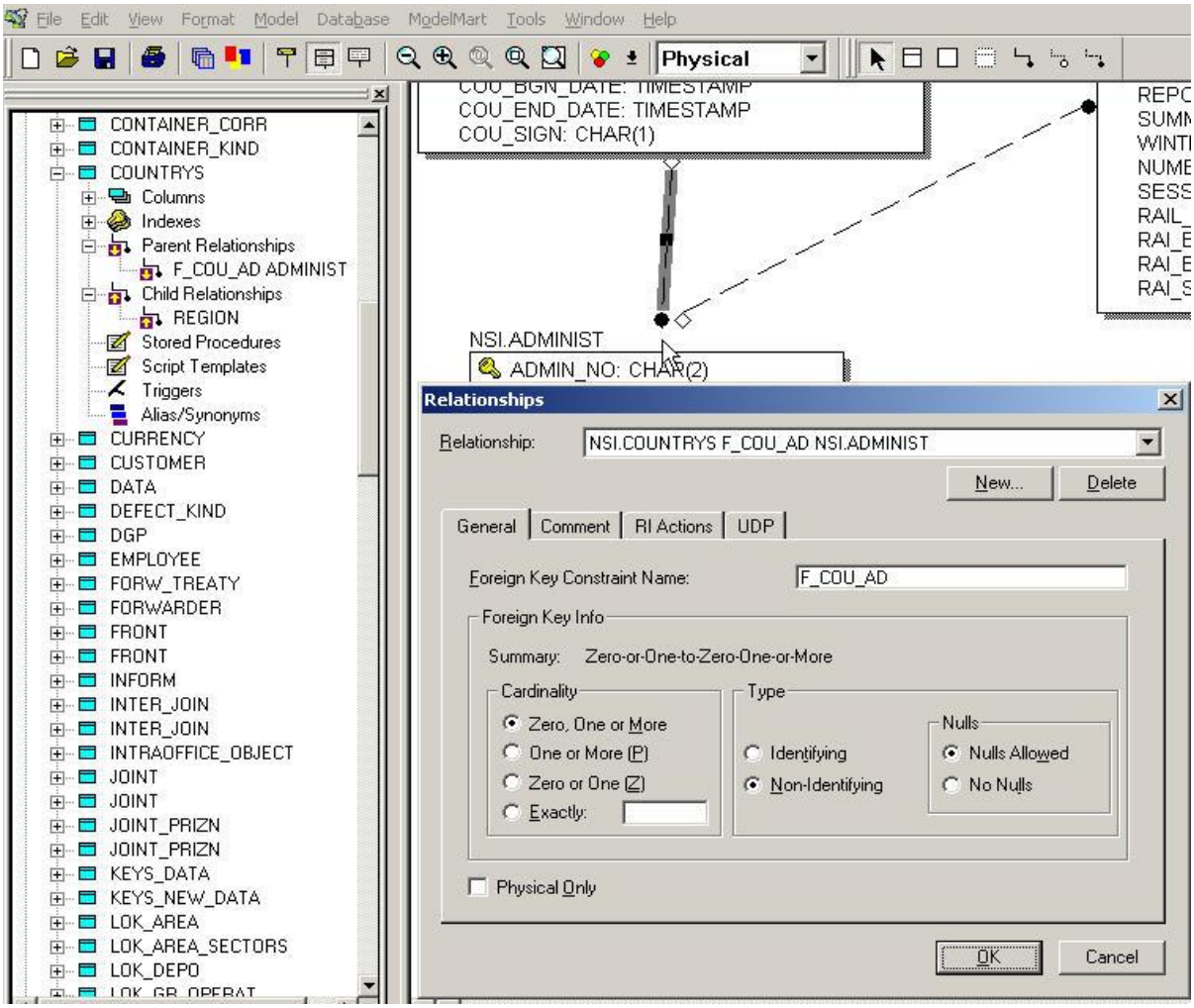


Рис. 5.33. Создание/корректировка связи в модели данных

Активизация этого окна производится в окне “Diagram Window” щелчком мышкой по выбранной линии связи или в окне “Model Explorer” выбором нужной таблицы, нажав для нее меню “Parent relationships” или “Child relationships”, опции “Property”/”New”.

5.8.8. Сохранение модели базы данных

Создание модели данных – это длительный, итерационный процесс, выполнение вышеизложенных функций и операций разработчиком БД может повторяться многократно для получения схемы модели данных, удовлетворяющей требованиям задач и пользователей на некотором этапе сопровождения БД.

При этом полученная модель всякий раз после окончания корректировки должна быть сохранена с помощью операций “Save” или “Save as” в виде AllFusion Erwin Data Modeler файла с расширением *.er1. Эта модель в последующем может быть выбрана для дальнейшей обработки с помощью операции “Open”. Все эти функции AllFusion Erwin Data Modeler достаточно подробно описаны в документе “AllFusion Erwin Data Modeler. Getting Started”.

AllFusion Erwin Data Modeler запоминает имена файлов для создаваемых моделей и их местоположение (пути доступа) и предлагает их для выбора при входе. Тем самым ускоряется процесс выборки требуемой модели и облегчается сопровождение БД.

5.8.9. Генерация операторов для создания базы данных

После окончательного определения и построения физической модели данных можно перейти к выполнению одной из наиболее важных функций – генерации схемы, а именно – генерации операторов на языке DDL, описывающих БД. Эти операторы используются затем для создания физической БД и ее поддержки на выбранном для ее размещения сервере.

AllFusion Erwin Data Modeler обеспечивает ведение следующих групп опций (режимов) для описания (указания) необходимых разработчику вариантов генерации схемы: “Schema”, “Summary table”, “View”, “Table”, “Column”, “Index”, “Referential Integrity”, “Trigger”, “Other options”. Включение параметров для указанных групп опций влияет на характеристики создаваемой физической БД, которые зависят от выбранной платформы СУБД. В связи с этим, не все группы опций используются одинаково для различных СУБД и, естественно, не все опции используются в данной разработке БД.

Генерация операторов на языке DDL производится при открытой модели БД, при этом необходимо выбрать тип “Physical” для модели. Перед генерацией необходимо выполнить выбор типа БД и ее версии для сервера назначения (рис. 5.34.). Для этого в меню “Database” выбрать “Choose Database”, в окне “Target Server” включить кнопки “DB2” и “DB2 version” – выбрать соответствующую версию DB2 для дальнейшей генерации операторов DDL. Следует заметить, что эту операцию необходимо выполнить только первоначально при генерации задания на языке DDL первый раз, либо при смене платформы СУБД или ее версии в дальнейшем, так как AllFusion Erwin Data Modeler запоминает предыдущее состояние введенных параметров и использует их при следующем входе.

Далее, для продолжения операции получения файла задания на языке DDL, необходимо перейти в меню “Tools”, подменю “Forward Engineer/Schema generation” – появляется окно, изображенное на рис. 5.35.

Для указанных в окне “Schema generation” режимов “Schema”, “View”, “Table”, “Columns”, “Index”, “Referential Integrity”, “Trigger”, “Other options” необходимо справа в окне “Schema” указать необходимость включения соответствующих последовательностей SQL операторов в файл задания на языке DDL. Это операторы Create, Drop и другие, которые будут сгенерированы AllFusion Erwin Data Modeler автоматически для схемы в целом, для табличных пространств и буферных пулов, таблиц, индексов, триггеров и других элементов схемы. Полностью состояние включения элементов в операторы на языке DDL можно посмотреть, щелкнув по кнопке “Summary”.

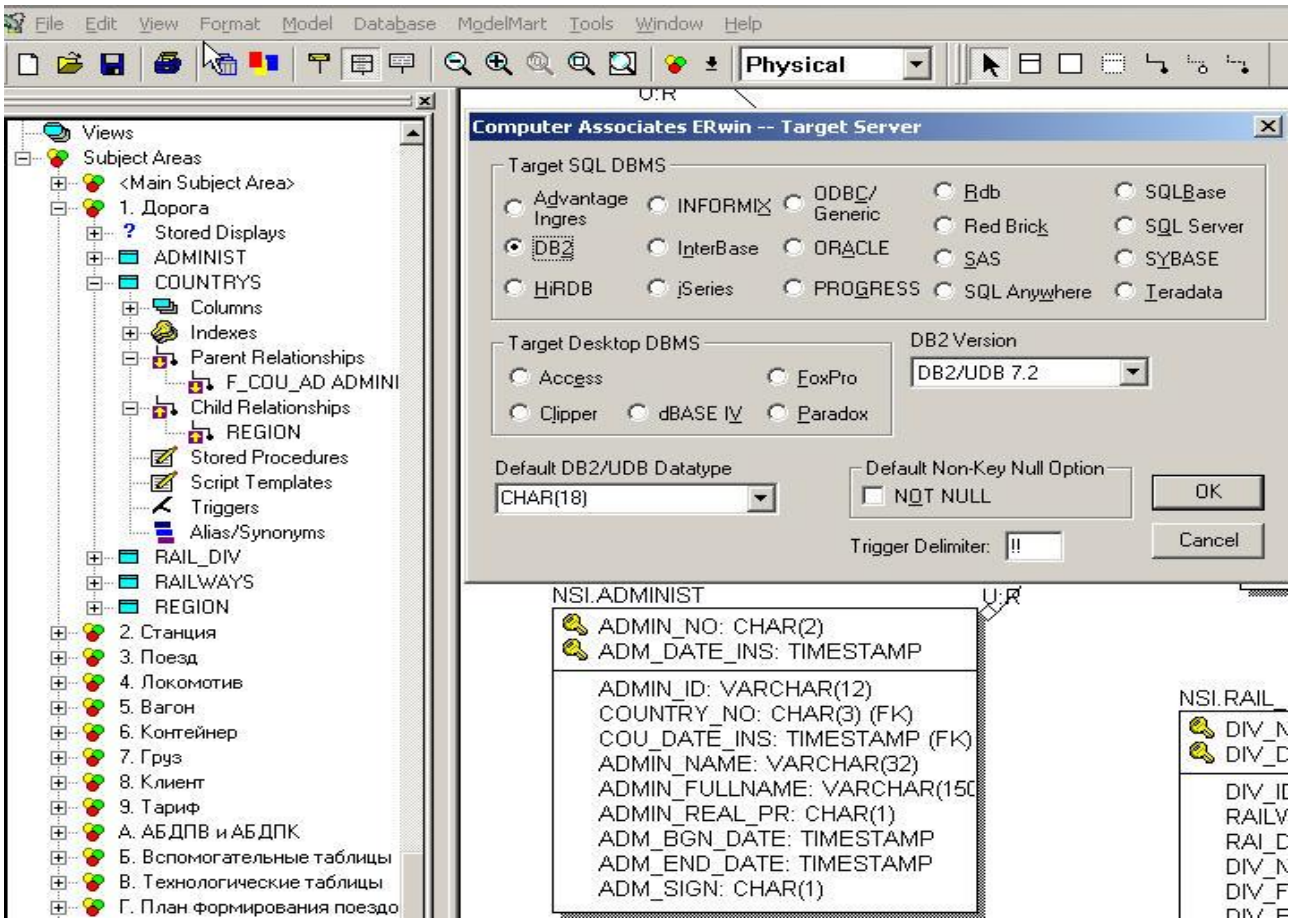


Рис. 5.34. Выбор БД и ее версии

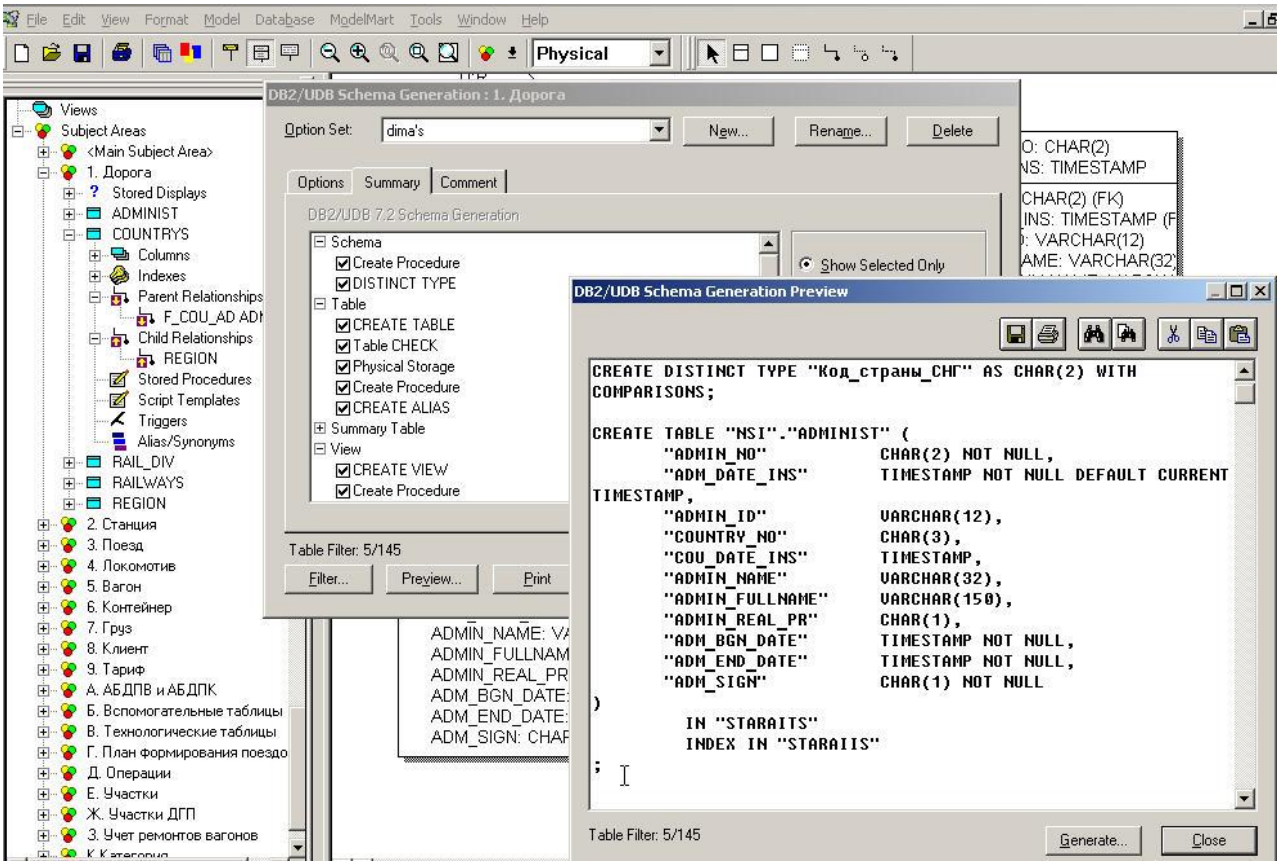


Рис. 5.35. Генерация файла с операторами DDL для БД

Для завершения операции – генерация файла скриптов и получения операторов DDL в виде файла – задания необходимо нажать кнопку “Report” в нижней части экрана. Появляется окно, в котором разработчик должен выбрать папку и указать имя файла для сохранения сгенерированных операторов DDL в виде задания (сеанса в терминах DB2) для создания БД на сервере.

Сгенерированные операторы можно просмотреть, нажав кнопку “Preview” в нижней части экрана. При этом используются сохраненные на этапе генерации значения параметров для переключения режимов генерации операторов DDL схемы (см. часть примера ниже).

```
CREATE TABLE "NSI"."ADMINIST" (
    "ADMIN_NO"          CHAR(2) NOT NULL,
    "ADM_DATE_INS"      IMESTAMP NOT NULL DEFAULT CURRENT
TIMESTAMP,
    "ADMIN_ID"          VARCHAR(12),
    "COUNTRY_NO"        CHAR(3),
    "COU_DATE_INS"      TIMESTAMP,
    "ADMIN_NAME"        VARCHAR(32),
    "ADMIN_FULLNAME"    VARCHAR(150),
    "ADMIN_REAL_PR"     CHAR(1),
    "ADM_BGN_DATE"      TIMESTAMP NOT NULL,
    "ADM_END_DATE"      TIMESTAMP NOT NULL,
```

```
    "ADM_SIGN"          CHAR(1) NOT NULL
)
  IN "STARAITS"
  INDEX IN "STARAIIS"
;
ALTER TABLE "NSI"."ADMINIST"
  DATA CAPTURE CHANGES
 ;
COMMENT ON TABLE "NSI"."ADMINIST" IS 'Администрация';
COMMENT ON COLUMN "NSI"."ADMINIST"."ADMIN_NO" IS 'Код администрации';
CREATE UNIQUE INDEX "NSI"."H_ADM_NO" ON "NSI"."ADMINIST"
(
    "ADMIN_NO"              ASC,
    "ADM_BGN_DATE"          ASC,
    "ADM_END_DATE"          ASC
);
COMMENT ON INDEX "NSI"."H_ADM_NO" IS 'Индекс историчности объекта';
CREATE INDEX "NSI"."F_COUADM" ON "NSI"."ADMINIST"
(
    "COUNTRY_NO"            ASC,
    "COU_DATE_INS"          ASC
);
```

5.8.10. Подготовка исходных данных для разработки новой версии БД

Исходные данные для получения новой версии БД определяются требованиями появляющихся новых функций и модификации задач по мере их развития. Назначение этапа – проверка и обеспечение достоверности схемы модели БД, проверка правильности исходных данных для корректировки и соответствия их схеме модели БД.

Исходные данные для внесения изменений и получения новой версии БД включают:

– схему эксплуатируемой предыдущей версии БД в среде AllFusion Erwin Data Modeler (по возможности) или структура таблиц данных конкретной БД (при первоначальном формировании схемы);

– совокупность отчетов по старой схеме БД в среде AllFusion Erwin Data Modeler;

– эксплуатируемую физическую БД;

– исходные данные для изменений по форме некоторого установленного документа пользователя;

– источник ввода данных и порядок их заполнения, порядок установления связей с другими таблицами (для вновь включаемых таблиц, столбцов, связей должен быть определен).

Формы документов, предназначенных для документального оформления требований по внесению изменений в эксплуатирующуюся БД, получению исходных данных для заполнения БД и выпуску новой ее версии, регламентируется внутренними распоряжениями или документами пользователя. Рекомендуется по возможности приближать вид этих документов к отчетам AllFusion Erwin Data Modeler для схем моделей БД.

На данном этапе можно выполнить определенные подготовительные работы, которые позволят в дальнейшем сократить время и улучшить качество работ по созданию и запуску новой версии БД. Необходимость их выполнения определяется разработчиком БД в зависимости от того, есть ли достоверная схема БД на данный момент, насколько она соответствует физической БД, насколько большой список изменений в БД и насколько тщательно подготовлены исходные данные для корректировки. Предлагается выполнить следующий перечень операций (рис. 5.36.):

– запустить CASE систему AllFusion Erwin Data Modeler;

– загрузить существующую модель старой БД в AllFusion Erwin Data Modeler или получить ее средствами AllFusion Erwin Data Modeler, операция “Reverse engineering”;

– получить доступ к эксплуатируемой БД;

– произвести полное сравнение существующей модели БД с физической БД средствами AllFusion Erwin Data Modeler для контроля их соответствия;

– при наличии отклонений произвести выяснение причин различий и привести в соответствии схемы модели БД структуре физической БД;

– получить полную логическую и физическую схемы БД со всеми ее подсхемами, таблицами, столбцами и их характеристиками;

– произвести анализ исходных данных для внесения изменений и определения оптимального варианта их включения в новую версию БД;

– на основе анализа получить более полный, исправленный и детализированный перечень вносимых изменений в БД по структуре отчетов AllFusion Erwin Data Modeler для БД (вставка, замена, удаление элементов схемы).

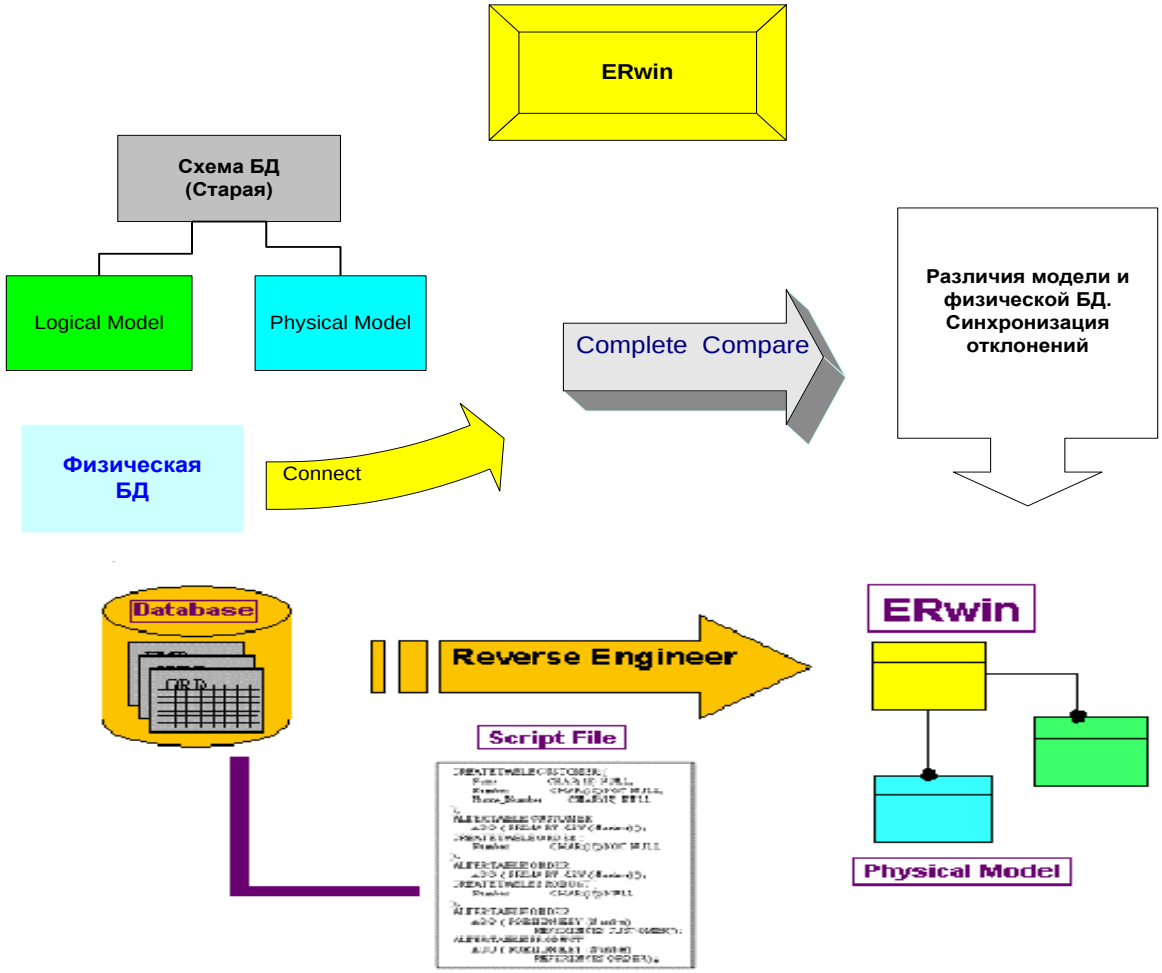


Рис. 5.36. Схема технологического процесса этапа подготовки исходных данных для новой версии БД

6. Язык UML, модели ПО, объектно–ориентированный анализ и проектирование ПО

Язык UML представляет собой общецелевой язык визуального моделирования, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения, бизнес–процессов и других систем. Язык UML является достаточно строгим и мощным средством моделирования, который может быть эффективно использован для построения концептуальных, логических и графических моделей сложных систем различного целевого назначения. Этот язык вобрал в себя наилучшие качества и опыт методов программной инженерии, которые с успехом использовались на протяжении последних лет при моделировании больших и сложных систем.

С точки зрения методологии ООАП достаточно полная модель сложной системы представляет собой определенное число взаимосвязанных представлений (views), каждое из которых адекватно отражает аспект поведения или структуры системы. При этом наиболее общими представлениями сложной системы принято считать статическое и динамическое, которые в свою очередь могут подразделяться на другие более частные.

Принцип иерархического построения моделей сложных систем предписывает рассматривать процесс построения моделей на разных уровнях абстрагирования или детализации в рамках фиксированных представлений.

Уровень представления (layer) — способ организации и рассмотрения модели на одном уровне абстракции, который представляет горизонтальный срез архитектуры модели, в то время как разбиение представляет ее вертикальный срез.

При этом исходная или первоначальная модель сложной системы имеет наиболее общее представление и относится к концептуальному уровню. Такая модель, получившая название концептуальной, строится на начальном этапе проектирования и может не содержать многих деталей и аспектов моделируемой системы. Последующие модели конкретизируют концептуальную модель, дополняя ее представлениями логического и физического уровня.

В целом же процесс ООАП можно рассматривать как последовательный переход от разработки наиболее общих моделей и представлений концептуального уровня к более частным и детальным представлениям логического и физического уровня. При этом на каждом этапе ООАП данные модели последовательно дополняются все большим количеством деталей, что позволяет им более адекватно отражать различные аспекты конкретной реализации сложной системы. Общая схема взаимосвязей моделей ООАП представлена на рис. 6.1.

Рис. 6.1. Взаимосвязь моделей ООАП

Для описания языка UML используются средства самого языка. К базовым средствам относится пакет, который служит для группировки элементов модели. При этом сами элементы модели, в том числе произвольные сущности, отнесенные к одному пакету, выступают в роли единого целого. При этом все разновидности элементов графической нотации языка UML организованы в пакеты.

Унифицированный язык моделирования (UML) является стандартным инструментом для моделирования программного обеспечения. С помощью UML можно визуализировать, специфицировать, конструировать и документировать артефакты программных систем.

UML можно применять для моделирования любых систем: от информационных систем масштаба предприятия до распределенных Web–приложений и даже встроенных систем реального времени. Это очень выразительный язык, позволяющий рассмотреть систему со всех точек зрения, имеющих отношение к ее разработке и последующему развертыванию. Несмотря на обилие выразительных возможностей, этот язык прост для понимания и использования.

Однако, UML – это всего лишь язык моделирования; он является одной из составляющих процесса разработки программного обеспечения. Хотя UML не зависит от моделируемой реальности, лучше всего применять его, когда процесс моделирования основан на рассмотрении вариантов использования, является итеративным и пошаговым, а сама система имеет четко выраженную архитектуру.

UML не является языком визуального программирования, но модели, созданные с его помощью, могут быть непосредственно переведены на различные

языки программирования. Иными словами, UML–модель можно отобразить на такие языки, как Java, C++, Visual Basic и даже на таблицы реляционной базы данных. Те понятия, которые предпочтительно передавать графически, так и представляются в UML; те же, которые лучше описывать в текстовом виде, выражаются с помощью языка программирования.

Такое отображение модели на язык программирования позволяет осуществлять прямое проектирование: генерацию кода из модели UML в какой–то конкретный язык. Можно решить и обратную задачу: реконструировать модель по имеющейся реализации.

Кроме того, UML позволяет решить проблему документирования системной архитектуры и всех ее деталей, предлагает язык для формулирования требований к системе и определения тестов и, наконец, предоставляет средства для моделирования работ на этапе планирования проекта и управления версиями.

6.1. Основные элементы языка UML

Словарь языка UML включает три вида строительных блоков:
– сущности (предметы);
– отношения;
– диаграммы.

6.1.1. Сущности

Сущности – это абстракции, являющиеся основными элементами модели. Отношения связывают различные сущности. Диаграмма – это графическое представление множества сущностей. Изображается она, чаще всего, как связный граф из вершин (сущностей) и дуг (отношений).

В UML имеется четыре типа сущностей:
– структурные;
– поведенческие;
– группирующие;
– аннотационные.

Сущности являются основными объектно–ориентированными блоками языка. С их помощью можно создавать корректные модели.

Структурные сущности – представляют собой статические части модели, соответствующие концептуальным или физическим элементам системы. Существует семь разновидностей структурных сущностей.

Класс (Class) – это описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Класс реализует один или несколько интерфейсов. Графически класс изображается в виде прямоугольника, в котором обычно записаны его имя, атрибуты и операции, как показано на рис. 6.2.

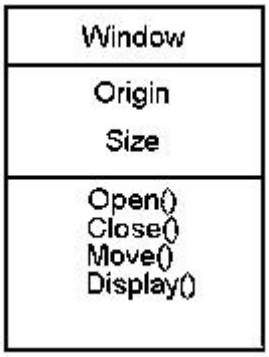


Рис. 6.2. Классы

Интерфейс (Interface) – это совокупность операций, которые определяют сервис (набор услуг), предоставляемый классом или компонентом. Таким образом, интерфейс описывает видимое извне поведение элемента. Интерфейс может представлять поведение класса или компонента полностью или частично; он определяет только спецификации операций (сигнатуры), но никогда – их реализации. Графически интерфейс изображается в виде круга, под которым пишется его имя, как показано на рис. 6.3. Интерфейс редко существует сам по себе – обычно он присоединяется к реализующему его классу или компоненту.

Рис. 6.3. Интерфейсы

Кооперация (Collaboration) определяет взаимодействие; она представляет собой совокупность ролей и других элементов, которые, работая совместно, производят некоторый кооперативный эффект, не сводящийся к простой сумме слагаемых. Кооперация, следовательно, имеет как структурный, так и поведенческий аспект. Один и тот же класс может принимать участие в нескольких кооперациях; таким образом, они являются реализацией образцов поведения, формирующих систему. Графически кооперация изображается в виде эллипса, изображенного пунктирной линией, в который обычно заключено только имя, как показано на рис. 6.4.

Рис. 6.4. Кооперативные диаграммы

Вариант использования (Use case) – это описание последовательности выполняемых системой действий, которая производит наблюдаемый результат,

значимый для какого–то определенного актера (Actor). Варианты использования реализуются посредством кооперативных диаграмм. Графически вариант использования изображается в виде изображенного непрерывной линией эллипса, обычно содержащего только его имя, как показано на рис. 6.5.

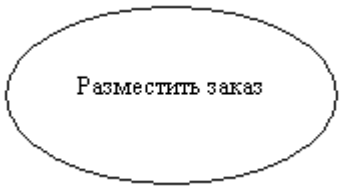


Рис. 6.5. Варианты использования

Три другие сущности – активные классы, компоненты и узлы – подобны классам: они описывают совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Тем не менее, они в достаточной степени отличаются друг от друга и от классов и, учитывая их важность при моделировании определенных аспектов объектно–ориентированных систем, заслуживают специального рассмотрения.

Активным классом (Active class) называется класс, объекты которого вовлечены в один или несколько процессов, или нитей (Threads), и поэтому могут инициировать управляющее воздействие. Активный класс во всем подобен обычному классу, за исключением того, что его объекты представляют собой элементы, деятельность которых осуществляется одновременно с деятельностью других элементов. Графически активный класс изображается так же, как простой класс, но ограничивающий прямоугольник рисуется жирной линией и обычно включает имя, атрибуты и операции.

Два оставшихся элемента – компоненты и узлы – также имеют свои особенности. Они соответствуют физическим сущностям системы, в то время как пять предыдущих – концептуальным и логическим сущностям.

Компонент (Component) – это физическая заменяемая часть системы, которая соответствует некоторому набору интерфейсов и обеспечивает его реализацию. В системе можно встретить различные виды устанавливаемых компонентов, такие как COM+ или Java Beans, а также компоненты, являющиеся артефактами процесса разработки, например файлы исходного кода. Компонент, как правило, представляет собой физическую упаковку логических элементов, таких как классы, интерфейсы и кооперативные диаграммы. Графически компонент изображается в виде прямоугольника с вкладками, содержащего обычно только имя, как показано на рис. 6.6.

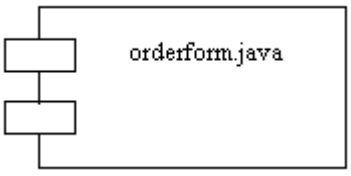


Рис. 6.6. Компоненты

Узел (Node) – это элемент реальной (физической) системы, который существует во время функционирования программного комплекса и представляет собой вычислительный ресурс, обычно обладающий как минимум некоторым объемом памяти, а часто еще и способностью обработки. Совокупность компонентов может размещаться в узле, а также мигрировать с одного узла на другой. Графически узел изображается в виде куба, обычно содержащего только имя, как показано на рис. 6.7.

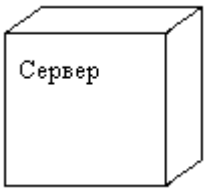


Рис. 6.7. Узлы

Существуют также разновидности этих сущностей: актеры, сигналы, утилиты (виды классов), процессы и нити (виды активных классов), приложения, документы, файлы, библиотеки, страницы и таблицы (виды компонентов).

Поведенческие сущности (Behavioral things) являются динамическими составляющими модели UML. Они описывают поведение модели во времени и пространстве. Существует всего два основных типа поведенческих сущностей.

Взаимодействие (Interaction) – это поведение, суть которого заключается в обмене сообщениями (Messages) между объектами в рамках конкретного контекста для достижения определенной цели. С помощью взаимодействия южно описать как отдельную операцию, так и поведение совокупности объектов. Взаимодействие предполагает ряд других элементов, таких как сообщения, последовательности действий (поведение, инициированное сообщением) и связи (между объектами). Графически сообщения изображаются в виде стрелки, над которой почти всегда пишется имя соответствующей операции (рис. 6.8.).

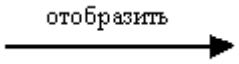


Рис. 6.8. Сообщения

Автомат (State machine) – это алгоритм поведения, определяющий последовательность состояний, через которые объект или взаимодействие проходят на протяжении своего жизненного цикла в ответ на различные события, а также реакции на эти события. С помощью автомата можно описать поведение отдельного класса или кооперативной диаграммы классов. С автоматом связан ряд других элементов: состояния, переходы (из одного состояния в другое), события (сущности, инициирующие переходы) и виды действий (реакция на переход). Графически состояние изображается в виде прямоугольника с закругленными углами, содержащего имя и, возможно, подсостояния (рис. 6.9.).

Эти два элемента – взаимодействия и автоматы – являются основными поведенческими сущностями, входящими в модель UML. Семантически они часто бывают связаны с различными структурными элементами, в первую очередь – классами, кооперациями и объектами.

Рис. 6.9. Состояния

Группирующие сущности являются организующими частями модели UML. Это блоки, на которые можно разложить модель. Есть только одна первичная группирующая сущность, а именно пакет.

Пакеты (Packages) представляют собой универсальный механизм организации элементов в группы. В пакет можно поместить структурные, поведенческие и даже другие группирующие сущности. В отличие от компонентов, существующих во время работы программы, пакеты носят чисто концептуальный характер, то есть существуют только во время разработки. Изображается пакет в виде папки с закладкой, содержащей, как правило, только имя и иногда – содержимое (рис. 6.10.).

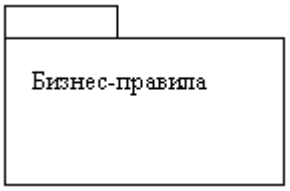


Рис. 6.10. Пакеты

Пакет – основной способ организации элементов модели в языке UML. Каждый пакет владеет всеми своими элементами, т. е. теми элементами, которые включены в него. Про соответствующие элементы пакета говорят, что они принадлежат пакету или входят в него. При этом каждый элемент может принадлежать только одному пакету. В свою очередь, одни пакеты могут быть вложены в другие.

Подпакет (Subpackage) — пакет, который является составной частью другого пакета.

По определению все элементы подпакета принадлежат и более общему пакету. Тем самым для элементов модели задается отношение вложенности пакетов, которое представляет собой иерархию.

Для графического изображения пакетов на диаграммах применяется большой прямоугольник с небольшим прямоугольником, присоединенным к левой части верхней стороны первого. Можно сказать, что визуально символ пакета напоминает пиктограмму папки в популярном графическом интерфейсе. Внутри большого прямоугольника может записываться информация, относящаяся к

данному пакету. Если такой информации нет, то внутри большого прямоугольника записывается имя пакета, которое должно быть уникальным в пределах рассматриваемой модели. Если же такая информация имеется, то имя пакета записывается в верхнем маленьком прямоугольнике (рис. 6.11.).

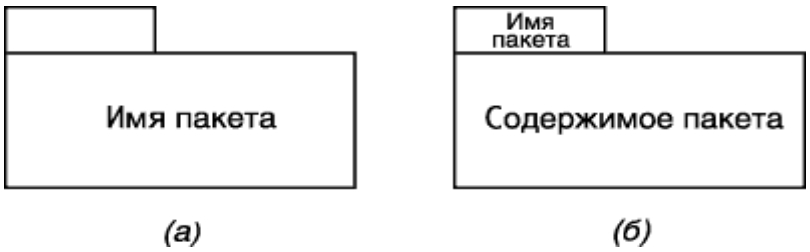


Рис. 6.11. Графическое изображение пакетов в языке UML

Перед именем пакета может помещаться строка текста, содержащая ключевое слово, заранее определенное в языке UML, и называемое стереотипом.

В языке UML определены следующие **стереотипы сообщений**:

<<call>> (вызвать) – сообщение, требующее вызова операции или процедуры объекта–получателя. Если сообщение с этим стереотипом рефлексивное, то оно инициирует локальный вызов операции у пославшего это сообщение объекта.

<<return>> (возвратить) – сообщение, возвращающее значение выполненной операции или процедуры вызвавшему ее объекту. Значение результата может инициировать ветвление потока управления.

<<create>> (создать) – сообщение, требующее создания другого объекта для выполнения определенных действий. Созданный объект может стать активным (ему передается поток управления), а может остаться пассивным.

<<destroy>> (уничтожить) – сообщение с явным требованием уничтожить соответствующий объект. Посылается в том случае, когда необходимо прекратить нежелательные действия со стороны существующего в системе объекта, либо когда объект больше не нужен и должен освободить задействованные им системные ресурсы.

<<send>> (послать) – обозначает посылку другому объекту сигнала, который асинхронно инициируется одним объектом и принимается (перехватывается) другим. Отличие сигнала от сообщения заключается в том, что сигнал должен быть явно описан в том классе, объект которого инициирует его передачу.

В качестве содержимого пакета могут выступать имена его отдельных элементов и их свойства, такие как видимость элементов за пределами пакета.

Одним из типов отношений между пакетами является отношение вложенности или включения пакетов друг в друга. В языке UML это отношение может быть изображено без использования линий простым размещением одного пакета–прямоугольника внутри другого пакета–прямоугольника. Так, в данном случае пакет с именем Пакет_1 содержит в себе два подпакета: Пакет_2 и Пакет_3 (рис. 6.12.).

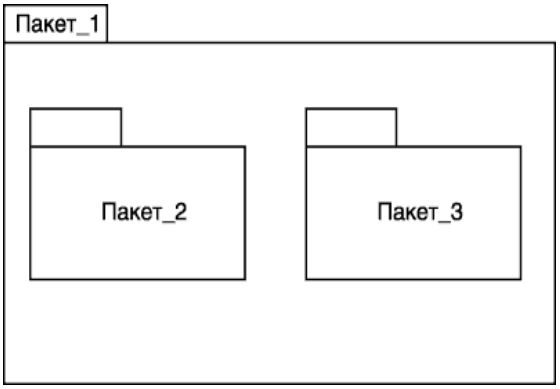


Рис. 6.12. Графическое изображение вложенности пакетов друг в друга

Кроме того, в языке UML это же отношение может быть изображено с помощью отрезков линий аналогично графическому представлению дерева. В этом случае наиболее общий пакет или контейнер изображается в верхней части рисунка, а его подпакеты – уровнем ниже. Контейнер соединяется с подпакетами сплошной линией, на конце которой, примыкающей к контейнеру, изображается специальный символ – ⊕. Он означает, что подпакеты "собственность" или часть контейнера, и, кроме этих подпакетов, контейнер не содержит никаких других. Рассмотренный выше пример может быть представлен с помощью явной визуализации отношения включения (рис. 6.13.).

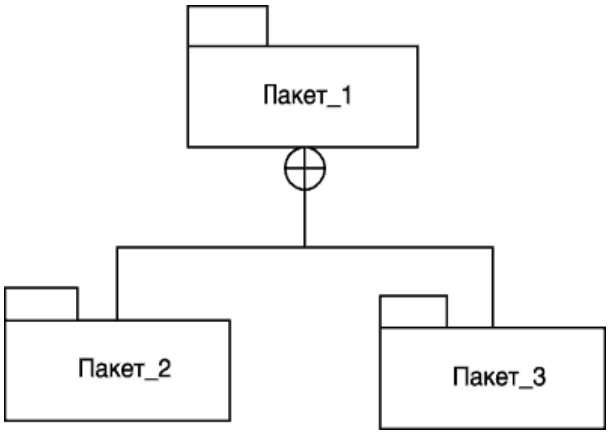


Рис. 6.13. Графическое изображение языка UML для вложенности пакетов друг в друга с помощью явной визуализации отношения включения

Пакеты – это основные группирующие сущности, с помощью которых можно организовать модель UML. Существуют также вариации пакетов, например каркасы (Frameworks), модели и подсистемы.

Модель является подклассом пакета и представляет собой абстракцию физической системы, которая предназначена для вполне определенной цели. Именно эта цель предопределяет те компоненты, которые должны быть включены в модель и те, рассмотрение которых не является обязательным. Другими словами, модель отражает релевантные аспекты физической системы, оказывающие

непосредственное влияние на достижение поставленной цели. В прикладных задачах цель обычно задается в форме исходных требований к системе, которые, в свою очередь, в языке UML записываются в виде вариантов использования системы.

В языке UML для одной и той же физической системы могут быть определены различные модели, каждая из которых специфицирует систему с различных точек зрения. Примерами таких моделей являются логическая модель, модель проектирования, модель вариантов использования и другие. При этом каждая такая модель имеет собственную точку зрения на физическую систему и свой уровень абстракции. Модели, как и пакеты, могут быть вложены друг в друга. Пакет может включать в себя несколько различных моделей одной и той же системы, и в этом состоит один из важнейших механизмов разработки моделей на языке UML. Общая модель системы в контексте языка UML содержит в себе модель анализа и модель проектирования, что явно отражает связь с ООАП (рис. 6.14.).

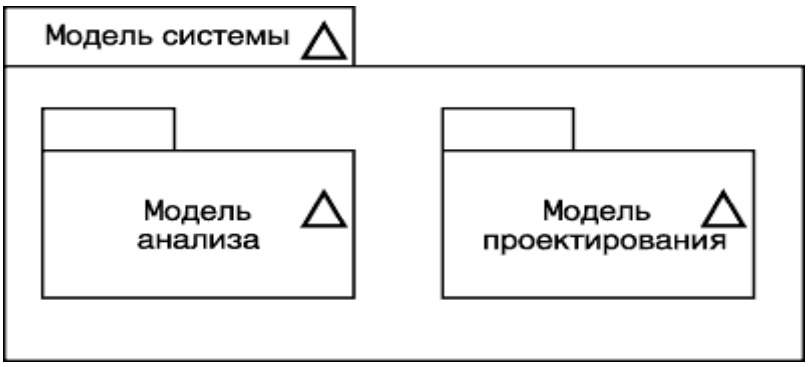


Рис. 6.14. Изображение модели системы в виде пакетов моделей анализа и проектирования

Подсистема есть просто группировка элементов модели, которые специфицируют простейшее поведение физической системы. При этом элементы подсистемы делятся на две части – спецификацию поведения и его реализацию. Для графического представления подсистемы применяется специальное обозначение – прямоугольник, как в случае пакета, но дополнительно разделенный на три секции. При этом в верхнем маленьком прямоугольнике изображается символ, по своей форме напоминающий "вилку" и указывающий на подсистему. Имя подсистемы вместе с необязательным ключевым словом или стереотипом записывается внутри большого прямоугольника. Однако при наличии строк текста внутри большого прямоугольника имя подсистемы может быть записано рядом с обозначением "вилки" (рис. 6.15.).

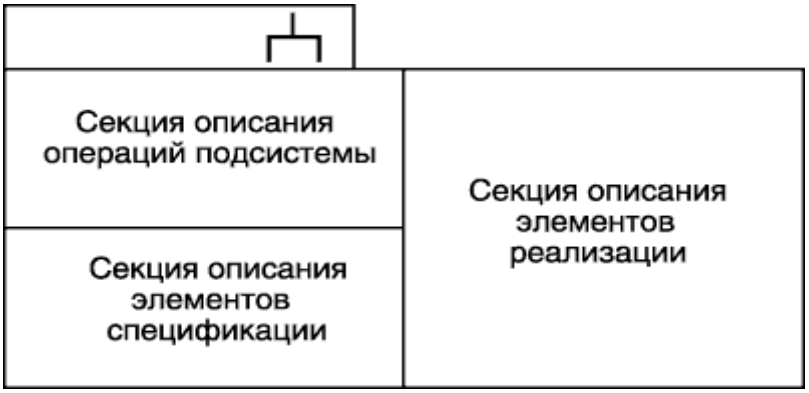


Рис. 6.15. Графическое изображение подсистемы в языке UML

Операции подсистемы записываются в левой верхней секции, ниже указываются элементы спецификации, а справа от вертикальной линии – элементы реализации. При этом два последних раздела помечаются соответствующими метками: "Элементы спецификации" и "Элементы реализации". Секция операций никак не помечается. Если в подсистеме отсутствуют те или иные секции, то они не отображаются на схеме.

Аннотационные сущности – пояснительные части модели UML. Это комментарии для дополнительного описания, разъяснения или замечания к любому элементу модели. Имеется только один базовый тип аннотационных элементов – примечание.

Примечание (Note) – это просто символ для изображения комментариев или ограничений, присоединенных к элементу или группе элементов. Графически примечание изображается в виде прямоугольника с загнутым краем, содержащим текстовый или графический комментарий, как показано на рис. 6.16.

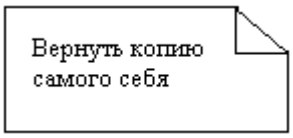


Рис. 6.16. Примечание

Этот элемент является основной аннотационной сущностью, которую можно включать в модель UML. Чаще всего примечания используются, чтобы снабдить диаграммы комментариями или ограничениями, которые можно выразить в виде неформального или формального текста. Существуют вариации этого элемента, например, требования, где описывают некое желательное поведение с точки зрения внешней по отношению к модели.

6.1.2. Отношения

В языке UML определены четыре типа отношений:
– зависимость;

– ассоциация;
– обобщение;
– реализация.

Эти отношения являются основными связующими строительными блоками в UML и применяются для создания моделей.

Зависимость (Dependency) – это семантическое отношение между двумя сущностями, при котором изменение одной из них, независимой, может повлиять на семантику другой, зависимой. Графически зависимость изображается в виде прямой пунктирной линии, часто со стрелкой, которая может содержать метку (рис. 6.17.).

Рис. 6.17. Зависимость

Ассоциация (Association) – структурное отношение, описывающее совокупность связей (соединение между объектами). Разновидностью ассоциации является агрегирование (Aggregation), определяющее структурное отношение между целым и его частями. Графически ассоциация изображается в виде прямой линии (иногда завершающейся стрелкой или содержащей метку), рядом с которой могут присутствовать дополнительные обозначения, на пример кратность и имена ролей (рис. 6.18.).

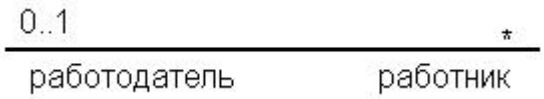


Рис. 6.18. Ассоциация

Обобщение (Generalization) – это отношение "специализация/обобщение", при котором объект специализированного элемента (потомок) может быть подставлен вместо объекта обобщенного элемента (родителя или предка). Таким образом, потомок (Child) наследует структуру и поведение своего родителя (Parent). Графически отношение обобщения изображается в виде линии с не закрашенной стрелкой, указывающей на родителя, как показано на рис. 6.19.

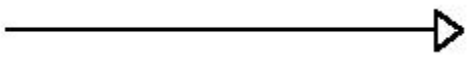


Рис. 6.19. Обобщение

Реализация (Realization) – это семантическое отношение между классификаторами, при котором один классификатор определяет "контракт", а другой гарантирует его выполнение. Отношения реализации встречаются в двух случаях: во–первых, между интерфейсами и реализующими их классами или компонентами, а во–вторых, между прецедентами и реализующими их кооперациями. Отношение реализации изображается в виде пунктирной линии с не

закрашенной стрелкой, как нечто среднее между отношениями обобщения и зависимости (рис. 6.20.).

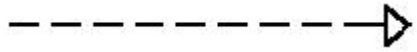


Рис. 6.20. Реализация

Четыре описанных элемента являются основными типами отношений, которые можно включать в модели UML.

6.1.3. Диаграммы

В рамках языка UML все представления о модели сложной системы фиксируются в виде специальных графических конструкций, получивших название диаграмм.

Диаграмма в UML – это графическое представление набора элементов, изображаемое чаще всего в виде графа с вершинами (сущностями) и ребрами (отношениями). Диаграммы рисуют для визуализации системы с разных точек зрения. Диаграмма – это в некотором смысле одна из проекций системы. Теоретически диаграммы могут содержать любые комбинации сущностей и отношений. На практике, однако, применяется сравнительно небольшое количество типовых комбинаций. В UML выделяют девять типов диаграмм:

диаграммы классов;
диаграммы объектов;
диаграммы вариантов использования;
диаграммы последовательностей;
кооперативные диаграммы;
диаграммы состояний;
диаграммы деятельностей;
диаграммы компонентов;
диаграммы размещения.

На **диаграмме классов** показывают классы, интерфейсы, объекты и кооперативные диаграммы, а также их отношения. При моделировании объектно–ориентированных систем этот тип диаграмм используют чаще всего. Диаграммы классов соответствуют статическому виду системы с точки зрения проектирования. Диаграммы классов, которые включают активные классы, соответствуют статическому виду системы с точки зрения процессов.

На **диаграмме объектов** представлены объекты и отношения между ними. Они являются статическими изображениями экземпляров сущностей, показанных на диаграммах классов. Диаграммы объектов, как и диаграммы классов, относятся к статическому виду системы с точки зрения проектирования или процессов, но с расчетом на настоящую или макетную реализацию.

На **диаграмме вариантов** использования представлены варианты использования и актеры (частный случай классов), а также отношения между ними.

Диаграммы вариантов использования относятся к статическому виду системы с точки зрения вариантов использования. Они особенно важны при организации и моделировании поведения системы.

Диаграммы последовательностей и кооперативные диаграммы являются частными случаями диаграмм взаимодействия. На диаграммах взаимодействия представлены связи между объектами; показаны, в частности, сообщения, которыми объекты могут обмениваться. Диаграммы взаимодействия относятся к динамическому виду системы. При этом диаграммы последовательности отражают временную упорядоченность сообщений, а кооперативные диаграммы – структурную организацию обменивающихся сообщениями объектов. Эти диаграммы являются изоморфными, то есть могут быть преобразованы друг в друга.

На **диаграммах состояний** представлен автомат, включающий в себя состояния, переходы, события и виды действий. Диаграммы состояний относятся к динамическому виду системы; особенно они важны при моделировании поведения интерфейса, класса или кооперативной диаграммы. Они акцентируют внимание на поведении объекта, зависящем от последовательности событий, что очень полезно для моделирования реактивных систем.

Диаграмма деятельности – это частный случай диаграммы состояний; на ней представлены переходы потока управления от одной деятельности к другой внутри системы. Диаграммы деятельностей относятся к динамическому виду системы; они наиболее важны при моделировании ее функционирования и отражают поток управления между объектами.

На **диаграмме компонентов** представлена организация совокупности компонентов и существующие между ними зависимости. Диаграммы компонентов относятся к статическому виду системы с точки зрения реализации. Они могут быть соотнесены с диаграммами классов, так как компонент обычно отображается на один или несколько классов, интерфейсов или кооперации.

На **диаграмме размещения** представлена конфигурация обрабатывающих узлов системы и размещенных в них компонентов. Диаграммы размещения относятся к статическому виду архитектуры системы с точки зрения размещения. Они связаны с диаграммами компонентов, поскольку в узле обычно размещаются один или несколько компонентов.

Инструментальные средства позволяют генерировать и другие диаграммы, но девять перечисленных встречаются на практике чаще всего.

В целом интегрированная модель сложной системы в нотации UML может быть представлена в виде совокупности указанных выше диаграмм (рис. 6.21.).

Рис. 6.21. Интегрированная модель сложной системы в нотации UML

Кроме графических элементов, которые определены для каждой канонической диаграммы, на них может быть изображена текстовая информация, которая расширяет семантику базовых элементов.

В последующем канонические диаграммы рассматриваются более подробно.

6.2. Диаграмма вариантов использования как концептуальное представление бизнес–системы в процессе ее разработки

Визуальное моделирование с использованием нотации UML можно представить как процесс поуровневого спуска от наиболее общей и абстрактной концептуальной модели исходной бизнес–системы к логической, а затем и к физической модели соответствующей программной системы. Для достижения этих целей вначале строится модель в форме так называемой диаграммы вариантов использования (use case diagram), которая описывает функциональное назначение системы или, другими словами, то, что бизнес–система должна делать в процессе своего функционирования.

Диаграмма вариантов использования (Use case diagram) — диаграмма, на которой изображаются отношения между актерами и вариантами использования.

Диаграмма вариантов использования – это исходное концептуальное представление или концептуальная модель системы в процессе ее проектирования и разработки. Создание диаграммы вариантов использования имеет следующие цели:

– Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы

– Сформулировать общие требования к функциональному поведению проектируемой системы

– Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей

– Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями

Назначение данной диаграммы состоит в следующем: проектируемая программная система представляется в форме так называемых вариантов использования, с которыми взаимодействуют внешние сущности или актеры. При этом актером или действующим лицом называется любой объект, субъект или система, взаимодействующая с моделируемой бизнес–системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая служит источником воздействия на моделируемую систему так, как определит разработчик. Вариант использования служит для описания сервисов, которые система предоставляет актеру. Другими словами каждый вариант использования определяет набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой и собственно выполнение вариантов использования.

Рассматривая диаграмму вариантов использования в качестве модели бизнес–системы, можно ассоциировать ее с "черным ящиком". Концептуальный характер этой диаграммы проявляется в том, что подробная детализация диаграммы или включение в нее элементов физического уровня представления на начальном этапе проектирования скорее имеет отрицательный характер, поскольку предопределяет способы реализации поведения системы. Эти аспекты должны быть сознательно скрыты от разработчика на диаграмме вариантов использования.

В самом общем случае, диаграмма вариантов использования представляет собой граф специального вида, который является графической нотацией для представления конкретных вариантов использования, актеров и отношений между этими элементами. При этом отдельные элементы диаграммы заключают в прямоугольник, который обозначает границы проектируемой системы. В то же время отношения, которые могут быть изображены на данном графе, представляют собой только фиксированные типы взаимосвязей между актерами и вариантами использования, которые в совокупности описывают сервисы или функциональные требования к моделируемой системе.

6.2.1. Базовые элементы диаграммы вариантов использования

Базовыми элементами диаграммы вариантов использования являются вариант использования и актер.

Вариант использования (Use case) — внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с актерами.

Вариант использования представляет собой спецификацию общих особенностей поведения или функционирования моделируемой системы без рассмотрения внутренней структуры этой системы. Несмотря на то, что каждый вариант использования определяет последовательность действий, которые должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим актером, сами эти действия не изображаются на рассматриваемой диаграмме.

Содержание варианта использования может быть представлено в форме дополнительного пояснительного текста, который раскрывает смысл или семантику действий при выполнении данного варианта использования. Такой пояснительный текст получил название текста–сценария или просто сценария. Далее в этой главе рассматривается один из шаблонов, который может быть рекомендован для написания сценариев вариантов использования.

Отдельный вариант использования обозначается на диаграмме эллипсом, внутри которого содержится его краткое имя в форме существительного или глагола с пояснительными словами. Сам текст имени варианта использования должен начинаться с заглавной буквы (рис. 6.22.).

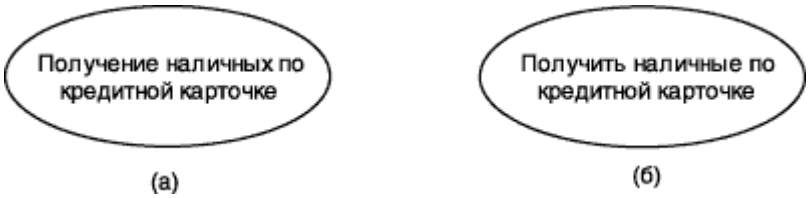


Рис. 6.22. Графическое обозначение варианта использования

Цель спецификации варианта использования заключается в том, чтобы зафиксировать некоторый аспект или фрагмент поведения проектируемой системы без указания особенностей реализации данной функциональности. В этом смысле каждый вариант использования соответствует отдельному сервису, который предоставляет моделируемая система по запросу актера, т. е. определяет один из способов применения системы. Сервис, который инициализируется по запросу актера, должен представлять собой законченную последовательность действий. Это означает, что после того как система закончит обработку запроса актера, она должна возвратиться в исходное состояние, в котором снова готова к выполнению следующих запросов.

Диаграмма вариантов использования содержит конечное множество вариантов использования, которые в целом должны определять все возможные стороны ожидаемого поведения системы. Для удобства множество вариантов использования может рассматриваться как отдельный пакет. Применение вариантов использования на всех этапах работы над проектом позволяет не только достичь требуемого уровня унификации обозначений для представления функциональности подсистем и системы в целом, но и является мощным средством последовательного уточнения требований к проектируемой системе на основе их итеративного обсуждения со всеми заинтересованными специалистами.

Примерами вариантов использования могут быть следующие действия: проверка состояния текущего счета клиента, оформление заказа на покупку товара, получение дополнительной информации о кредитоспособности клиента, отображение графической формы на экране монитора и другие действия.

Актер (Actor) — согласованное множество ролей, которые играют внешние сущности по отношению к вариантам использования при взаимодействии с ними.

Актер представляет собой любую внешнюю по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач. Каждый актер может рассматриваться как некая отдельная роль относительно конкретного варианта использования. Стандартным графическим обозначением актера на диаграммах является фигурка "человечка", под которой записывается имя актера (рис. 6.23.).

Рис. 6.23. Графическое обозначение актера

В некоторых случаях актер может обозначаться в виде прямоугольника класса со стереотипом <<actor>> и обычными составляющими элементами класса. Имена актеров должны начинаться с заглавной буквы и следовать рекомендациям использования имен для типов и классов модели. При этом символ отдельного актера связывает соответствующее описание актера с конкретным именем.

Имя актера должно быть достаточно информативным с точки зрения семантики. Для этой цели подходят наименования должностей в компании (например, продавец, кассир, менеджер, президент). Не рекомендуется давать актерам имена собственные или названия моделей конкретных устройств, даже если это с очевидностью следует из контекста проекта. Дело в том, что одно и то же лицо может выступать в нескольких ролях и, соответственно, обращаться к различным сервисам системы.

Актеры используются для моделирования внешних по отношению к проектируемой системе сущностей, которые взаимодействуют с системой. В качестве актеров могут выступать другие системы, в том числе подсистемы проектируемой системы или ее отдельные классы. Важно понимать, что каждый актер определяет согласованное множество ролей, в которых могут выступать пользователи данной системы в процессе взаимодействия с ней. В каждый момент времени с системой взаимодействует вполне определенный пользователь, при этом он играет или выступает в одной из таких ролей. Наиболее наглядный пример актера — конкретный посетитель web–сайта в Интернет со своими параметрами аутентификации.

Поскольку в общем случае актер всегда находится вне системы, его внутренняя структура никак не определяется. Для актера имеет значение только его внешнее представление, т.е. то, как он воспринимается со стороны системы.

Актеры взаимодействуют с системой посредством передачи и приема сообщений от вариантов использования. Сообщение представляет собой запрос актером сервиса от системы и получение этого сервиса. Это взаимодействие может быть выражено посредством ассоциаций между отдельными актерами и вариантами использования. Кроме этого, с актерами могут быть связаны интерфейсы, которые определяют, каким образом другие элементы модели взаимодействуют с этими актерами.

6.2.2. Отношения на диаграмме вариантов использования

Отношение (Relationship) — семантическая связь между отдельными элементами модели.

Между элементами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов. Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной системы. В свою очередь один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них свой сервис.

В то же время два варианта использования, определенные в рамках одной моделируемой системы, также могут взаимодействовать друг с другом, однако характер этого взаимодействия будет отличаться от взаимодействия с актерами. Однако в обоих случаях способы взаимодействия элементов модели предполагают обмен сигналами или сообщениями, которые инициируют реализацию функционального поведения моделируемой системы.

В языке UML имеется несколько стандартных видов отношений между актерами и вариантами использования:

– ассоциации (association relationship)
– включения (include relationship)
– расширения (extend relationship)
– обобщения (generalization relationship)

При этом общие свойства вариантов использования могут быть представлены тремя различными способами, а именно — с помощью отношений включения, расширения и обобщения.

6.2.2.1. Отношение ассоциации

Отношение ассоциации (Association) – одно из фундаментальных понятий в языке UML и в той или иной степени используется при построении всех

графических моделей систем в форме канонических диаграмм. Применительно к диаграммам вариантов использования ассоциация служит для обозначения специфической роли актера при его взаимодействии с отдельным вариантом использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь некоторые дополнительные обозначения, например, имя и кратность (рис. 6.24.).

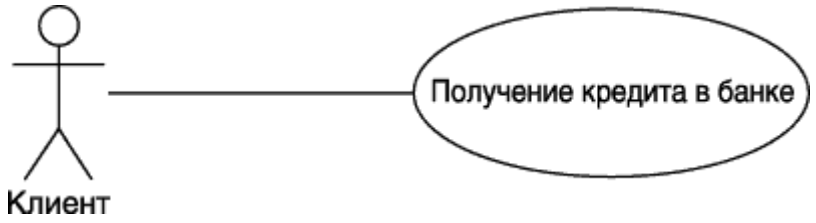


Рис. 6.24. Пример графического представления отношения ассоциации между актером и вариантом использования

В контексте диаграммы вариантов использования отношение ассоциации между актером и вариантом использования может указывать на то, что актер инициирует соответствующий вариант использования. Такого актера называют главным. В других случаях подобная ассоциация может указывать на актера, которому предоставляется справочная информация о результатах функционирования моделируемой системы. Таких актеров часто называют второстепенными. Более детальное описание семантических особенностей отношения ассоциации будет дано при рассмотрении других диаграмм в последующих лекциях.

6.2.2.2. Отношение включения

Отношение включения (Include) в языке UML — это разновидность отношения зависимости между базовым вариантом использования и его специальным случаем. При этом отношением зависимости (dependency) является такое отношение между двумя элементами модели, при котором изменение одного элемента (независимого) приводит к изменению другого элемента (зависимого).

Отношение включения устанавливается только между двумя вариантами использования и указывает на то, что заданное поведение для одного варианта использования включается в качестве составного фрагмента в последовательность поведения другого варианта использования. Данное отношение является направленным бинарным отношением.

Так, например, отношение включения, направленное от варианта использования "Предоставление кредита в банке" к варианту использования "Проверка платежеспособности клиента", указывает на то, что каждый экземпляр первого варианта использования всегда включает в себя функциональное поведение или выполнение второго варианта использования. В этом смысле поведение второго варианта использования является частью поведения первого варианта использования на данной диаграмме. Графически данное отношение обозначается как отношение зависимости в форме пунктирной линии со стрелкой, направленной от базового варианта использования к включаемому варианту использования. При этом данная линия помечается стереотипом <<include>>, как показано на рис. 6.25.

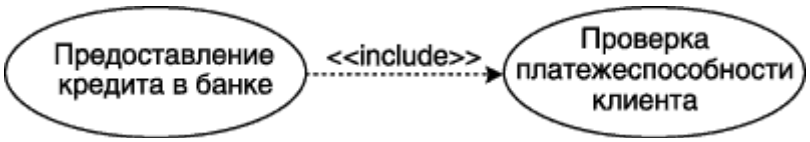


Рис. 6.25. Пример графического изображения отношения включения между вариантами использования

Семантика этого отношения определяется следующим образом. Процесс выполнения базового варианта использования включает в себя как собственное подмножество последовательность действий, которая определена для включаемого варианта использования. При этом выполнение включаемой последовательности действий происходит всегда при инициировании базового варианта использования.

Один вариант использования может входить в несколько других вариантов, а также содержать в себе другие варианты. Включаемый вариант использования является независимым от базового варианта в том смысле, что он предоставляет последнему инкапсулированное поведение, детали реализации которого скрыты от последнего и могут быть легко перераспределены между несколькими включаемыми вариантами использования. Более того, базовый вариант зависит только от результатов выполнения включаемого в него варианта использования, но не от структуры включаемых в него вариантов.

6.2.2.3. Отношение расширения

Отношение расширения (Extend) определяет взаимосвязь базового варианта использования с другим вариантом использования, функциональное поведение которого задействуется базовым не всегда, а только при выполнении дополнительных условий.

В языке UML отношение расширения является зависимостью, направленной к базовому варианту использования и соединенной с ним в так называемой точке расширения. Отношение расширения между вариантами использования обозначается как отношение зависимости в форме пунктирной линии со стрелкой, направленной от того варианта использования, который является расширением для базового варианта использования. Данная линия со стрелкой должна быть помечена стереотипом <<extend>>, как показано на рис. 6.26.

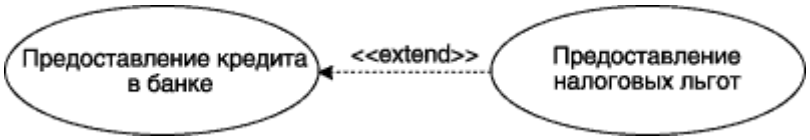


Рис. 6.26. Пример графического изображения отношения расширения между вариантами использования

В изображенном фрагменте имеет место отношение расширения между базовым вариантом использования "Предоставление кредита в банке" и вариантом использования "Предоставление налоговых льгот". Это означает, что свойства поведения первого варианта использования в некоторых случаях могут быть дополнены функциональностью второго варианта использования. Для того чтобы это расширение имело место, должно быть выполнено определенное логическое условие данного отношения расширения.

Отношение расширения позволяет моделировать таким образом, что один из вариантов использования должен присоединять к своему поведению дополнительное поведение, определенное для другого варианта использования. В то же время, данное отношение всегда предполагает проверку условия и ссылку на точку расширения в базовом варианте использования. Точка расширения определяет место в базовом варианте использования, в которое должно быть помещено расширение при выполнении соответствующего логического условия. При этом один из вариантов использования может быть расширением для нескольких базовых вариантов, а также иметь в качестве собственных расширений другие варианты. Базовый вариант использования не зависит от своих расширений.

Семантика отношения расширения определяется следующим образом. Если базовый вариант использования выполняет некоторую последовательность действий, которая определяет его поведение, и при этом имеется точка расширения на экземпляр другого варианта использования, которая является первой из всех точек расширения у базового варианта, то проверяется логическое условие данного отношения. Если это условие выполняется, исходная последовательность действий расширяется посредством включения действий другого варианта использования. Следует заметить, что условие отношения расширения проверяется лишь один раз — при первой ссылке на точку расширения, и если оно выполняется, то все расширяющие варианты использования вставляются в базовый вариант.

6.2.2.4. Отношение обобщения

Два и более актера могут иметь общие свойства, т. е. взаимодействовать с одним и тем же множеством вариантов использования одинаковым образом. Такая общность свойств и поведения представляется в виде **отношения обобщения (Generalization)** с другим.

Графически отношение обобщения обозначается сплошной линией со стрелкой в форме не закрашенного треугольника, которая указывает на родительский вариант использования (рис. 6.27.). Эта линия со стрелкой имеет специальное название — стрелка–обобщение.

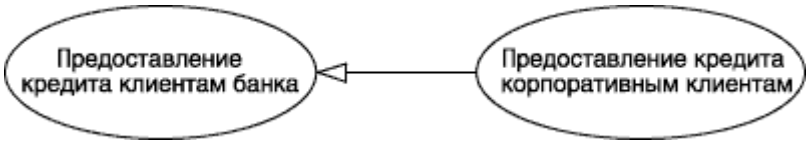


Рис. 6.27. Пример графического изображения отношения обобщения между вариантами использования

В данном примере отношение обобщения указывает на то, что вариант использования "Предоставление кредита корпоративным клиентам" – специальный случай варианта использования "Предоставление кредита клиентам банка". Другими словами, первый вариант использования является специализацией второго варианта использования. При этом вариант использования "Предоставление кредита клиентам банка" еще называют предком или родителем по отношению к варианту использования "Предоставление кредита корпоративным клиентам", а последний вариант называют потомком по отношению к первому варианту использования. Следует подчеркнуть, что потомок наследует все свойства поведения своего родителя, а также может обладать дополнительными особенностями поведения.

Отношение обобщения между вариантами использования применяется в том случае, когда необходимо отметить, что дочерние варианты использования обладают всеми особенностями поведения родительских вариантов. При этом дочерние варианты использования участвуют во всех отношениях родительских вариантов. В свою очередь, дочерние варианты могут наделяться новыми свойствами поведения, которые отсутствуют у родительских вариантов использования, а также уточнять или модифицировать наследуемые от них свойства поведения.

6.2.3. Дополнительные обозначения языка UML для бизнес–моделирования

Язык UML включает в себя специальные механизмы расширения, которые позволяют ввести в рассмотрение дополнительные графические обозначения, ориентированные для решения задач из определенной предметной области.

Примеры подобных обозначений, которые используются для моделирования бизнес–систем и могут быть изображены на диаграммах вариантов использования: бизнес–актер, сотрудник и бизнес–вариант использования (рис. 6.28.).

Бизнес–актер (Business actor) – индивидуум, группа, организация, компания или система, которые взаимодействуют с моделируемой бизнес–системой, но не входят в нее, т.е. не являются частью моделируемой системы. Примерами бизнес–актеров являются клиенты, покупатели, поставщики, партнеры. Общее свойство бизнес–актеров состоит в том, что они являются инициаторами или клиентами бизнес–процессов моделируемой системы.

Сотрудник (Business worker) – индивидуум, который действует внутри моделируемой бизнес–системы, взаимодействует с другими сотрудниками и является участником бизнес–процесса моделируемой системы. Примерами сотрудников являются менеджеры, администраторы, кассиры, инженеры. Общее свойство сотрудников заключается в том то, что они являются субъектами и входят в состав моделируемой системы.

Бизнес–вариант использования (Business use case) — вариант использования, определяющий последовательность действий моделируемой системы, направленных на выполнение отдельного бизнес–процесса. Общее свойство бизнес–вариантов использования состоит в том, что они являются концептуальной моделью отдельных бизнес–процессов моделируемой системы.

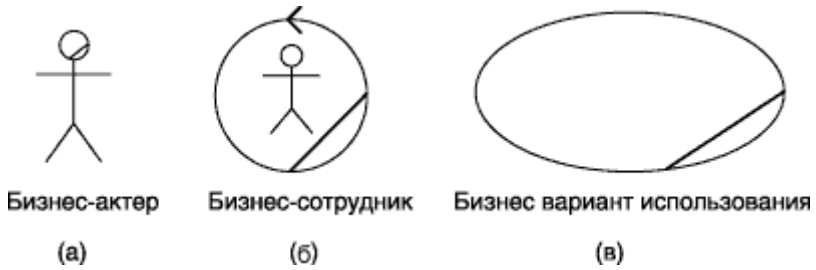


Рис. 6.28. Графические изображения бизнес–актера (а), бизнес–сотрудника (б) и бизнес–варианта использования (в)

6.2.4. Примеры USE CASE и их реализация

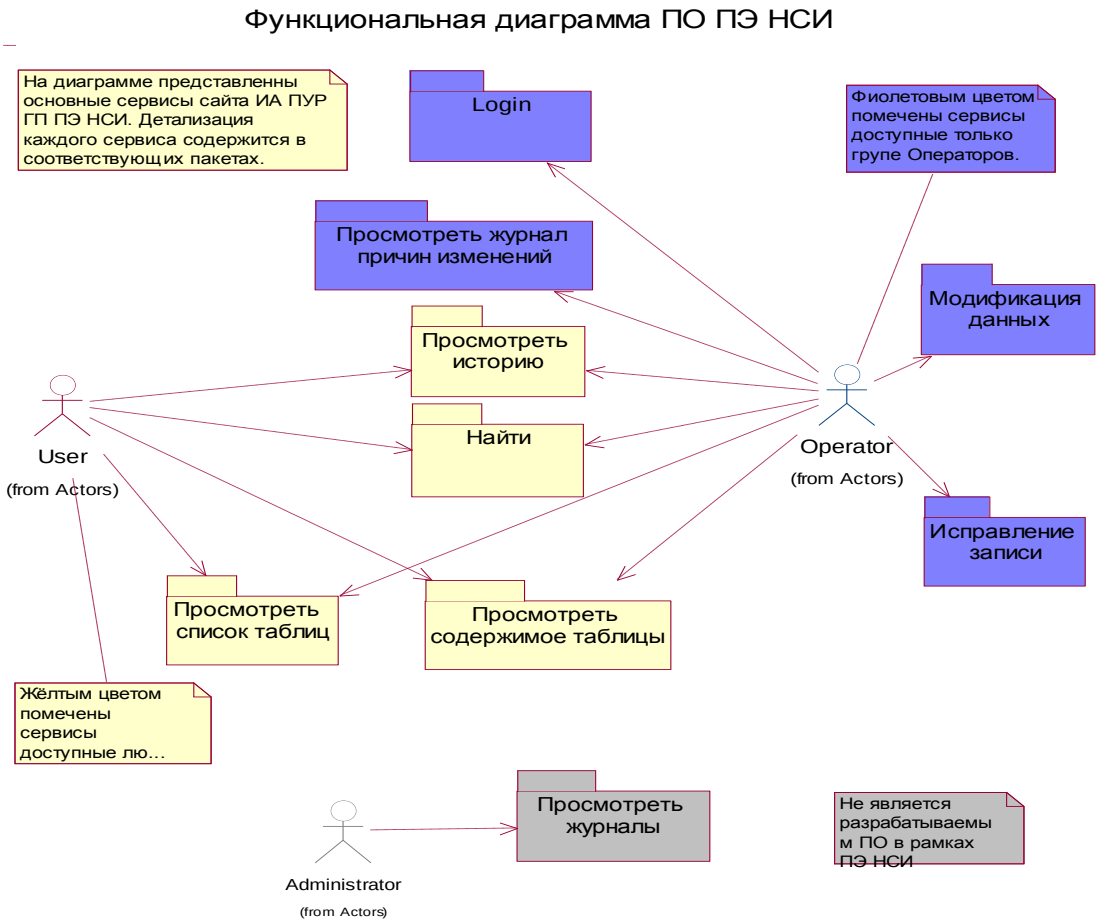


Рис. 6.29. Пример использования пакетов

Краткое описание рис. 6.29:

В данном use case пользователь (Actor) может получить доступ к сайту, и к базе данных НСИ – согласно зарегистрированному имени.

Actors: User, Operator, and Administrator

Поток событий: Основной поток

Начало: Use Case начинается, когда пользователь вводит Login и Password в соответствующие формы и подтверждает ввод.

Проверка: Система проверяет введённые пользователем данные и разрешает доступ.

Альтернативный поток:

Альтернативный поток1:

Отказать в доступе. Введённый пользователем Login и Password не верны. Система предлагает повторить ввод или зарегистрироваться.

Альтернативный поток 2:

Пользователь выбирает сервис "Зарегистрироваться". Система предлагает заполнить форму регистрации и при верном её заполнении создаёт новый аккаунт.

Альтернативный поток 3:

В любой момент времени пользователь может выбрать сервис «Выйти». Текущая сессия пользователя завершается. Соединение с сервером разрывается.

Специфические требования:

Использование IE. Разрабатываемая система предполагает использование браузера Microsoft Internet Explorer v.7.0 и выше.

Постусловия:

Постусловие1:

Загрузка основной страницы. После авторизации пользователя система загружает главную страницу сайта.

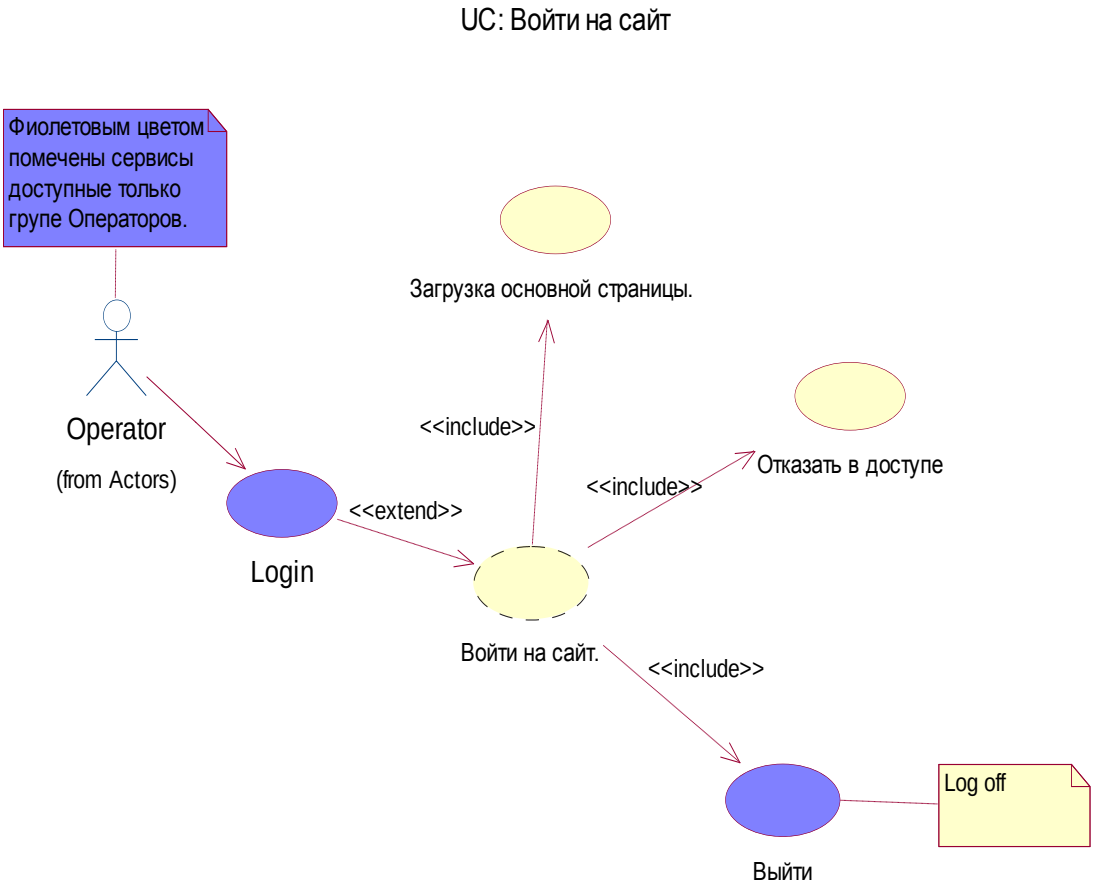


Рис. 6.30. Use Case «Войти на сайт»

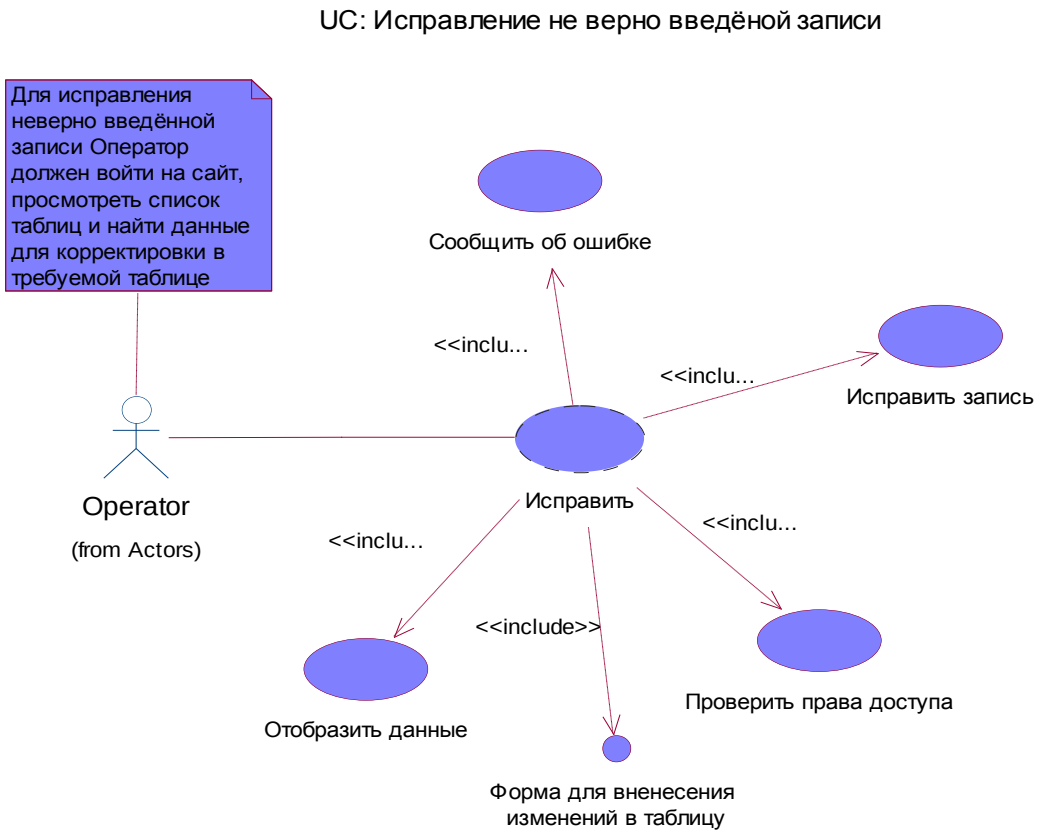


Рис. 6.31. Use Case «Исправление не верно введенной записи»

Краткое описание рис. 6.31:

Use Case стартует, когда Оператор выбирает сервис "Исправить". Назначением данного сервиса является исправление активной записи в БД ПЭ НСИ без создания истории. Оператор выбирает поле, которое должно быть исправлено, вносит в форму ввода новое значение. После ввода значения, сервис "Исправить" вносит изменения в соответствующую запись таблицы БД ПЭ НСИ.

Actors: Оператор

Поток событий:

Основной поток:

Выбор поля:

Оператор выбирает сервис "Исправить". Сервис предлагает выбрать требуемое поле исправляемой записи, отображая список возможных для исправления полей и их текущее значение.

Внесение изменения:

Оператор выбирает поле для исправления и заносит новое значение выбранного поля в форму ввода. Сервис отправляет запрос к БД ПЭ НСИ. СУБД DB2 проверяет права пользователя и, если они являются достаточными, разрешает изменение.

Форма изменения:

С точки зрения разрабатываемой ПЭ НСИ, исправление ошибки не является исторически–информационным и не влечёт за собой появление новых записей и изменения в связанных записях.

Альтернативные потоки:

Альтернативный поток1:

Сообщить об ошибке. При выборе сервиса "Исправление записи" система возвращает Пользователю сообщение об ошибках:

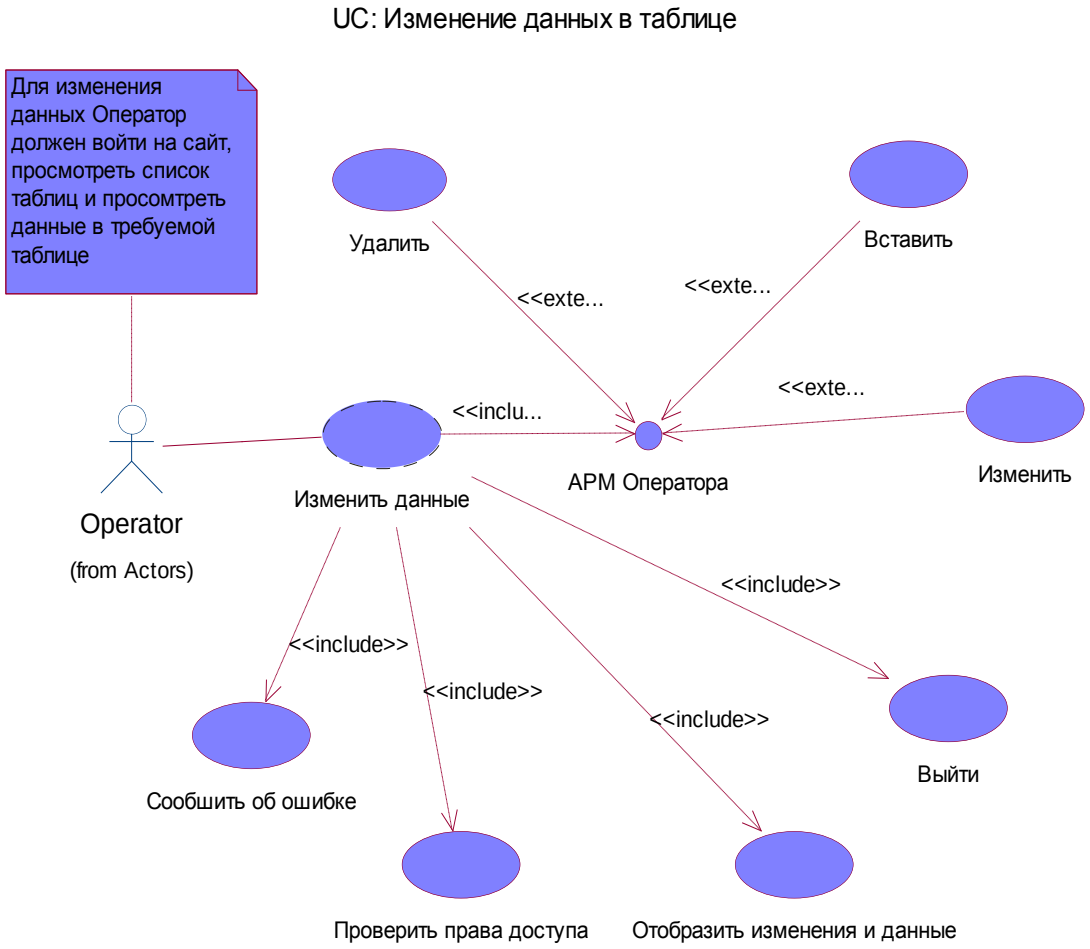


Рис. 6.32. Use Case «Изменение данных в таблице»

Краткое описание рис. 6.32:

Use Case стартует, когда Оператор выбирает сервис "Модификация данных". Оператор может выбрать: вставить новую запись; удалить существующую запись; изменить существующую запись. Система контролирует действия Оператора, и правильность введённых данных (по типам)

Actors: Operator

Поток событий:

Основной поток:

Выбрать действие. Оператор выбирает возможное действие: вставить новую запись; удалить существующую запись; изменить существующую запись.

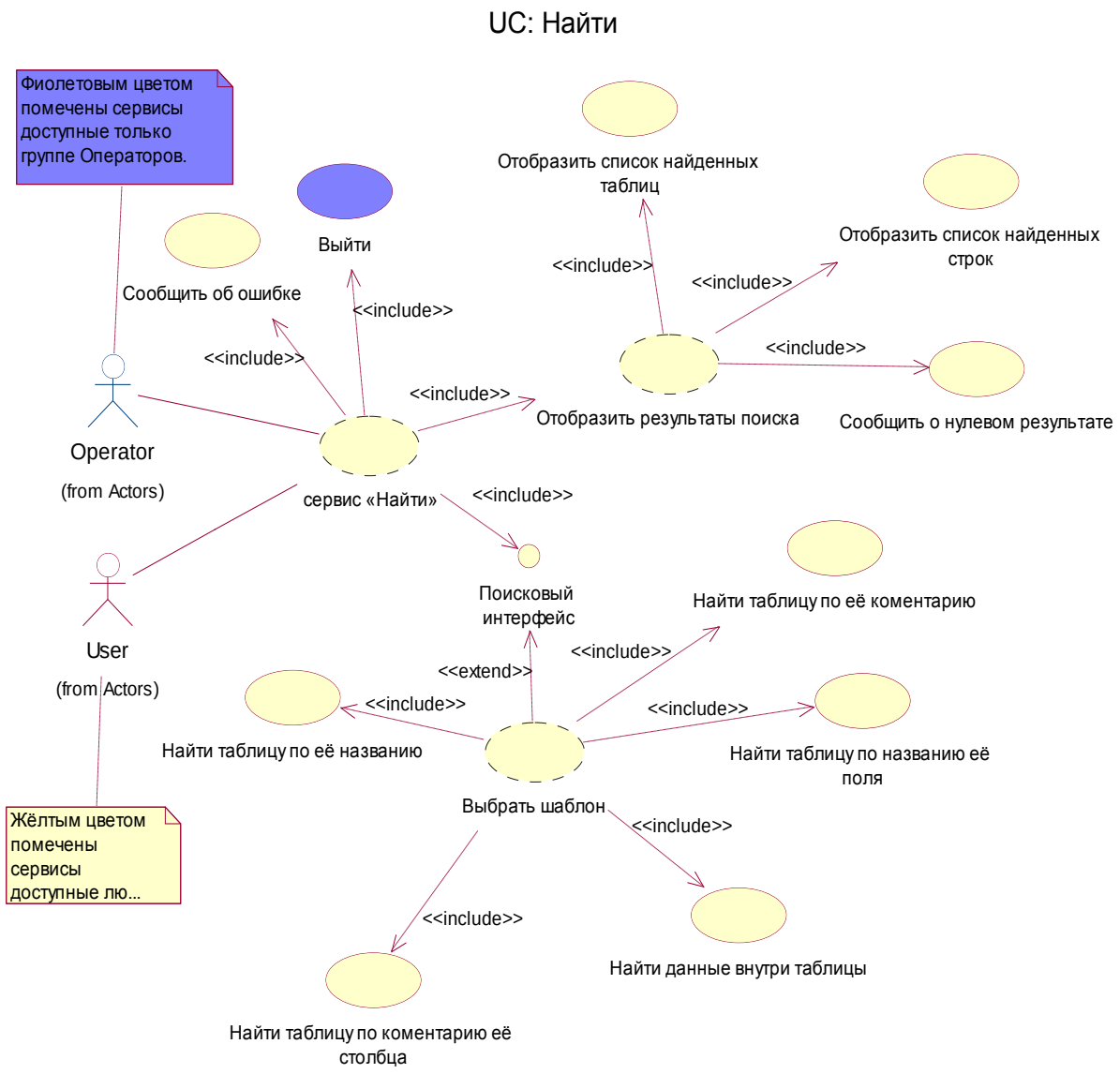


Рис. 6.33. Use Case «Найти»

6.3. Диаграммы последовательности

Диаграмма последовательности (Sequence diagram) – диаграмма, на которой показаны взаимодействия объектов, упорядоченные по времени их проявления.

Диаграммы последовательности характеризуются двумя особенностями.

Во–первых, на них показана линия жизни объекта.

Линия жизни объекта (Object lifeline) – вертикальная линия на диаграмме последовательности, которая представляет существование объекта в течение определенного периода времени. Линия жизни объекта изображается пунктирной вертикальной линией, ассоциированной с единственным объектом на диаграмме

последовательности. Линия жизни служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей рабочей области диаграммы последовательности от самой верхней ее части до самой нижней.

Большая часть объектов, представленных на диаграмме взаимодействий, существует на протяжении всего взаимодействия, поэтому их изображают в верхней части диаграммы, а их линии жизни прорисованы сверху донизу. Объекты могут создаваться и во время взаимодействий. Линии жизни таких объектов начинаются с получения сообщения со стереотипом create. Объекты могут также уничтожаться во время взаимодействий; в таком случае их линии жизни заканчиваются получением сообщения со стереотипом destroy, а в качестве визуального образа используется большая буква X, обозначающая конец жизни объекта. (Обстоятельства жизненного цикла объекта можно указывать с помощью ограничений new, destroyed и transient.)

Если объект на протяжении своей жизни изменяет значения атрибутов, состояние или роль, это можно показать, поместив копию его пиктограммы на линии жизни в точке изменения.

Вторая особенность этих диаграмм – **фокус управления**. Он изображается в виде вытянутого прямоугольника, показывающего промежуток времени, в течение которого объект выполняет какое–либо действие, непосредственно или с помощью подчиненной процедуры. Верхняя грань прямоугольника выравнивается по временной оси с моментом начала действия, нижняя – с моментом его завершения (и может быть помечена сообщением о возврате). Вложенность фокуса управления, вызванную рекурсией (то есть обращением к собственной операции) или обратным вызовом со стороны другого объекта, можно показать, расположив другой фокус управления чуть правее своего родителя (допускается вложенность произвольной глубины). Если место расположения фокуса управления требуется указать с максимальной точностью, можно заштриховать область прямоугольника, соответствующую времени, в течение которого метод действительно работает и не передает управление другому объекту.

Крайним слева на диаграмме последовательности изображается **объект – инициатор** моделируемого процесса взаимодействия (объект a на рис. 6.34.). Правее – другой объект, который непосредственно взаимодействует с первым. Таким образом, порядок расположения объектов на диаграмме последовательности определяется исключительно соображениями удобства визуализации их взаимодействия друг с другом.

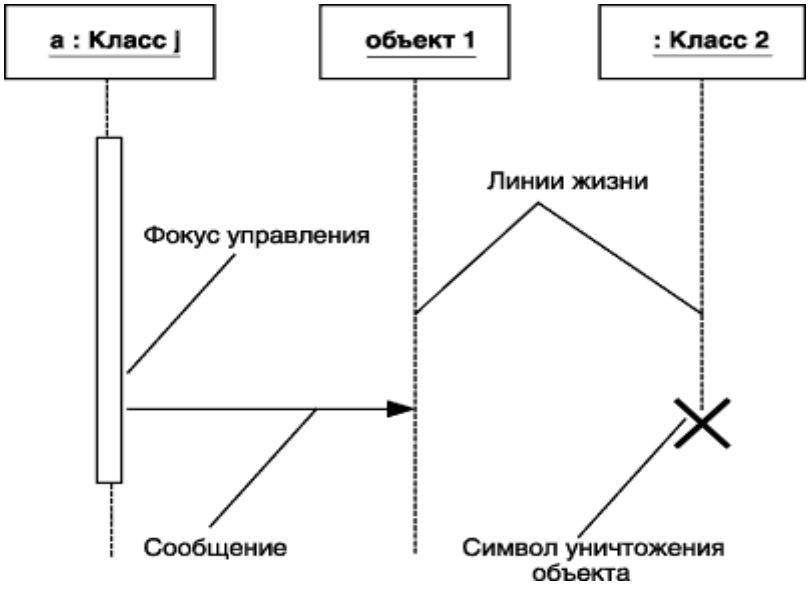


Рис. 6.34. Графические элементы диаграммы последовательности

Начальному моменту времени соответствует самая верхняя часть диаграммы. При этом процесс взаимодействия объектов реализуется посредством сообщений, которые посылаются одними объектами другим. Сообщения изображаются в виде горизонтальных стрелок с именем сообщения и образуют определенный порядок относительно времени своей инициализации. Другими словами, сообщения, расположенные на диаграмме последовательности выше, передаются раньше тех, которые расположены ниже. При этом масштаб на оси времени не указывается, поскольку диаграмма последовательности моделирует лишь временную упорядоченность взаимодействий типа "раньше–позже".

В отдельных случаях инициатором взаимодействия в системе может быть актер или внешний пользователь. При этом актер изображается на диаграмме последовательности самым первым объектом слева со своим фокусом управления (рис. 6.35.). Наиболее часто актер и его фокус управления будут существовать в системе постоянно, отмечая характерную для пользователя активность в инициировании взаимодействий с системой. Актер может иметь собственное имя либо оставаться анонимным.

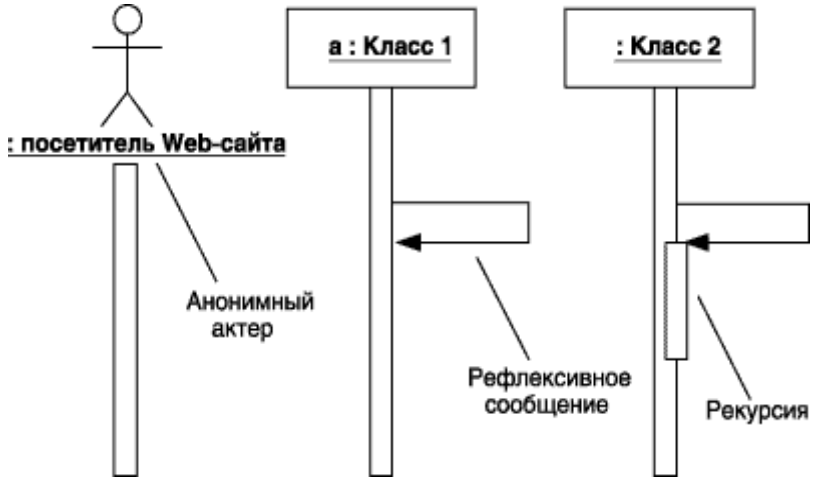


Рис. 6.35. Графическое изображение актера, рефлексивного сообщения и рекурсии на диаграмме последовательности

6.3.1. Сообщения на диаграмме последовательности

На диаграмме последовательности все сообщения упорядочены по времени своей передачи в моделируемой системе, хотя номера у них могут не указываться.

На диаграммах последовательности могут присутствовать три разновидности сообщений, каждое из которых имеет свое графическое изображение (рис. 6.36.).

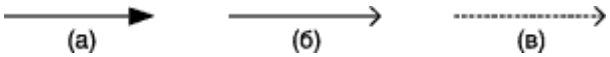


Рис. 6.36. Графическое изображение различных видов сообщений между объектами на диаграмме последовательности

Первая разновидность сообщения (рис. 6.36, а) наиболее распространена и используется для вызова процедур, выполнения операций или обозначения отдельных вложенных потоков управления. Начало этой стрелки, как правило, соприкасается с фокусом управления того объекта–клиента, который инициирует это сообщение. Конец стрелки соприкасается с линией жизни того объекта, который принимает это сообщение и выполняет в ответ определенные действия. При этом принимающий объект может получить фокус управления, становясь в этом случае активным. Передающий объект может потерять фокус управления или остаться активным.

Вторая разновидность сообщения (рис. 6.36, б) используется для обозначения простого асинхронного сообщения, которое передается в произвольный момент времени. Передача такого сообщения обычно не сопровождается получением фокуса управления объектом–получателем.

Третья разновидность сообщения (рис. 6.36, в) используется для возврата из вызова процедуры. Примером может служить простое сообщение о завершении вычислений без предоставления результата расчетов объекту–клиенту. В

процедурных потоках управления эта стрелка может быть опущена, поскольку ее наличие неявно предполагается в конце активизации объекта. В то же время считается, что каждый вызов процедуры имеет свою пару – возврат вызова. Для непроцедурных потоков управления, включая параллельные и асинхронные сообщения, стрелка возврата должна указываться явным образом.

Обычно сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности. При этом неявно предполагается, что время передачи сообщения достаточно мало по сравнению с процессами выполнения действий объектами. Считается также, что за время передачи сообщения с соответствующими объектами не может произойти никаких событий. Другими словами, состояния объектов не изменяются. Если же это предположение не может быть признано справедливым, то стрелка сообщения изображается под наклоном, так чтобы конец стрелки располагался ниже ее начала.

Каждое сообщение на диаграмме последовательности ассоциируется с определенной операцией, которая должна быть выполнена принявшим его объектом. При этом операция может иметь аргументы или параметры, значения которых влияют на получение различных результатов. Соответствующие параметры операция будет иметь и вызывающее это действие сообщение. Более того, значения параметров отдельных сообщений могут содержать условные выражения, образуя ветвление или альтернативные пути основного потока управления.

На диаграммах последовательностей внимание акцентируется, прежде всего, на временной упорядоченности сообщений. На рис. 6.37. показано, что для создания такой диаграммы надо, прежде всего, расположить объекты, участвующие во взаимодействии, в верхней ее части вдоль оси X. Обычно инициирующий взаимодействие объект размещают слева, а остальные – правее (тем дальше, чем более подчиненным является объект). Затем вдоль оси Y размещаются сообщения, которые объекты посылают и принимают, причем более поздние оказываются ниже. Это дает читателю наглядную картину, позволяющую понять развитие потока управления во времени.

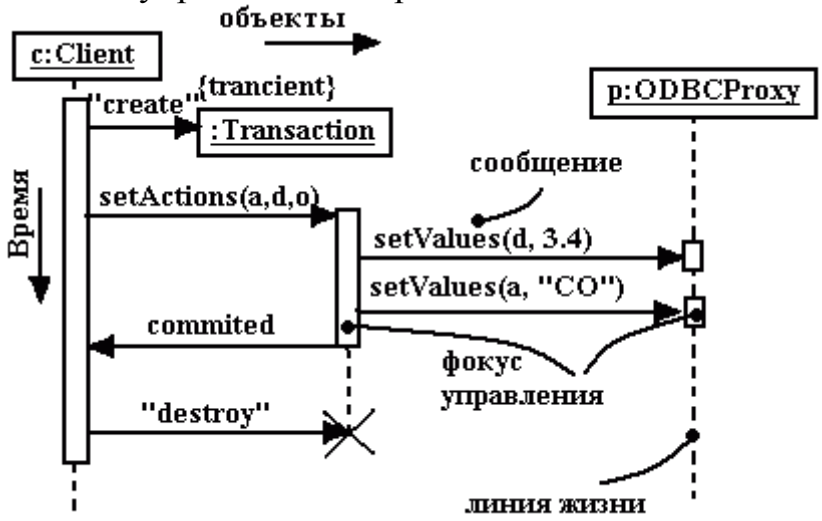


Рис. 6.37. Диаграмма последовательностей

В отдельных случаях объект может посылать сообщения самому себе, инициируя так называемые рефлексивные сообщения. Для этой цели служит специальное изображение (сообщение у объекта а на рис. 6.35.). Такие сообщения изображаются в форме сообщения, начало и конец которого соприкасаются с линией жизни или фокусом управления одного и того же объекта.

6.3.2. Ветвление потока управления

Одна из особенностей диаграммы последовательности – возможность визуализировать простое ветвление процесса. Для изображения ветвления используются две или более стрелки, выходящие из одной точки фокуса управления объекта (объект ob1 на рис. 6.38.). При этом рядом с каждой из них должно быть явно указано соответствующее условие ветви в форме булевского выражения.

Количество ветвей может быть произвольным, однако наличие ветвлений может существенно усложнить интерпретацию диаграммы последовательности. Предложение–условие должно быть явно указано для каждой ветви и записывается в форме обычного текста, псевдокода или выражения языка программирования. Это выражение всегда должно возвращать некоторое булевское выражение. Запись этих условий должна исключать одновременную передачу альтернативных сообщений по двум и более ветвям. В противном случае на диаграмме последовательности может возникнуть конфликт ветвления.

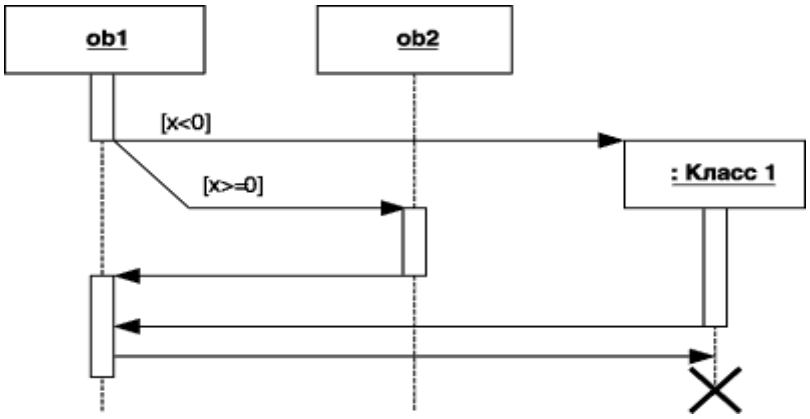


Рис. 6.38. Графическое изображение бинарного ветвления потока управления на диаграмме последовательности

С помощью ветвления можно изобразить и более сложную логику взаимодействия объектов между собой (объект ob1 на рис. 6.39.). Если условий более двух, то для каждого из них необходимо предусмотреть ситуацию единственного выполнения. Описанный ниже пример относится к моделированию взаимодействия программной системы обслуживания клиентов в банке. В этом

примере диаграммы последовательности объект ob1 вызывает выполнение действий у одного из трех других объектов.

Условием ветвления может служить сумма снимаемых клиентом средств со своего текущего счета. Если эта сумма превышает 1500$, то могут потребоваться дополнительные действия, связанные с созданием и последующим разрушением объекта Класса 1. Если же сумма превышает 100$, но не превышает 1500$, то вызывается операция или процедура объекта ob3. И, наконец, если сумма не превышает 100$, то вызывается операция или процедура объекта ob2. При этом объекты ob1, ob2 и ob3 постоянно существуют в системе. Последний объект создается от Класса 1 только в том случае, если справедливо первое из альтернативных условий. В противном случае он может быть никогда не создан.

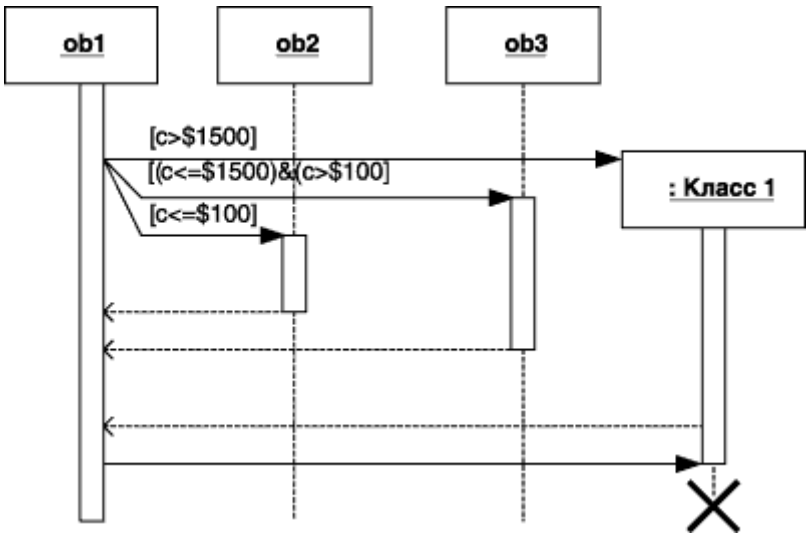


Рис. 6.39. Графическое изображение тернарного ветвления потока управления на диаграмме последовательности

Объект ob1 имеет постоянный фокус управления, а все остальные объекты – получают фокус управления только для выполнения ими соответствующих операций.

На диаграммах последовательности при записи сообщений также могут использоваться стереотипы. Их семантика и синтаксис остаются без изменения, как они определены в нотации языка UML. Ниже представлена диаграмма последовательности для описанного выше случая ветвления, дополненная стереотипными значениями отдельных сообщений (рис. 6.40.). Очевидно, эта диаграмма последовательности является более выразительной и простой для своей содержательной интерпретации.

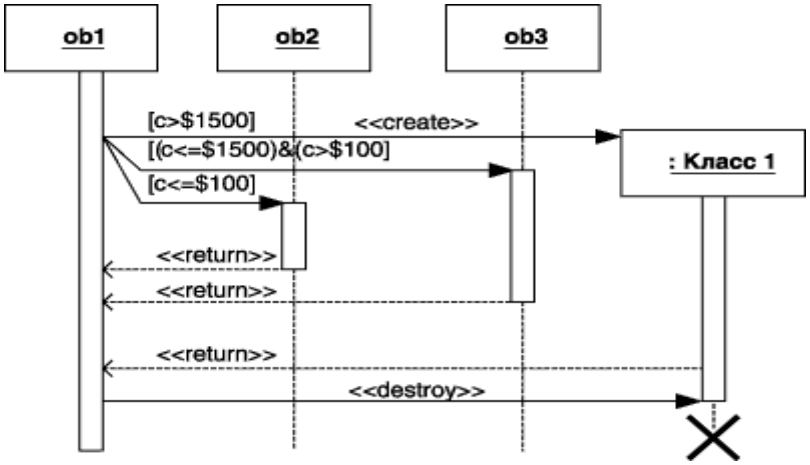


Рис. 6.40. Диаграмма последовательности со стереотипными значениями сообщений

6.3.3. Пример диаграммы последовательности

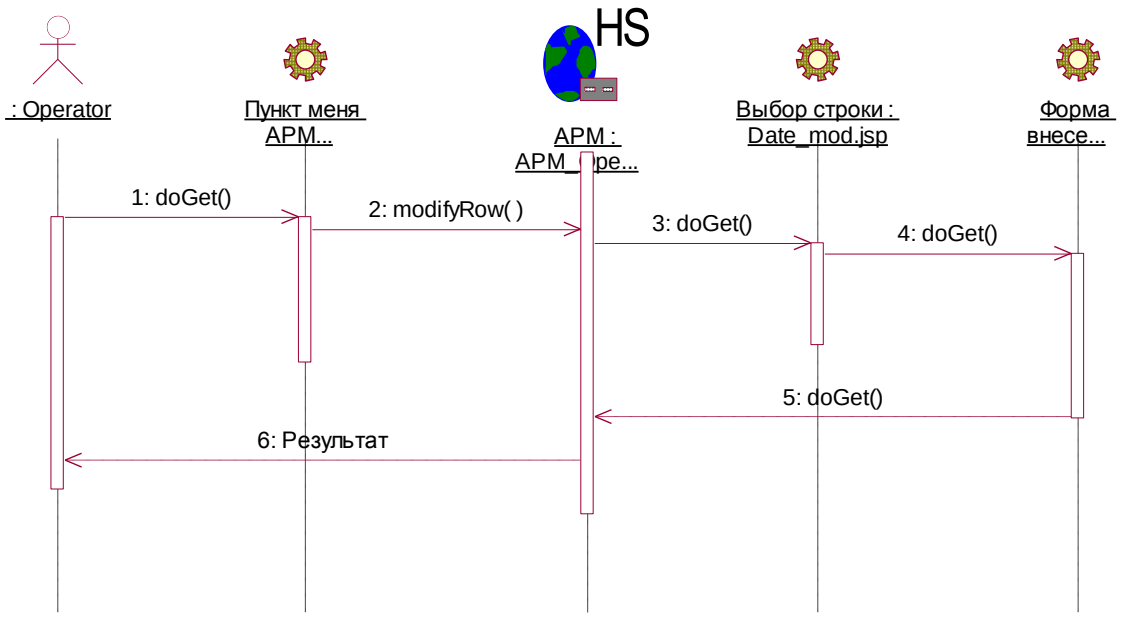


Рис. 6.41

6.4. Диаграмма кооперации

Диаграмма кооперации (Collaboration diagram) предназначена для описания поведения системы на уровне отдельных объектов, которые обмениваются между собой сообщениями, чтобы достичь нужной цели или реализовать некоторый вариант использования.

С точки зрения аналитика или архитектора системы в проекте важно представить структурные связи отдельных объектов между собой. Такое представление структуры модели как совокупности взаимодействующих объектов и обеспечивает диаграмма кооперации.

Диаграммой последовательностей (Sequence diagram) называется диаграмма взаимодействий, акцентирующая внимание на временной упорядоченности сообщений.

А кооперативной диаграммой (Collaboration diagram) называется диаграмма взаимодействий, основное внимание в которой уделяется структурной организации объектов, принимающих и отправляющих сообщения. Графически такая диаграмма представляет собой граф из вершин и ребер (рис. 6.42.).

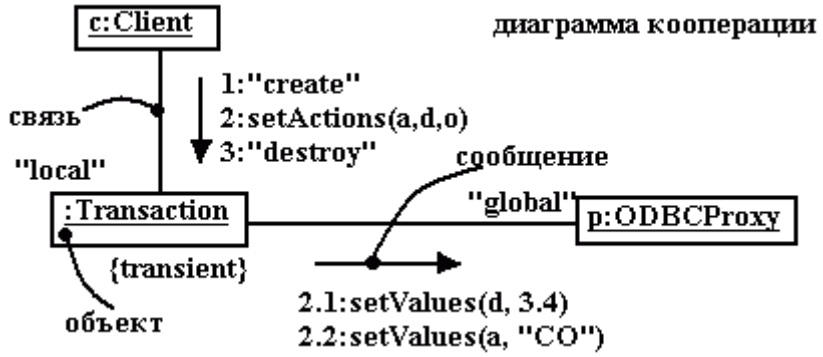


Рис. 6.42. Диаграммы кооперации

6.4.1. Объекты диаграммы кооперации и их графическое изображение

Объект (Object) — сущность с хорошо определенными границами и индивидуальностью, которая инкапсулирует состояние и поведение.

В контексте языка UML любой объект является экземпляром класса, описанного в модели и представленного на диаграмме классов. Объект создается на этапе реализации модели или выполнения программы. Он имеет собственное имя и конкретные значения атрибутов.

Следует рассмотреть особенности семантики и графической нотации объектов, из которых строятся диаграммы.

Для диаграмм полное имя объекта в целом представляет собой строку текста, разделенную двоеточием и записанную в формате: <собственное имя объекта >'/'<Имя роли класса>:<Имя класса >.

Имя роли класса указывается в том случае, когда соответствующий класс отсутствует в модели или разработчику необходимо акцентировать внимание на особенности его использования в рассматриваемом контексте моделирования взаимодействия. Имя класса – это имя одного из классов, представленного на диаграмме классов. Важно отметить, что вся запись имени объекта подчеркивается, что является визуальным признаком объектов на различных диаграммах языка UML.

Если указано собственное имя объекта, то оно должно начинаться со строчной буквы. В тоже время имя объекта, имя роли с символом "/" или имя класса могут отсутствовать. Однако двоеточие всегда должно стоять перед именем класса, а косая черта – перед именем роли.

Таким образом, на диаграммах могут встретиться следующие варианты:

o : C– объект с собственным именем o, экземпляр класса C.

: C– анонимный объект, экземпляр класса C.

o :(или просто o) — объект–сирота с собственным именем o.

o / R : C— объект с собственным именем o, экземпляр класса C, играющий роль R.

/ R : C— анонимный объект, экземпляр класса C, играющий роль R.

o / R— объект–сирота с собственным именем o, играющий роль R.

/ R— анонимный объект и одновременно объект–сирота, играющий роль R.

Примеры изображения объектов на диаграммах кооперации приводятся на рис. 6.43.

Рис. 6.43. Примеры графических изображений объектов на диаграммах кооперации и последовательности

Если собственное имя объекта отсутствует, то такой объект принято называть анонимным. Однако в этом случае обязательно ставится двоеточие перед именем соответствующего класса. Отсутствовать может и имя класса – такой объект называется сиротой. Для него записывается только собственное имя объекта, двоеточие не ставится, имя класса не указываются. Если для объектов указываются атрибуты, то в большинстве случаев они принимают конкретные значения. Для отдельных объектов могут быть дополнительно указаны роли, которые они играют в кооперации.

В контексте языка UML все объекты делятся на две категории: пассивные и активные. **Пассивный объект** оперирует только данными и не может

инициировать деятельность по управлению другими объектами. Однако пассивные объекты могут посылать сигналы в процессе выполнения запросов, которые они обрабатывают. На диаграмме кооперации пассивные объекты изображаются обычным образом без дополнительных стереотипов.

Активный объект (Active object) имеет собственный процесс управления и может инициировать деятельность по управлению другими объектами.

Активный объект на диаграмме кооперации обозначаются прямоугольником с утолщенными границами (рис. 6.44.). Каждый активный объект является владельцем определенного процесса управления.

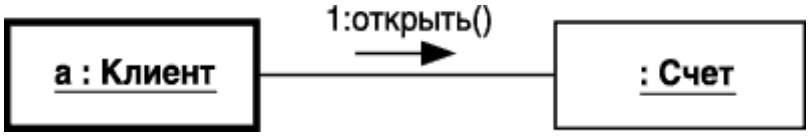


Рис. 6.44. Графическое изображение активного объекта (слева) на диаграмме кооперации

В данном фрагменте диаграммы кооперации активный объект a : Клиент является инициатором открытия счета, который представлен анонимным объектом : Счет.

6.4.2. Кооперация объектов

Кооперация (Collaboration) — спецификация множества объектов отдельных классов, совместно взаимодействующих с целью реализации отдельных вариантов использования в общем контексте моделируемой системы.

Понятие кооперации – одно из фундаментальных в языке UML. Цель самой кооперации состоит в том, чтобы специфицировать особенности реализации отдельных вариантов использования или наиболее значимых операций в системе. Кооперация определяет структуру поведения системы в терминах взаимодействия участников этой кооперации.

На диаграмме кооперации размещаются объекты, представляющие собой экземпляры классов, связи между ними, которые в свою очередь являются экземплярами ассоциаций и сообщения. Связи дополняются стрелками сообщений, при этом показываются только те объекты, которые участвуют в реализации моделируемой кооперации. Далее, как и на диаграмме классов, показываются структурные отношения между объектами в виде различных соединительных линий. Связи могут дополняться именами ролей, которые играют объекты в данной взаимосвязи. И, наконец, изображаются динамические взаимосвязи — потоки сообщений в форме стрелок с указанием направления рядом с соединительными линиями между объектами, при этом задаются имена сообщений и их порядковые номера в общей последовательности сообщений.

Одна и та же совокупность объектов может участвовать в реализации различных кооперaций. В зависимости от рассматриваемой кооперации, могут изменяться как связи между отдельными объектами, так и поток сообщений между ними. Именно это отличает диаграмму кооперации от диаграммы классов, на которой должны быть указаны все без исключения классы, их атрибуты и операции, а также все ассоциации и другие структурные отношения между элементами модели.

На кооперативных диаграммах линию жизни объекта явным образом не показывают, хотя можно показать сообщения create и destroy. Не показывают там и фокус управления, однако порядковые номера сообщения могут отображать вложенность.

В языке UML предусмотрены стандартные действия, выполняемые в ответ на получение соответствующего сообщения. Они могут быть явно указаны на диаграмме кооперации в форме стереотипа перед именем сообщения, к которому они относятся, или выше его. В этом случае они записываются в угловых кавычках.

Кооперативная диаграмма акцентирует внимание на организации объектов, принимающих участие во взаимодействии. Как показано на рис. 6.45., для создания кооперативной диаграммы нужно расположить участвующие во взаимодействии объекты в виде вершин графа. Затем связи, соединяющие эти объекты, изображаются в виде дуг этого графа. Наконец, связи дополняются сообщениями, которые объекты принимают и посылают. Это дает пользователю ясное визуальное представление о потоке управления в контексте структурной организации кооперирующихся объектов.

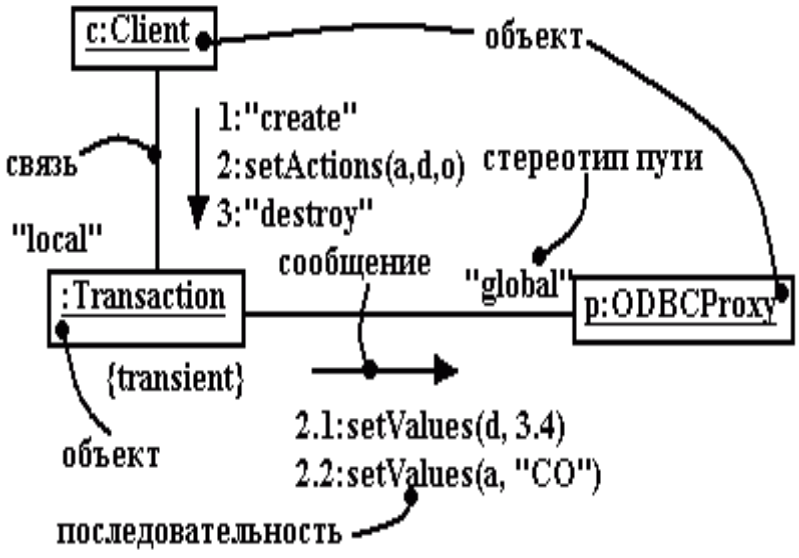


Рис. 6.45. Кооперативная диаграмма

6.4.3. Пример совместного использования диаграмм кооперации и последовательности

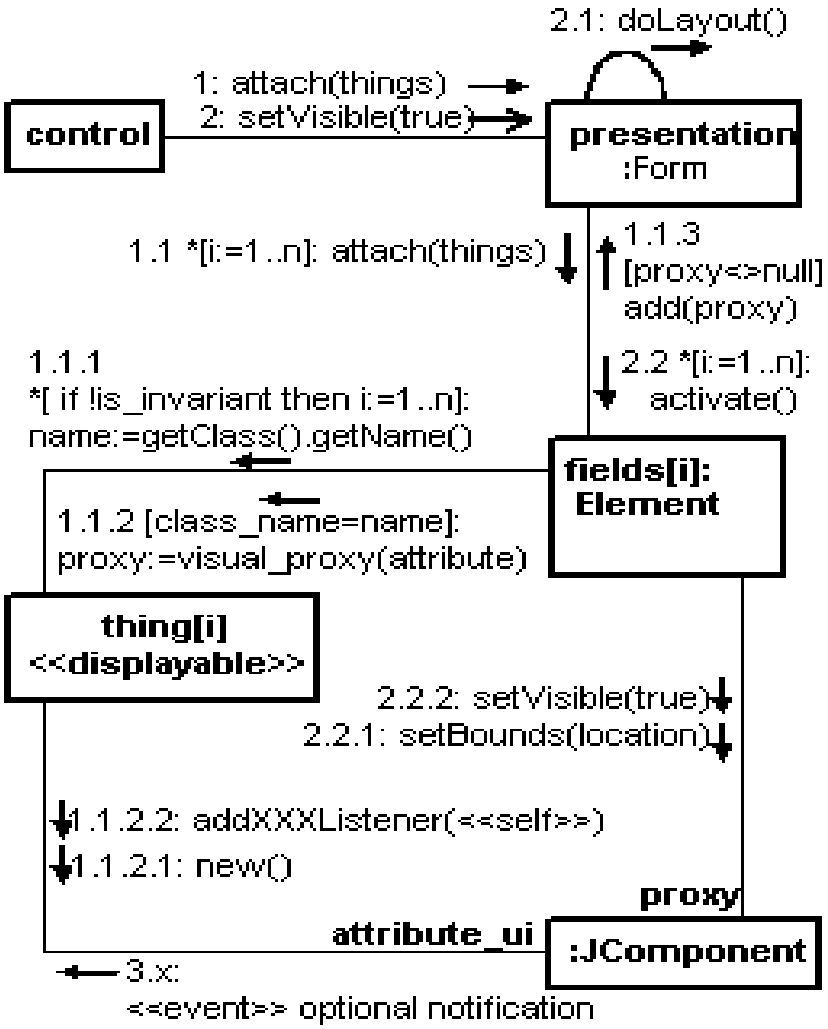


Рис. 6.46.

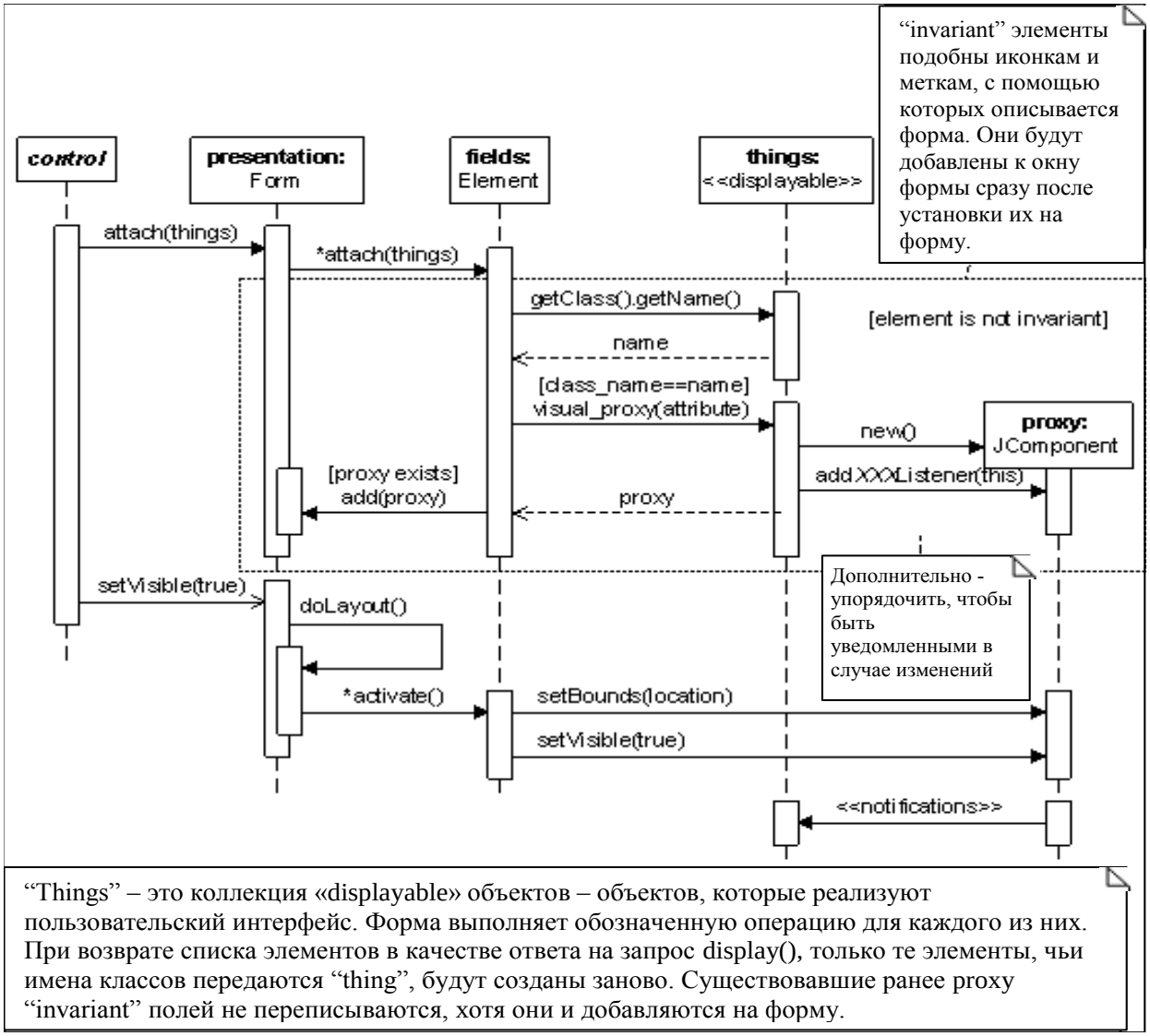


Рис. 6.47.

6.5. Сравнение диаграммы последовательности и диаграммы кооперации

Диаграммы последовательностей и кооперативные диаграммы (и те, и другие называются диаграммами взаимодействий) относятся к числу пяти видов диаграмм, применяемых в UML для моделирования динамических аспектов системы.

В моделирование динамических аспектов системы входит моделирование конкретных и прототипических экземпляров классов, интерфейсов, компонентов и узлов, а также сообщений, которыми они обмениваются, – и все это в контексте сценария, иллюстрирующего данное поведение. Диаграммы взаимодействий могут существовать автономно и служить для визуализации, специфицирования, конструирования и документирования динамики конкретного сообщества

объектов, а могут использоваться для моделирования отдельного потока управления в составе прецедента.

Диаграммы взаимодействий важны не только для моделирования динамических аспектов системы, но и для создания исполняемых систем посредством прямого и обратного проектирования.

Диаграмме взаимодействий присущи общие для всех диаграмм свойства: имя и графическое содержание, являющееся одной из проекций модели. От других диаграмм ее отличает содержание.

На диаграммах взаимодействий показывают связи, включающие множество объектов и отношений между ними, в том числе сообщения, которыми объекты обмениваются. При этом диаграмма последовательностей акцентирует внимание на временной упорядоченности сообщений, а кооперативная диаграмма – на структурной организации посылающих и принимающих сообщения объектов.

У кооперативной диаграммы есть **два свойства**, которые отличают их от диаграмм последовательностей.

Первое – это путь. Для описания связи одного объекта с другим к дальней концевой точке этой связи можно присоединить стереотип пути (например, local, показывающий, что помеченный объект является локальным по отношению к отправителю сообщения). Имеет смысл явным образом изображать путь связи только в отношении путей типа local, parameter, global и self (но не associations).

Второе свойство – это порядковый номер сообщения. Для обозначения временной последовательности перед сообщением можно поставить номер (нумерация начинается с единицы), который должен постепенно возрастать для каждого нового сообщения (2, 3 и. т.д.). Для обозначения вложенности используется десятичная нотация Дьюи (1 – первое сообщение; 1.1– первое сообщение, вложенное в сообщение 1; 1.2 – второе сообщение, вложенное в сообщение 1 и т.д.). Уровень вложенности не ограничен. Для каждой связи можно показать несколько сообщений (вероятно, посылаемых разными отправителями), и каждое из них должно иметь уникальный порядковый номер.

Поскольку диаграммы последовательностей и кооперативные диаграммы используют одну и ту же информацию из метамодели UML, они семантически эквивалентны. Это означает, что можно преобразовать диаграмму одного типа в другой, без какой–либо потери информации. Это не означает, однако, что на обеих диаграммах представлена в точности одна и та же информация. Диаграммы обоих типов используют одну модель, но визуализируют разные ее особенности.

Диаграммы взаимодействий используются для моделирования динамических аспектов системы. Речь идет о взаимодействии экземпляра, любой разновидности в любом представлении системной архитектуры, включая экземпляры классов, в том числе активных, интерфейсов, компонентов и узлов.

Моделирование динамических аспектов системы с помощью диаграммы взаимодействий возможно в контексте системы в целом, подсистемы, операции или класса. Диаграммы взаимодействий можно присоединять также к прецедентам (для моделирования сценария) и к кооперациям (для моделирования динамических аспектов сообщества объектов).

При моделировании динамических аспектов системы диаграммы взаимодействий обычно используются двояко:

– для моделирования временной упорядоченности потоков управления. С этой целью используют диаграммы последовательностей. При этом внимание акцентируется на передаче сообщений во времени, что бывает особенно полезно для визуализации динамического поведения в контексте вариантов использования. Простые итерации и ветвления на диаграммах последовательностей отображать удобнее, чем на кооперативных диаграммах;

– для моделирования структурной организации потоков управления. В этом случае нужны кооперативные диаграммы. Основное внимание при этом уделяется моделированию структурных отношений между взаимодействующими экземплярами, вдоль которых передаются сообщения. Для визуализации сложных итераций, ветвлений и параллельных потоков управления кооперативные диаграммы подходят лучше, чем диаграммы последовательностей.

Моделирование временной упорядоченности потока управления осуществляется следующим образом:

– установите контекст взаимодействия, будь то система, подсистема, операция, класс или один из сценариев варианта использования либо кооперативные диаграммы.

– определите сцену для взаимодействия, выяснив, какие объекты принимают в нем участие.

– разместите их на диаграмме последовательностей слева направо так, чтобы более важные объекты были расположены левее.

– проведите для каждого объекта линию жизни. Чаще всего объекты существуют на протяжении всего взаимодействия. Для тех же объектов, которые создаются или уничтожаются в ходе взаимодействия, явно отметьте на линиях жизни моменты рождения и смерти с помощью подходящих стереотипных сообщений.

– начав с сообщения, инициирующего взаимодействие, расположите все последующие сообщения сверху вниз между линиями жизни объектов. Если необходимо объяснить семантику взаимодействия, покажите свойства каждого сообщения (например, его параметры).

– если требуется показать вложенность сообщений или точный промежуток времени, когда происходят вычисления, дополните линии жизни объектов фокусами управления.

– если необходимо специфицировать временные или пространственные ограничения, дополните сообщения отметками времени и присоедините соответствующие ограничения.

– для более строгого и формального описания потока управления присоедините к каждому сообщению пред– и постусловия.

– на одной диаграмме последовательностей можно показать только один поток управления (хотя с помощью нотации UML для итераций и ветвлений можно проиллюстрировать простые вариации). Поэтому, как правило, создают несколько диаграмм взаимодействий, одни из которых считаются основными, а другие

описывают альтернативные пути и исключительные условия. Такой набор диаграмм последовательностей можно организовать в пакет, дав каждой диаграмме подходящее имя, отличающее ее от остальных.

– рассмотрите объекты, существующие в контексте системы, подсистемы, операции или класса. Рассмотрите также объекты и роли, принимающие участие в варианте использования или кооперативной диаграмме. Для моделирования потока управления, проходящего через эти объекты и роли, применяются диаграммы взаимодействий – при этом, чтобы показать передачу сообщений в контексте данной структуры, используют их разновидность – кооперативные диаграммы.

Моделирование структурной организации потоков управления состоит из следующих шагов:

– установите контекст взаимодействия. Это может быть система, подсистема, операция, класс или один из сценариев варианта использования либо кооперативная диаграмма.

– определите сцену для взаимодействия, выяснив, какие объекты принимают в нем участие. Разместите их на кооперативной диаграмме в виде вершин графа так, чтобы более важные объекты оказались в центре диаграммы, а их соседи – по краям.

– определите начальные свойства каждого из этих объектов. Если значения атрибутов, помеченные значения, состояния или роли объектов изменяются во время взаимодействия, поместите на диаграмму дубликаты с новыми значениями и соедините их сообщениями со стереотипами become и copy, сопроводив их соответствующими порядковыми номерами.

– детально опишите связи между объектами, вдоль которых передаются сообщения. Для этого:

– сначала нарисуйте связи–ассоциации. Они наиболее важны, поскольку представляют структурные соединения;

– после этого нарисуйте остальные связи, дополнив их соответствующими стереотипами пути (такими, как global или local), чтобы явным образом показать, как объекты связаны друг с другом.

– начав с сообщения, инициирующего взаимодействие, присоедините все последующие сообщения к соответствующим связям, задав порядковые номера. Вложенность показывайте с помощью нотации Дьюи.

– если требуется специфицировать временные или пространственные ограничения, дополните сообщения отметками времени и присоедините нужные ограничения.

– если требуется описать поток управления более формально, присоедините к каждому сообщению пред– и постусловия.

Как и в случае диаграмм последовательностей, на одной кооперативной диаграмме можно показать только один поток управления (хотя нотация UML для итераций и ветвлений помогает проиллюстрировать простые вариации). Поэтому, как правило, создают несколько диаграмм взаимодействий, одни из которых считаются основными, а другие описывают альтернативные пути и

исключительные условия. Такие наборы кооперативных диаграмм можно организовать в пакеты, дав каждой диаграмме подходящее имя, отличающее ее от остальных.

Создавая диаграммы взаимодействий в UML, помните, что и диаграммы последовательностей, и кооперативные диаграммы являются проекциями динамических аспектов системы на одну и ту же модель. Ни одна диаграмма взаимодействий, взятая в отдельности, не может охватить все динамические аспекты. Для моделирования динамики системы в целом, равно как и ее подсистем, операций, классов, вариантов использования и кооперативных диаграмм, лучше использовать сразу несколько диаграмм взаимодействий.

Хорошо структурированная диаграмма взаимодействий обладает следующими свойствами:

акцентирует внимание только на одном аспекте динамики системы;

содержит только такие варианты использования и актеры, которые важны для понимания этого аспекта;

содержит только такие детали, которые соответствуют данному уровню абстракции, и только те дополнения, которые необходимы для понимания системы;

не настолько лаконична, чтобы ввести читателя в заблуждение относительно важных аспектов семантики.

6.6. Диаграммы состояний

Диаграмма состояний (Statechart diagram) показывает автомат, фокусируя внимание на потоке управления от состояния к состоянию. Диаграмма состояний изображается в виде графа с вершинами и ребрами.

Автомат (State machine) – это описание последовательности состояний, через которые проходит объект на протяжении своего жизненного цикла, реагируя на события, – в том числе описание реакций на эти события.

На диаграмме состояний отображают жизненный цикл одного объекта (или класса объектов), начиная с момента его создания и заканчивая разрушением. Поэтому с помощью таких диаграмм удобно моделировать динамику поведения объекта класса. Например, объект Сотрудник может быть нанят, уволен, проходить испытательный срок, находиться в отпуске или в отставке. В каждом из этих состояний объект класса Сотрудник может вести себя по–разному.

Диаграмму состояний не требуется создавать для каждого класса. Многие проекты вообще обходятся без них. Если динамика класса важна, то для него полезно создать диаграмму состояний. Такие классы обычно имеют много различных состояний, то есть поведение объекта такого класса существенно, и есть смысл его моделировать. Поведение таких классов, естественно, управляется событиями.

Два основных элемента диаграммы состояний – это состояния и переходы между ними.

Состояние (State) – это ситуация в жизни объекта, на протяжении которой он удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает какого–то события.

Состоянием называется одно из возможных условий, в которых может находиться объект класса между двумя событиями. На языке UML состояние изображают в виде прямоугольника с закругленными краями (рис. 6.48.).

Рис. 6.48. Состояние

Например, телефон находится в состоянии ожидания, если трубка положена.

Имя состояния должно быть уникальным в своем классе. На значках некоторых состояний иногда полезно указать действия (детали), которые ассоциированы с этими состояниями объекта (это данные, связанные с состоянием).

Действие (Action) – это атомарное вычисление, которое приводит к смене состояния или возврату значения.

Действия могут быть трех типов: входные, выходные и деятельность. Рассмотрим типы действий позднее.

Переход (Transition) – это отношение между двумя состояниями, показывающее, что объект, находящийся в первом состоянии, должен выполнить некоторые действия и перейти во второе состояние, как только произойдет определенное событие и будут выполнены заданные условия.

Переходом называется перемещение объекта из одного состояния в другое. Переход изображается в виде линии со стрелкой (рис. 6.49.).

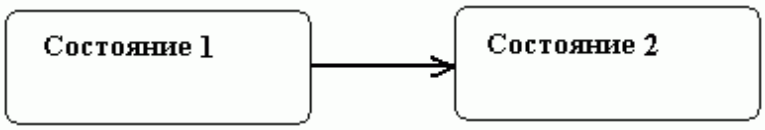


Рис. 6.49. Переход

Переходы могут быть рефлексивными (рис. 6.50.), то есть объект переходит в то же состояние, в котором он находится.

Рис. 6.50. Рефлексный переход

На переходах можно записывать события и действия (рис. 6.51.).

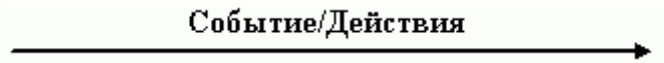


Рис. 6.51. Переход с событием и действием

Событие (Event) – это спецификация существенного факта, который происходит во времени и пространстве. В контексте автоматов событие – это стимул, способный вызвать срабатывание перехода.

Событие – определяет условие, когда переход может быть выполнен. Событием может быть объект (класс) или операция (чаще всего). Событие можно описать обычной фразой. Например: “Выполнить заказ ”, “Отменить заказ”.

Если событием является операция, то у нее могут быть аргументы.

Большинство переходов должно иметь события, так как именно они инициируют переход. Тем не менее, бывают и автоматические переходы, которые не имеют событий, то есть объект сам перемещается из одного состояния в другое согласно действиям, записанным внутри значка состояния: деятельность, входные или выходные.

После событий в квадратных скобках может быть записано ограждающее условие, которое определяет, когда переход выполняется, а когда нет. Их задавать необязательно. Чаще используются для автоматических переходов (когда на переходе не указано событие), особенно если их несколько из одного состояния.

Действие – это непрерываемое поведение, которое выполняется во время перехода. Эти действия не должны выполняться при входе или выходе из состояния (входные и выходные действия показываются внутри состояния). Действием перехода обычно является или вызов метода, или порождение другого объекта (события), или останов процесса.

Например, при переходе объекта "Счёт" из открытого в закрытое состояние должно выполняться действие "Сохранить дату закрытия счёта" (рис. 6.52.).

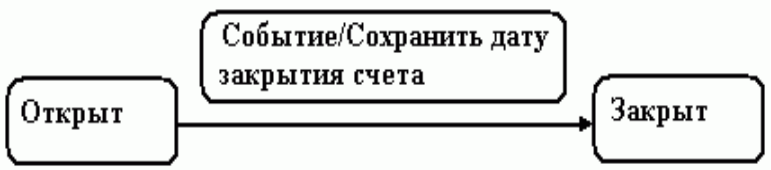


Рис. 6.52. Переход объекта из открытого в закрытое состояние

Такие действия размещают вдоль линии перехода после имени события (ему предшествует косая черта “/”).

Рассмотрим более подробно детали (действия), которые объект может выполнять, находясь в конкретном состоянии. Итак, с состоянием можно связать данные (операции) трех типов:

деятельность;

входное действие;

выходное действие.

Деятельность (Activity) – это продолжающееся неатомарное вычисление внутри автомата.

Деятельностью называется поведение, которое реализует объект, находясь в данном состоянии. Деятельность может быть прервана переходом объекта в другое состояние, а может, выполняться до конца в данном состоянии, но до перехода. Обычно деятельности предшествует слово do (рис. 6.53.).

Рис. 6.53. Деятельность

Деятельность может выполняться также в результате получения объектом некоторого сообщения (операции).

Входное действие – это поведение, которое всегда выполняется, когда объект переходит в данное состояние (независимо, из какого состояния). Входное действие осуществляется не после того, как объект перешел в состояние, а скорее как часть перехода в данное состояние. И поэтому оно рассматривается как непрерываемое. Ему на состоянии предшествует слово entry (рис. 6.54.).

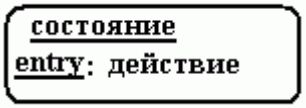


Рис. 6.54. Входное действие

Выходное действие – это действие подобно входному действию, но осуществляется как составная часть процесса выхода из состояния, независимо от того, куда объект переходит (в какое состояние), то есть выходное действие является частью процесса перехода объекта в другое состояние и является поэтому непрерываемым. Ему предшествует слово exit (рис. 6.55.).

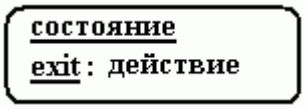


Рис. 6.55. Выходное действие

Поведение объекта во время деятельности, входных и выходных действий может включать в себя отправку сообщения другому объекту. Например, do: ^цель.Операция (Аргументы), где знак "^" указывает, что объект в данном состоянии отправляет сообщение объекту «цель».

На диаграмме состояний обычно указывают два специальных состояния объекта – начальное и конечное (Start и Stop).

Начальное состояние – это то состояние, в котором объект находится сразу после своего создания. Изображается в виде закрашенного кружочка (рис. 6.56.).

Рис. 6.56. Начальное состояние

От него проводится переход к первоначальному состоянию. Начальное состояние должно быть обязательно на диаграмме, и только одно.

Конечное состояние – это то состояние, в котором объект находится непосредственно перед уничтожением. Его изображают в виде значка «бычий глаз» (рис. 6.57.).

Рис. 6.57. Конечное состояние

Конечное состояние не является обязательным, и их может быть сколько угодно.

Для упорядочивания состояний на диаграмме их можно вкладывать друг друга. Вложенные состояния называются подсостояниями (Substates), а те, в которые они вложены, – суперсостояниями (Superstates).

Состояния обычно вкладываются (группируются), если у них имеются идентичные переходы. Тогда есть смысл сгруппировать их вместе в суперсостояние, которое будет поддерживать одинаковые переходы вложенных состояний. На диаграмме состояний будет меньше переходов, то есть говорят «навести порядок» на диаграмме.

От начального состояния можно провести переход к любому подсостоянию, и от любого подсостояния или суперсостояния можно добавить переход в конечное состояние. Может быть несколько уровней вложенности состояний.

6.6.1. Составное состояние и подсостояние

Моделирование сложных объектов и систем, как правило, связано с многоуровневым представлением их состояний. В этом случае возникает необходимость детализировать отдельные состояния, сделав их составными.

Составное состояние (Composite state) – сложное состояние, которое состоит из других вложенных в него состояний.

Составное состояние называют также состоянием–композитом. Вложенные состояния выступают по отношению к составному состоянию как **подсостояния (Substate)**. И хотя между ними имеет место отношение композиции, графически все вершины диаграммы, которые соответствуют вложенным состояниям, изображаются внутри символа составного состояния (рис. 6.58.). В этом случае размеры графического символа составного состояния увеличиваются, так чтобы вместить в себя все подсостояния.

Составное состояние может содержать или несколько последовательных подсостояний, или несколько параллельных конечных подавтоматов. Каждое состояние–композит может уточняться только одним из указанных способов. При этом любое из подсостояний, в свою очередь, может быть состоянием–композитом

и содержать внутри себя другие вложенные подсостояния. Количество уровней вложенности составных состояний в языке UML не фиксировано.

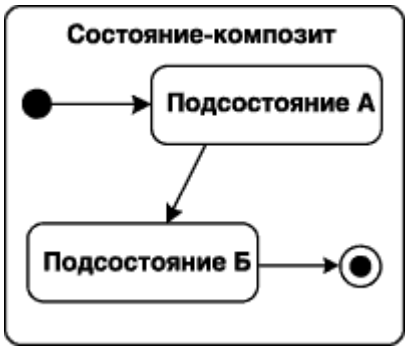


Рис. 6.58. Графическое представление составного состояния с двумя вложенными в него последовательными подсостояниями

6.6.1.1. Последовательные подсостояния

Последовательные подсостояния (Sequential substates) – вложенные состояния состояния–композита, в рамках которого в каждый момент времени объект может находиться в одном и только одном подсостоянии.

Поведение объекта в этом случае представляет собой последовательную смену подсостояний, от начального до конечного. Моделируемый объект или система продолжает находиться в составном состоянии, тем не менее, введение в рассмотрение последовательных подсостояний позволяет учесть более тонкие логические аспекты его внутреннего поведения.

В качестве примера моделируемой системы стоит рассмотреть обычный телефонный аппарат. Он может находиться в различных состояниях, в частности в состоянии дозвона до абонента. Очевидно, для того чтобы позвонить, необходимо снять телефонную трубку, услышать тоновый сигнал, после чего набрать нужный телефонный номер. Таким образом, состояние дозвона до абонента является составным и состоит из двух последовательных подсостояний: Телефонная трубка поднята и Набор телефонного номера. Фрагмент диаграммы состояний для этого примера содержит одно состояние–композит, которое состоит из двух последовательных подсостояний рис. 6.59.

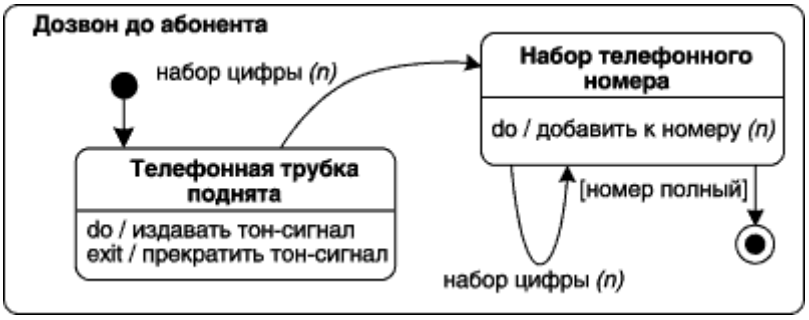


Рис. 6.59. Пример составного состояния с двумя вложенными последовательными подсостояниями

Некоторых пояснений могут потребовать переходы.

Два из них специфицируют событие–триггер, которое имеет имя: набор цифры(n) с параметром n. В качестве параметра, как нетрудно предположить, выступает отдельная цифра на диске телефонного аппарата. Переход из начального подсостояния не содержит никакой строки текста. Последний переход в конечное подсостояние также не имеет события–триггера, но имеет сторожевое условие, проверяющее полноту набранного номера абонента. Только в случае истинности этого условия телефонный аппарат может перейти в конечное состояние для состояния–композита Дозвон до абонента.

Каждое составное состояние должно содержать в качестве вложенных состояний начальное и конечное состояния. При этом начальное подсостояние является исходным, когда происходит переход объекта в данное составное состояние. Если составное состояние содержит внутри себя конечное состояние, то переход в это вложенное конечное состояние означает завершение нахождения объекта в данном составном состоянии. Важно помнить, что для последовательных подсостояний начальное и конечное состояния должны быть единственными в каждом составном состоянии.

Это можно объяснить следующим образом. Каждая совокупность вложенных последовательных подсостояний представляет собой конечный подавтомат того конечного автомата, которому принадлежит рассматриваемое составное состояние. Поскольку каждый конечный автомат может иметь по определению единственное начальное и единственное конечное состояния, то для любого его конечного подавтомата это условие также должно выполняться.

6.6.1.2. Параллельные подсостояния

Параллельные подсостояния (Concurrent substates) – вложенные состояния, используемые для спецификации двух и более конечных подавтоматов, которые могут выполняться параллельно внутри составного состояния.

Каждый из конечных подавтоматов занимает некоторую графическую область внутри составного состояния, которая отделяется от остальных горизонтальной пунктирной линией. Если на диаграмме состояний имеется составное состояние с вложенными параллельными подсостояниями, то объект может одновременно находиться в каждом из этих подсостояний.

Отдельные параллельные подсостояния могут, в свою очередь, состоять из нескольких последовательных подсостояний (рис. 6.60.). В этом случае по определению моделируемый объект может находиться только в одном из последовательных подсостояний каждого подавтомата. Таким образом, для фрагмента диаграммы состояний (рис. 6.60.) допустимо одновременное нахождение объекта только в следующих подсостояниях: (А, В, Г), (Б, В, Г), (А, В, Д), (Б, В, Д).

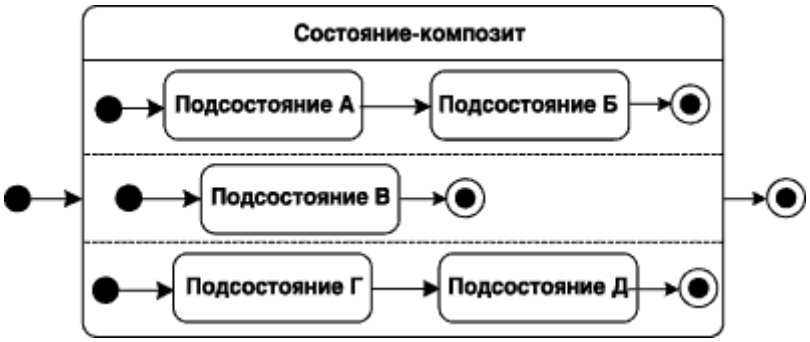


Рис. 6.60. Графическое изображение состояния–композита с вложенными параллельными подсостояниями

6.6.1.3. Несовместимые подсостояния

Несовместимое подсостояние (Disjoint substate) – подсостояние, в котором подсистема не может находиться одновременно с другими подсостояниями одного и того же составного состояния.

В этом контексте недопустимо нахождение объекта одновременно в несовместимых подсостояниях (А, Б, В) или (В, Г, Д).

Поскольку каждый регион вложенного состояния специфицирует некоторый конечный подавтомат, то для каждого из вложенных конечных подавтоматов могут быть определены собственные начальное и конечное состояния (рис. 6.61.). При переходе в данное составное состояние каждый из конечных подавтоматов оказывается в своем начальном состоянии. Далее происходит параллельное выполнение каждого из этих конечных подавтоматов, причем выход из составного состояния будет возможен лишь в том случае, когда все конечные подавтоматы будут находиться в своих конечных состояниях. Если какой–либо из конечных подавтоматов пришел в свое финальное состояние раньше других, то он должен ожидать, пока и другие подавтоматы не придут в свои финальные состояния.

В некоторых случаях бывает желательно скрыть внутреннюю структуру составного состояния.

Например, отдельный конечный подавтомат, специфицирующий составное состояние, может быть настолько большим по масштабу, что его визуализация затруднит общее представление диаграммы состояний. В подобной ситуации допускается не раскрывать на исходной диаграмме состояний данное составное состояние, а указать в правом нижнем углу специальный символ–пиктограмму (рис. 6.61.). В последующем диаграмма состояний для соответствующего конечного подавтомата может быть изображена отдельно от основной диаграммы с необходимыми комментариями.

Рис. 6.61. Составное состояние со скрытой внутренней структурой и специальной пиктограммой

6.6.2. Исторические состояния

Обычный конечный автомат не позволяет учитывать предысторию в процессе моделирования поведения систем и объектов. Однако функционирование ряда систем основано на возможности выхода из отдельного состояния–композита с последующим возвращением в это же состояние. Может оказаться необходимым учесть ту часть деятельности, которая была выполнена на момент выхода из этого состояния–композита, чтобы не начинать ее выполнение сначала. Для этой цели в языке UML существует историческое состояние.

Историческое состояние (History state) – псевдосостояние, используемое для запоминания того из последовательных подсостояний, которое было текущим в момент выхода из составного состояния.

Историческое состояние применяется только в контексте составного состояния. При этом существует две разновидности исторического состояния: неглубокое или недавнее и глубокое или давнее (рис. 6.62.).

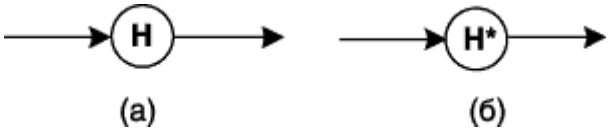


Рис. 6.62. Графическое изображение недавнего (а) и давнего (б) исторического состояния

Неглубокое историческое состояние (Shallow history state) обозначается в форме небольшой окружности, в которую помещена латинская буква "H" (рис. 6.62, а). Это состояние обладает следующей семантикой. Во–первых, оно является первым подсостоянием в составном состоянии, и переход извне в рассматриваемое составное состояние должен вести непосредственно в данное историческое состояние. Во–вторых, при первом попадании в неглубокое историческое состояние оно не хранит никакой истории. Другими словами, при первом переходе в недавнее историческое состояние оно заменяет собой начальное состояние соответствующего конечного подавтомата.

Далее могут последовательно изменяться вложенные подсостояния. Если в некоторый момент происходит выход из составного состояния (например, в случае наступления некоторого события), то рассматриваемое историческое состояние запоминает то из подсостояний, которое было текущим на момент выхода из данного составного состояния. При последующем входе в это составное состояние неглубокое историческое подсостояние имеет непустую историю и сразу отправляет конечный подавтомат в запомненное подсостояние, минуя все предшествующие ему подсостояния.

Историческое состояние теряет свою историю в тот момент, когда конечный подавтомат доходит до своего конечного состояния. При этом неглубокое историческое состояние запоминает историю только того конечного подавтомата, к которому оно относится. Другими словами, этот тип псевдосостояния способен запомнить историю только одного с ним уровня вложенности.

Если запомненное подсостояние также является составным состоянием, а при выходе из исходного составного состояния необходимо запомнить подсостояние второго уровня вложенности, то в этом случае следует воспользоваться более сильным псевдосостоянием – **глубоким историческим состоянием.**

Глубокое историческое состояние (Deep history state или состояние глубокой истории) также обозначается в форме небольшой окружности, в которую помещена латинская буква "H" с дополнительным символом "*" (рис. 6.62, б) и служит для запоминания всех подсостояний любого уровня вложенности для исходного составного состояния.

6.6.3. Сложные переходы и псевдосостояния

Рассмотренное выше понятие перехода вполне достаточно для большинства типичных расчетно–аналитических задач. Однако современные программные системы могут реализовывать сложную логику поведения отдельных своих компонентов. Иногда для адекватного представления процесса изменения состояний семантика обычного перехода для них недостаточна. С этой целью в языке UML специфицированы дополнительные обозначения и свойства, которыми могут обладать отдельные переходы на диаграмме состояний.

В отдельных случаях возникает необходимость явно показать ситуацию, когда переход может иметь несколько исходных состояний или целевых состояний. Такой переход получил название – **параллельный переход.** Введение в рассмотрение параллельных переходов может быть обусловлено необходимостью синхронизировать и/или разделить отдельные процессы управления на параллельные нити без спецификации дополнительной синхронизации в параллельных конечных подавтоматах.

Графически такой переход изображается вертикальной черточкой, аналогично обозначению перехода в известном формализме сетей Петри. Если параллельный переход имеет две или более исходящих из него дуг (рис. 6.63, а), то его называют разделением (fork). Если же он имеет две или более входящие дуги (рис. 6.63, б), то

его называют слиянием (join). Текстовая строка спецификации параллельного перехода записывается рядом с черточкой и относится ко всем входящим или исходящим дугам.

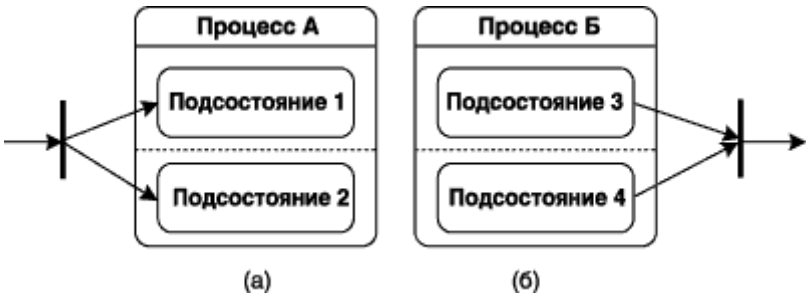


Рис. 6.63. Графическое изображение перехода–разделения в параллельные подсостояния (а) и перехода–слияния из параллельных подсостояний (б)

Срабатывание параллельного перехода происходит следующим образом. В первом случае происходит разделение составного конечного автомата на два конечных подавтомата, образующих параллельные ветви вложенных подпроцессов. При этом после срабатывания перехода–разделения моделируемая система или объект одновременно будет находиться во всех целевых подсостояниях этого параллельного перехода (подсостояния 1 и 2). Далее процесс изменения состояний будет протекать согласно ранее рассмотренным правилам для составных состояний.

Во втором случае переход–слияние срабатывает, если имеет место событие–триггер для всех исходных состояний этого перехода, и выполнено (при его наличии) сторожевое условие. При срабатывании перехода–слияния одновременно покидаются все исходные подсостояния перехода (подсостояния 3 и 4) и происходит переход в целевое состояние. При этом каждое из исходных подсостояний перехода должно принадлежать отдельному конечному подавтомату, входящему в составной конечный автомат (процессу Б).

Переход, стрелка которого соединена с границей составного состояния, обозначает переход в это составное состояние (переход а на рис. 6.64.). Он эквивалентен переходу в начальное состояние каждого из конечных подавтоматов (единственному на рис. 6.64.), входящих в состав данного состояния–композита. Переход f, выходящий из составного состояния (рис. 6.64.), относится к каждому из вложенных состояний. Это означает, что моделируемая система или объект при наступления события f может покинуть данное составное состояние, находясь в любом из его вложенных состояний В и Г.

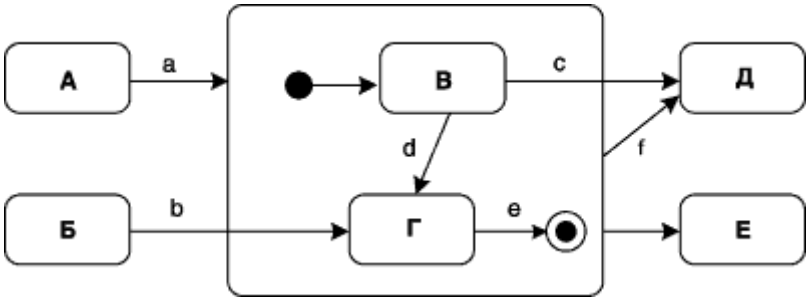


Рис. 6.64. Различные варианты переходов в составное состояние и из составного состояния

Иногда желательно реализовать ситуацию, когда выход из отдельного вложенного состояния соответствовал бы также выходу из составного состояния. В этом случае изображают переход, который непосредственно выходит из вложенного состояния и пересекает границу состояния–композита (переход С на рис. 6.64.). Аналогично, допускается изображение переходов, входящих извне состояния–композита в отдельное вложенное состояние (переход b на рис. 6.64.).

Переход d является внутренним для рассматриваемого состояния–композита и никак не влияет на выход из состояния–композита. Выход из данного составного состояния также возможен при наступлении события e, которое приводит в его конечное состояние, а из него – в состояние Е, находящееся вне данного состояния–композита.

В общем случае поведение параллельных конечных подавтоматов происходит независимо друг от друга, что позволяет, например, моделировать многозадачность в программных системах. Однако в отдельных ситуациях может возникнуть необходимость учесть в модели синхронизацию наступления отдельных событий и срабатывание соответствующих переходов. Для этой цели в языке UML имеется псевдосостояние, которое называется синхронизирующим состоянием или состоянием синхронизации.

6.6.4. Состояние синхронизации

Состояние синхронизации (Synch state) – псевдосостояние в конечном автомате, которое используется для синхронизации параллельных областей конечного автомата.

Синхронизирующее состояние обозначается небольшой окружностью, внутри которой помещен символ звездочки "*". Оно используется совместно с переходом–слиянием или переходом–разделением для того, чтобы явно указать события в других конечных подавтоматах, оказывающие непосредственное влияние на поведение данного подавтомата.

Так, например, при включении компьютера с некоторой сетевой операционной системой параллельно начинается выполнение нескольких процессов. В частности, происходит проверка пароля пользователя и запуск различных служб. При этом работа пользователя на компьютере станет возможной только в случае успешной его аутентификации, в противном случае компьютер может быть выключен. Рассмотренные особенности синхронизации этих параллельных процессов учтены на соответствующей диаграмме состояний с помощью синхронизирующего состояния (рис. 6.65.).

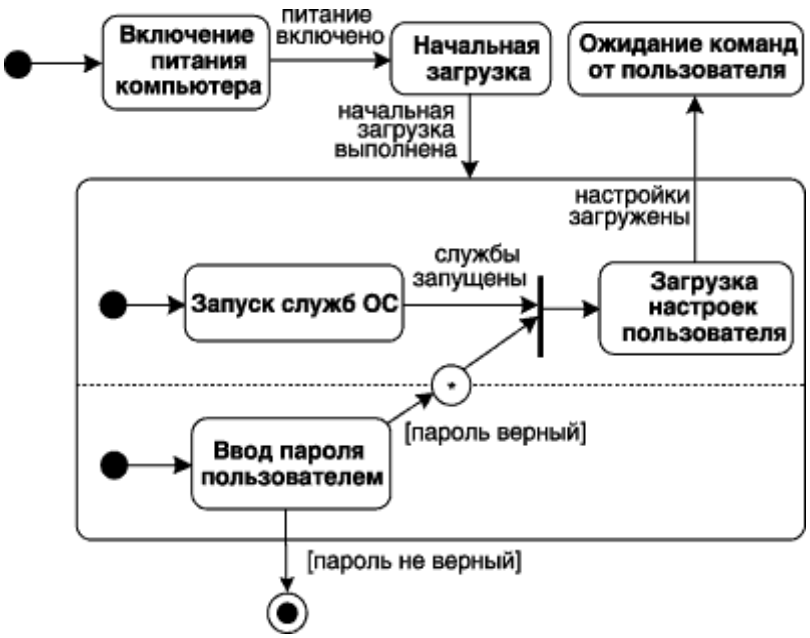


Рис. 6.65. Диаграмма состояний для примера включения компьютера

6.6.5. Рекомендации по построению диаграмм состояний

По своему назначению диаграмма состояний не является обязательным представлением в модели и как бы "присоединяется" к тому элементу, который, по замыслу разработчиков, имеет нетривиальное поведение в течение своего жизненного цикла. Наличие у системы нескольких состояний, отличающихся от простой дихотомии "исправен – неисправен", "активен – неактивен", "ожидание – реакция на внешние действия", уже служит признаком необходимости построения диаграммы состояний. В качестве начального варианта диаграммы

состояний, если нет очевидных соображений по поводу состояний объекта, можно воспользоваться подобными состояниями, в качестве составных, уточняя их (детализируя их внутреннюю структуру) по мере рассмотрения логики поведения моделируемой системы или объекта.

При выделении состояний и переходов следует помнить, что длительность срабатывания отдельных переходов должна быть существенно меньшей, чем нахождение моделируемых элементов в соответствующих состояниях. Каждое из состояний должно характеризоваться определенной устойчивостью во времени.

Другими словами, из каждого состояния на диаграмме не может быть самопроизвольного перехода в какое бы то ни было другое состояние. Все переходы должны быть явно специфицированы, в противном случае построенная диаграмма состояний является либо неполной (неадекватной), либо ошибочной с точки зрения нотации языка UML (ill formed).

При разработке диаграммы состояний нужно постоянно следить, чтобы объект в каждый момент мог находиться только в единственном состоянии. Если это не так, то данное обстоятельство может быть как следствием ошибки, так и неявным признаком наличия параллельности поведения у моделируемого объекта. В последнем случае следует явно специфицировать необходимое число конечных подавтоматов, вложив их в то составное состояние, которое характеризуется нарушением условия одновременности.

Следует произвести обязательную проверку, чтобы никакие два перехода из одного состояния не могли сработать одновременно. Другими словами, необходимо выполнить требование отсутствия конфликтов у всех переходов, выходящих из одного и того же состояния. Наличие такого конфликта может служить признаком ошибки, либо параллельности или ветвления рассматриваемого процесса. Если параллельность по замыслу разработчика отсутствует, то следует ввести дополнительные сторожевые условия либо изменить существующие, чтобы исключить конфликт переходов. При наличии параллельности следует заменить конфликтующие переходы одним параллельным переходом типа ветвления.

Использование исторических состояний оправдано в том случае, когда необходимо организовать обработку исключительных ситуаций (прерываний) без потери данных или выполненной работы. При этом применять исторические состояния, особенно глубокие, необходимо с известной долей осторожности. Нужно помнить, что каждый из конечных подавтоматов может иметь только одно историческое состояние. В противном случае возможны ошибки, особенно, когда подавтоматы изображаются на отдельных диаграммах состояний.

6.6.6. Примеры диаграмм состояний

Пример: Ниже приведен пример диаграммы состояний (простая форма) для класса Course (Учебный курс) (рис. 6.66.). Курс может быть открыт, закрыт, отменен или завершен

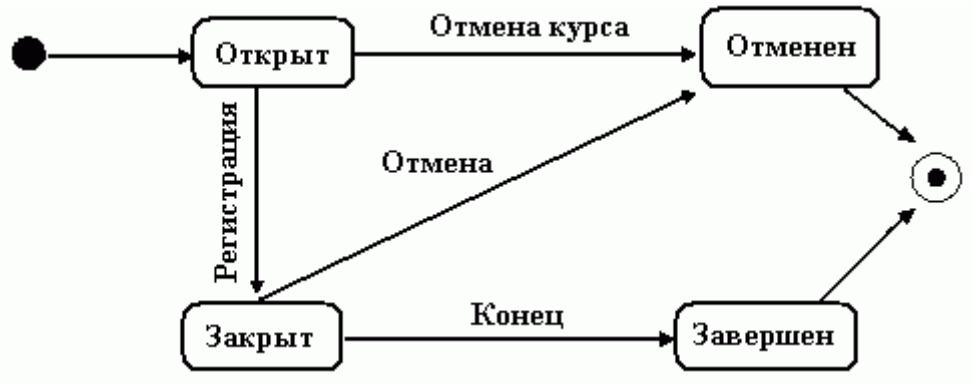


Рис. 6.66. Пример диаграммы состояний

Из этой диаграммы состояний видно, что состояния Открыт и Закрыт можно поместить в одно суперсостояние (например, Активный курс). Станет проще и понятнее, да и переходов будет меньше.

Пример: Показать диаграмму состояний для объектов класса Account (Счет) системы банковского автомата (ATM). Счет может находиться в разных состояниях (открыт, закрыт, превышен), и в каждом из них ведет себя по–разному (рис. 6.67.)

Рис. 6.67. Диаграмма состояний для объектов банковского автомата

Пример: Изобразить диаграмму состояний для телефона (рис. 6.68.)

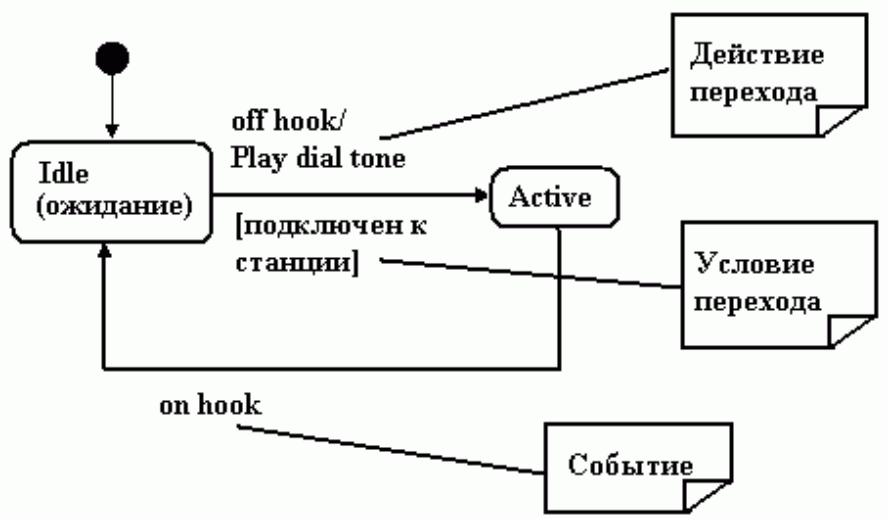


Рис. 6.68. Диаграмма состояний для телефона

Только что включенный в сеть телефон находится в начальном состоянии, то есть его предыдущее поведение несущественно. В начальном состоянии телефон готов к тому, чтобы можно было позвонить или принять звонок. Если поднять трубку, то телефон перейдет в новое состояние: готовности к набору номера. И в этом состоянии мы не ожидаем, что телефон зазвонит. Если кто–то наберет ваш номер, и телефон находится в начальном состоянии, то при поднятии трубки телефон перейдет в новое состояние: состояние с установленным соединением.

Пример: Создать диаграмму состояний для класса Заказ. Заказ может быть выполнен, отменен, выполнение заказа приостановлено (остаются не заказанные позиции в заказе) и при инициализации объекта “Заказ” сохранить дату заказа (входное действие), добавить к заказу новые позиции (деятельность) и т.д. (рис. 6.69.)

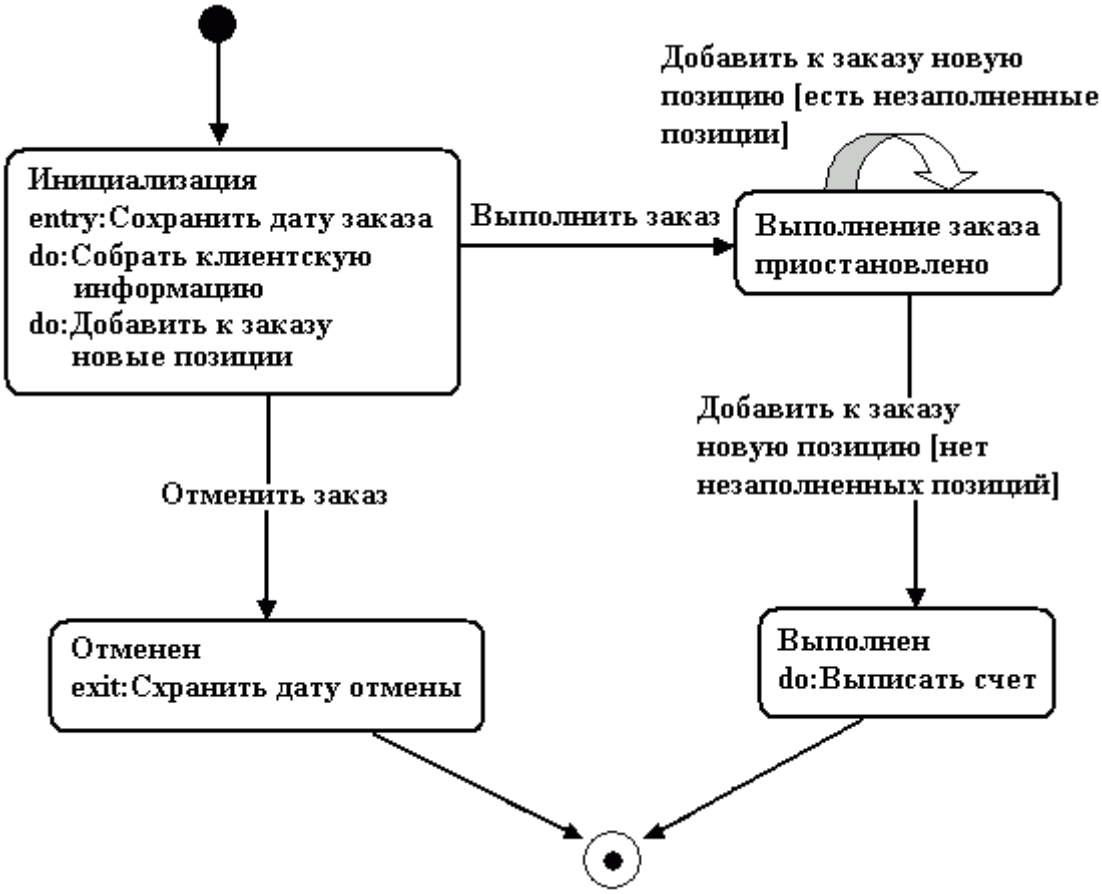


Рис. 6.69. Диаграмма состояний для класса Заказ

6.7. Диаграммы деятельностей

Диаграммы деятельностей (Activity diagram) используются для описания поведения систем.

В отличие от большинства других средств UML диаграммы деятельностей не имеют отношения к работам «троих друзей». Диаграммы деятельностей сочетают в себе идеи различных методов: диаграмм событий Джима Оделла, диаграмм состояний SDL и сетей Петри. Эти диаграммы особенно полезны в сочетании с потоками работ, а также в описании поведения, которое включает параллельные процессы.

Основным элементом диаграммы деятельностей является деятельность. Причем диаграммы деятельностей, как и диаграммы классов, могут строиться с трех различных точек зрения: с концептуальной, с точки зрения спецификации и с точки зрения реализации. В соответствии с точкой зрения деятельность рассматривается по–разному.

На концептуальной диаграмме деятельность – это некоторая задача, которую необходимо автоматизировать или выполнить вручную.

На диаграмме, построенной с точки зрения спецификации или реализации, деятельность – это некоторый метод над классом.

Рассмотрим пример диаграммы деятельностей (рис. 6.70.).

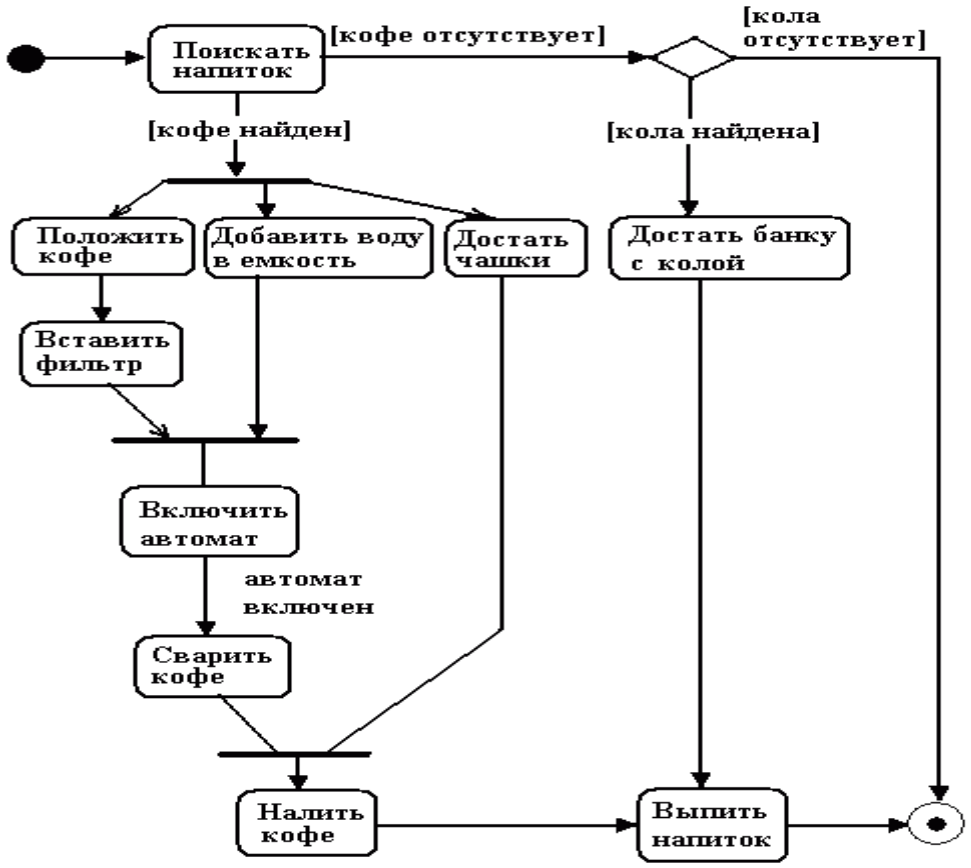


Рис. 6.70. Диаграмма деятельностей

Мы видим на примере, что за каждой деятельностью может следовать другая деятельность. В этом случае они образуют простую последовательность. Например, за деятельностью «Положить Кофе в Фильтр» следует деятельность «Вставить Фильтр в Автомат». При таком подходе диаграмма деятельностей напоминает блок–схему.

Разницу можно обнаружить, если посмотреть на деятельность «Поискать Напиток». Эта деятельность активизирует два действия: каждое действие содержит условие, которое может принимать одно из двух значений: «истина» или «ложь» (так же, как на диаграмме состояний). У нас Личность осуществляет деятельность «Поискать Напиток», выбирая между кофе и колой.

Предположим, что мы отыскали кофе, то есть идем вниз по «кофейному» маршруту. Этот путь ведет к линейке синхронизации, с которой связана активизация трех деятельностей: «Положить Кофе в Фильтр», «Добавить Воду в Ёмкость» и «Достать Чашки». Диаграмма указывает, что эти три деятельности могут выполняться параллельно. Причем порядок их выполнения не играет роли: их можно выполнять в любой последовательности, а можно и чередовать друг с другом. Например, достать одну чашку, затем добавить немного воды в ёмкость;

достать другую чашку, добавить еще немного воды и т.д. А можно делать некоторые вещи одновременно: одной рукой наливать воду, а другой доставать чашки. Любой из этих вариантов допустим. Это и показывает линейка синхронизации, то есть выбор порядка выполнения действий остается за Личностью (за тем, кто выполняет эти действия). В этом заключается главное различие между диаграммой деятельностей и блок–схемой, то есть блок–схемы обычно показывают последовательные процессы, а диаграммы деятельностей могут поддерживать дополнительно параллельные процессы.

Поэтому диаграммы деятельностей являются полезными при параллельном программировании. Можно графически изобразить все ветви и показать, когда их необходимо синхронизировать. Такая возможность важна также при моделировании бизнес – процессов, так как среди бизнес–процессов часто встречаются такие, которые не обязаны выполняться последовательно: диаграмма побуждает людей искать возможности делать дела параллельно.

Итак, если при описании поведения системы выявляются параллельные процессы (деятельности), то их необходимо синхронизировать.

Простая линейка синхронизации (как в примере) показывает, что выходная деятельность («Включать Автомат») активизируется только тогда, когда выполнены обе входные деятельности: «Добавить Воду в Емкость» и «Вставить Фильтр в Автомат». Линейки могут быть и более сложными.

Обратите внимание, что второе решение (первое – «Поискать Напиток») относится к составным. Если кофе нет, то приходим ко второму решению, связанному с колой. При такой последовательности решений второе решение обозначается ромбом. Количество вложенных решений может быть любым.

Далее обратите внимание, что деятельность «Выпить Напиток» имеет два входа. Это означает, что она будет выполнена в любом случае, то есть рассматривается как условие «ИЛИ» (выполняется, если выполняется хотя бы одна из двух входящих деятельностей). Линейку же синхронизации можно рассматривать как условие «И».

Диаграммы деятельностей полезны для описания сложных методов. Их также можно применять где угодно. Например, для описания вариантов использования, если вариант использования представляет собой сложный процесс (функцию).

Например, вариант использования, который связан с обработкой заказа, можно изобразить следующей диаграммой деятельностей (рис. 6.71.).

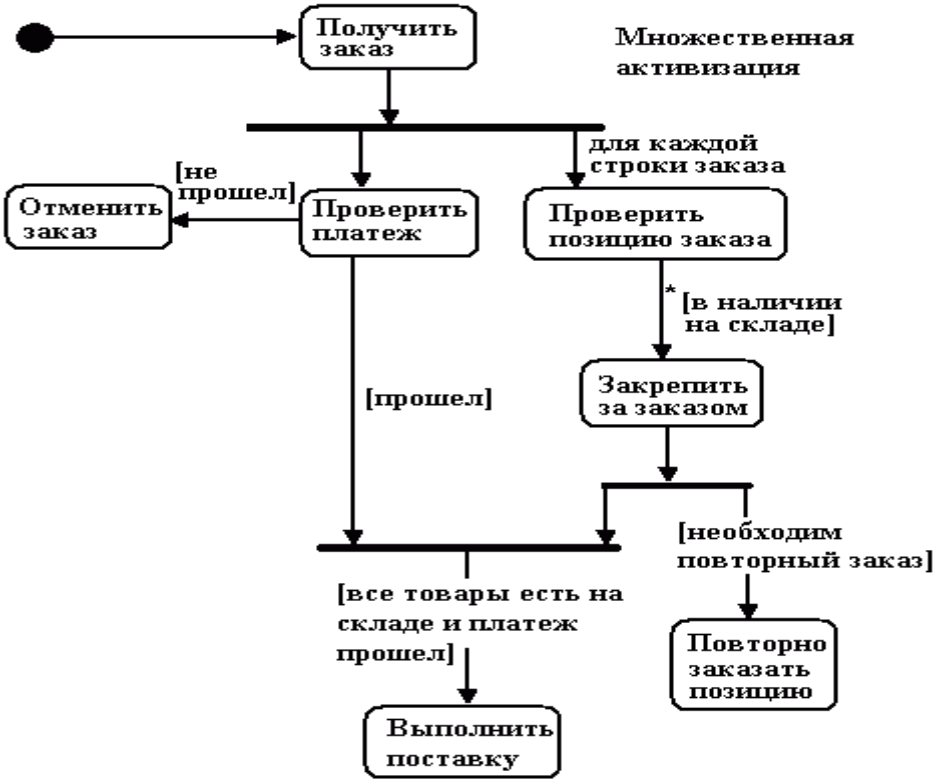


Рис. 6.71. Диаграмма деятельностей, изображающая обработку заказа.

Следующая диаграмма деятельностей может отражать вариант использования «Получение поставки» (рис. 6.72.).

Здесь введена новая конструкция: деятельность «Проверить Позицию Заказа», которая помечена символом «*». Это, так называемый, маркер множественности. Он показывает, что при "получении заказа" должны выполнить деятельность «Проверить Позицию Заказа» для каждой строки заказа. Это означает, что за деятельностью «Получить Заказ» следует один вызов деятельности «Проверить Платеж» и много вызовов деятельности «Проверить Позицию Заказа». Все эти вызовы производятся параллельно.

Этот случай подчеркивает второй источник параллелизма на диаграмме деятельностей. Параллельные деятельности могут быть связаны не только с линейной синхронизацией, но и с множественной активизацией. Для множественной активизации обязательно показывается ее основа. В нашем примере – «для каждой строки заказов».

Как видно из примера, диаграмма деятельностей необязательно включает явно определенную конечную точку (где заканчивается выполнение всех деятельностей). У нас деятельности не заканчиваются вместе.

В нашем примере не сможем выполнить Заказ, пока не пополнится запас на складе (после поставки следующей партии товара). Этой деятельности («Получение Поставки») может соответствовать отдельный вариант использования. Этот вариант использования покажет деятельность, которая связана с ожиданием заказа до получения очередной поставки.

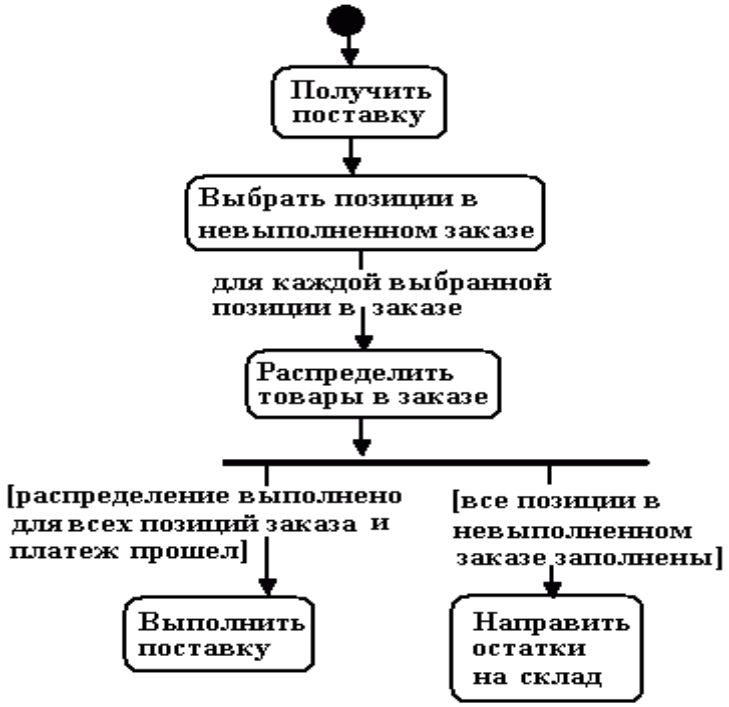


Рис. 6.72. Диаграмма деятельностей, отражающая «Получение поставки»

Если посмотреть две диаграммы деятельностей (рис. 6.71.) и (рис. 6.72.), приведенные выше, то видно, что их можно объединить в одну. У них общая деятельность «выполнить поставку» (рис. 6.73.).

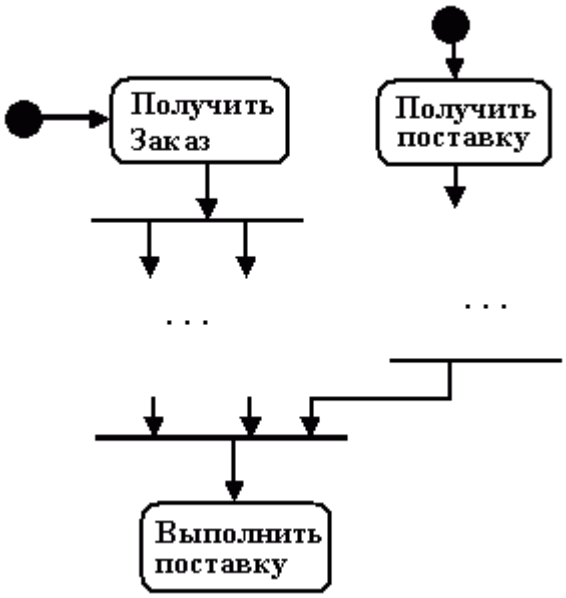


Рис. 6.73. Наложение диаграмм деятельностей

Такое объединение диаграмм называется наложением отдельных диаграмм деятельностей для обоих вариантов использования. Такие диаграммы могут содержать несколько начальных точек. Таким образом, с помощью диаграмм

деятельностей можно одновременно описывать поведение нескольких вариантов использования. Можно отразить поведение системы целиком в одной диаграмме, а можно в нескольких.

Следует отметить, что диаграммы деятельностей отражают происходящие события. Однако они ничего не говорят о том, кто участвует в реализации процессов, то есть диаграмма деятельностей не показывает, какой класс (объект класса) отвечает за выполнение деятельности. Один из способов решить эту проблему – снабдить каждую деятельность меткой соответствующего класса (или человека), а затем дополнить диаграммой взаимодействий (последовательностей), где показать обмен сообщениями между объектами.

Другой способ – это, так называемые, «плавательные дорожки» (рис. 6.74.). В этом случае диаграмма деятельностей разделяется на вертикальные зоны с помощью пунктирных линий. Каждая зона представляет собой зону ответственности каждого класса или конкретного подразделения фирмы. Такой способ позволяет совместить логику диаграмм деятельностей с ответственностью классов, которая отражена на диаграммах взаимодействий. Но построить такую диаграмму не всегда легко, так как надо, чтобы все деятельности попали в свои зоны ответственности.

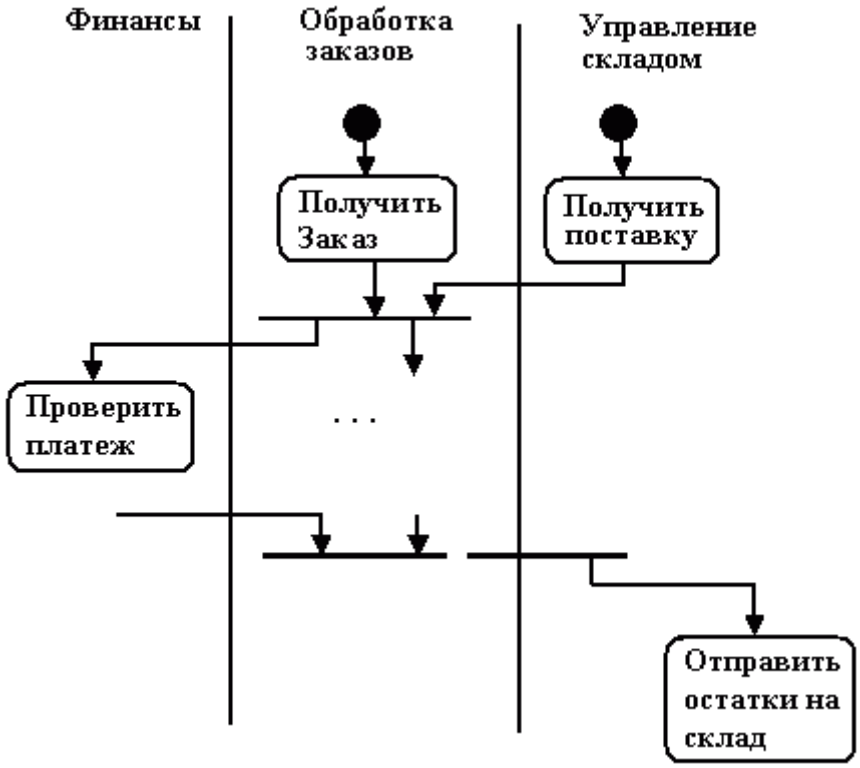


Рис. 6.74. Диаграмма деятельностей, использующая «плавательные дорожки»

Не всегда полезно одновременно заниматься решением двух проблем: строить диаграммы деятельностей и связывать поведение с конкретными объектами. Но в любом случае, важно то, чтобы связать процессы с классами. В противном случае

невозможно построить общую модель – базовую архитектуру программной системы (диаграмма классов, построенная с концептуальной точки зрения).

Наконец, любая деятельность может быть декомпозирована, то есть уточнена. Уточнить деятельность можно либо в виде текста, либо в виде другой диаграммы деятельностей. Сложную логику можно изображать другими удобными средствами. Например, с помощью Р–схемы, или записать на псевдокоде.

Используя диаграмму деятельности, руководствуйтесь следующими принципами:

– дайте диаграмме имя, соответствующее ее назначению;

– начинайте с моделирования главного потока. Ветвления, параллельность и траектории объектов являются второстепенными деталями, которые можно изобразить на отдельной диаграмме;

– располагайте элементы так, чтобы число пересечений было минимальным;

– используйте примечания и закраску, чтобы привлечь внимание к важным особенностям диаграммы.

6.7.1. Примеры диаграмм деятельностей

Пример: Диаграмма описания алгоритма функции «Модификация данных»

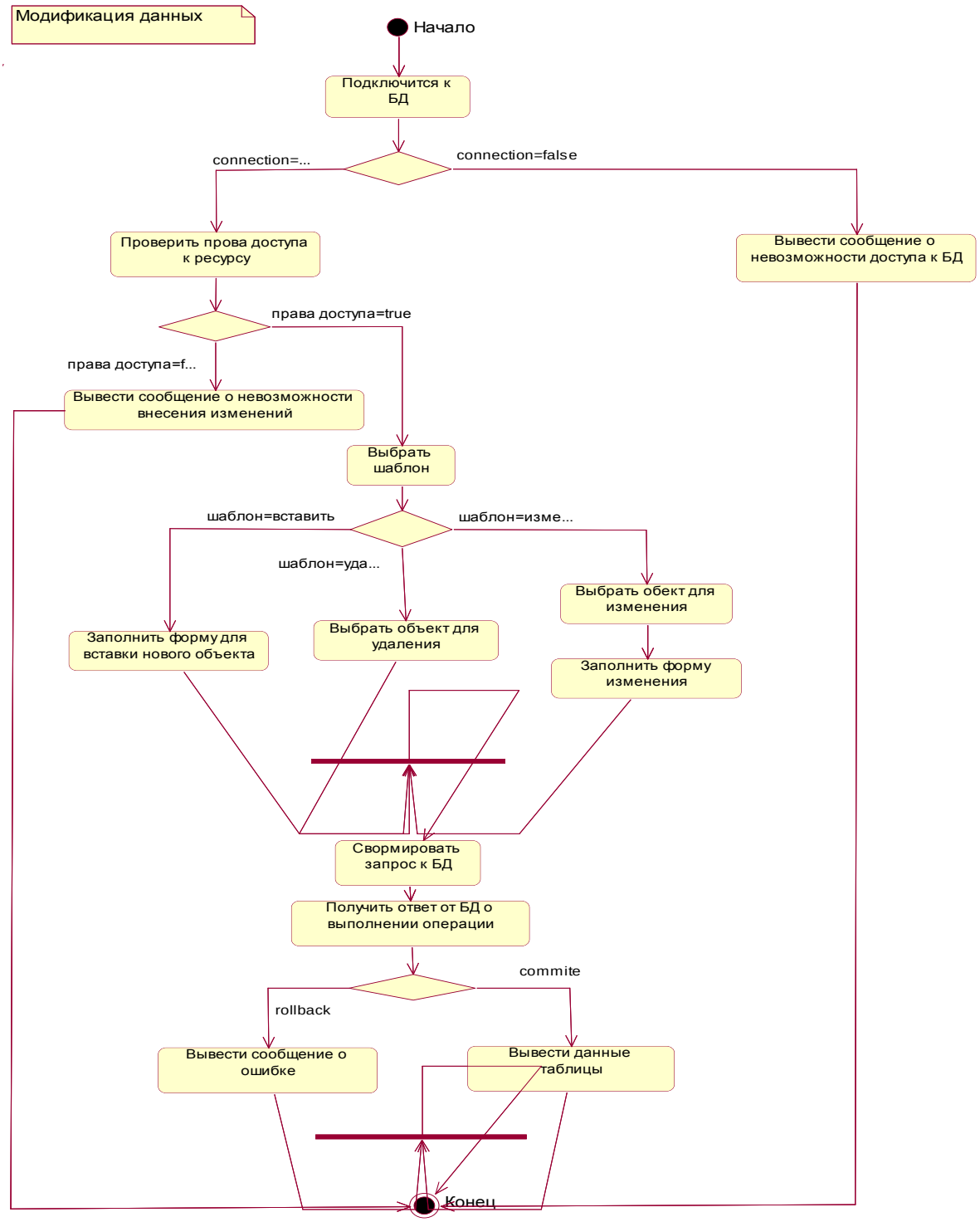


Рис. 6.75. Модификация данных

Пример: Диаграмма описания алгоритма функции «Удалить запись»

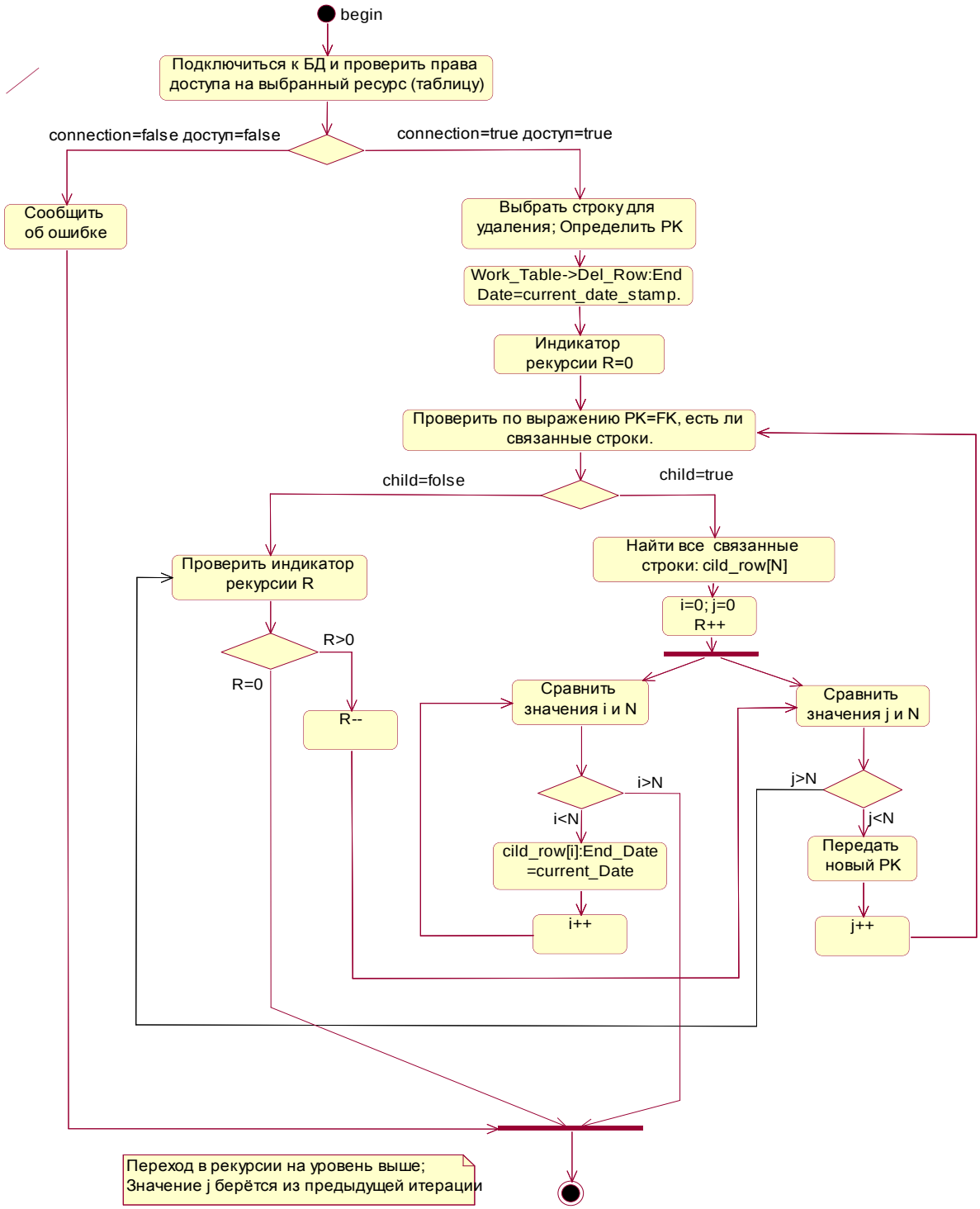


Рис. 6.76. Удалить запись

6.8. Классы

Классы – это самые важные строительные блоки любой объектно–ориентированной системы.

Классом (Class) называется описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Правильно сконструированный класс должен представлять одну и только одну абстракцию. Хорошо структурированные классы характеризуются четкими границами и помогают формировать сбалансированное распределение обязанностей в системе.

Класс представляет не индивидуальный объект, а целую их совокупность.

Многие языки программирования поддерживают концепцию классов. В этом случае создаваемые абстракции могут быть непосредственно отображены в конструкциях языка программирования, даже если речь идет об абстракциях не программных сущностей типа "покупатель", "торговля" или "разговор".

Графическое изображение класса в UML показано на рис. 6.77. Такое обозначение абстракции независимо от конкретного языка программирования и подчеркивает ее наиболее важные характеристики: имя, атрибуты и операции.

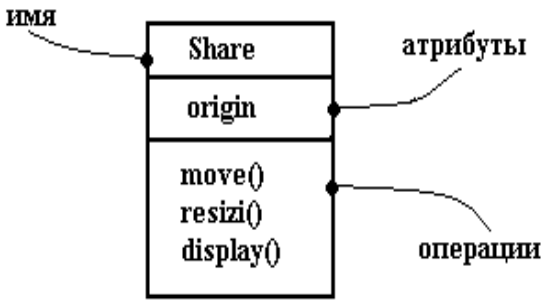


Рис. 6.77. Классы

У каждого класса есть **имя**, отличающее его от других классов. Имя класса – это текстовая строка. Взятое само по себе, оно называется простым именем. К составному имени впереди добавлено имя пакета, в который входит класс. Имя класса в объемлющем пакете должно быть уникальным. Например, java::awt::Rectangle. Имя класса может состоять из любого числа букв, цифр и ряда знаков препинания (за исключением таких, например, как двоеточие, которое применяется для отделения имени класса от имени объемлющего пакета). Имя может занимать несколько строк. На практике для именования класса используют одно или несколько коротких существительных, взятых из словаря моделируемой системы. Обычно каждое слово в имени класса пишется с заглавной буквы, например: Customer (Клиент).

Атрибут – это именованное свойство класса, включающее описание множества значений, которые могут принимать экземпляры этого свойства. Класс может иметь любое число атрибутов или не иметь их вовсе. Атрибут представляет некоторое свойство сущности, общее для всех объектов данного класса. В каждый момент времени любой атрибут объекта, принадлежащего данному классу,

обладает вполне определенным значением. Атрибуты представлены их именами в разделе, который расположен под именем класса (рис. 6.78.).

Имя атрибута, как и имя класса, может быть произвольной текстовой строкой. На практике для именования атрибута используют одно или несколько коротких существительных, соответствующих некоторому свойству объемлющего класса.

При описании атрибута можно явным образом указывать его класс и начальное значение, принимаемое по умолчанию, как показано на рис. 6.78.

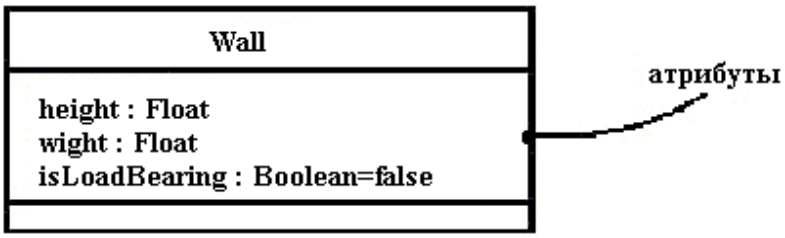


Рис. 6.78. Атрибуты и их класс

Операцией называется реализация услуги, которую можно запросить у любого объекта класса для воздействия на поведение. Иными словами, операция – это абстракция того, что позволено делать с объектом. У всех объектов класса имеется общий набор операций. Класс может содержать любое число операций или не содержать их вовсе. Часто (хотя не всегда) обращение к операции объекта изменяет его состояние или его данные. Операции класса изображаются в разделе, расположенном ниже раздела с атрибутами. При этом можно ограничиться только именами, как показано на рис. 6.79. Имя операции, как и имя класса, может быть произвольной текстовой строкой. На практике для именования операций используют короткий глагол или глагольный оборот, соответствующий определенному поведению объемлющего класса. Каждое слово в имени операции, кроме самого первого, обычно пишут с заглавной буквы, например move или isEmpty.

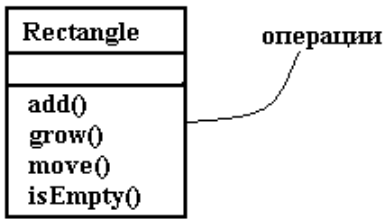


Рис. 6.79. Операции

Операцию можно описать более подробно, указав ее сигнатуру, в которую входят имена и типы всех параметров, их значения, принятые по умолчанию, а применительно к функциям – тип возвращаемого значения, как показано на рис. 6.80.

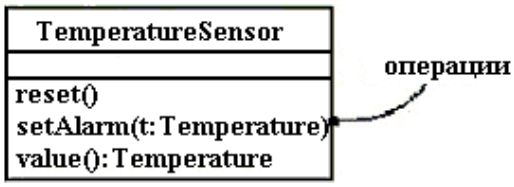


Рис. 6.80. Операции и их сигнатуры

При изображении класса необязательно показывать все его атрибуты и операции. Их может быть много для одного рисунка. Поэтому нужно указывать наиболее важные атрибуты и операции или совсем опускать их. Таким образом, пустой раздел в соответствующем месте прямоугольника может означать не отсутствие атрибутов или операций, а только то, что их не сочли нужным изобразить. Явным образом наличие дополнительных атрибутов или операций можно обозначить, поставив в конце списка многоточие.

Для лучшей организации списков атрибутов и операций можно снабдить каждую группу дополнительным описанием, воспользовавшись стереотипами.

Обязанности (Responsibilities) класса – это своего рода контракт, которому должен подчиняться класс. Соответствующие атрибуты и операции являются теми свойствами, посредством которых выполняются обязанности класса.

Моделирование классов лучше всего начинать с определения обязанностей сущностей. Число обязанностей класса может быть произвольным, но на практике хорошо структурированный класс имеет по меньшей мере одну обязанность. С другой стороны, их не должно быть и слишком много. При уточнении модели обязанности класса преобразуются в совокупность атрибутов и операций, которые должны наилучшим образом обеспечить их выполнение.

Графически обязанности изображают в особом разделе в нижней части пиктограммы класса (рис. 6.81.). Их можно указать также в примечании.

Рис. 6.81. Обязанности

Кроме того, иногда приходится визуализировать или специфицировать и другие особенности классов: видимость отдельных атрибутов и операций, специфические для конкретного языка программирования свойства операции или исключения, которые объекты класса могут возбуждать или обрабатывать.

Классы редко существуют сами по себе. При построении моделей следует обращать внимание на группы взаимодействующих между собой классов. В языке

UML такие сообщества принято называть кооперациями и изображать на диаграммах классов.

Надо избегать слишком больших или, наоборот, чересчур маленьких классов. Каждый класс должен хорошо делать что–то одно. Язык UML способен помочь в визуализации и специфицировании баланса обязанностей.

Хорошо структурированный класс обладает следующими свойствами:

является четко очерченной абстракцией некоторого понятия из словаря проблемной области или области решения;

содержит небольшой, точно определенный набор обязанностей и выполняет каждую из них;

поддерживает четкое разделение спецификаций абстракции и ее реализации;

понятен и прост, но в то же время допускает расширение и адаптацию к новым задачам.

6.8.1. Области видимости и действия, кратность и иерархия классов

Одна из деталей, наиболее существенных для атрибутов и операций классификаторов, – их видимость. Видимость свойства говорит о том, может ли оно использоваться другими классификаторами. Естественно, это подразумевает видимость самого классификатора. Один классификатор может "видеть" другой, если тот находится в области действия первого и между ними существует явное или неявное отношение. В языке UML можно определить **три уровня видимости:**

public (открытый) – любой внешний классификатор, который "видит" данный, может пользоваться его открытыми свойствами. Обозначается знаком + (плюс) перед именем атрибута или операции;

protected (защищенный) – любой потомок данного классификатора может пользоваться его защищенными свойствами. Обозначается знаком # (диез);

private (закрытый) – только данный классификатор может пользоваться закрытыми свойствами. Обозначается символом – (минус).

На рис. 6.82. показаны открытые, защищенные и закрытые атрибуты и методы для класса Toolbar.

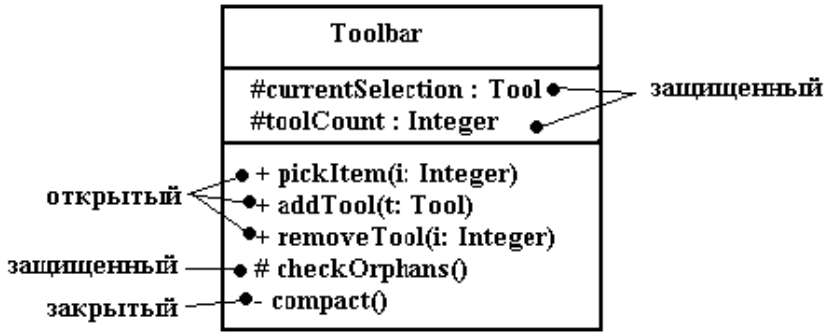


Рис. 6.82. Видимость

Видимость свойств классификатора определяют для того, чтобы скрыть детали его реализации и показать только те особенности, которые необходимы для осуществления обязанностей, продекларированных абстракцией. Если символ видимости явно не указан, обычно предполагается, что свойство является открытым. Отношения дружественности (Friendship) позволяют классификатору показывать другим свои закрытые детали.

Еще одной важной характеристикой атрибутов и операций классификатора является **область действия (Scope)**. Задавая область действия некоторого свойства, тем самым указывают, будет ли оно проявлять себя по–разному в каждом экземпляре классификатора, или одно и то же значение свойства будет разделяться (то есть совместно использоваться) всеми экземплярами. В UML определены два вида областей действия:

instance (экземпляр) – у каждого экземпляра классификатора есть собственное значение данного свойства;

classifier (классификатор) – все экземпляры классификатора совместно используют общее значение данного свойства.

Имя свойства, которое имеет область действия classifier, подчеркивается. Если подчеркивание отсутствует, предполагается область действия instance. Как правило, свойства моделируемых классификаторов имеют область действия экземпляра. Свойства с областью действия классификатора чаще всего применяются для описания закрытых атрибутов, общих для всех экземпляров, например для генерации уникальных идентификаторов или в операциях, создающих экземпляры класса.

При моделировании **иерархии классов**, верхние уровни занимают общие абстракции, а ниже находятся специализированные классы. Внутри этой иерархии некоторые классы определяют как абстрактные, то есть не имеющие непосредственных экземпляров. В языке UML имя абстрактного класса пишут курсивом. Наоборот, конкретными называются классы, которые могут иметь непосредственные экземпляры.

При моделировании класса часто возникает потребность задать ему свойства, унаследованные от более общих классов и, наоборот, предоставить возможность более специализированным классам наследовать особенности данного. Такая семантика легко обеспечивается для классов средствами UML. Можно определить и такие классы, у которых нет потомков. Они называются листовыми и задаются в UML с помощью свойства leaf (лист), написанного под именем класса.

Реже используется возможность задать класс, не имеющий родителей. Такой класс называется корневым и специфицируется с помощью свойства root (узел), записанного под его именем. Если имеется несколько независимых иерархий наследования, то начало каждой удобно обозначать таким способом.

Операции могут иметь сходные свойства. Как правило, операции являются полиморфными, – это значит, что в различных местах иерархии классов можно определять операции с одинаковыми сигнатурами. При этом те операции, которые определены в классе–потомке, перекрывают действие тех, что определены в родительских классах. Когда во время исполнения системы поступает какое–то

сообщение, операция по его обработке вызывается полиморфно, – иными словами, выбирается та, которая соответствует типу объекта. В UML имена абстрактных операций пишутся курсивом, как и в случае с классами. Не полиморфные операции (листовые) обозначаются при помощи слова leaf. Это означает, что данная операция не может быть перекрыта другой.

При работе с классом разумно предположить, что может существовать любое количество его экземпляров – если, конечно, это не абстрактный класс, у которого вообще не существует непосредственных экземпляров, хотя у его потомков их может быть любое количество. В некоторых случаях, однако, число экземпляров класса нужно ограничить. Чаще всего возникает необходимость задать класс, у которого:

нет ни одного экземпляра – тогда класс становится служебным (Utility), содержащим только атрибуты и операции с областью действия класса;

ровно один экземпляр – такой класс называют одиночным (Singleton);

заданное число экземпляров;

произвольное число экземпляров – вариант по умолчанию.

Количество экземпляров класса называется его **кратностью**. В общем смысле кратность – это диапазон возможных кардинальных чисел некоторой сущности. В языке UML кратность класса задается выражением, написанным в правом верхнем углу его пиктограммы. Кратность применима и к атрибутам. Кратность атрибута записывают в виде выражения, заключенного в квадратные скобки и расположенного сразу после имени атрибута. Например, как показано на рис. 6.83.

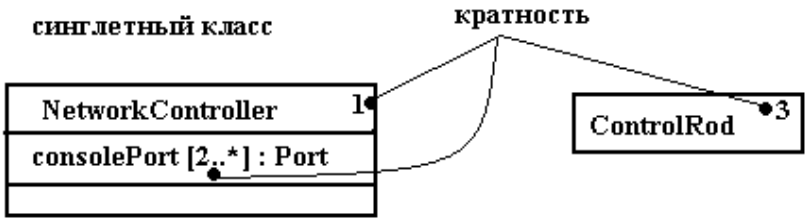


Рис. 6.83. Кратность

Таким образом, на самом высоком уровне абстракции, моделируя структурные свойства класса (то есть атрибуты), можно просто записать их имена. В дополнение к этому, можно определить видимость, область действия и кратность каждого атрибута. Кроме того, можно задать тип, начальное значение и изменяемость атрибутов. А для обозначения множества логически связанных атрибутов допустимо использовать стереотипы.

В UML определены четыре стандартных стереотипа, применимые к классам:

metaclass – определяет классификатор, все объекты которого являются классами;

powertype – определяет классификатор, все объекты которого являются потомками данного родителя;

stereotype – определяет, что данный классификатор является стереотипом, который можно применить к другим элементам;

utility – определяет класс, атрибуты и операции которого находятся в области действия всех классов.

Суммируя сказанное, получаем, что хорошо структурированный классификатор обладает следующими свойствами:

наделен как структурными, так и поведенческими аспектами;

внутренне согласован и слабо связан с другими классификаторами;

раскрывает только те особенности, которые необходимы для, использующих класс клиентов, и скрывает остальные;

его семантика и назначение не допускают неоднозначного толкования;

не настолько формализован, чтобы лишить всякой свободы тех, кто будет его реализовывать;

специфицирован в достаточной степени, чтобы исключить неоднозначное толкование его назначения.

Изображая классификатор в UML, принимают во внимание следующие рекомендации:

показывать только те его свойства, которые необходимы для понимания абстракции в контексте класса;

использовать такие стереотипы, которые наилучшим образом отражают назначение классификатора.

6.8.2. Отношения между классами

Кроме внутреннего устройства классов важную роль при разработке проектируемой системы имеют различные отношения между классами, которые также могут быть изображены на диаграмме классов. Совокупность допустимых типов таких отношений строго фиксирована в языке UML и определяется самой семантикой этих отношений. Базовые отношения, изображаемые на диаграммах классов:

Отношение ассоциации (Association relationship)

Отношение обобщения (Generalization relationship)

Отношение агрегации (Aggregation relationship)

Отношение композиции (Composition relationship)

Каждое из этих отношений имеет собственное графическое представление, которое отражает семантический характер взаимосвязи между объектами соответствующих классов.

6.8.2.1. Отношение ассоциации

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. Данное отношение, как уже отмечалось, обозначается сплошной линией со стрелкой или без нее и с дополнительными символами, которые характеризуют специальные свойства ассоциации. Ассоциации рассматривались при изучении элементов диаграммы вариантов использования, применительно к диаграммам классов, тем не менее, семантика этого типа отношений значительно шире. В качестве дополнительных специальных символов могут использоваться имя ассоциации, символ навигации, а также имена и кратность классов–ролей ассоциации.

Имя ассоциации – необязательный элемент ее обозначения. Однако если оно задано, то записывается с заглавной буквы рядом с линией ассоциации. Отдельные классы ассоциации могут играть определенную роль в соответствующем отношении, на что явно указывает имя концевых точек ассоциации на диаграмме.

Наиболее простой случай данного отношения – **бинарная ассоциация (Binary association)**, которая служит для представления произвольного отношения между двумя классами. Она связывает в точности два различных класса и может быть ненаправленным (симметричным) или направленным отношением. Частный случай бинарной ассоциации – **рефлексивная ассоциация**, которая связывает класс с самим собой. Ненаправленная бинарная ассоциация изображается линией без стрелки. Для нее на диаграмме может быть указан порядок чтения классов с использованием значка в форме треугольника рядом с именем данной ассоциации.

В качестве простого примера ненаправленной бинарной ассоциации можно рассмотреть отношение между двумя классами – классом Компания и классом Сотрудник (рис. 6.84.). Они связаны между собой бинарной ассоциацией Работает, имя которой указано на рисунке рядом с линией ассоциации. Для данного отношения определен следующий порядок чтения следования классов – сотрудник работает в компании.

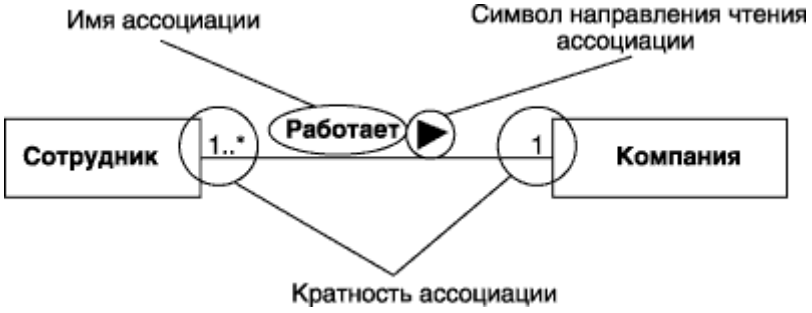


Рис. 6.84. Графическое изображение ненаправленной бинарной ассоциации между классами

Направленная бинарная ассоциация изображается сплошной линией с простой стрелкой на одной из ее концевых точек. Направление этой стрелки указывает на то, какой класс является первым, а какой – вторым.

В качестве простого примера направленной бинарной ассоциации можно рассмотреть отношение между двумя классами – классом Клиент и классом Счет

(рис. 6.85.). Они связаны между собой бинарной ассоциацией с именем Имеет, для которой определен порядок следования классов. Это означает, что конкретный объект класса Клиент всегда должен указываться первым при рассмотрении взаимосвязи с объектом класса Счет. Другими словами, эти объекты классов образуют кортеж элементов, например, <клиент, счет_1, счет_2,…, счет_n>.

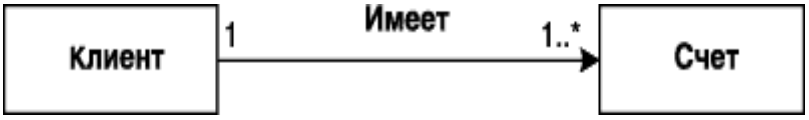


Рис. 6.85. Графическое изображение направленной бинарной ассоциации между классами

Частный случай отношения ассоциации – так называемая **исключающая ассоциация (Xor–association)**. Семантика данной ассоциации указывает на то, что из нескольких потенциально возможных вариантов данной ассоциации в каждый момент времени может использоваться только один. На диаграмме классов исключающая ассоциация изображается пунктирной линией, соединяющей две и более ассоциации (рис. 6.86.), рядом с которой записывается ограничение в форме строки текста в фигурных скобках: {xor}.

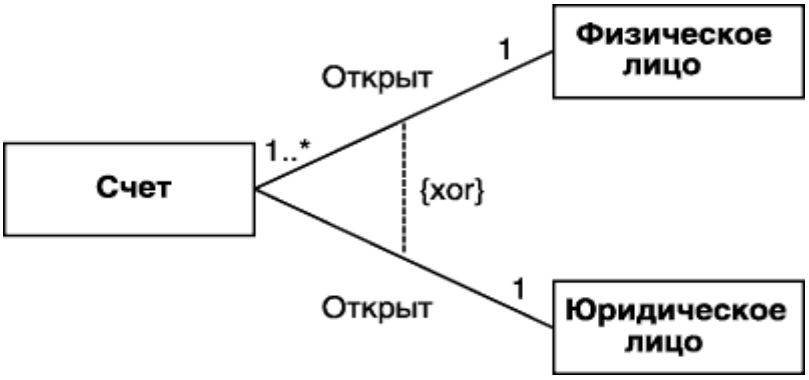


Рис. 6.86. Графическое изображение исключающей ассоциации между тремя классами

Тернарная ассоциация связывает отношением три класса. Ассоциация более высокой арности называется n–арной ассоциацией.

n–арная ассоциация (n–ary association) – ассоциация между тремя и большим числом классов.

Каждый экземпляр такой ассоциации представляет собой упорядоченный набор (кортеж), содержащий n экземпляров из соответствующих классов. Такая ассоциация связывает отношением более чем три класса, при этом класс может участвовать в ассоциации более чем один раз. Каждый экземпляр n–арной ассоциации представляет собой n–арный кортеж, состоящий из объектов соответствующих классов. В этом контексте бинарная ассоциация является

частным случаем n–арной ассоциации, когда значение n=2, но имеет собственное обозначение. Бинарная ассоциация – это специальный случай n–арной ассоциации.

Графически n–арная ассоциация обозначается ромбом, от которого ведут линии к символам классов данной ассоциации. Сам же ромб соединяется с символами классов сплошными линиями. Обычно линии проводятся от вершин ромба или от середины его сторон. Имя n–арной ассоциации записывается рядом с ромбом соответствующей ассоциации. Однако порядок классов в n–арной ассоциации, в отличие от порядка множеств в отношении, на диаграмме не фиксируется.

В качестве примера тернарной ассоциации можно рассмотреть отношение между тремя классами: Сотрудник, Компания и Проект. Данная ассоциация указывает на наличие отношения между этими тремя классами, которое может представлять информацию о проектах, реализуемых в компании, и о сотрудниках, участвующих в выполнении отдельных проектов (рис. 6.87.).

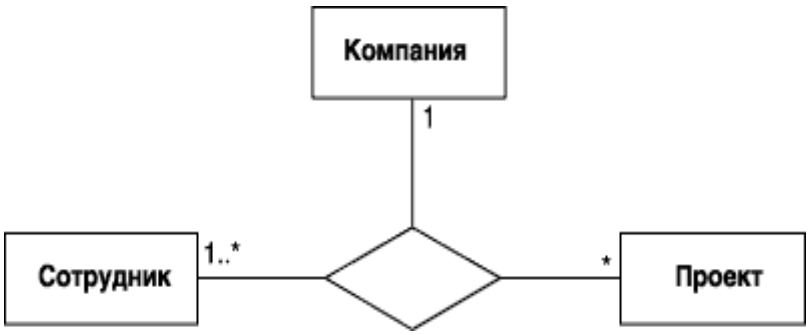


Рис. 6.87. Графическое изображение тернарной ассоциации между тремя классами

Класс может быть присоединен к линии ассоциации пунктирной линией. Это означает, что данный класс обеспечивает поддержку свойств соответствующей n–арной ассоциации, а сама n–арная ассоциация имеет атрибуты, операции и/или ассоциации. Другими словами, такая ассоциация является классом с соответствующим обозначением в виде прямоугольника и самостоятельным элементом языка UML – ассоциацией классом (Association class).

Ассоциация–класс (Association class) – модельный элемент, который одновременно является ассоциацией и классом. Ассоциация класс может рассматриваться как ассоциация, которая обладает свойствами класса, или как класс, имеющий также свойства ассоциации.

Как уже упоминалось, отдельный класс в ассоциации может играть определенную роль в данной ассоциации. Эта роль может быть явно специфицирована на диаграмме классов. С этой целью в языке UML вводится в рассмотрение специальный элемент – концевая точка ассоциации или конец ассоциации (Association end), который графически соответствует точке соединения линии ассоциации с отдельным классом.

Конец ассоциации (Association end) – концевая точка ассоциации, которая связывает ассоциацию с классификатором.

Конец ассоциации – часть ассоциации, но не класса. Каждая ассоциация может иметь два или больше концов ассоциации. Наиболее важные свойства ассоциации указываются на диаграмме рядом с этими элементами ассоциации и должны перемещаться вместе с ними. Одним из таких дополнительных обозначений является имя роли отдельного класса, входящего в ассоциацию.

Роль (Role) – имеющее имя специфическое поведение некоторой сущности, рассматриваемой в определенном контексте. Роль может быть статической или динамической.

Имя роли представляет собой строку текста рядом с концом ассоциации для соответствующего класса. Она указывает на специфическую роль, которую играет класс, являющийся концом рассматриваемой ассоциации. Имя роли не обязательный элемент обозначений и может отсутствовать на диаграмме.

Следующий элемент обозначений – **кратность ассоциации**. Кратность относится к концам ассоциации и обозначается в виде интервала целых чисел, аналогично кратности атрибутов и операций классов, но без прямых скобок. Этот интервал записывается рядом с концом соответствующей ассоциации и означает потенциальное число отдельных экземпляров класса, которые могут иметь место, когда остальные экземпляры или объекты классов фиксированы.

Так, для примера (рис. 6.87.) кратность "1" для класса Компания означает, что каждый сотрудник может работать только в одной компании. Кратность "1..*" для класса Сотрудник означает, что в каждой компании могут работать несколько сотрудников, общее число которых заранее неизвестно и ничем не ограничено. Вместо кратности "1..*" нельзя записать только символ "*", поскольку последний означает кратность "0..*". Для данного примера это означало бы, что отдельные компании могут совсем не иметь сотрудников в своем штате. Такая кратность приемлема в ситуациях, когда в компании вообще не выполняется никаких проектов.

Имя ассоциации рассматривается в качестве отдельного атрибута у соответствующих классов ассоциаций и может быть указано на диаграмме явным способом в определенной секции прямоугольника класса.

Ассоциация является наиболее общей формой отношения в языке UML. Все другие типы рассматриваемых отношений можно считать частным случаем данного отношения. Однако важность выделения специфических семантических свойств и дополнительных характеристик для других типов отношений обусловливают необходимость их самостоятельного изучения при построении диаграмм классов. Поскольку эти отношения имеют специальные обозначения и относятся к базовым понятиям языка UML, следует перейти к их последовательному рассмотрению.

6.8.2.2. Отношение обобщения

Отношение обобщения является обычным таксономическим отношением или отношением классификации между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком).

Обобщение (Generalization) – таксономическое отношение между более общим понятием и менее общим понятием.

Менее общий элемент модели должен быть согласован с более общим элементом и может содержать дополнительную информацию. Данное отношение используется для представления иерархических взаимосвязей между различными элементами языка UML, такими как пакеты, классы, варианты использования.

Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения.

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

Согласно одному из главных принципов методологии ООАП – наследованию, класс–потомок обладает всеми свойствами и поведением класса–предка, а также имеет собственные свойства и поведение, которые могут отсутствовать у класса–предка.

Родитель, предок (Parent) – в отношении обобщения более общий элемент. Потомок (Child) – специализация одного из элементов отношения обобщения, называемого в этом случае родителем.

На диаграммах отношение обобщения обозначается сплошной линией с треугольной стрелкой на одном из концов (рис. 6.88.). Стрелка указывает на более общий класс (класс–предок или суперкласс), а ее начало – на более специальный класс (класс–потомок или подкласс).

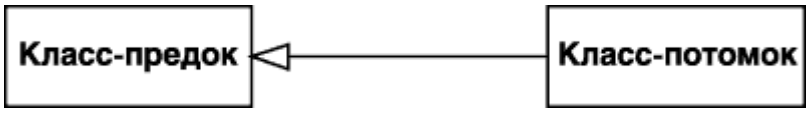


Рис. 6.88. Графическое изображение отношения обобщения в языке UML

От одного класса–предка одновременно могут наследовать несколько классов–потомков, что отражает таксономический характер данного отношения. В этом случае на диаграмме классов для подобного отношения обобщения указывается несколько линий со стрелками.

Например, класс Транспортное средство (курсив обозначает абстрактный класс) может выступать в качестве суперкласса для подклассов, соответствующих конкретным транспортным средствам, таким как: Автомобиль, Автобус, Трактор и другим. Это может быть представлено графически в форме диаграммы классов следующего вида (рис. 6.89.).

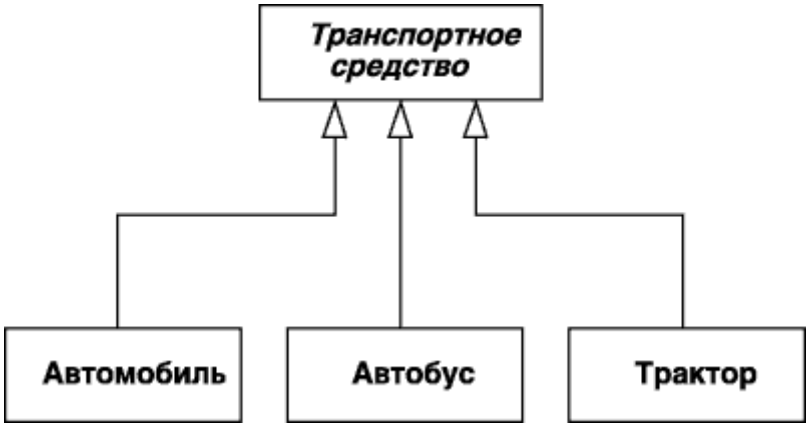


Рис. 6.89. Пример графического изображения отношения обобщения для нескольких классов–потомков

С целью упрощения обозначений на диаграмме классов и уменьшения числа стрелок–треугольников и совокупности линий, обозначающих одно и то же отношение обобщения, может быть просто изображена стрелка. В этом случае отдельные линии сходятся к стрелке, которая имеет с этими линиями единственную точку пересечения (рис. 6.90.).

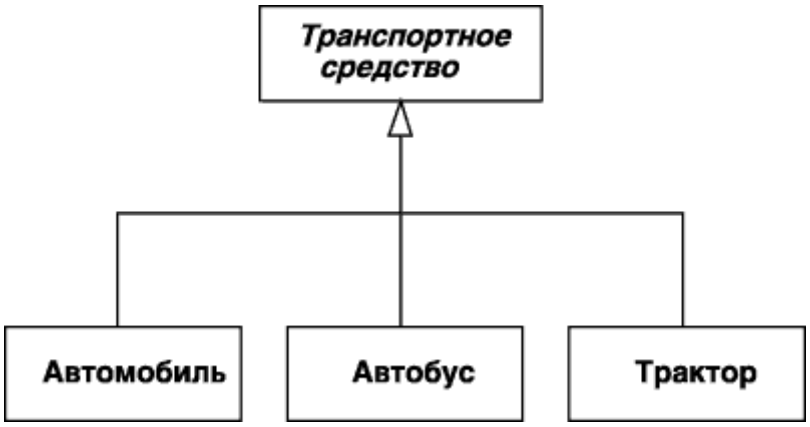


Рис. 6.90. Альтернативный вариант графического изображения отношения обобщения классов для случая объединения отдельных линий

Графическое изображение отношения обобщения по форме соответствует графу специального вида, а именно – иерархическому дереву. Как нетрудно заметить, в этом случае класс–предок является корнем дерева, а классы–потомки – его листьями. Отличие заключается в возможности указания на диаграмме классов дополнительной семантической информации, которая может отражать различные теоретико–множественные характеристики данного отношения. При этом класс–предок на диаграмме может занимать произвольное положение относительно своих классов–потомков, определяемое лишь соображениями удобства.

В дополнение к простой стрелке обобщения может быть присоединена строка текста, указывающая на специальные свойства этого отношения в форме ограничения. Этот текст будет относиться ко всем линиям обобщения, которые

идут к классам–потомкам. Поскольку отмеченное свойство касается всех подклассов данного отношения, именно поэтому спецификация этого свойства осуществляется в форме ограничения, которое должно быть записано в фигурных скобках.

В качестве ограничений могут быть использованы следующие ключевые слова языка UML:

{complete} – означает, что в данном отношении обобщения специфицированы все классы–потомки, и других классов–потомков у данного класса–предка быть не может.

{incomplete} – означает случай, противоположный первому. А именно, предполагается, что на диаграмме указаны не все классы–потомки. В последующем возможно разработчик восполнит их перечень, не изменяя уже построенную диаграмму.

{disjoint} – означает, что классы–потомки не могут содержать объектов, одновременно являющихся экземплярами двух или более классов.

{overlapping} – случай, противоположный предыдущему. А именно, предполагается, что отдельные экземпляры классов–потомков могут принадлежать одновременно нескольким классам.

С учетом дополнительного использования стандартного ограничения диаграмма классов (рис. 6.90.) может быть уточнена (рис. 6.91.).

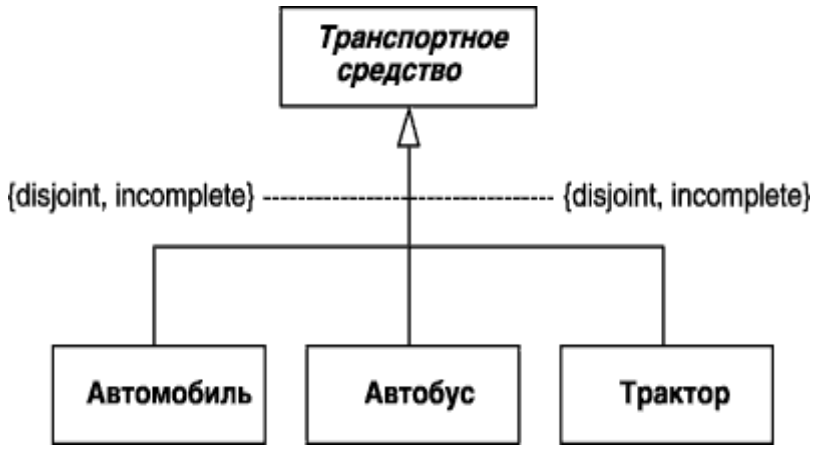


Рис. 6.91. Вариант уточненного графического изображения отношения обобщения классов с использованием строки–ограничения

6.8.2.3. Отношение агрегации

Агрегация (Aggregation) – специальная форма ассоциации, которая служит для представления отношения типа "часть–целое" между агрегатом (целое) и его составной частью.

Отношение агрегации имеет место между несколькими классами в том случае, если один из классов представляет собой сущность, которая включает в себя в

качестве составных частей другие сущности. Данное отношение имеет **фундаментальное значение** для описания структуры сложных систем, поскольку применяется для представления **системных взаимосвязей типа "часть–целое"**. Раскрывая внутреннюю структуру системы, отношение агрегации показывает, из каких элементов состоит система, и как они связаны между собой.

С точки зрения модели отдельные части системы могут выступать в виде, как элементов, так и подсистем, которые, в свою очередь, тоже могут состоять из подсистем или элементов. Таким образом, данное отношение 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.

Очевидно, что рассматриваемое в таком аспекте деление системы на составные части представляет собой иерархию, но принципиально отличную от той, которая порождается 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 существенно отличаются от атрибутов и операций целого.

Графически отношение агрегации изображается сплошной линией, один из концов которой представляет собой не закрашенный внутри ромб. Этот ромб указывает на тот класс, который представляет собой "целое" или класс–контейнер. Остальные классы являются его "частями" (рис. 6.92.).

Рис. 6.92. Графическое изображение отношения агрегации в языке UML

В качестве примера отношения агрегации можно рассмотреть взаимосвязь типа "часть–целое", которая имеет место между классом Системный блок персонального компьютера и его составными частями: Процессор, Материнская плата, Оперативная память, Жесткий диск и Дисковод гибких дисков. Используя обозначения языка UML, компонентный состав системного блока можно представить в виде соответствующей диаграммы классов (рис. 6.93.), которая в данном случае иллюстрирует отношение агрегации.

Рис. 6.93. Диаграмма классов для иллюстрации отношения агрегации на примере системного блока ПК

6.8.2.4. Отношение композиции

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.

Отношение композиции – ***частный случай отношения 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**.

Композит (Composite) – класс, который связан отношением композиции с одним или большим числом классов.

Графически отношение композиции изображается сплошной линией, один из концов которой представляет собой закрашенный внутри ромб. Этот ромб указывает на тот класс, который представляет собой класс–композит. Остальные классы являются его "частями" (рис. 6.94.).

Рис. 6.94. Графическое изображение отношения композиции в языке UML

Практический пример этого отношения – окно графического интерфейса программы, которое может состоять из строки заголовка, полос прокрутки, главного меню, рабочей области и строки состояния. Нетрудно заключить, что подобное окно представляет собой класс, а его составные элементы также являются отдельными классами. Последнее обстоятельство характерно для отношения композиции, поскольку отражает различные способы представления данного отношения.

Для отношений композиции и агрегации могут использоваться дополнительные обозначения, применяемые для отношения ассоциации. А именно, могут указываться кратности отдельных классов, которые в общем случае не обязательны. Применительно к описанному выше примеру класс Окно программы является классом–композитом, а взаимосвязи составляющих его частей могут быть изображены следующей диаграммой классов (рис. 6.95).

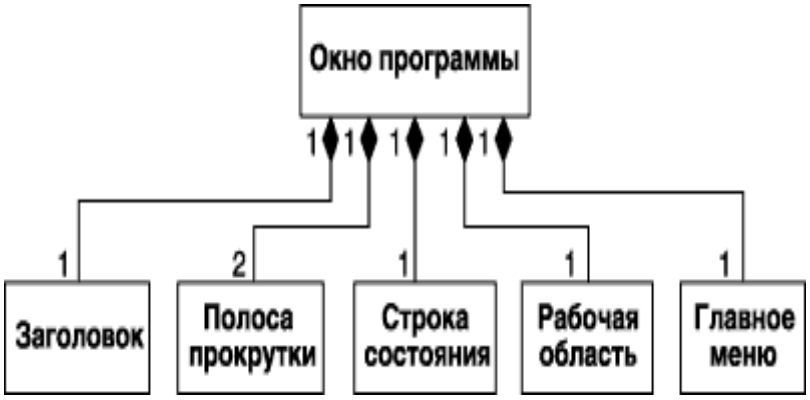


Рис. 6.95. Диаграмма классов для иллюстрации отношения композиции на примере класса–композита Окно программы

6.8.3. Примеры диаграмм классов

Логическая структура сайта ПЭ НСИ:

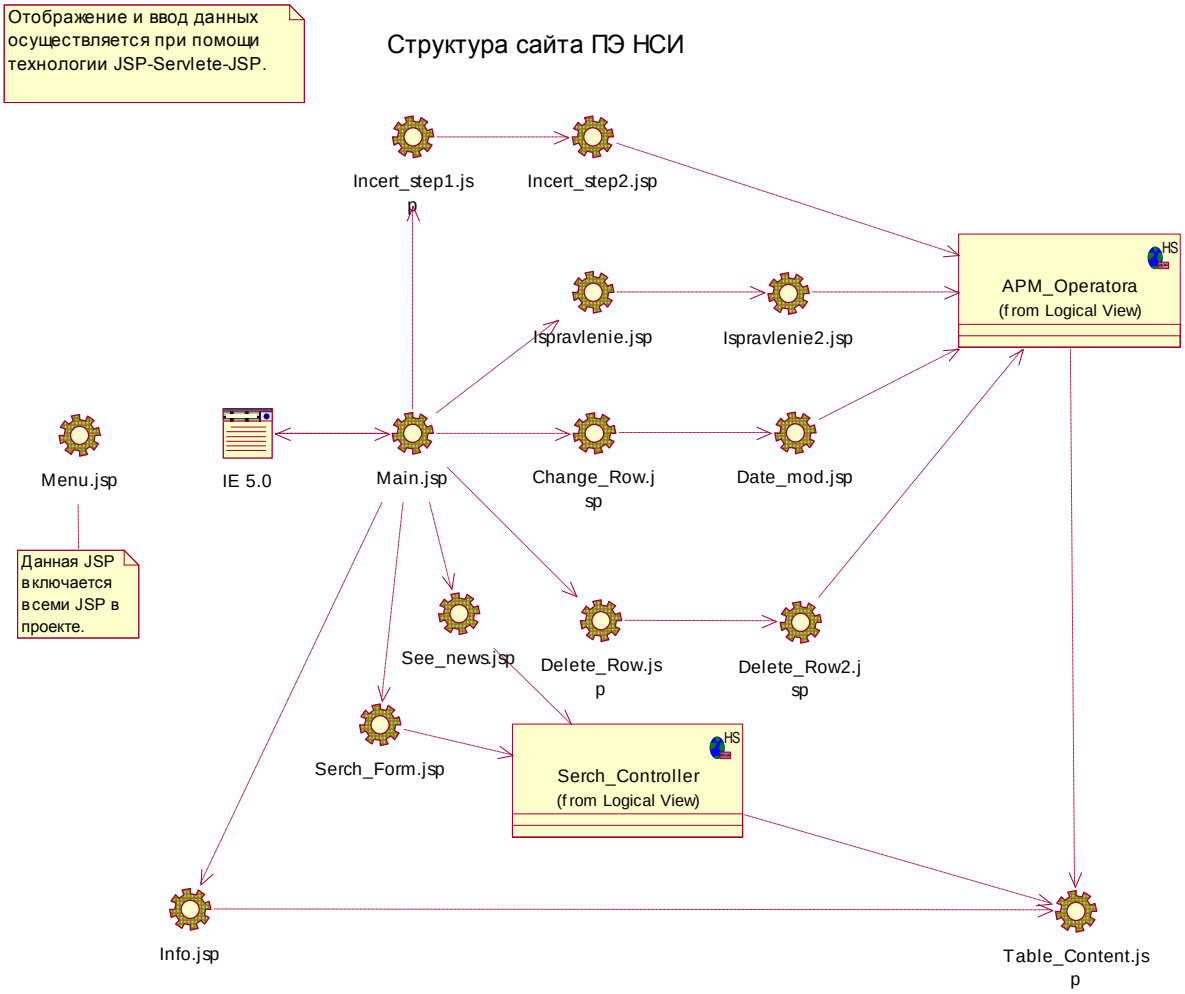


Рис. 6.96.

Логическая структура сайта ПЭ НСИ:

Уровень "Отображение" – представляет возможность конечному пользователю работать с БД ПЭ НСИ при помощи удаленного доступа. Отвечает за отображение информации ПЭ НСИ и обработку запросов пользователя. Отображение и ввод данных осуществляется при помощи технологии JSP–Servlete–JSP. На диаграмме представлены страницы JSP (аналог html–страниц, но в отличие от последних обрабатываются не только браузером но и сервером) и схема вызовов каждой из страниц.

Диаграмма логической структуры сайта ПЭ НСИ. Часть 1. Сервер:

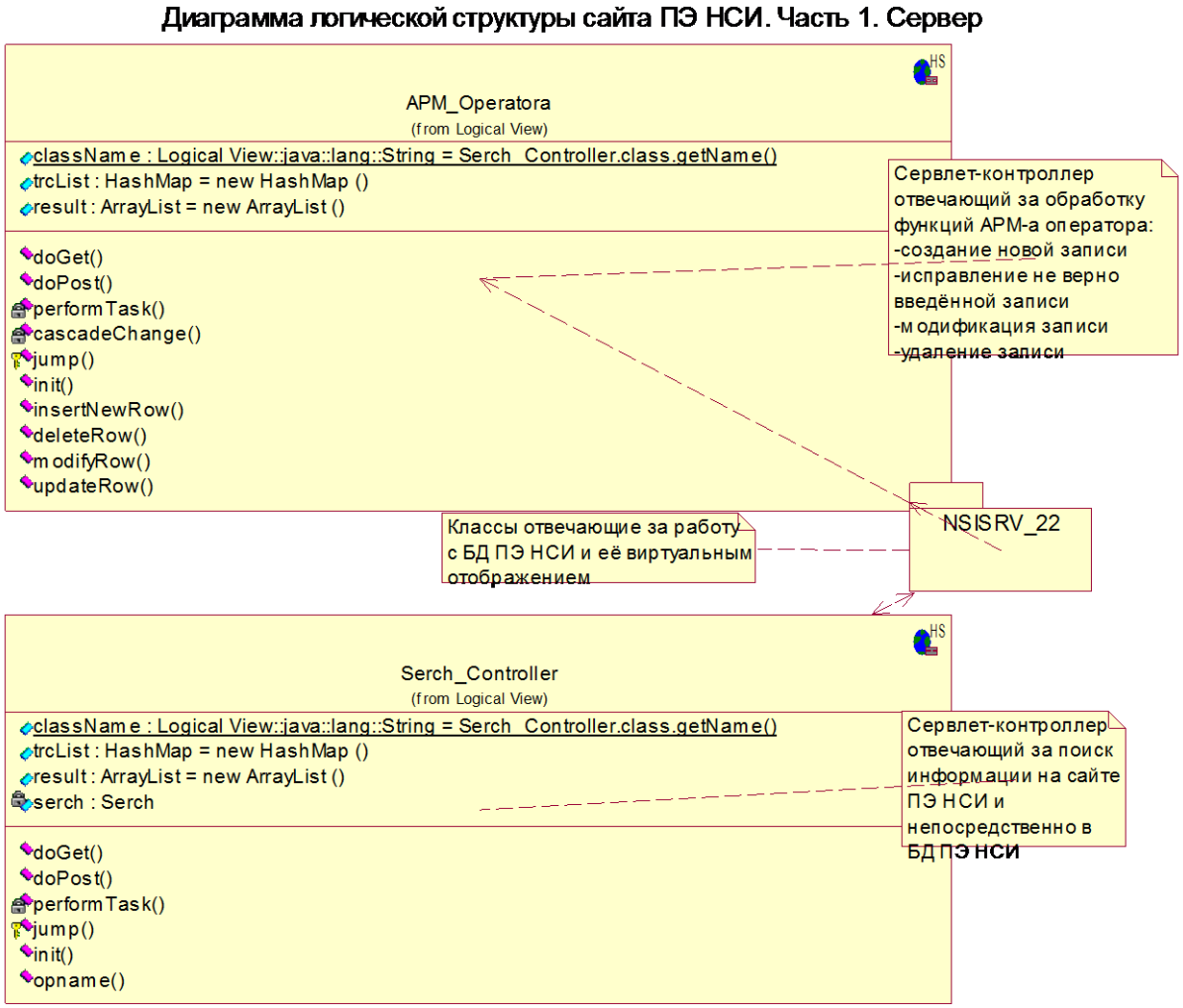


Рис. 6.97.

Диаграмма логической структуры сайта ПЭ НСИ. Часть 1. Сервер:

Представлены два сервлета – контроллера, реализующих обработку запросов пользователя. При помощи классов пакета NSISRV_22 сервлеты – контроллеры получают доступ к БД ПЭ НСИ и её виртуальному отображению.

Логическая структура ПО ПЭ НСИ. Часть2. Бизнес логика:

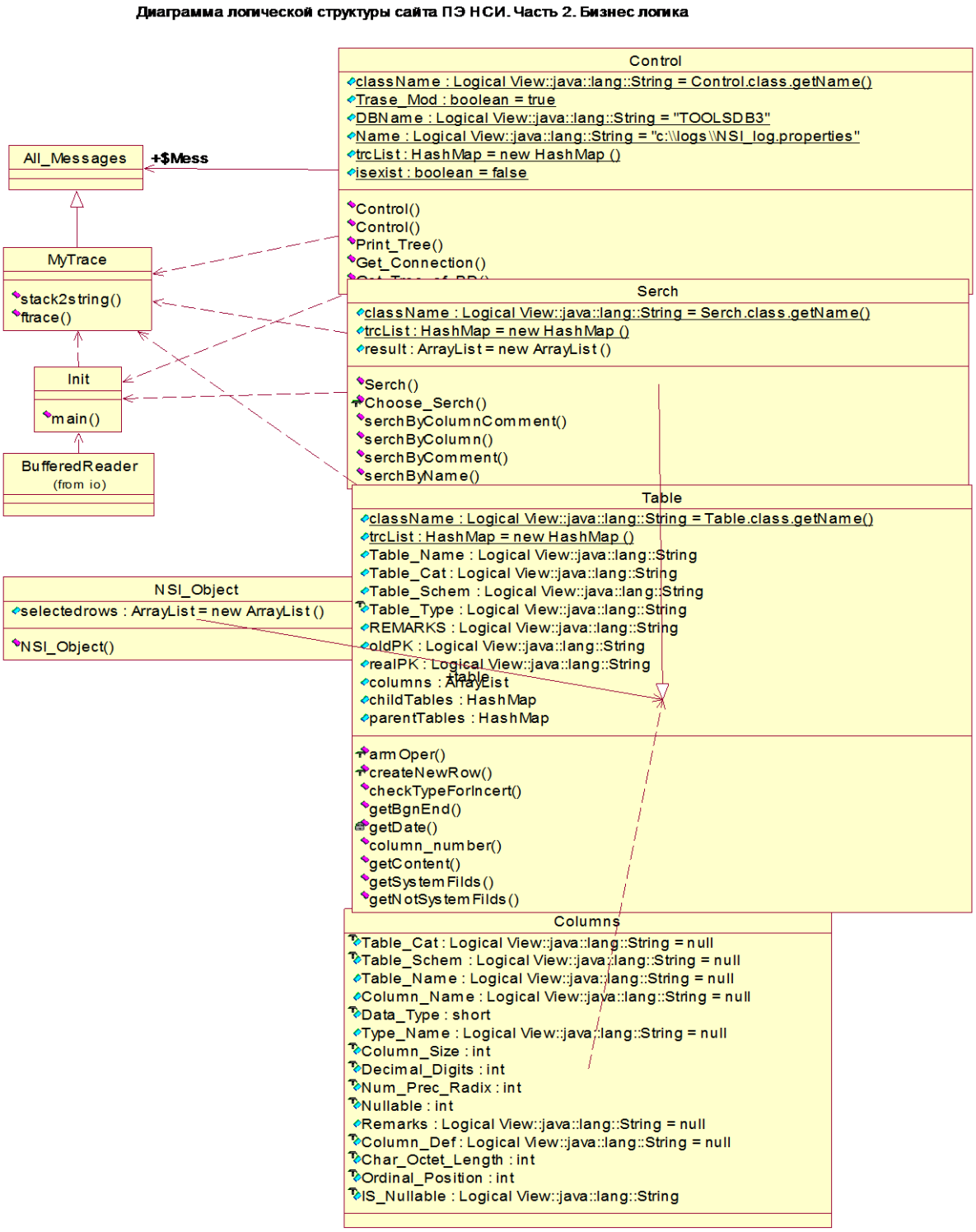


Рис. 6.98.

Диаграмма логической структуры сайта ПЭ НСИ. Часть 2. Бизнес логика:
Представлены классы, реализующие бизнес–логику приложения:
– формирование sql запросов к БД ПЭ НСИ;
– обработка результатов sql запросов;
– виртуализация БД ПЭ НСИ;
– поиск информации;
– изменение информации;
– логирование.

6.9. Компоненты

Компонент (Component) – это физическая заменяемая часть системы, совместимая с одним набором интерфейсов и обеспечивающая реализацию какого–либо другого. Компонент изображается в виде прямоугольника с вкладками (рис. 6.99.).

Компоненты используются для моделирования физических сущностей, размещенных в узле: исполняемых модулей, библиотек, таблиц, файлов и документов. Обычно компонент представляет собой физическую пакет логических элементов, таких как классы, интерфейсы и кооперативные диаграммы.

Многие операционные системы и языки программирования непосредственно поддерживают понятие компонента. Объектные библиотеки, исполняемые программы, компоненты Enterprise JavaBeans – все эти примеры сущностей, которые могут быть непосредственно представлены компонентами в смысле UML. Компоненты могут использоваться не только для моделирования такого рода сущностей, но и для представления иных элементов работающей системы – к примеру, таблиц, файлов и документов.

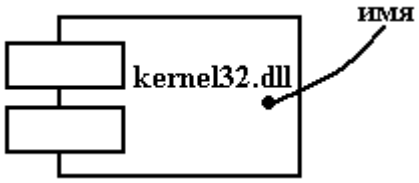


Рис. 6.99. Компонент

Графически компонент в UML изображается так, как показано на рис. 6.102. Это обозначение позволяет рассматривать компонент без привязки к конкретной операционной системе или языку программирования. С помощью стереотипов – одного из механизмов расширения UML – можно приспособить эту нотацию для представления конкретных видов компонентов.

У каждого компонента должно быть имя, отличающее его от других компонентов. Имя – это текстовая строка; взятое само по себе, оно называется простым. В составном имени спереди от простого добавлено имя пакета, в котором

находится компонент. Имя компонента должно быть уникальным внутри объемлющего пакета. Обычно при изображении компонента указывают только его имя, как видно из рис. 6.100. Тем не менее, как и в случае с классами, вы можете снабжать компоненты помеченными значениями или дополнительными разделами, чтобы показать детали.

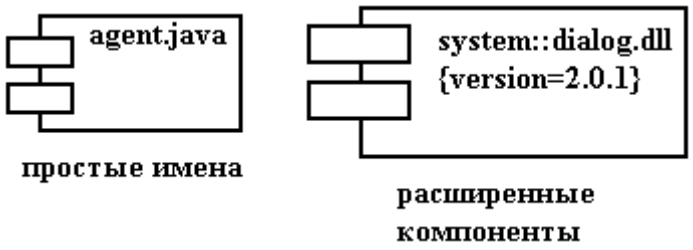


Рис. 6.100. Простое и расширенное изображение компонентов

Имя компонента может состоять из любого числа букв, цифр и некоторых знаков препинания (за исключением таких, как двоеточия, которые применяются для отделения имени компонента от имени объемлющего пакета). Имя может занимать несколько строк. Как правило, для именования компонентов используют одно или несколько коротких существительных, взятых из словаря реализации, и в зависимости от выбранной операционной системы добавляют расширения имен файлов (например, java или dll).

Изображая компонент в UML, руководствуйтесь следующими правилами:

– применяйте свернутую форму интерфейса, если только не возникает острой необходимости раскрыть операции, предлагаемые этим интерфейсом;

– показывайте только те интерфейсы, которые необходимы для понимания назначения компонента в данном контексте;

– в тех случаях, когда вы используете компоненты для моделирования библиотек и исходного кода, указывайте помеченные значения, относящиеся к контролю версий.

Хорошо структурированный компонент обладает следующими свойствами:

предоставляет четкую абстракцию некоторой сущности, которая является частью физического аспекта системы;

предоставляет реализацию небольшого, хорошо определенного набора интерфейсов;

включает набор классов, которые, действуя совместно, реализуют семантику интерфейсов изящно и экономно;

слабо связан с другими компонентами; как правило, компоненты моделируются только совместно, с отношениями зависимости и реализации.

6.9.1. Виды компонентов

Можно выделить три вида компонентов.

Во–первых, это **компоненты размещения (Deployment components),** которые необходимы и достаточны для построения исполняемой системы. К их числу относятся динамически подключаемые библиотеки (DLL) и исполняемые программы (EXE). Определение компонентов в UML достаточно широко, чтобы охватить как классические объектные модели, вроде COM+, CORBA и Enterprise JavaBeans, так и альтернативные, возможно содержащие динамические Web–страницы, таблицы базы данных и исполняемые модули, где используются закрытые механизмы коммуникации.

Во–вторых, есть компоненты – **рабочие продукты (Work product components)**. По сути дела, это побочный результат процесса разработки. Сюда можно отнести файлы с исходными текстами программ и данными, из которых создаются компоненты размещения. Такие компоненты не принимают непосредственного участия в работе исполняемой системы, но являются рабочими продуктами, из которых исполняемая система создается.

В–третьих, существуют **компоненты исполнения (Execution components);** Они создаются как результат работы системы. Примером может служить объект COM+, экземпляр которого создается из DLL.

Все механизмы расширения UML применимы и к компонентам. Чаще всего используются помеченные значения для расширения свойств компонентов (например, для задания версии компонента) и стереотипы для задания новых видов компонентов (например, зависимых от операционной системы).

В UML определены пять стандартных стереотипов, применимых к компонентам:

executable (исполнимый) – определяет компонент, который может исполняться в узле;

library (библиотека) – определяет статическую или динамическую объектную библиотеку;

table (таблица) – определяет компонент, представляющий таблицу базы данных;

file (файл) – определяет компонент, представляющий документ, который содержит исходный текст или данные;

document (документ) – определяет компонент, представляющий документ.

6.9.2. Отношения между компонентами

Вы можете организовывать компоненты, группируя их в пакеты, так же, как это делается для классов.

При организации компонентов между ними можно специфицировать отношения зависимости, обобщения, ассоциации (включая агрегирование) и реализации.

Другим случаем отношения зависимости на диаграмме компонентов является отношение программного вызова и компиляции между различными видами компонентов. Для рассмотренного фрагмента диаграммы компонентов (рис. 6.101.)

наличие подобной зависимости означает, что исполнимый компонент Control.exe использует или импортирует некоторую функциональность компонента Library.dll, вызывает страницу гипертекста Home.html и файл помощи Search.hlp, а исходный текст этого исполнимого компонента хранится в файле Control.cpp. При этом характер отдельных видов зависимостей может быть отмечен дополнительно с помощью текстовых стереотипов.

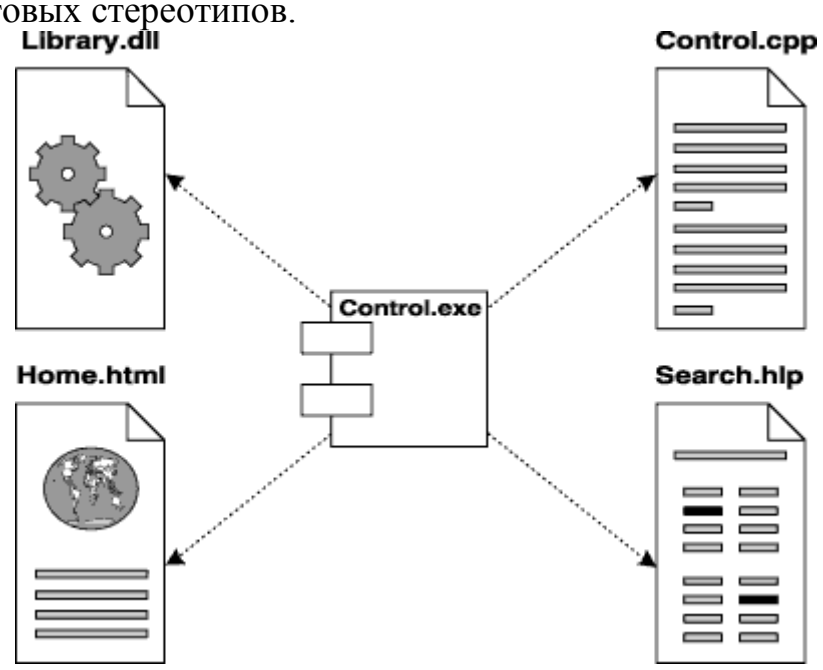


Рис. 6.101. Графическое изображение отношения зависимости между компонентами

На диаграмме компонентов могут быть также представлены отношения зависимости между компонентами и реализованными в них классами. Эта информация имеет значение для обеспечения согласования логического и физического представлений модели системы. Разумеется, изменения в структуре описаний классов могут привести к изменению этой зависимости. Ниже приводится фрагмент зависимости подобного рода, когда исполнимый компонент Control.exe зависит от соответствующих классов (рис. 6.102.).

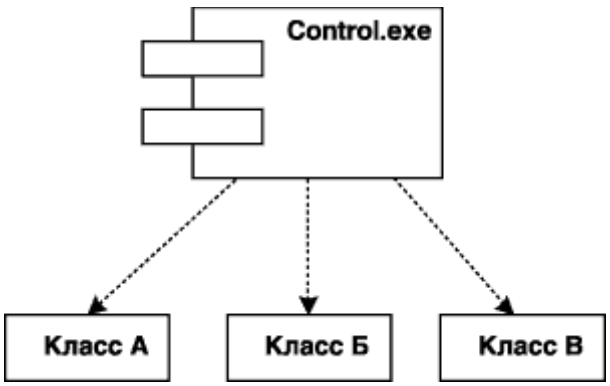


Рис. 6.102. Графическое изображение зависимости между компонентом и классами

6.9.3. Компоненты и классы

Во многих отношениях компоненты подобны классам. Те и другие наделены именами, могут реализовывать набор интерфейсов, вступать в отношения зависимости, обобщения и ассоциации, быть вложенными, иметь экземпляры и принимать участие во взаимодействиях. Однако между компонентами и классами есть существенные различия: классы представляют собой логические абстракции, а компоненты – физические сущности. Таким образом, компоненты могут размещаться в узлах, а классы – нет; компоненты представляют собой физическую упаковку логических сущностей и, следовательно, находятся на другом уровне абстракции; классы могут обладать атрибутами и операциями. Компоненты обладают только операциями, доступными через их интерфейсы.

Первое отличие является самым важным. При моделировании системы решение о том, что использовать – класс или компонент, – очевидно: если моделируемая сущность непосредственно размещается в узле, то это компонент, в противном случае – класс.

Второе различие предполагает существование некоторого отношения между классами и компонентами, а именно: компонент – это физическая реализация множества логических элементов, таких как классы и кооперативные диаграммы. Как показано на рис. 6.103., отношение между компонентом и классом, который он реализует, может быть явно изображено с помощью отношения зависимости. Как правило, вам не придется рассматривать такие отношения; лучше хранить их как часть спецификации компонента.

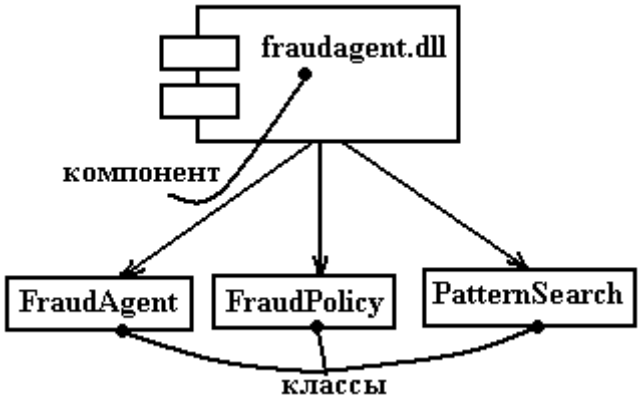


Рис. 6.103. Компоненты и классы

Третье различие подчеркивает, что интерфейсы являются мостом между компонентами и классами. В следующем разделе более подробно объясняется, что, хотя и компоненты, и классы могут реализовывать интерфейсы, в отличие от класса услуги компонента обычно доступны только через его интерфейсы.

6.9.4. Компоненты и интерфейсы

Интерфейс – это набор операций, которые описывают услуги, предоставляемые классом или компонентом. В общем случае интерфейс графически изображается окружностью, которая соединяется с компонентом отрезком линии без стрелок (рис. 6.104, а). При этом имя интерфейса, которое рекомендуется начинать с заглавной буквы "I", записывается рядом с окружностью. Семантически линия означает реализацию интерфейса, а наличие интерфейсов у компонента означает, что данный компонент реализует соответствующий набор интерфейсов.

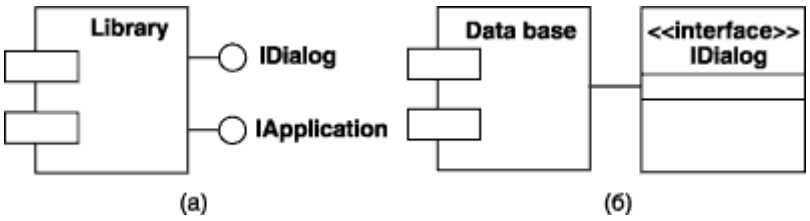


Рис. 6.104. Графическое изображение интерфейсов на диаграмме компонентов

Кроме того, интерфейс на диаграмме компонентов может быть изображен в виде прямоугольника класса со стереотипом <<interface>> и секцией поддерживаемых операций (рис. 6.104, б). Как правило, этот вариант обозначения используется для представления внутренней структуры интерфейса.

Как видно из рис. 6.105., отношения между компонентом и его интерфейсами можно изобразить двумя способами. Первый, наиболее распространенный, состоит в том, что интерфейс рисуется в свернутой (elided) форме. Компонент, реализующий интерфейс, присоединяется к нему с помощью отношения свернутой реализации. Во втором случае интерфейс рисуется в развернутом виде, возможно с раскрытием операций. Реализующий его компонент присоединяется с помощью отношения полной реализации. В обоих случаях компонент, получающий доступ к услугам других компонентов через этот интерфейс, подключается к нему с помощью отношения зависимости.

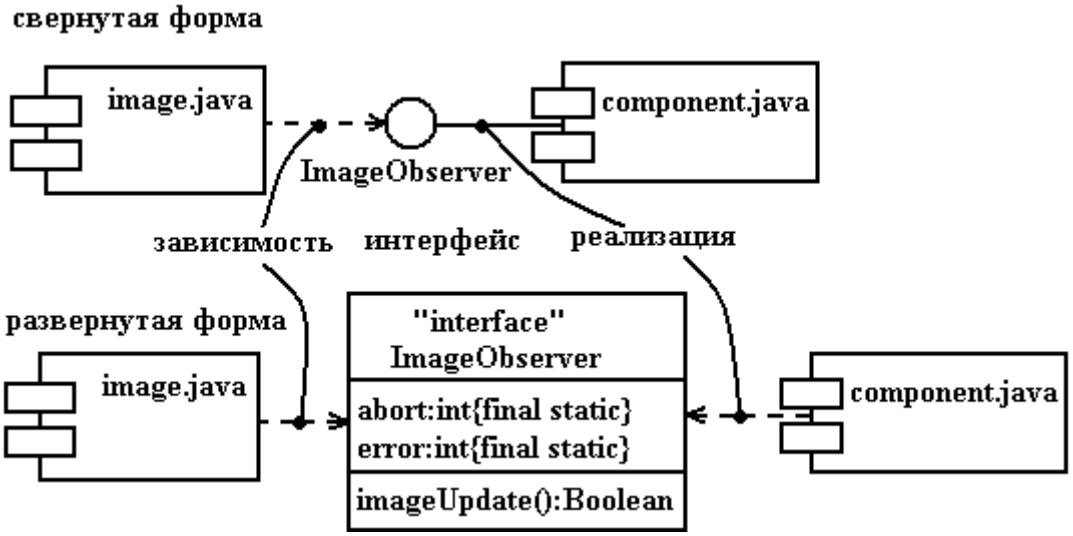


Рис. 6.105. Компоненты и интерфейсы

Интерфейс, реализуемый компонентом, называется экспортируемым интерфейсом (Export interface). Это означает, что компонент через данный интерфейс предоставляет ряд услуг другим компонентам. Компонент может экспортировать много интерфейсов. Интерфейс, которым компонент пользуется, называется импортируемым (Import interface). Это означает, что компонент совместим с таким интерфейсом и зависит от него при выполнении своих функций. Компонент может импортировать различные интерфейсы, причем ему разрешается одновременно экспортировать и импортировать интерфейсы.

Конкретный интерфейс может экспортироваться одним компонентом и импортироваться другим. Если между двумя компонентами располагается интерфейс, их непосредственная взаимозависимость разрывается. Компонент, использующий данный интерфейс, будет функционировать корректно (притом безразлично, каким именно компонентом реализован интерфейс). Разумеется, компонент можно использовать в некотором контексте только тогда, когда все импортируемые им интерфейсы экспортируются какими–либо другими компонентами.

Главная задача каждого компонентно–ориентированного средства в любой операционной системе – обеспечить возможность сборки приложений из заменяемых двоичных частей. Это означает, что вы можете создать систему из компонентов, а затем развивать ее, добавляя новые компоненты или заменяя старые, – без перекомпиляции. Именно интерфейсы позволяют достичь этого. Специфицируя интерфейс, вы можете вставить в уже работающую систему любой компонент, который совместим с этим интерфейсом или предоставляет его. Систему можно расширять, подставляя компоненты, обеспечивающие новые услуги с помощью дополнительных интерфейсов, а также компоненты, способные распознать и использовать эти новые интерфейсы. Такая семантика объясняет, что стоит за определением компонентов в UML. Компонент – это физическая заменяемая часть системы, которая совместима с одними интерфейсами и реализует другие.

Во–первых, компонент имеет физическую природу. Он существует в реальном мире битов, а не в мире концепций.

Во–вторых, компонент заменяем. Вместо одного компонента можно подставить другой, если он совместим с тем же набором интерфейсов.

Обычно механизм добавления или замены компонента с целью формирования исполняемой системы прозрачен для пользователя и обеспечивается либо объектными моделями (такими, как COM+ или Enterprise JavaBeans), которые часто совсем не требуют внешнего вмешательства, либо инструментальными средствами, автоматизирующими этот механизм.

В–третьих, компонент – это часть системы. Компонент редко выступает в отрыве от остальных: обычно он работает совместно с другими компонентами и, стало быть, встраивается в архитектурный или технологический контекст, для которого предназначен. Компонент является логически и физически способным к сцеплению, то есть представляет собой значимый структурный и/или поведенческий фрагмент некоторой большей системы. Компонент можно повторно использовать в различных системах. Таким образом, компоненты представляют собой фундаментальные строительные блоки, из которых собираются системы. Это определение рекурсивно – система, рассматриваемая на одном уровне абстракции, может быть всего лишь компонентом на более высоком уровне.

В–четвертых, как упоминалось выше, компонент совместим с одним набором интерфейсов и реализует другой набор.

6.9.5. Варианты графического изображения компонентов

Поскольку компонент как элемент модели может иметь различную физическую реализацию, иногда его изображают в форме специального графического символа, иллюстрирующего конкретные особенности реализации. Строго говоря, эти дополнительные обозначения не специфицированы в нотации языка UML. Однако, удовлетворяя общим механизмам расширения языка UML, они упрощают понимание диаграммы компонентов, существенно повышая наглядность графического представления.

Для более наглядного изображения компонентов были предложены и стали общепринятыми следующие графические стереотипы:

Во–первых, стереотипы для компонентов развертывания, которые обеспечивают непосредственное выполнение системой своих функций. Такими компонентами могут быть динамически подключаемые библиотеки (рис. 6.106, а), Web–страницы на языке разметки гипертекста (рис. 6.106, б) и файлы справки (рис. 6.106, в).

Во–вторых, стереотипы для компонентов в форме рабочих продуктов. Как правило – это файлы с исходными текстами программ (рис. 6.106, г).

\

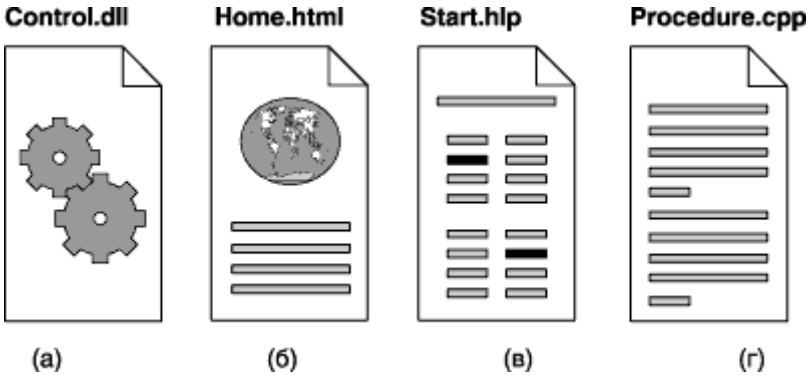


Рис. 6.106. Варианты графического изображения компонентов на диаграмме компонентов

Эти элементы иногда называют артефактами, подчеркивая при этом их законченное информационное содержание, зависящее от конкретной технологии реализации соответствующих компонентов. Более того, разработчики могут для этой цели использовать самостоятельные обозначения, поскольку в языке UML нет строгой нотации для графического представления артефактов.

Другой способ спецификации различных видов компонентов — указание текстового стереотипа компонента перед его именем. В языке UML для компонентов определены следующие стереотипы:

<<file>> (файл) – определяет наиболее общую разновидность компонента, который представляется в виде произвольного физического файла.

<<executable>> (исполнимый) – определяет разновидность компонента–файла, который является исполнимым файлом и может выполняться на компьютерной платформе.

<<document>> (документ) – определяет разновидность компонента–файла, который представляется в форме документа произвольного содержания, не являющегося исполнимым файлом или файлом с исходным текстом программы.

<<library>> (библиотека) – определяет разновидность компонента–файла, который представляется в форме динамической или статической библиотеки.

<<source>> (источник) – определяет разновидность компонента–файла, представляющего собой файл с исходным текстом программы, который после компиляции может быть преобразован в исполнимый файл.

<<table>> (таблица) – определяет разновидность компонента, который представляется в форме таблицы базы данных.

Отдельными разработчиками предлагались собственные графические стереотипы для изображения тех или иных типов компонентов, однако, за небольшим исключением они не нашли широкого применения. В свою очередь ряд инструментальных CASE–средств также содержат дополнительный набор графических стереотипов для обозначения компонентов.

6.9.6. Пример диаграммы компонентов

Диаграмма компонент ПЭ НСИ:

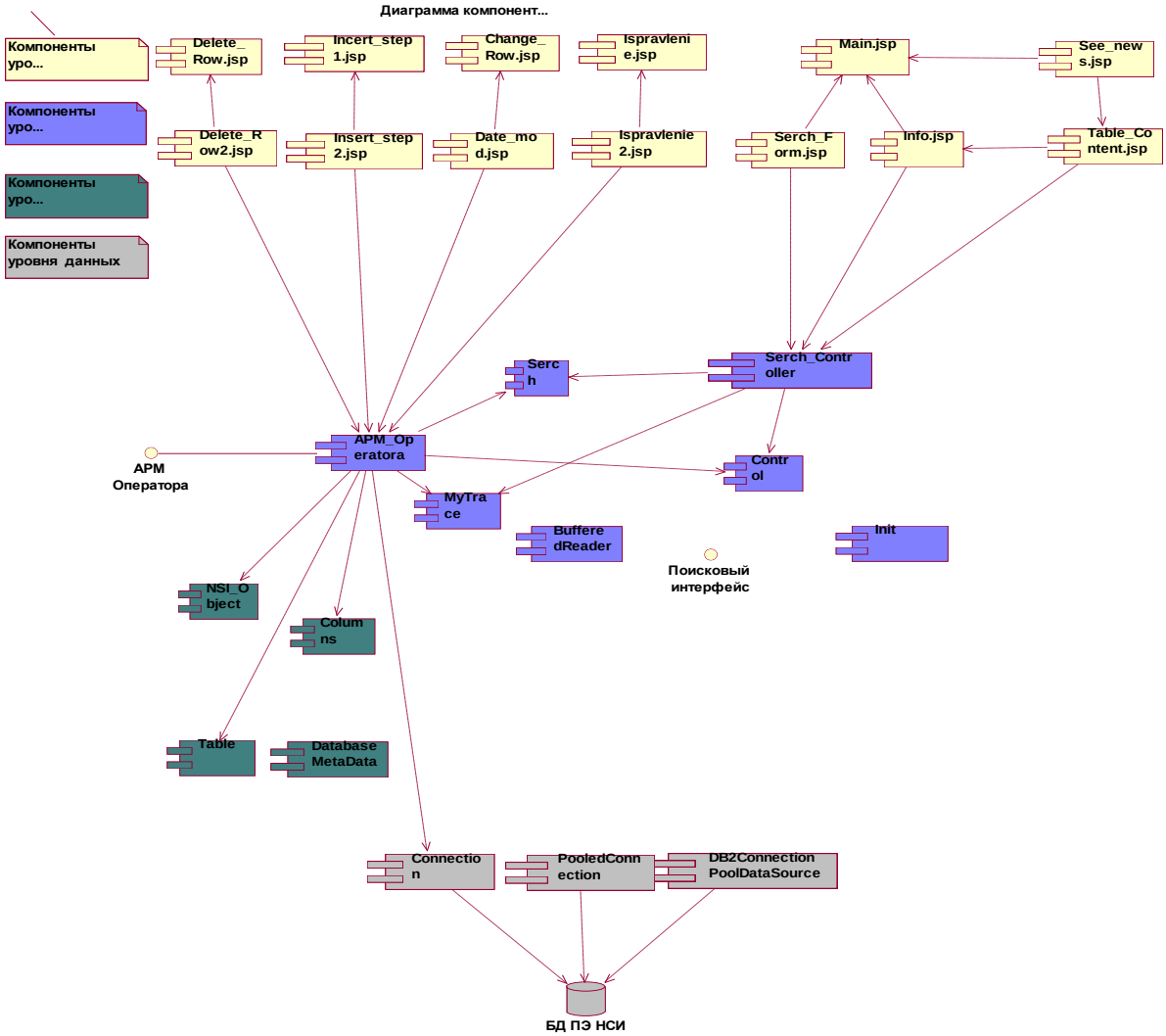


Рис. 6.107

Диаграмма компонент ПЭ НСИ:

Компоненты ПО ПЭ НСИ и взаимодействие между ними.

Компоненты уровня пользовательского представления: Страницы JSP, отображающие информацию и собирающие первичные данные пользовательских запросов.

Компоненты уровня управления: Классы, отвечающие за реализацию запросов пользователя.

Компоненты уровня виртуальной базы: Классы, реализующие виртуализацию БД ПЭ НСИ.

Компоненты уровня данных: Таблицы БД ПЭ НСИ и классы, реализующие непосредственную работу с ними.

6.10. Диаграмма развертывания

Сложные программные системы могут реализовываться в сетевом варианте, на различных вычислительных платформах и технологиях доступа к распределенным базам данных. Наличие локальной корпоративной сети требует решения целого комплекса дополнительных задач рационального размещения компонентов по узлам этой сети, что определяет общую производительность программной системы.

Интеграция программной системы с интернетом определяет необходимость решения дополнительных вопросов при проектировании системы, таких как обеспечение безопасности и устойчивости доступа к информации для корпоративных клиентов. Эти аспекты в немалой степени зависят от реализации проекта в форме физически существующих узлов системы, таких как серверы, рабочие станции, брандмауэры, каналы связи и хранилища данных.

Технологии доступа и манипулирования данными в рамках общей схемы "клиент–сервер" также требуют размещения больших баз данных в различных сегментах корпоративной сети, их резервного копирования, архивирования, кэширования для обеспечения необходимой производительности системы в целом. С целью спецификации программных и технологических особенностей реализации распределенных архитектур необходимо визуальное представление этих аспектов.

Первой из диаграмм физического представления является диаграмма компонентов. Вторая форма физического представления программной системы – это диаграмма развертывания (размещения).

Диаграмма развертывания (Deployment diagram) – диаграмма, на которой представлены узлы выполнения программных компонентов реального времени, а также процессов и объектов.

Диаграмма развертывания применяется для представления общей конфигурации и топологии распределенной программной системы и содержит изображение размещения компонентов по отдельным узлам системы. Кроме того, диаграмма развертывания показывает наличие физических соединений – маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы.

6.10.1. Узел диаграммы развертывания

Узел (Node) – это физический элемент, который существует во время выполнения и представляет вычислительный ресурс, обычно обладающий как минимум некоторым объемом памяти, а зачастую также и процессором. Графически узел изображается в виде куба.

В качестве вычислительного ресурса узла может рассматриваться один или несколько процессоров, а также объем электронной или магнитооптической памяти. Однако в языке UML понятие узла включает в себя не только вычислительные устройства (процессоры), но и другие механические или

электронные устройства, такие как датчики, принтеры, модемы, цифровые камеры, сканеры и манипуляторы.

Графически узел на диаграмме развертывания изображается в форме трехмерного куба. Узел имеет имя, которое указывается внутри этого графического символа. Сами узлы могут представляться как на уровне типа (рис. 6.108, а), так и на уровне экземпляра (рис. 6.108, б).

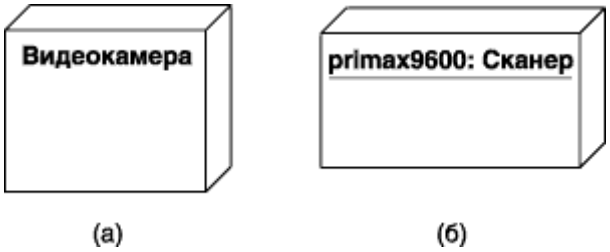


Рис. 6.108. Графическое изображение узла на диаграмме развертывания

Второй способ разрешает показывать на диаграмме развертывания узлы с вложенными изображениями компонентов (рис. 6.108, б). Важно помнить, что в качестве таких вложенных компонентов могут выступать только исполняемые компоненты и динамические библиотеки.

Каждый узел должен иметь имя, отличающее его от прочих узлов. Имя – это текстовая строка. Взятое само по себе, оно называется простым именем. Составное имя – это простое имя узла, к которому спереди добавлено имя пакета, в котором он находится. Имя узла должно быть уникальным внутри объемлющего пакета. Обычно при изображении узла указывают только его имя, как видно из рис. 6.109. Но, как и в случае с классами, вы можете снабжать узлы помеченными значениями или дополнительными разделами, чтобы показать детали.

Имя узла может состоять из любого числа букв, цифр и некоторых знаков препинания (за исключением таких, как двоеточия, которые применяются для отделения имени узла от имени объемлющего пакета). Имя может занимать несколько строк. На практике для именования узлов используют одно или несколько коротких существительных, взятых из словаря реализации.

В качестве дополнения к имени узла могут использоваться различные текстовые стереотипы, которые явно специфицируют назначение этого узла. Для этой цели были предложены следующие текстовые стереотипы: "processor" (процессор), "sensor" (датчик), "modem" (модем), "net" (сеть), "printer" (принтер) и другие, смысл которых понятен из контекста.

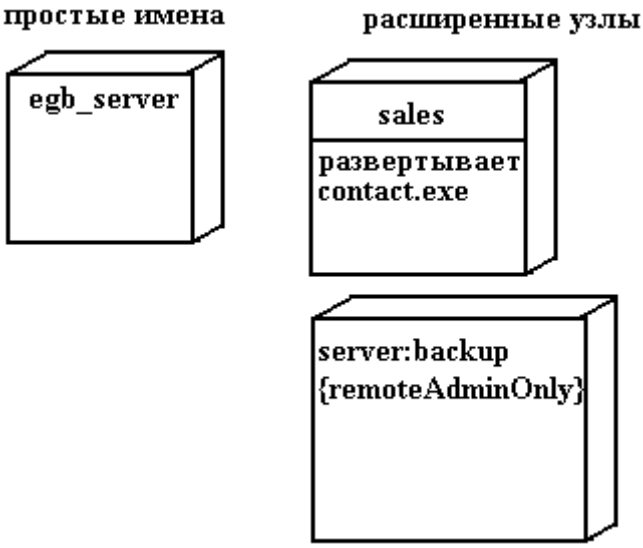


Рис. 6.109. Простое и расширенное изображение узлов.

Во многих отношениях узлы подобны компонентам. Те и другие наделены именами, могут быть участниками отношений зависимости, обобщения и ассоциации, бывают вложенными, могут иметь экземпляры и принимать участие во взаимодействиях. Однако между ними есть и существенные различия:

– компоненты принимают участие в исполнении системы; узлы – это сущности, которые исполняют компоненты;

– компоненты представляют физическую упаковку логических элементов; узлы представляют средства физического размещения компонентов.

Первое из этих отличий самое важное. Здесь все просто – узлы исполняют компоненты, компоненты исполняются в узлах.

Второе различие предполагает наличие некоего отношения между классами, компонентами и узлами. В самом деле, компонент – это материализация множества других логических элементов, таких как классы и кооперативные диаграммы, а узел – место, на котором развернут компонент. Класс может быть реализован одним или несколькими компонентами, а компонент, в свою очередь, развернут в одном или нескольких узлах. Как показано на рис. 6.110., отношение между узлом и компонентом, который на нем развернут, может быть явно изображено с помощью отношения зависимости. Как правило, вам не придется визуализировать такие отношения. Лучше хранить их как часть спецификации узла.

Множество объектов или компонентов, приписанных к узлу как группа, называется элементом распределения (Distribution unit).

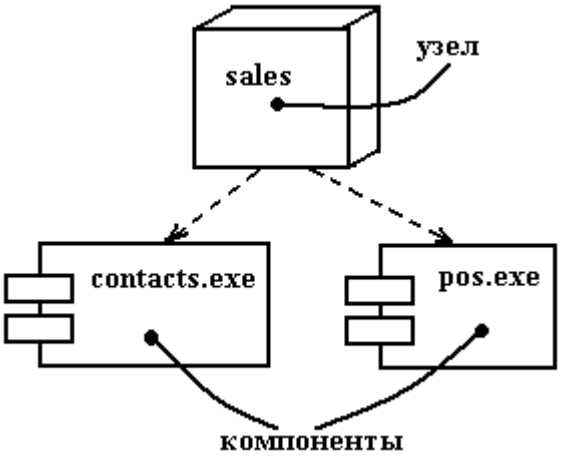


Рис. 6.110. Узлы и компоненты

Узлы можно организовывать, группируя их в пакеты, точно так же, как это делается с классами и компонентами.

Можно организовывать узлы, специфицируя отношения зависимости, обобщения и ассоциации (включая агрегирование), существующие между ними. Самый распространенный вид отношения между узлами – это ассоциация.

Компоненты не обязательно должны быть статически распределены по узлам системы. В UML можно моделировать динамическую миграцию компонентов из одного узла в другой, как бывает в системах, включающих агенты, или в системах повышенной надежности, в состав которых входят кластерные серверы и реплицируемые базы данных.

Хорошо структурированный узел обладает следующими свойствами:

– представляет четкую абстракцию некоей сущности из словаря аппаратных средств области решения;

– декомпозирован только до уровня, необходимого для того, чтобы донести ваши идеи до читателя;

– раскрывает только те атрибуты и операции, которые относятся к моделируемой области;

– явно показывает, какие компоненты на нем развернуты;

– связан с другими узлами способом, отражающим топологию реальной системы.

Изображая узел в UML, руководствуйтесь следующими принципами:

– определите для своего проекта или организации в целом набор стереотипов с подходящими пиктограммами, которые несут очевидную для читателя смысловую нагрузку;

– показывайте только те атрибуты и операции (если таковые существуют), которые необходимы для понимания назначения узла в данном контексте.

6.10.2. Отношения между узлами диаграммы

На диаграмме развертывания кроме изображения узлов указываются отношения между ними. В качестве отношений выступают физические соединения между узлами, а также зависимости между узлами и компонентами, которые допускается изображать на диаграммах развертывания.

Соединения являются разновидностью ассоциации и изображаются отрезками линий без стрелок. Наличие такой линии указывает на необходимость организации физического канала для обмена информацией между соответствующими узлами. Характер соединения может быть дополнительно специфицирован примечанием, стереотипом, помеченным значением или ограничением. Так, на представленном ниже фрагменте диаграммы развертывания (рис. 6.111.) явно определены рекомендации по технологии физической реализации соединений в форме примечания.

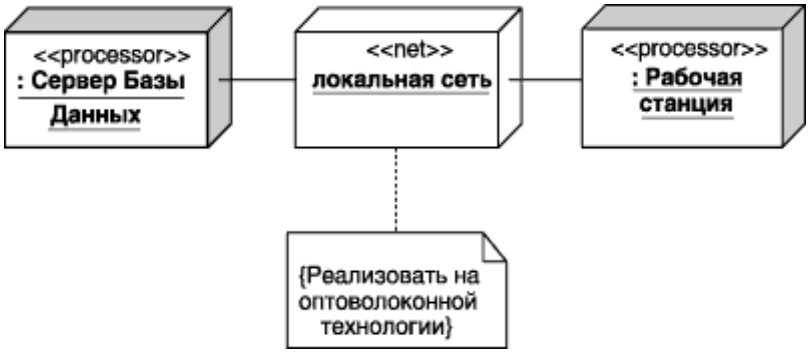


Рис. 6.111. Фрагмент диаграммы развертывания с соединениями между узлами

6.10.3. Пример диаграммы развертывания

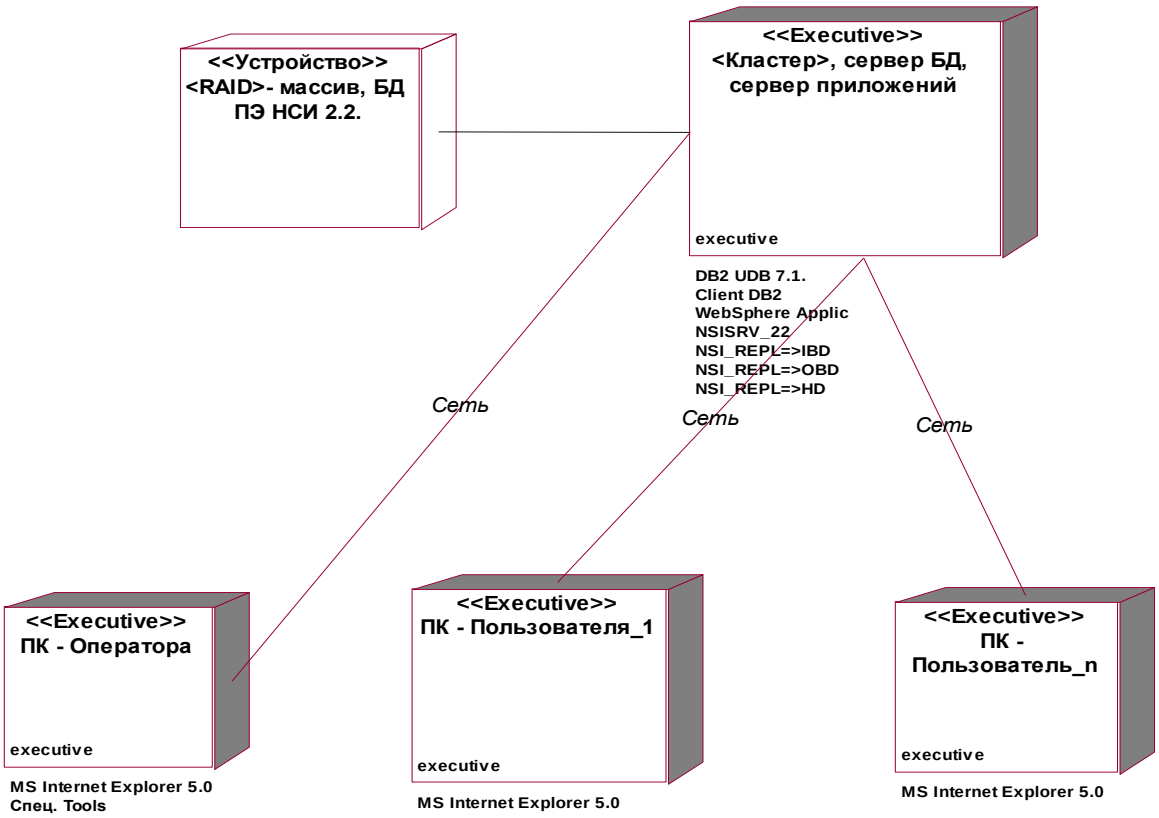


Рис. 6.112.

7. Сервис-ориентиированная архитектура (SOA)

7.1. Основные проблемы ИТ

Разработка большого количества систем ведет к беспрецедентной сложности организации их взаимодействия (см. рис. 7.1.). Паутина технологий и поставщиков приела к невозможности сопровождения ИТ подразделениями всей номенклатуре приложений, более 50% бюджета крупных ИТ компаний направлено на сопровождение (т.е. на back office). И как следствие:

- несоизмеримость затрат на сопровождение со значением для бизнеса;
- длительный возврат инвестиций;
- не достигаются требования бизнеса;
- наблюдается запаздывание с внедрением новых услуг.

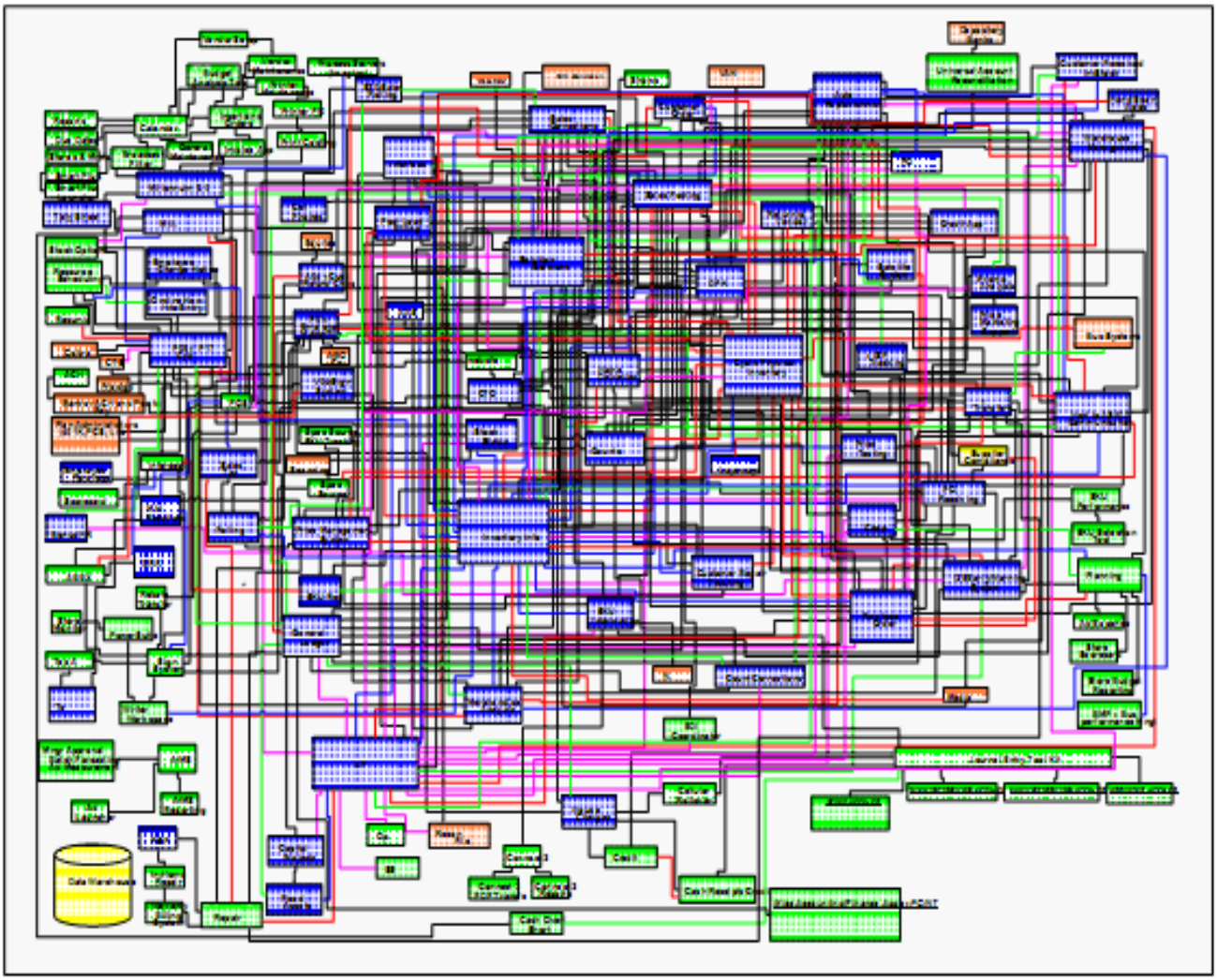


Рис. 7.1. Сложность взаимодействия ПО

7.2. Основные факторы внедрения SOA

На рис. 7.2 приведены основные факторы, влияющие на реализацию приложений на основе SOA.

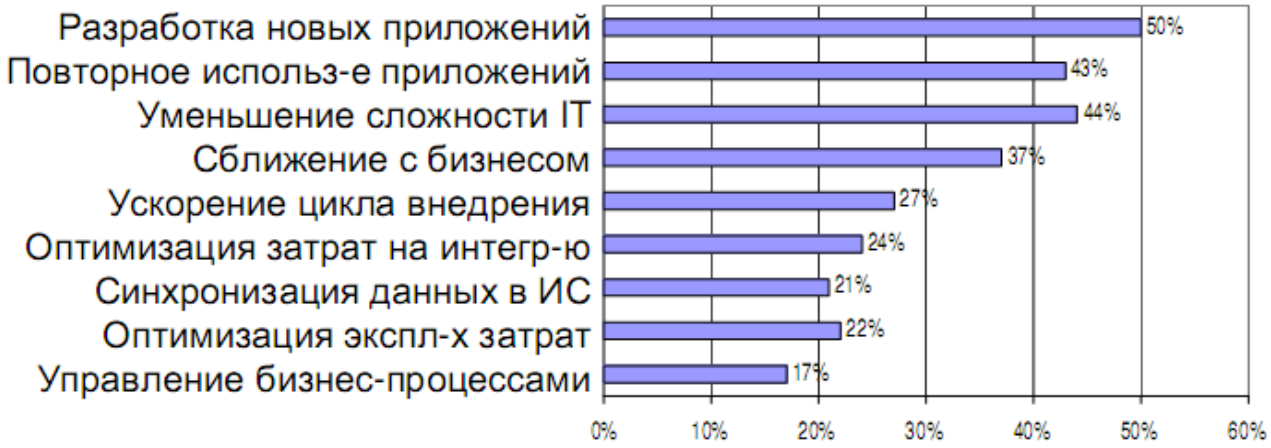


Рис. 7.2. Основные факторы внедрения SOA

7.3. Определения, цели и принципы SOA

Се́рвис-ориенти́рованная архитекту́ра (англ. **SOA**, service-oriented architecture) — модульный подход к разработке программного обеспечения (в дальнейшем ПО), основанный на использовании сервисов (служб) со стандартизированными интерфейсами, которые иденцифицируются web-адресом.

Веб-служба является единицей для обеспечения модульности.

Сервис-ориентированная архитектура (SOA) представляет собой стиль создания архитектуры ИТ, направленный на превращение бизнеса в ряд связанных сервисов - стандартных бизнес-задач, которые можно при необходимости вызывать через сеть. Это может быть Интернет, локальная сеть или географически распределённая сеть, объединяющая различные технологии и сочетающая сервисы из Нью-Йорка, Лондона и Гонконга так, как если бы они были установлены на локальной машине. Для выполнения определенной бизнес-задачи можно будет объединять множество таких сервисов. Это позволит компании быстро адаптироваться под изменение условий и требований рынка.

В основе SOA лежат принципы многократного использования функциональных элементов информационных технологий (в дальнейшем ИТ), ликвидации дублирования функциональности в ПО, унификации типовых

операционных процессов, обеспечения перевода операционной модели компании на централизованные процессы и функциональную организацию на основе промышленной платформы интеграции.

Архитектура не привязана к какой-то определённой технологии. Она может быть реализована с использованием широкого спектра технологий, включая такие технологии как REST, RPC, DCOM, CORBA или веб-сервисы. SOA может быть реализована используя один из этих протоколов и, например, может использовать, дополнительно, механизм файловой системы, для обмена данными.

Таким образом, системы, основанные на SOA, могут быть независимы от технологий разработки и платформ (таких как Java, .NET и т. д.). К примеру, сервисы, написанные на C#, работающие на платформах .NET и сервисы на Java, работающие на платформах Java EE, могут быть с одинаковым успехом вызваны общим составным приложением.

Концепция SOA предоставляет каркас для наилучшей интеграции систем, удовлетворяющий требованиям бизнеса. Эта концепция является усовершенствованием практических методов, которые уже существуют в таких методах программной интеграции, как управляемые процессы «сверху вниз», «снизу вверх» и «от краев к центру». Сейчас она стала реальной с развитием технологий и с появлением настоящей совместимости.

Основная причина резко возросшего интереса к SOA, наблюдаемого в последние годы, заключается в том, что вместе с SOA впервые открылась возможность преодолеть хронический разрыв между бизнесом и информационными технологиями. Не требует доказательств, что подлинная интеграция бизнеса с IT открывает огромные перспективы, но ее реализация требует преодоления серьезных препятствий. Начать следует хотя бы с того, что проектирование SOA предполагает наличие бизнес-проекта (business design).

Цели

Для крупных информационных систем уровня предприятия и выше:

- сокращение издержек при разработке приложений, за счёт упорядочивания процесса разработки;
- расширение повторного использования кода;
- независимость от используемых платформ, инструментов, языков разработки;
- повышение масштабируемости создаваемых систем;
- улучшение управляемости создаваемых систем.

Принципы SOA

- Архитектура, как таковая, не привязана к какой-то определённой технологии.
- Независимость организации системы от используемой вычислительной платформы (платформ).

- Независимость организации системы от применяемых языков программирования.
- Использование сервисов, независимых от конкретных приложений, с единообразными интерфейсами доступа к ним.
- Организация сервисов как слабосвязанных компонентов для построения систем.

7.4. Жизненный цикл сервисов

Жизненный цикл сервиса является частью этапа внедрения, показанного ниже на диаграмме жизненного цикла SOA (см. рис. 7.3.).

Рис. 7.3. Жизненный цикл SOA

Жизненный цикл сервиса начинается в момент его ввода в эксплуатацию (определения) и заканчивается в момент его вывода из эксплуатации или переориентирования. Жизненный цикл сервиса предполагает управление сервисом на трех этапах: этапе формирования требований и анализа, этапе проектирования и разработки и этапе эксплуатации в ИТ-среде.

Рис. 7.4. Три этапа жизненного цикла сервисов

Диаграмма на рис. 7.4. показывает три этапа жизненного цикла сервисов и указывает на необходимость наличия репозитория сервисов уровня предприятия, необходимого для управления сервисами.

Формирование требований и анализ: сначала определяются бизнес-цели и расставляются приоритеты. Исходя из установленных приоритетов, нетехнический персонал, совместно с бизнес-аналитиками, начинает работу по **документированию бизнес-процессов, правил и требований**. Основные требования включают следующие:

1. Визуальное представление бизнес-процессов начиная с уровня 0 и далее
2. Определение каждого бизнес-процесса
3. Определение владельцев каждого из процессов
4. Определение целей и текущих недостатков бизнес-сервисов
5. Определение входных и выходных элементов данных
6. Расстановка приоритетов бизнес-процессов и бизнес-сервисов
7. Проработка всех аспектов определений бизнес-сервисов
8. Имитация пользовательских интерфейсов и/или бизнес-процессов.

Проектирование и разработка: на стадии проектирования бизнес-аналитики работают с системным архитектором для передачи последнему бизнес-требований. Архитектор ответствен за выполнение основной оценки, проектирование и передачу сервисов на разработку. Разработчики ответственны за разработку, сборку, тестирование и предоставление

составного приложения для его эксплуатации в ИТ-среде. Некоторые основные требования к проектированию следующие:

1. Проверка требований и определение альтернативных вариантов для каждого бизнес-процесса
2. Проектирование и оценка компонентов каждого сервиса, таких как портал, интеграция, инфраструктура, данные, политики и бизнес-сервисы (логические сервисы)
3. Определение возможностей повторного использования бизнес-сервисов
4. Разработка и ввод в эксплуатацию в соответствии с подробным планом реализации проекта
5. Отслеживание процессов и предоставление отчетности руководству (в т.ч. ИТ-руководителям)
6. Получение одобрения со стороны руководства при поставке каждого бизнес-сервиса.

Эксплуатация в ИТ-среде: группа, ответственная за эксплуатацию, осуществляет тестирование сервиса и его подготовку к внедрению, а также подготовку производственной среды. При этом, основное внимание уделяется подготовке производственной среды. Кроме того, данная группа рассчитывает структуру сети и центров обработки данных, а также отвечает за развертывание, мониторинг и предоставление начальной поддержки по всем приложениям, поддержка которых осуществляется ИТ-структурой. Некоторые основные требования к проектированию следующие:

1. Проверка требований и определение требований к инфраструктуре
2. Создание системной среды, включающей развертывание системы, проверку степени интеграции системы, тестирование производительности, одобрение системы пользователями и создание программной среды
3. Оказание помощи группам разработчиков при конфигурировании систем и приложений, создании новых сборок и планировании производительности
4. Отслеживание и управление зависимостями между сервисами и ресурсами
5. Развертывание и управление бизнес-сервисами в производстве
6. Предоставление поддержки по приложениям для бизнес-сервисов исходя из приоритетов

Более подробное описание каждого из этапов будет приведено далее в данном разделе. Ниже на рис. 7.5. следует описание основных ИТ-процессов,

нацеленных на предоставление комплексных решений для разработки бизнес-сервисов.

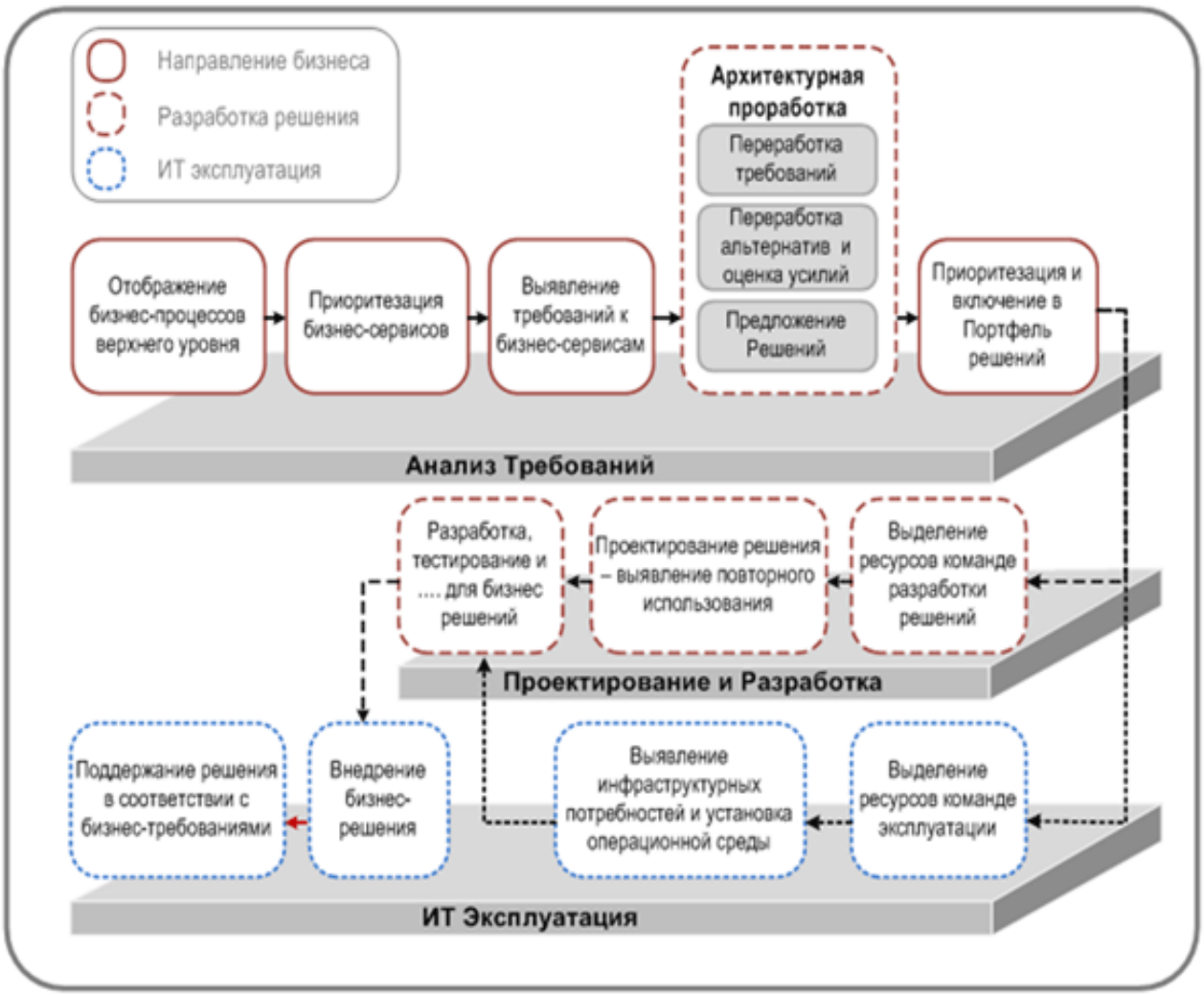


Рис. 7.5. ИТ-процесс предоставления комплексных решений

Данный процесс также определяет роль каждой из трех организационных групп в предоставлении бизнес-решений.

7.4.1. Управление жизненным циклом сервиса

Управление — это процессы, средства и организационная структура, наличие которых необходимо для успешной реализации SOA. Эффективное повторное использование сервиса может быть достигнуто только в случае следования стандартам и процедурам в течение всего жизненного цикла этого сервиса. Ввиду совместного использования сервисов приложениями, они должны проектироваться, разрабатываться и развертываться с особой

тщательностью, т.к. только так можно избежать негативного воздействия на уже имеющихся пользователей.

При использовании сервисов различными организационными единицами возникает конфликт приоритетов. Эффективное управление позволяет максимально обеспечить возможность повторного использования сервисов при минимальных потерях. Среди основных задач управления SOA можно выделить следующие:

- публикация стандартов и руководств по SOA;
- определение и реализация процессов, способствующих использованию и повторному использованию сервисов в рамках проекта;
- контроль всех находящихся в разделяемом доступе сервисов предприятия или подразделения;
- поощрение создания стандартов и руководств в рамках организации;
- информирование о результатах SOA в рамках организации;
- управление сервисами укрепляет полный жизненный цикл сервисов.

7.4.2. Формирование требований и анализ

Бизнес-аналитики работают вместе с представителями бизнеса и определяют бизнес-требования, желательно в форме бизнес-процессов. Для начального проекта SOA, участники обычно фокусируются на бизнес-процессах, которые не затрагивают всё предприятие или подразделение полностью, а ограничены рамками, заданными руководящей группой, и укладываются в рамки финансирования. Группа определяет бизнес-логику комплексного приложения, которое должно быть представлено.

После определения бизнес-процессов, аналитики находят все процессы, дублирующиеся в рамках предприятия или подразделения. Затем проводится поиск сервисов, которые могут быть повторно использованы разработчиками. После этого бизнес-аналитики загружают артефакты в репозиторий SOA, что, в свою очередь, запускает процесс управления.

Процесс управления определяется организацией и проектом. Группы участников не должны рассматривать данный этап как завершенный без получения всех разрешений, особенно от владельцев бизнеса.

7.4.3. Проектирование

Бизнес-аналитики передают требования и бизнес-процессы системным архитекторам чтобы те разработали приложение. Каждая ИТ-структура, как правило, имеет собственный подход к разработке приложений.

В ходе данного этапа архитекторы определяют сервисы и способы их внедрения. Затем, по репозиторию осуществляется поиск сервисов, которые

могут быть использованы повторно. Архитекторам не стоит ограничивать поиск только теми сервисами, находящимися в производстве в настоящее время, можно расширить поиск по сервисам, которые разрабатываются другими группами разработчиков.

На конечной стадии данного процесса, архитекторы уже могут иметь представление об уже имеющихся сервисах, которые могут быть использованы повторно, сервисах, которые должны быть модифицированы для создания новой версии, сервисах, которые необходимо развернуть, и сервисах, которые должны быть выведены из эксплуатации.

Архитекторы загружают всех проектные артефакты в репозиторий SOA, что, в свою очередь, запускает процесс управления, включающий в себя получение разрешений от подразделений, занимающихся инспекцией архитектуры, руководителей проекта и служб эксплуатации. Кроме того, руководители проекта также могут использовать данную информацию для распределения заданий по разработке.

7.4.4. Разработка сервиса

Системный архитектор направляет группам разработчиков план разработки, причем, желательно из реопзитория SOA. Группы разработчиков могут находиться в разных местах, а каждая группа может специализироваться на определенной сфере деятельности или на определенном продукте.

Группы разработчиков разрабатывают и многократно тестируют комплексное приложение, после чего загружают артефакты в репозиторий предприятия. Когда разработчики объявляют о том, что сервис готов к развертыванию, запускается процесс управления.

7.4.5. Эксплуатация в ИТ-среде

Группа, ответственная за эксплуатацию, как правило, обеспечивает разработку, контроль качества, подготовку к внедрению и подготовку производственной среды. При получении плана разработки от архитекторов, группы, ответственные за эксплуатацию в ИТ-среде, создают среду для разработки. Кроме того, эти группы часто управляют средой контроля качества, т.к. она должна быть идентична производственной среде.

Группа разработчиков, как правило, предоставляют группе эксплуатации определенную сборку продукта. Для комплексных приложений, состоящих из сервисов, разработчики предоставляют сборку сервисов. Лучшим вариантом, который можно рекомендовать, будет сборка сервисов исходя из информации, хранящейся в репозитории SOA.

После того, как группа эксплуатации завершила сборку, сервисы разворачиваются в целевом узле. Политики функционирования, безопасности и управления определяются бизнес-аналитиком и архитектором на более ранних стадиях. В обязанности группы эксплуатации также входит осуществление мониторинга и предоставления отчетных данных. Это делается в целях отслеживания ключевых показателей производительности и контроля выполнения соглашений об оказании ИТ-услуг.

Кроме того, хорошей практикой является предоставление данных о продукте (включая данные об аппаратном обеспечении, имени узла, версии продукта и версии приложения) в репозиторий SOA.

7.4.6. Мониторинг процессов

Бизнесу требуется различная информация, которая сочетает в себе данные систем контроля, эксплуатационные данные и данные пакетной обработки. Подобная информация может относиться к одной из следующих категорий:

Соглашение о предоставлении ИТ-услуг

Контроль бизнес-активности

Управление политиками

Модель зрелости сервиса (матрица, необходимая для контроля жизненного цикла сервиса, и атрибут, по которому может осуществляться поиск в репозитории сервисов).

7.4.7. Эталонная модель сервис-ориентированных архитектур

Цель данной эталонной модели состоит в определении сущности сервис-ориентированной архитектуры и обозначении терминологии, а также описании общего понимания SOA.

Модель определяет связи, которые являются значимыми для SOA, как абстрактной модели, независимой от варианта реализации и от постоянно развивающихся технологий, которые могли бы повлиять на внедрение SOA.

На Рис. 7.6. показано как эталонная модель SOA относится к другим частям архитектуры распределенных систем. Концепции и связи, определенные эталонной моделью создают базис для описаний эталонных архитектур и моделей, которые будут определять более специфические виды SOA проектирования. Определенные архитектуры появляются из комбинаций эталонных архитектур, моделей архитектур и дополнительных требований, которые накладываются техническим окружением

Архитектура должна отвечать целям, мотивам и требованиям, которые решают актуальные проблемы, тогда как эталонная модель определят базовые классы решений, конкретная архитектура описывает определенное решение.

Зачастую архитектура разрабатывается в контексте предопределенной конфигурации, включающей протоколы, профили, спецификации и стандарты.

Внедрение SOA подразумевает комбинирование всех элементов от общих архитектурных принципов, специфичных инфраструктур, которые определяют текущие потребности до своеобразных методов внедрения, которые будут созданы и использованы в определенных условиях.

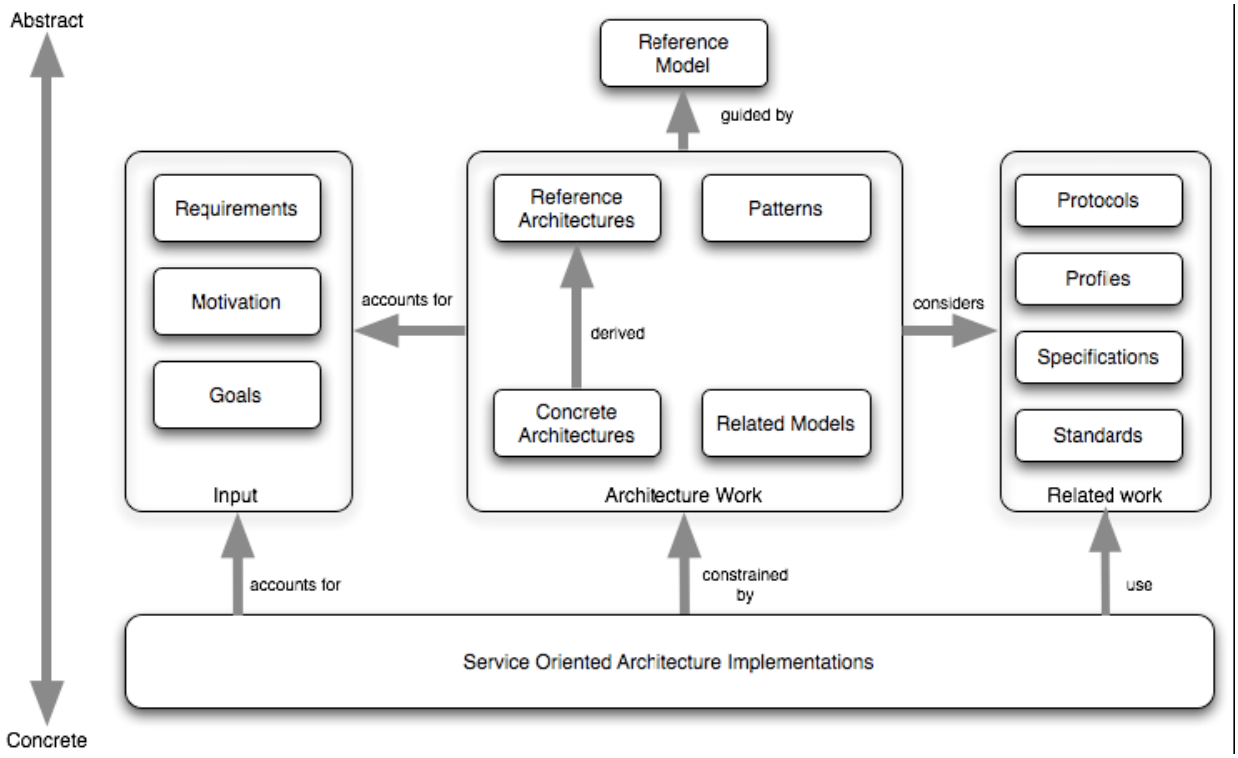


Рис. 7.6. Связи эталонной модели с частями ее реализации

8. WEB-Сервисы

«Web-сервисы обеспечивают однородное представление неоднородной среды» - Джеймс Гослинг (вице-президент компании Sun Microsystems, один из создателей языка Java).

В индустрии программного обеспечения всегда решались задачи интеграции программных приложений функционирующих на различных операционных системах, языках программирования и аппаратных платформах.

Традиционно приложения были жестко связаны, когда при удаленном сетевом вызове одно приложение жестко привязано к другому названием функции

с ее параметрами. В настоящее время большинство приложений разрабатываются в распределенной, неоднородной среде (см. рис. 8.1.) и проблема разработки слабосвязанных приложений является одной из основных, которая стоит перед разработчиками ПО.

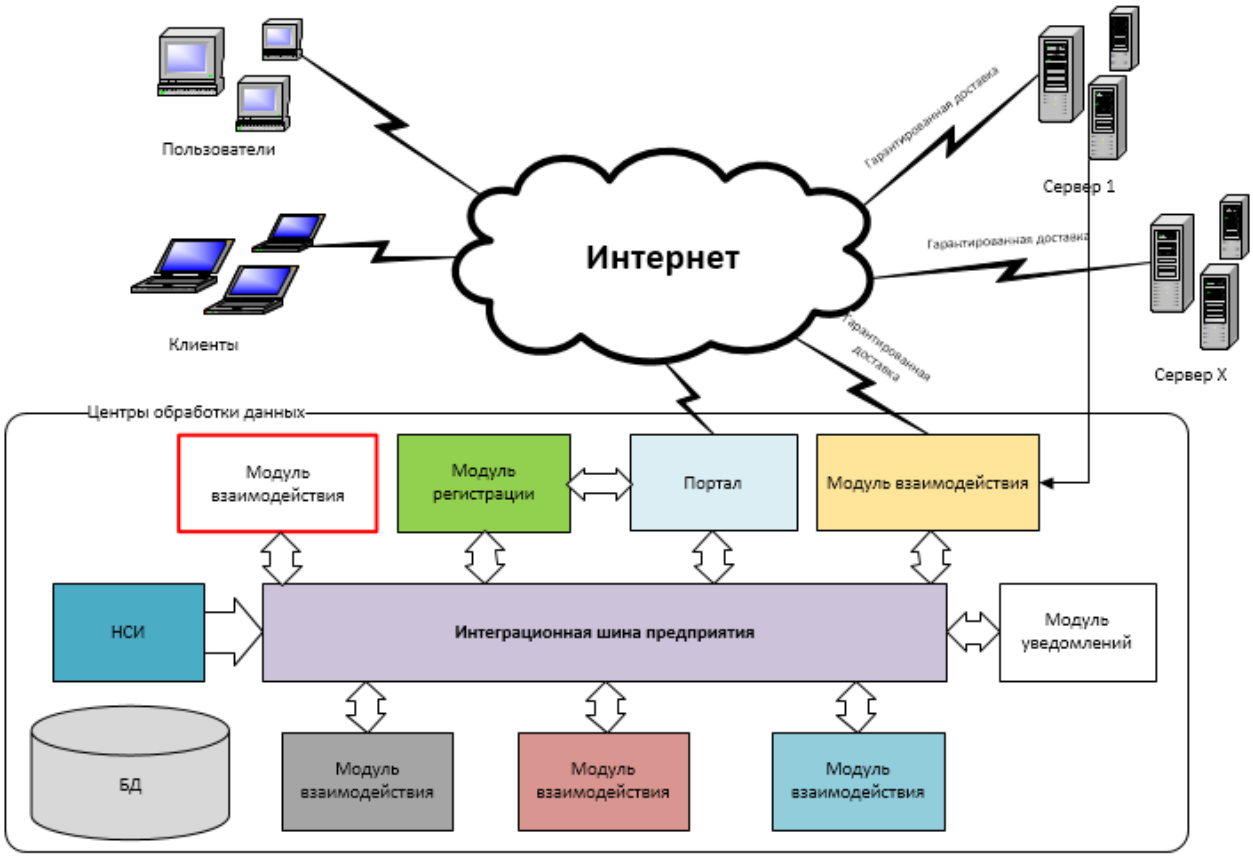


Рис. 8.1 Распределенная, неоднородная среда

С распределенными системами связан ряд трудностей:

- сложность самой среды;
- трудность интеграции неоднородных компонентов;
- обеспечение безопасности;
- управление ресурсами (выделение, организация одновременного доступа, поддержка пулов ресурсов);
- масштабируемость;

- поддержка транзакций;
- управление удаленными объектами (активация, создание, удаление, обращение по имени).

8.1. Понятия технологии WEB-Сервисов

Web-сервисы (Web-службы) это технология, которая позволяет приложениям взаимодействовать друг с другом независимо от платформы, на которой они развернуты, а также от языка программирования, на котором они написаны. **Web-сервис** - это программный интерфейс, который описывает набор операций, которые могут быть вызваны удаленно по сети посредством стандартизированных XML сообщений.

Web-сервисы используют XML. XML позволяет описать любые данные независимым от платформы способом, что, в свою очередь, приводит к слабо-связным приложениям. Кроме того, Web-сервисы могут функционировать на более высоком уровне абстракции, анализируя, модифицируя или обрабатывая типы данных динамическим образом по требованию. Значит, с технической точки зрения, Web-сервисы могут обрабатывать данные значительно легче, предоставляя возможность программному обеспечению взаимодействовать более открыто (см. рис. 8.2).

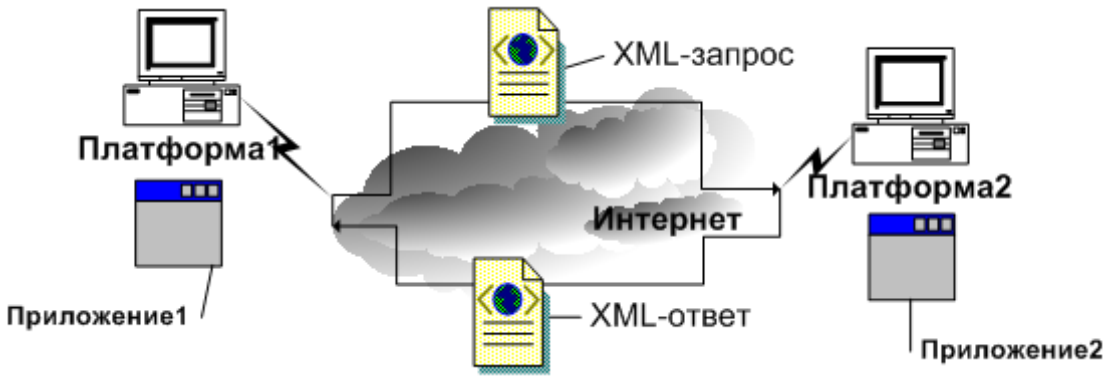


Рис. 8.2. Взаимодействие Web-сервисов

На самом высоком концептуальном уровне мы можем рассматривать Web-сервисы как единицы приложения, каждая из которых занимается выполнением определенной функциональной задачи. Если подняться на уровень выше, то эти задачи можно объединить в бизнес-ориентированные задачи для выполнения определенных бизнес операций. Таким образом, после того как технические

специалисты разработали Web-сервисы, архитекторы бизнес процессов могут объединить их для решения конкретных бизнес задач. Если взять за аналогию автомобиль, то при сборке кузова, двигателя, трансмиссии и других составляющих, архитектор бизнес процессов может брать двигатель целиком, не вдаваясь в подробности тех составляющих, из которых собран каждый двигатель. Кроме того, динамическая платформа означает, что двигатель может работать с трансмиссией или другими компонентами автомобиля от других производителей.

Из последнего рассмотренного аспекта вытекает, что Web-сервисы помогают в рамках организации преодолеть разрыв между техническими специалистами и людьми, далекими от тонкостей технологий. Web-сервисы упрощают процесс понимания технических операций. Последние могут описывать события и различные типы деятельности, а технические специалисты могут ассоциировать их с соответствующими сервисами.

При использовании универсально описанных интерфейсов в хорошо спроектированных задачах Web-сервисы становится значительно легче использовать повторно, а, следовательно, и приложения, которые они представляют. Повторное использование программных приложений означает лучший возврат инвестиций в программное обеспечение, поскольку используя те же ресурсы, вы можете получить больше. Это позволяет руководителям рассматривать новые возможности использования существующих приложений или предложить их партнерам для использования в новых задачах, повышая тем самым степень взаимодействия между партнерами.

Следовательно, *основной проблемой, для которой Web-сервисы могут служить решением, является интеграция данных и приложений,* а также преобразования технических функций в бизнес-ориентированные задачи. Эти два аспекта позволяют различным компаниям взаимодействовать на уровне процессов или приложений с их партнерами, оставляя при этом возможность динамически адаптироваться к новым ситуациям или работать с различными партнерами по требованию.

8.2. Основные технологии Web-сервисов

Основными технологиями, используемые для построения Web-сервисов являются:

- **XML**: Extensible Markup Language (расширяемый язык разметки, XML) – это язык разметки, который лежит в основе большинства спецификаций, используемых в Web-сервисах. XML – это исходный язык, с помощью которого можно описать любые данные в структурированном виде, независимо от их представления в конкретном устройстве;
- **SOAP**: Simple Object Access Protocol (простой протокол доступа к объектам, SOAP) – это сетевой, транспортный и программный язык, а также не зависимый от платформы протокол, позволяющий клиенту вызвать удаленный сервис. Сообщения имеют формат XML;

- **WSDL**: Web Services Description Language (язык описания Web-сервиса, WSDL) – это интерфейс, основанный на XML, а также язык описания реализации. Поставщик сервиса использует WSDL-документ для описания операций, выполняемых Web-сервисом, а также параметров и типов данных, которые она использует. WSDL- документ также содержит информацию для доступа к сервису;
- WSIL: Web Services Inspection Language (язык обследования Web-сервисов, WSIL) – это спецификация на основе XML, которая позволяет находить Web-сервис без использования UDDI. Тем не менее, WSIL также можно использовать в сочетании с UDDI, т.е. он является не зависимым от UDDI и не подменяет его;
- **UDDI**: Universal Description, Discovery, and Integration (универсальное описание, поиск и взаимодействие, UDDI) – это клиентский API, а также серверная реализация на основе SOAP, которые можно использовать для хранения и получения информации о поставщиках сервисов и Web-сервисах (см. рис. 8.3.)

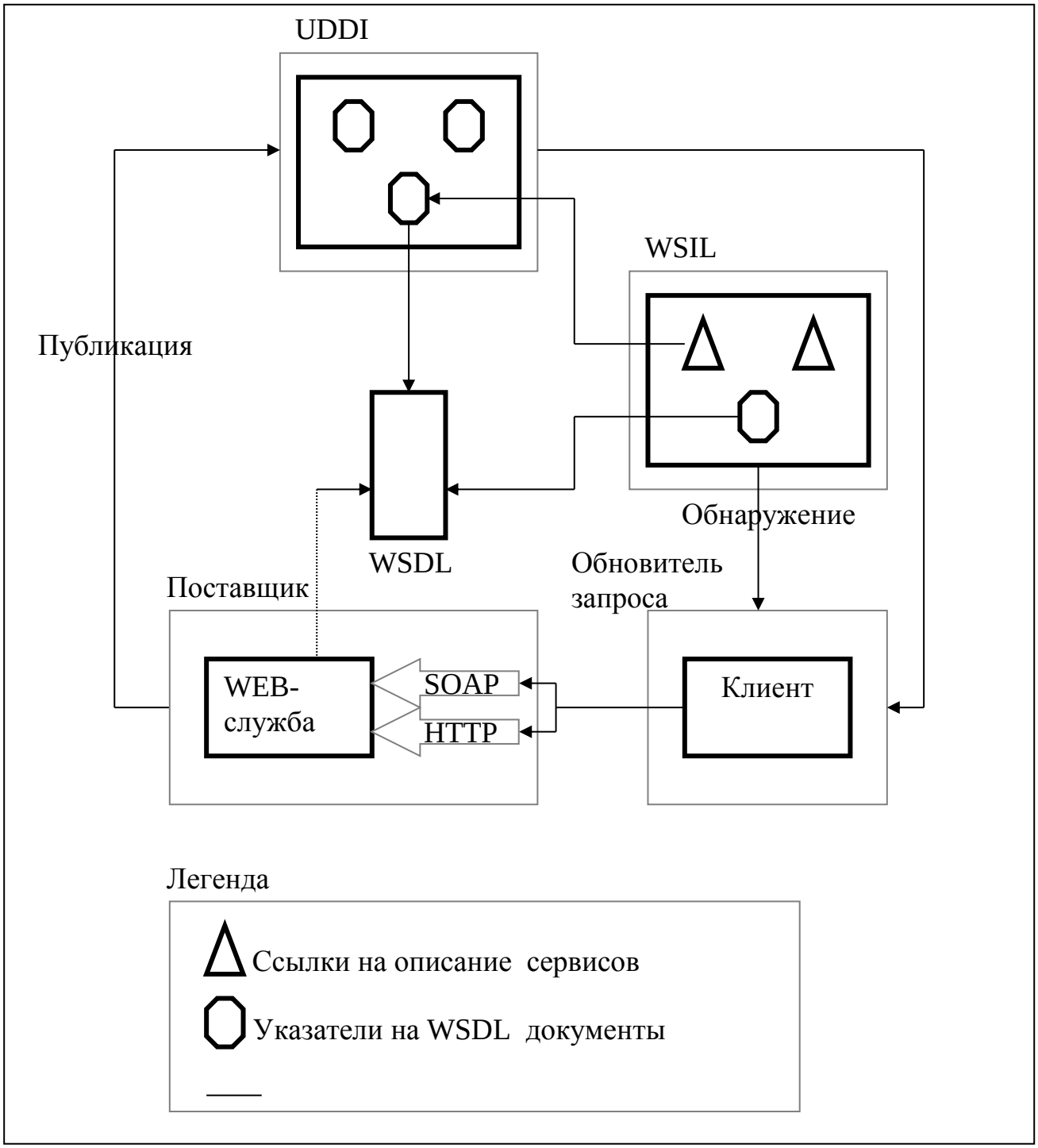


Рис. 8.3. Основные технологии для построения Web-сервисов

8.3 Использование WEB-Сервисов

Web-сервис позволяет компьютерным программам стандартизировано общаться между собой.

Web-сервис является абстрактным компонентом - равно как абстрактна концепция диалога между людьми. Во время «диалога» - программы общаются, используя знакомый им язык. Благодаря Web-сервисам программы могут, как находиться как на одном компьютере, так и размещаться на разных машинах в разных точках земного шара, соединённые через интернет. Программы и компьютеры не обязательно должны быть одинаковыми: это могут быть как две программы Microsoft .NET на одном ноутбуке, так и программа Java на канадском сервере iSeries, программа C++ на компьютере или ОС Linux из Китая.

Как было сказано ранее: в коммуникациях посредством Web-сервисов используются следующие стандартные технологии (см. рис. 8.4):

- XML;
- Протокол SOAP;
- Библиотека описаний Web-сервисов (WSDL) .

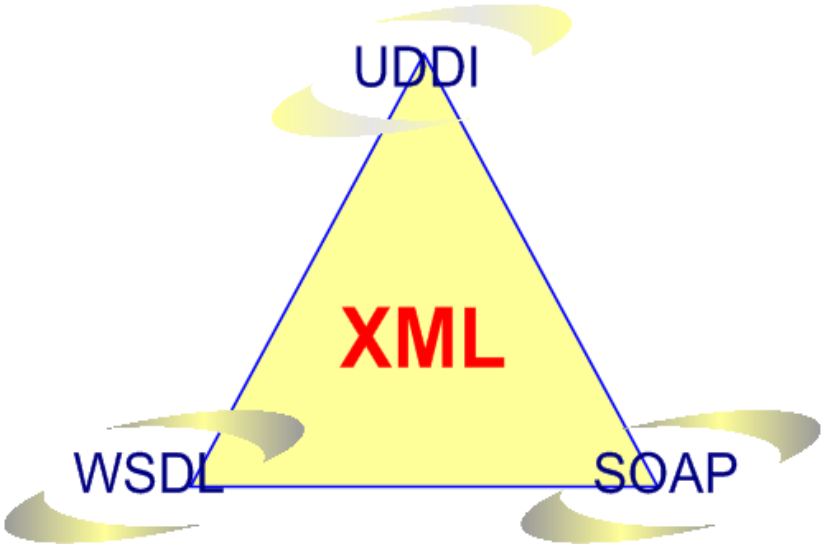


Рис. 8.4 Пирамида взаимодействия

8.3.1 Пример XML

Язык XML используется для общения между компонентами Web-сервисов. Сообщения, пересылаемые между приложениями, равно как определяющие Web-сервис файлы, имеют формат XML. Ниже приведена структура простого файла XML.

```
?xml version=”1.0” encoding=”UTF-8”?>
        <person xmlns=”um:mydomain:people” job=”developer”>
            <firstname>John</firstname>
            <lastname>Doe</lastname>
            < birthday >1970-01-25</birthday>
        </person>
```

Как можно видеть, некоторая информация в файле (такая как имя, фамилия) окружена тегами, заключёнными в треугольные скобки. Имя John показано как <firstname>John</firstname>. Ещё есть элементы, в которые вложены другие элементы, например в элемент <person> вложены элементы <firstname>, <lastname> и <birthday>.

Разработка Web-сервисов на языке XML даёт следующие преимущества:

- Структура и грамматика XML аналогична структуре остальных языков программирования, поэтому взаимодействующим с Web-сервисами программам не надо проводить структурный анализ файлов XML напрямую.
- Файлы XML текстовые, и их может прочитать человек (т.е., зная язык XML, можно открыть файл XML в текстовом редакторе и понять его содержимое). Это может помочь при отладке программ, использующих XML.
- XML позволяет использовать в сообщениях любую стандартную кодировку, поэтому сообщения можно писать как на английском, так и на русском или японском языках.
- XML позволяет пользоваться так называемым пространством имен, в котором вы можете предопределить желаемую структуру файлового элемента с определённым именем. Например, вы можете определить элемент Price, который всегда должен быть числом с плавающей точкой, или PersonName, включающий в себя два строковых подэлемента FirstName и LastName.

Недостатками XML являются:

- Язык XML жёсткий, поэтому любое неправильное форматирование сообщения XML приводит к сбою анализа всего сообщения (даже если проблему легко интерпретировать или пропустить). Однако если вы используете стандартную библиотеку для генерации файлов XML (что и делается при создании Web-сервисов), библиотека сама проверяет правильность форматирования.
- Сообщение XML хранится в обычном текстовом файле, а потому занимает больше места, чем его эквивалент в другом формате (в таких, как разделённый, двоичный или "самодельный" формат).

Но эти проблемы не значительны по сравнению с преимуществами формата XML.

8.3.2 Описание SOAP

Как было приведено выше, общение Web-сервисов ведётся в формате XML, однако это решает лишь половину проблемы. Приложения могут разобрать сообщение, однако им неизвестно, что делать с полученным после анализа результатом. Инструкция, описывающая правила форматирования сообщений XML для Web-сервисов известна как SOAP. В нем определена структура сообщений, благодаря чему программы знают, как отправлять и интерпретировать данные. Базовая структура сообщения SOAP показана на рисунке 8.5.

Рис. 8.5. Базовая структура сообщения SOAP

В базовом случае пакет SOAP, включает в себя тело SOAP, в котором находятся передаваемые данные. Иногда ещё есть необязательный заголовок SOAP (внутри пакета перед телом), содержащий дополнительную информацию.

Примеры на языке XML:
Простой пример, содержит только тело пакета.

```
<SOAP-ENV:Body>
        <ns1:GetStockInfo xmlns:ns1="urn:thisNamespace">
                <symbol>FOO</symbol>
        </ns1:GetStockInfo>
 </SOAP-ENV:Body>
```

Следующий пример иллюстрирует запрос SOAP, называемый GetLastTradePrice, который позволяет клиенту послать запрос о последних котировках определенных акций.

```
POST/StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <SOAP-ENV:Body>
            <m:GetLastTradePrice xmlns:m="Some-URI">
                  <symbol>DIS</symbol>
            </m:GetLastTradePrice>
      </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

В первых пяти строчках (часть заголовка HTTP) указывается тип сообщения (POST), хост, тип и длина информационного наполнения, а заголовок SOAPAction определяет цель запроса SOAP. Само сообщение SOAP представляет собой документ XML, где сначала идет конверт SOAP, затем элемент XML, который указывает пространство имен SOAP и атрибуты, если таковые имеются. Конверт SOAP может включать в себя заголовок (но не в данном случае), за которым следует тело SOAP. В нашем примере в теле содержится запрос GetLastTradePrice и символ акций, для которых запрашиваются последние котировки. Ответ на этот запрос может выглядеть следующим образом.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    <SOAP-ENV:Body>
        <m:GetLastTradePriceResponse xmlns:m="Some-URI">
            <Price>34,5</Price>
        </m:GetLastTradePriceResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

И опять-таки первые три строки — это часть заголовка HTTP; само сообщение SOAP состоит из конверта, который содержит ответ на исходный запрос, помеченный GetLastTradePriceResponse, и включает в себя возвращаемое значение, в нашем случае 34,5.

Примечание:

Хотя формат SOAP стандартен и имеет одинаковые инструкции, необходимо помнить, что разные производители могут немного по-разному воплощать эти инструкции. Например, структура именных пространств и XML в сообщении SOAP, сгенерированном Apache Axis может сильно отличаться от структуры, сгенерированной Microsoft .NET. Однако правильно написанный клиент или сервер может обработать любое правильно написанное в соответствии с инструкциями SOAP сообщение.

На случай проблем в теле SOAP содержится информация об ошибке в форме SOAP Fault. Fault - это структура XML, содержащая описание ошибки, например:

```
<SOAP-ENV:Fault>
    <faultcode>SOAP-ENV:Server</faultcode>
    <faultstring>Server Error</faultstring>
    <detail>Database not available</detail>
</SOAP-ENV:Fault>
```

Хотя сообщение SOAP и хранится в текстовом виде в формате XML, из-за специфичности определений пространств имен и свойств элементов его порой бывает непросто понять или интерпретировать вручную. К счастью, программ и библиотек, способных создать или интерпретировать вам сообщение SOAP, много, и они берут на себя большую часть сложностей.

8.3.3 WSDL: Определение Web-сервисов

Для того чтобы человек или компания смогла вызвать Web-сервис, необходимо знать основную информацию, такую как: где сервис расположен, как сервис может быть связан, и подобная информация. Подобная информация предоставлена языком описания Web-сервисов WSDL.

WSDL – язык, основанный на XML, используемый для создания документов, которые содержат важную информацию о том, как Web-сервисы могут быть расположены и работать. WSDL (англ. Web Services Description Language) часто произносится как "Whiz-dull".

Когда говорят о **публикации** Web-сервиса, имеется в виду программа, публикующая файл WSDL и позволяющая иным программам пользоваться соответствующим сервисом. Программы, публикующие Web-сервисы, называются провайдерами. Говоря об **использовании** Web-сервиса, имеется в виду программа, отправляющая вызов к Web-сервису на другой машине. Пользователи Web-сервисов называются клиентами.

Для выполнения Web-сервиса, клиент должен определить местонахождение, а после его нахождения загрузить документ WSDL, в котором описаны подробности использования сервиса.

Клиент может получить файл WSDL многими различными способами, например:

- документ может быть расположен в доступном для поиска, общественном каталоге UDDI. В этом случае, любой клиент может найти Web-сервис и выполнить его;
- документы WSDL могут быть найдены другими способами, например, через запросы HTTP, FTP, и даже через электронную почту. Они также могут располагаться в частных каталогах UDDI.

8.3.3.1 Структура документа WSDL

В документе WSDL провайдером определены методы сервиса, которыми могут пользоваться другие программы. Также в документе есть информация о таких правилах форматирования сообщений SOAP, как используемые

пространства имен, порядок и структура параметров, и даже о дополнительной информации, которую необходимо включить в заголовок SOAP или HTTP.

В документе WSDL определяется web-сервис с помощью следующих основных элементов (см. таблица):

Элемент	Определяет
<portType>	Методы, предоставляемые web-сервисом
<message>	Сообщения, используемые web-сервисом
<types>	Типы данных, используемые web-сервисом
<binding>	Протоколы связи, используемые web-сервисом

Общая структура документа WSDL выглядит следующим образом:

```
<definitions>
<types>
  определение типов.....
</types>

<message>
  определение сообщения....
</message>

<portType>
  определение порта.....
</portType>

<binding>
  определение связей.....
</binding>

</definitions>
```

Документ WSDL может также содержать другие элементы, например элементы расширения и элемент service, который позволяет объединить вместе в одном отдельном документе WSDL определения нескольких web-сервисов.

Порты WSDL

Элемент *<portType>* является наиболее важным элементом в WSDL. Он определяет сам web-сервис, предоставляемые им операции и используемые сообщения.

Элемент *<portType>* можно сравнить с библиотекой функций (модулем, классом) в традиционном языке программирования.

Сообщения WSDL

Элемент *<message>* определяет элементы данных операции. Каждое сообщение может содержать одну или несколько частей. Эти части можно сравнить с параметрами вызываемых функций в традиционном языке программирования.

Типы WSDL

Элемент *<types>* определяет тип данных, используемых web-сервисом. Для максимальной платформо-независимости WSDL использует синтаксис XML Schema для определения типов данных.

Связи WSDL

Элемент *<binding>* определяет формат сообщения и детали протокола для каждого порта.

8.3.3.2 Пример WSDL

Это простейшая фрагмент WSDL-документа:

```
<message name="getTermRequest">
  <part name="term"  type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value"  type="xs:string"/>
</message>

<portType name="glossaryTerms">
 <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
 </operation>
</portType>
```

В этом примере элемент *<portType>* определяет *"glossaryTerms"* как имя порта, и *"getTerm"* как имя операции.

Операция *"getTerm"* имеет входящее сообщение, называемое *"getTermRequest"*, и исходящее сообщение *"getTermResponse"*.

Элементы "*message"* определяют части каждого сообщения и ассоциированные типы данных.

Если сравнивать с традиционным программированием, то *"glossaryTerms"* это библиотека функций, а *"getTerm"* --- функция с параметром *"getTermRequest"*, которая возвращает *"getTermResponse"* после выполнения.

8.3.3.3 Типы Операций

Запрос-ответ (request-response) это самый распространенный тип операций. Всего же в WSDL определено четыре типа (см. таблица):

Тип операции	Описание
Однонаправленный (One-way)	Операция может принимать сообщение, но не будет возвращать ответ
Запрос-ответ (Request-response)	Операция может принимать запрос и возвратит ответ
Вопрос-ответ (Solicit-response)	Операция может послать запрос и будет ждать ответ
Извещение (Notification)	Операция может послать сообщение, но не будет ожидать ответ

Однонаправленная (One-Way) Операция

Пример однонаправленной операции:

```
<message name="newTermValues">
  <part name="term" type="xs:string"/>
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="setTerm">
    <input name="newTerm" message="newTermValues"/>
  </operation>
```

```
</portType >
```

В этом примере порт *"glossaryTerms"* определяет однонаправленную операцию под названием *"setTerm"*. Операция *"SetTerm"* позволяет ввод новых словарных термов с помощью сообщения *"newTermValues"* со входными параметрами *"term"* и "value". Однако, данная операция не предусматривает какого-либо выходного сообщения.

Операция типа "Запрос-Ответ"

Пример операции типа "запрос-ответ":

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
 <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
 </operation>
</portType>
```

В этом примере порт *"glossaryTerms"* определяет операцию типа "запрос-ответ" с именем *"getTerm"*. Операция *"getTerm"* принимает на вход сообщение с именем *"getTermRequest"* и параметром *"term"* и возвращает сообщение с именем *"getTermResponse"* и параметром *"value"*.

Связи WSDL определяют формат сообщения и подробности протокола для каждого порта.

Связь с SOAP

Пример операции запроса-ответа:

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>
```

```
<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

<binding type="glossaryTerms" name="b1">
<soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
 <operation>
  <soap:operation
   soapAction="http://example.com/getTerm"/>
  <input>
   <soap:body use="literal"/>
  </input>
  <output>
   <soap:body use="literal"/>
  </output>
 </operation>
</binding>
```

Элемент *“binding”* имеет два атрибута --- атрибут *“name”* и атрибут *“type”*. Атрибут *“name”* определяет название (можно использовать любое) связи, а атрибут *“type”* --- точку монтирования для связи, в данном случае это порт *"golossaryTerms"*.

Элемент *“soap:binding”* имеет два атрибута --- атрибут *“style”* и атрибут *“transport”*. Элемент *“style”* может быть либо *"rpc"*, либо *"document"*. В данном случае используется значение *“document”*.

Атрибут *“transport”* определяет используемый SOAP протокол. В данном случае это HTTP.

Элемент *“operation”* описывает все операции, с которыми порт может работать.

Для каждой операции должно быть определено в соответствие действие SOAP. Также необходимо указать кодировку ввода и вывода. В этом случае используется *"literal"*.

8.3.3.4 Синтаксис WSDL

Ниже приведен листинг полного синтаксиса WSDL, который предлагается такими компаниями, как Ariba, IBM и Microsoft, для описания сервисов в рамках деятельности W3C XML Activity по Протоколам XML.

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

   <import namespace="uri" location="uri"/>*

   <wsdl:documentation .... /> ?

   <wsdl:types> ?
      <wsdl:documentation .... />?
      <xsd:schema .... />*
      <-- extensibility element --> *
   </wsdl:types>

   <wsdl:message name="nmtoken"> *
      <wsdl:documentation .... />?
      <part name="nmtoken" element="qname"? type="qname"?/> *
   </wsdl:message>

   <wsdl:portType name="nmtoken">*
      <wsdl:documentation .... />?
      <wsdl:operation name="nmtoken">*
         <wsdl:documentation .... /> ?
         <wsdl:input name="nmtoken"? message="qname">?
            <wsdl:documentation .... /> ?
         </wsdl:input>
         <wsdl:output name="nmtoken"? message="qname">?
            <wsdl:documentation .... /> ?
         </wsdl:output>
         <wsdl:fault name="nmtoken" message="qname"> *
            <wsdl:documentation .... /> ?
         </wsdl:fault>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="nmtoken" type="qname">*
      <wsdl:documentation .... />?
      <-- extensibility element --> *
      <wsdl:operation name="nmtoken">*
```

```
        <wsdl:documentation .... /> ?
        <-- extensibility element --> *
        <wsdl:input> ?
           <wsdl:documentation .... /> ?
           <-- extensibility element -->
        </wsdl:input>
        <wsdl:output> ?
           <wsdl:documentation .... /> ?
           <-- extensibility element --> *
        </wsdl:output>
        <wsdl:fault name="nmtoken"> *
           <wsdl:documentation .... /> ?
           <-- extensibility element --> *
        </wsdl:fault>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="nmtoken"> *
      <wsdl:documentation .... />?
      <wsdl:port name="nmtoken" binding="qname"> *
        <wsdl:documentation .... /> ?
        <-- extensibility element -->
      </wsdl:port>
      <-- extensibility element -->
   </wsdl:service>
   <-- extensibility element --> *
</wsdl:definitions>
```

8.3.3.5 Примеры WSDL

Пример определения математических операций посредством WSDL

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/">

   <types>
```

```
  <xs:schema targetNamespace="http://example.org/math/types/"
    xmlns="http://example.org/math/types/"
    elementFormDefault="unqualified" attributeFormDefault="unqualified">

  <xs:complexType name="MathInput">
        <xs:sequence>
          <xs:element name="x" type="xs:double"/>
          <xs:element name="y" type="xs:double"/>
        </xs:sequence>
      </xs:complexType>

      <xs:complexType name="MathOutput">
        <xs:sequence>
          <xs:element name="result" type="xs:double"/>
        </xs:sequence>
      </xs:complexType>

      <xs:element name="Add" type="MathInput"/>
      <xs:element name="AddResponse" type="MathOutput"/>
      <xs:element name="Subtract" type="MathInput"/>
      <xs:element name="SubtractResponse" type="MathOutput"/>
      <xs:element name="Multiply" type="MathInput"/>
      <xs:element name="MultiplyResponse" type="MathOutput"/>
      <xs:element name="Divide" type="MathInput"/>
      <xs:element name="DivideResponse" type="MathOutput"/>
    </xs:schema>

</types>


<message name="AddMessage">
    <part name="parameters" element="ns:Add"/>
</message>
<message name="AddResponseMessage">
    <part name="parameters" element="ns:AddResponse"/>
</message>
<message name="SubtractMessage">
    <part name="parameters" element="ns:Subtract"/>
</message>
<message name="SubtractResponseMessage">
    <part name="parameters" element="ns:SubtractResponse"/>
</message>
<message name="MultiplyMessage">
    <part name="parameters" element="ns:Multiply"/>
```

```
</message>
<message name="MultiplyResponseMessage">
  <part name="parameters" element="ns:MultiplyResponse"/>
</message>
<message name="DivideMessage">
  <part name="parameters" element="ns:Divide"/>
</message>
<message name="DivideResponseMessage">
  <part name="parameters" element="ns:DivideResponse"/>
</message>

<portType name="MathInterface">
  <operation name="Add">
    <input message="y:AddMessage"/>
    <output message="y:AddResponseMessage"/>
  </operation>

  <operation name="Subtract">
    <input message="y:SubtractMessage"/>
    <output message="y:SubtractResponseMessage"/>
  </operation>

  <operation name="Multiply">
    <input message="y:MultiplyMessage"/>
    <output message="y:MultiplyResponseMessage"/>
  </operation>

  <operation name="Divide">
    <input message="y:DivideMessage"/>
    <output message="y:DivideResponseMessage"/>
  </operation>

</portType>

<binding name="MathSoapHttpBinding" type="y:MathInterface">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="Add">
    <soap:operation soapAction="http://example.org/math/#Add"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
```

```
      </output>
    </operation>
    <operation name="Subtract">
      <soap:operation soapAction="http://example.org/math/#Subtract"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
    <operation name="Multiply">
      <soap:operation soapAction="http://example.org/math/#Multiply"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
    <operation name="Divide">
      <soap:operation soapAction="http://example.org/math/#Divide"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="MathService">
    <port name="MathEndpoint" binding="y:MathSoapHttpBinding">
      <soap:address location="http://localhost/math/math.asmx"/>
    </port>
  </service>
</definitions>
```

Пример описание сервисов доступа к БД посредством WSDL

Пример Web-сервиса для доступа к данным в БД по протоколу SOAP, включающего в себя набор методов:

1)

- getAllTables:

String[] getAllTables(String user)

этот метод позволяет получить одномерный список всех таблиц БД. Входным параметром данного метода является имя пользователя (String user) (пользовательской системы). Выходным параметром – одномерный список таблиц БД (String[]).

2)

- getTableInfo:

TableRow[] getTableInfo(String user, String table_name)

этот метод позволяет получить метаинформацию об интересующей пользователя таблице БД. Входным параметром данного метода является имя пользователя (пользовательской системы) (String user) и имя интересующей пользователя таблицы БД (String table_name). Выходным параметром – двумерный список, описывающий структуру таблицы БД (TableRow[] –содержит внутри String[]).

3)

- getTableData:

TableRow[] getTableData(String user, String table_name, String[] columns, String[] sql_filtre)

этот метод является основным методом сервиса и позволяет выгружать данные из интересующей пользователя таблицы БД. Входным параметром данного метода является имя пользователя (пользовательской системы) (String user), имя интересующей пользователя таблицы БД (String table_name), перечень интересующих столбцов (не обязательный параметр и может быть равен NULL – в этом случае будут выбраны все столбцы таблицы БД) (String[] columns) и одномерный набор SQL-фильтров для уточнения выборки (String[] sql_filtre). SQL-фильтры это синтаксически верные части SQL кода, которые будут подставлены под условие WHERE в запросе для выгрузки данных. Части фильтров соединяются между собой условием AND. Набор фильтров так же не являются обязательным параметром, и может принимать значение NULL. Выходным параметром – двумерный список данных таблицы БД.

Методы сервиса могут быть расширены или изменены. При изменении методов все зарегистрированные пользователи БД будут заранее уведомлены об изменениях и их структуре.

Работа с Web-сервисами осуществляется посредством стандартного описания WSDL. Пример файла WSDL для описанного выше сервиса представлен ниже.

Листинг WSDL:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://services.nsi"
xmlns:impl="http://services.nsi" xmlns:intf="http://services.nsi"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <wsdl:types>
 <schema elementFormDefault="qualified" targetNamespace="http://services.nsi"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:impl="http://services.nsi"
xmlns:intf="http://services.nsi" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <element name="getTableInfo">
 <complexType>
 <sequence>
 <element name="user" nillable="true" type="xsd:string"/>
 <element name="table_name" nillable="true" type="xsd:string"/>
 </sequence>
 </complexType>
 </element>
 <complexType name="TableRow">
 <sequence>
 <element maxOccurs="unbounded" name="row" nillable="true" type="xsd:string"/>
 </sequence>
 </complexType>
 <element name="getTableInfoResponse">
 <complexType>
 <sequence>
 <element maxOccurs="unbounded" minOccurs="0" name="getTableInfoReturn"
type="impl:TableRow"/>
 </sequence>
 </complexType>
 </element>
 <element name="getTableData">
 <complexType>
 <sequence>
 <element name="user" nillable="true" type="xsd:string"/>
```

```
 <element name="table_name" nillable="true" type="xsd:string"/>
 <element maxOccurs="unbounded" minOccurs="0" name="columns"
type="xsd:string"/>
 <element maxOccurs="unbounded" minOccurs="0" name="sql_filtre"
type="xsd:string"/>
 </sequence>
 </complexType>
 </element>
 <element name="getTableDataResponse">
 <complexType>
 <sequence>
 <element maxOccurs="unbounded" minOccurs="0" name="getTableDataReturn"
type="impl:TableRow"/>
 </sequence>
 </complexType>
 </element>
 <element name="getAllTables">
 <complexType>
 <sequence>
 <element name="user" nillable="true" type="xsd:string"/>
 </sequence>
 </complexType>
 </element>
 <element name="getAllTablesResponse">
 <complexType>
 <sequence>
 <element maxOccurs="unbounded" minOccurs="0" name="getAllTablesReturn"
type="xsd:string"/>
 </sequence>
 </complexType>
 </element>
 </schema>
 </wsdl:types>

 <wsdl:message name="getTableDataRequest">
 <wsdl:part element="impl:getTableData" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="getTableDataResponse">
 <wsdl:part element="impl:getTableDataResponse" name="parameters"/>
 </wsdl:message>

 <wsdl:message name="getTableInfoResponse">
 <wsdl:part element="impl:getTableInfoResponse" name="parameters"/>
 </wsdl:message>
```

```
<wsdl:message name="getAllTablesResponse">
<wsdl:part element="impl:getAllTablesResponse" name="parameters"/>
</wsdl:message>

<wsdl:message name="getAllTablesRequest">
<wsdl:part element="impl:getAllTables" name="parameters"/>
</wsdl:message>
<wsdl:message name="getTableInfoRequest">
<wsdl:part element="impl:getTableInfo" name="parameters"/>
      </wsdl:message>

<wsdl:portType name="LoadData">
<wsdl:operation name="getTableInfo">
<wsdl:input message="impl:getTableInfoRequest" name="getTableInfoRequest"/>
<wsdl:output message="impl:getTableInfoResponse" name="getTableInfoResponse"/>
</wsdl:operation>
<wsdl:operation name="getTableData">
<wsdl:input message="impl:getTableDataRequest" name="getTableDataRequest"/>
<wsdl:output message="impl:getTableDataResponse"
name="getTableDataResponse"/>
</wsdl:operation>
<wsdl:operation name="getAllTables">
<wsdl:input message="impl:getAllTablesRequest" name="getAllTablesRequest"/>
<wsdl:output message="impl:getAllTablesResponse" name="getAllTablesResponse"/>
</wsdl:operation>
</wsdl:portType>

<wsdl:binding name="LoadDataSoapBinding" type="impl:LoadData">
<wsdlsoap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="getTableInfo">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="getTableInfoRequest">
<wsdlsoap:body use="literal"/>
</wsdl:input>
<wsdl:output name="getTableInfoResponse">
<wsdlsoap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="getTableData">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="getTableDataRequest">
<wsdlsoap:body use="literal"/>
</wsdl:input>
```

```
<wsdl:output name="getTableDataResponse">
<wsdlsoap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="getAllTables">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="getAllTablesRequest">
<wsdlsoap:body use="literal"/>
</wsdl:input>
<wsdl:output name="getAllTablesResponse">
<wsdlsoap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="LoadDataService">
<wsdl:port binding="impl:LoadDataSoapBinding" name="LoadData">
<wsdlsoap:address location="http://ip адрес сервера:порт
                                подключения/WEBNSI/services/LoadData"/>
</wsdl:port>
</wsdl:service>

</wsdl:definitions>
```

Для работы с Web-сервисом на стороне пользователя должен существовать клиент, работающий по протоколу SOAP, который получает доступ к описанному выше WSDL файлу и на основе него строит соответствующие клиентские методы вызова и выгружает данные из БД.

8.3.4 UDDI

UDDI - это стандарт для создания каталога Web-сервисов, поставляемых любым количеством программ (Universal Description, Discovery and Integration (UDDI)). Это нечто вроде телефонной книги поставщиков Web-сервисов. Клиенты могут искать необходимую им информацию в реестре UDDI, а реестр возвращает им необходимые данные для подключения к сервису.

Хотя UDDI и довольно важный стандарт для определения Web-сервисов, его значимость сильно уменьшена за счёт того, что он является необязательным элементом Web-сервисов, а когда есть выбор, использовать или нет, многие решают не использовать.

Большинство организованных корпоративных сред с большим количеством внутренних Web-сервисов имеют реестры UDDI. Замечательно, когда есть корпоративный сайт UDDI, содержащий информацию о доступных в вашей

компании Web-сервисах. Собирая вместе все сервисы, UDDI позволяет без проблем и незаметно менять их поставщиков. Если клиенты ищут сервисы через UDDI, то вызовы SOAP автоматически отправляются к новому поставщику. Однако этот компонент не обязателен в архитектуре Web-сервисов.

What is UDDI

UDDI is a platform-independent framework for describing services, discovering businesses, and integrating business services by using the Internet.

UDDI stands for Universal Description, Discovery and Integration

UDDI is a directory for storing information about web services

UDDI is a directory of web service interfaces described by WSDL

UDDI communicates via SOAP

UDDI is built into the Microsoft .NET platform

What is UDDI Based On?

UDDI uses World Wide Web Consortium (W3C) and Internet Engineering Task Force (IETF) Internet standards such as XML, HTTP, and DNS protocols.

UDDI uses WSDL to describe interfaces to web services

Additionally, cross platform programming features are addressed by adopting SOAP, known as XML Protocol messaging specifications found at the W3C Web site.

UDDI Benefits

Any industry or businesses of all sizes can benefit from UDDI.

Before UDDI, there was no Internet standard for businesses to reach their customers and partners with information about their products and services. Nor was there a method of how to integrate into each other's systems and processes.

Problems the UDDI specification can help to solve:

Making it possible to discover the right business from the millions currently online

Defining how to enable commerce once the preferred business is discovered

Reaching new customers and increasing access to current customers

Expanding offerings and extending market reach

Solving customer-driven need to remove barriers to allow for rapid participation in the global Internet economy

Describing services and business processes programmatically in a single, open, and secure environment

---

How can UDDI be Used

If the industry published an UDDI standard for flight rate checking and reservation, airlines could register their services into an UDDI directory. Travel agencies could then search the UDDI directory to find the airline's reservation interface. When the interface is found, the travel agency can communicate with the service immediately because it uses a well-defined reservation interface.

Who is Supporting UDDI?

UDDI is a cross-industry effort driven by all major platform and software providers like Dell, Fujitsu, HP, Hitachi, IBM, Intel, Microsoft, Oracle, SAP, and Sun, as well as a large community of marketplace operators, and e-business leaders.

Over 220 companies are members of the UDDI community.

WSDL Summary

This tutorial has taught you how to create WSDL documents that describes a web service. It also specifies the location of the service and the operations (or methods) the service exposes.

You have learned how to define the message format and protocol details for a web service.

You have also learned that you can register and search for web services with UDDI.

8.4 Безопастность WEB-сервисов

Как проводится аутентификация при вызовах сервисов, если поставщик работает с закрытой информацией? Ведь понятно, что не все Web-сервисы доступны широкой публике, не так ли?

Это важный вопрос, однозначно ответить на который непросто. Есть различные схемы, которые вы можете использовать, в зависимости от ситуации, например:

- Могут ли сообщения SOAP приходить в виде текста или они должны быть зашифрованы?
- Достаточно ли вам простой аутентификации по логину и паролю или же она должна быть устойчивой и маркерной?
- Если используются маркеры, необходимы ли на них подписи, и как правильно их включить в сообщение SOAP?
- А что если клиент отправляет сообщения SOAP не напрямую, а через какую-то промежуточную структуру, такую как очередь сообщений или через ещё какой-нибудь Web-сервис?

Кроме того, при обмене сообщениями не всегда может использоваться HTTP, поэтому у вас не выйдет попросту использовать системы безопасности Web-сервисов в дополнение к существующим системам безопасности HTTP.

Протоколы безопасности для Web-сервисов начинаются со спецификации WS-Security, которая опреляет маркерную (token) архитектуру для безопасного взаимодействия. На этой базе построено шесть основных компонентных спецификаций:

- WS-Policy и связанные с ней спецификации, которые определяют политики правил, по которым взаимодействуют Web-сервисы;
- WS-Trust, которая определяет доверительную модель для безопасного обмена;
- WS-Privacy, которая определяет, как обеспечивается конфиденциальноть информации;

- WS-Secure Conversation, которая определяет как, используя правила, описанные в WS-Policy, WS-Trust, and WS-Privacy установить безопасную сессию между сервисами для обмена информации;
- WS-Federation, которая определяет правила распределенной идентификации и управление ею;
- WS-Authorization, которая занимается обработкой авторизации для доступа и обмена данными.

За моделью безопасности следуют прикладные спецификации, включая Business Process Execution Language for Web Services (BPEL4WS), которые определяют операции потоков работ, WS-Transaction и WS-Coordination, которые вместе отвечают за обработку распределенных транзакций.

В настоящее время в разработке находится спецификация Web Services Distributed Management для администрирования программного обеспечения всех сервисов и сервис-ориентированной архитектуры. В конце идут спецификации пользовательского интерфйса (WS-InteractiveApplications) и удаленного доступа к Web-сервисам (WS-RemotePortals).

8.5 WEB-Сервисы и SOA

SOA представляет собой общую архитектуру интеграции приложений. Web-сервисы представляют собой конкретный набор стандартов и спецификаций, являющихся одним из методов реализации SOA. Между Web- сервисами и SOA существует много логических связей, которые предполагают, что они могут дополнять друг друга:

- Web-сервисы представляют собой модель на основе открытых стандартов, которая может быть прочитана автоматически и которая позволяет создавать формальные описания интерфейсов для Web- сервисов.
- Web-сервисы предоставляют механизмы коммуникации, которые обеспечивают прозрачность местоположения и возможность взаимодействия.
- Web-сервисы развиваются, появляются такие технологии, как Business Process Execution Language for Web Services (WS-BPEL), SOAP в стиле документов, Web Services Definition Language (WSDL), а также WS-ResourceFramework, которые обеспечивают гибкую техническую реализацию хорошо спроектированных сервисов, включающих в себя и моделирующих многократно используемую функцию.

Работая вместе, Web-сервисы и SOA имеют потенциал для решения многих технических проблем, связанных с попыткой создать среду, работающую «по требованию».

8.6 Свойства WEB-Сервисов

Общие свойства Web-сервисов:

- **Web-сервисы являются самостоятельными.** С клиентской стороны не требуется никакого дополнительного программного обеспечения. Для начала работы достаточно иметь язык программирования с поддержкой XML и HTTP. На серверной стороне требуются только HTTP-сервер и SOAP-сервер. Можно использовать Web-сервисы в приложении, не написавши единой строчки кода.
- **Web-сервисы описывают сами себя.** Определение формата сообщения передается вместе с сообщением. Не требуется никаких внешних хранилищ метаданных или средств генерации кода.
- **Web-сервисы могут публиковаться, обнаруживаться и вызываться через Интернет.** Эта технология использует простые установившиеся стандарты, такие как HTTP. Используется существующая инфраструктура. Необходимы также некоторые дополнительные стандарты, такие как SOAP, WSDL и UDDI.
- **Web-сервисы являются модульными.** Простые Web-сервисы могут объединяться в более сложные либо при помощи технологий рабочих процессов (workflow), либо путем вызова низкоуровневых Web-сервисов из реализации высокоуровневых Web-сервисов. Web-сервисы могут образовывать цепочки, выполняющие более высокоуровневые бизнес-функции. Это уменьшает время разработки и позволяет создавать лучшие в своем роде реализации.
- **Web-сервисы не зависят от языка программирования и способны взаимодействовать.** Клиент и сервер могут быть реализованы в разных средах. Не нужно изменять существующий код, чтобы использовать Web-сервис. По существу, вы можете использовать для реализации клиентов и серверов Web-сервисов любой язык.
- **Web-сервисы по сути своей открыты и основаны на стандартах.** Главной технической основой Web-сервисов являются XML и HTTP. Существенная часть технологии Web-сервисов создана с использованием проектов с открытым исходным кодом. Следовательно, независимость от производителя и возможность взаимодействия – вполне реалистичные цели.
- **Web-сервисы имеют слабые связи.** Традиционно дизайн приложения зависит от жестких связей. Для Web-сервисов требуется более простой уровень координации, который позволяет осуществлять более гибкую реконфигурацию для обеспечения интеграции нужных сервисов.
- **Web-сервисы являются динамическими.** Динамичный электронный бизнес при использовании Web-сервисов становится реальностью, поскольку с UDDI и WSDL описание Web-сервисов и их обнаружение можно автоматизировать. Кроме того, Web-сервисы можно реализовывать и распространять, не влияя на работу клиентов, которые их используют.
- **Web-сервисы обеспечивают программный доступ.** Данный подход не предоставляет графического пользовательского интерфейса. Он работает на уровне

кода. Потребители сервисов должны знать интерфейс Web-сервисов, но не должны знать детали его реализации.

- **Web-сервисы могут заключать в себя существующие приложения.** Уже существующее самостоятельное приложение может быть интегрировано в сервис-ориентированную архитектуру путем реализации Web-сервисов в качестве интерфейса этого приложения.

8.7 Заключение

Работа с Web-сервисами заключается в том, что клиент читает файл WSDL поставщика, в соответствии с ним формирует и отправляет сообщение SOAP, позже получает другое сообщение SOAP в ответ.

Так почему же сервисы так важны? Частично важность сервисов заключена в том, что они предоставляют стандартный путь для общения между программами, вне зависимости от языков, на которых они написаны и платформ, на которых они работают. Ранее нам приходилось работать с форматами данных, уникальными для разных программ, или же с функциями API-уровня, с которыми не могли работать программы на других языках. Из использования XML во всех стандартах Web-сервисов означает, что все сервисы доступны и понятно определены.

Фактически, такой принцип позволяет совсем разным программам легко общаться друг с другом на понятном им всем языке. Одной из главных сложностей при работе с разными технологиями от разных производителей всегда была необходимость заставить все эти разные программы общаться между собой и обмениваться данными. Теперь, когда все приложения могут поставлять и/или использовать Web-сервисы, наладка взаимодействие между ними невероятно упростилось.

Ещё одним преимуществом Web-сервисов является то, что клиенты и поставщики могут находиться на разных машинах, пользоваться разными аппаратными и программными средствами, и общению это не мешает. Программы могут использоваться другими программами в рамках одной машины, или с других машин, но с использованием определённого формата передачи данных. Web-сервисам нужны только подключение к сети и обработчик XML.

Если все эти факторы учитывать вместе, результат получается значительным. При наличии стандартного средства для общения между приложениями по сети, можно по-другому строить программы. Вместо написания монолитных программ, в которых каждый раз заново изобретается колесо, можно писать программы, состоящие из модулей.

Будучи профессионалом в сфере IT можно заниматься как разработкой интерфейса, так и сервисов, или и тем и другим. Понимание того, как всё это работает вместе (или хотя бы знание, что это такое) важно для работы в подобных проектах.

9. Язык моделирования бизнес процессов BPMN

BPMN (Business Process Modeling Notation) – нотация, графический язык для наглядного изображения бизнес-процессов.

Основные назначения стандарта - создание нотации, понятной всем участникам бизнес-сферы, от бизнес-аналитиков, создающих первоначальные эскизы процессов, технических разработчиков, ответственных за внедрение технологии, в которой будут представлены данные процессы, и, наконец, до бизнесменов, которые будут управлять этими процессами, а также осуществлять их мониторинг. Таким образом, **BMPN является стандартизованным связующим звеном между разработкой бизнес процессов и их реализацией.**

Другой не менее важной целью является визуализация посредством бизнес ориентированной нотации языков XML, таких как BPEL4WS (Business Process Execution Language for Web Services – язык реализации бизнес процессов для Web - служб), разработанных для выполнения бизнес процессов.

Цель BPMN – стандартизировать нотацию моделирования бизнес процессов при наличии множества различных нотаций и точек зрения на моделирование. Использование BPMN обеспечит легкую передачу информации по процессам другим участникам бизнес сферы, специалистам по внедрению процессов, клиентам и поставщикам.

Рабочая группа BPMI(Business Process Management Initiative) проанализировала опыт многих существующих нотаций и попробовала объединить лучшие концепции разных нотаций в одну стандартную нотацию. Были рассмотрены следующие нотации и методологии: UML Activity Diagram, UML EDOC Business Processes, IDEF, ebXML BPSS, Activity-Decision Flow (ADF) Diagram, RosettaNet, LOVeM, and Event-Process Chains (EPCs). Результатом кропотливой работы стала нотация BPMN.

9.1 Назначение BPMN

Моделирование бизнес процессов предназначено для предоставлениия разнообразной информации различным специалистам (технологам, архитекторам, ИТ проектировщикам и т.д.) и выработки общих решений. BPMN применимо для описания различных типов моделей, однако некоторые типы моделей, разрабатываемые в организациях в рамках деловой деятельности, не будут рассмотрены в рамках BPMN. В рамках BPMN не будут включаться следующие виды моделирования:

- Организационные структуры и ресурсы;
- Функциональные схемы;
- Модели данных и информационные модели;
- Стратегии;
- Бизнес правила.

BPMN допускает создание сквозных бизнес процессов. Структурные элементы BPMN позволяют легко проводить различия между участками схемы BPMN.

Существует **три основных типа подмоделей в рамках сквозной модели BPMN:**

- Private (частны или внутренние) бизнес процессы;
- Public Processes (публичный или открытый) бизнес процессы;
- Collaboration (совместные или глобальные) бизнес процессы.

9.1.1 Частные (Внутренние) бизнес процессы

Частные бизнес процессы являются внутренними для определенной организации, данный тип бизнес процессов обычно называют workflow или процессы BPM (управление деловыми процессами, приведен пример на рис. 9.1). Один частный бизнес процесс может быть отображен в одном или более BPMN документах.

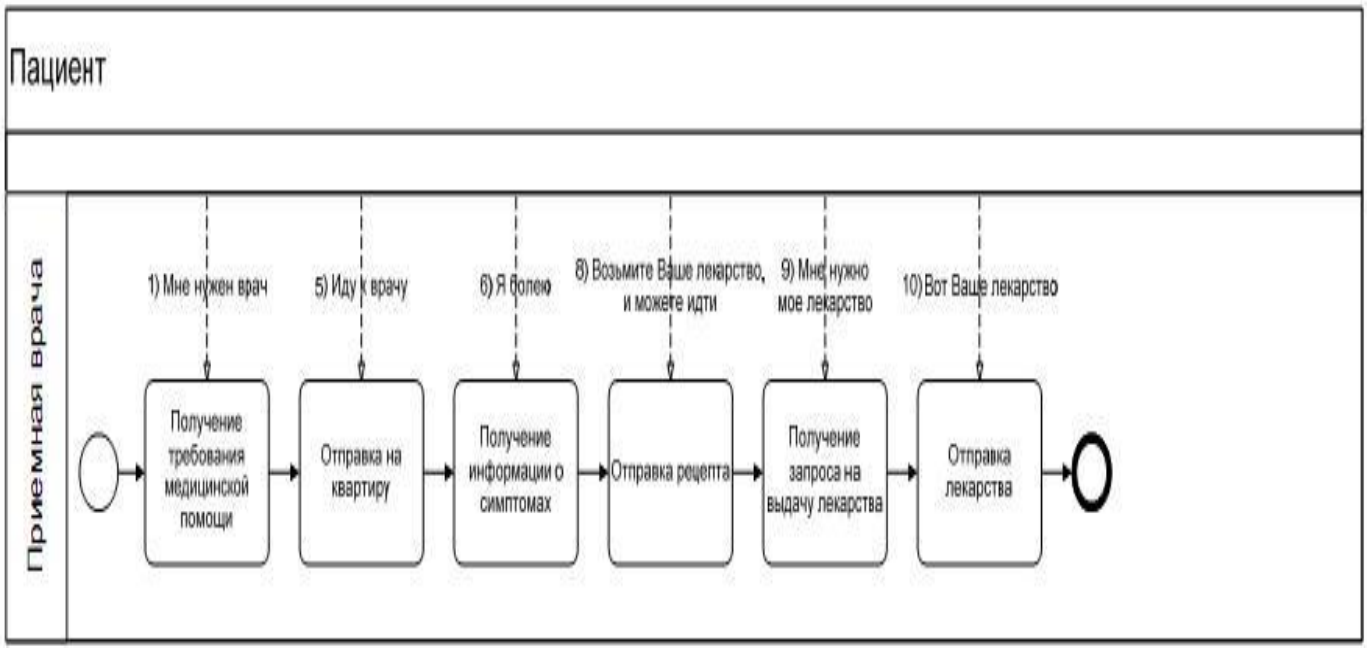
Если используются дорожки, то частный бизнес процесс будет помещен в одну область. Таким образом, последовательный поток процесса содержится внутри области и не может пересекать ее границы. Поток сообщений может пересекать границы области с целью указания на взаимодействия, существующие между отдельными частными бизнес процессами. Таким образом, одна схема бизнес процесса может содержать множество частных бизнес процессов, каждый из которых имеет свою схему на BPMN.

Рис. 9.1 Пример частного бизнес процесса

9.1.2 Публичные (Абстрактные) бизнес процессы

Публичные бизнес процессы представляют собой взаимодействие между частным бизнес процессом и другим процессом или участником (см. рис. 9.2.). Публичные считаются только те бизнес процессы, действия которых имеют связи за пределами частного бизнес процесса, также к ним относятся соответствующие механизмы контроля потока. Все остальные «внутренние» действия частного бизнес процесса не отображаются в публичном бизнес процессе. Таким образом, публичный бизнес процесс показывает последовательность сообщений, которые должны взаимодействовать с данным бизнес процессом

Публичные процессы содержатся внутри области и могут моделироваться отдельно или внутри большей схемы BPMN для демонстрации потока сообщений между блоками публичного процесса и другими объектами. Если публичный процесс размещается на той же схеме, что и соответствующий ему
частный процесс, то блоки, общие для обоих процессов, могут быть объединены. Публичный процесс может быть промоделирован, как отдельно, так и внутри обощенного процесса.



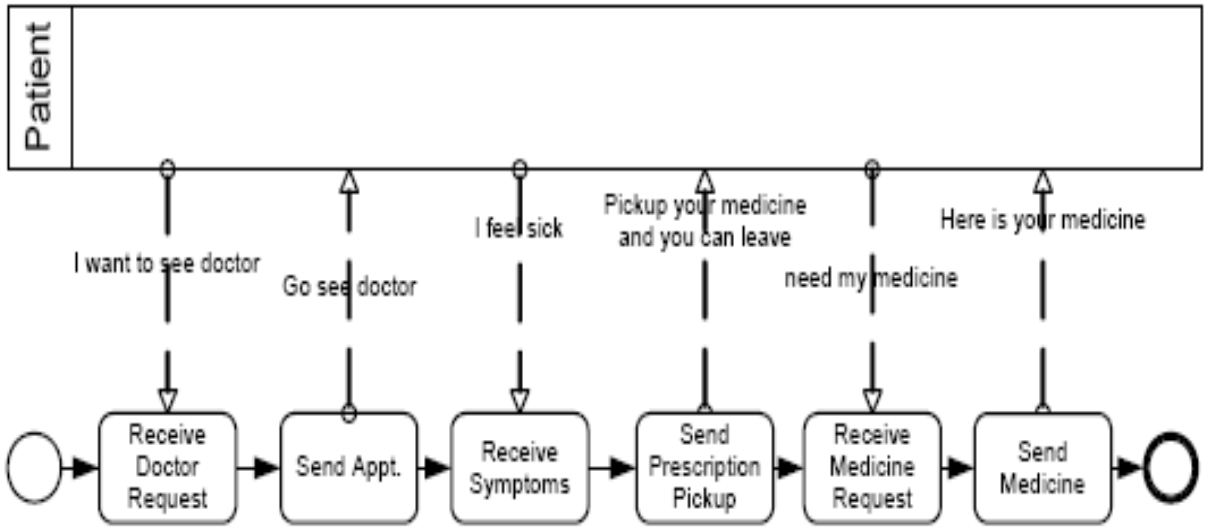


Рис. 9.2 Пример абстрактного процесса

9.1.3 Совместные (хореографии) процессы

Совместный процесс отображает *взаимодействие* между двумя и более бизнес объектами (или участнками). Эти *взаимодействия состоят в обмене сообщениями между данными объектами*. Один и тот же совместный процесс может быть выполнен на разных языках, таких как ebXML BPSS, RosettaNet или продукте рабочей группы W3C Choreography Working Group.

Совместный процесс можно изобразить в виде двух или более взаимодействующих абстрактных процессов (см. рис. 9.3). В приватном (абстрактном) процессе действия участников совместной работы могут рассматриваться как «точки касания» между участниками. Фактические (выполняемые) процессы, по сути, более подробны и обладают большим количеством действий по сравнению с абстрактными процессами.

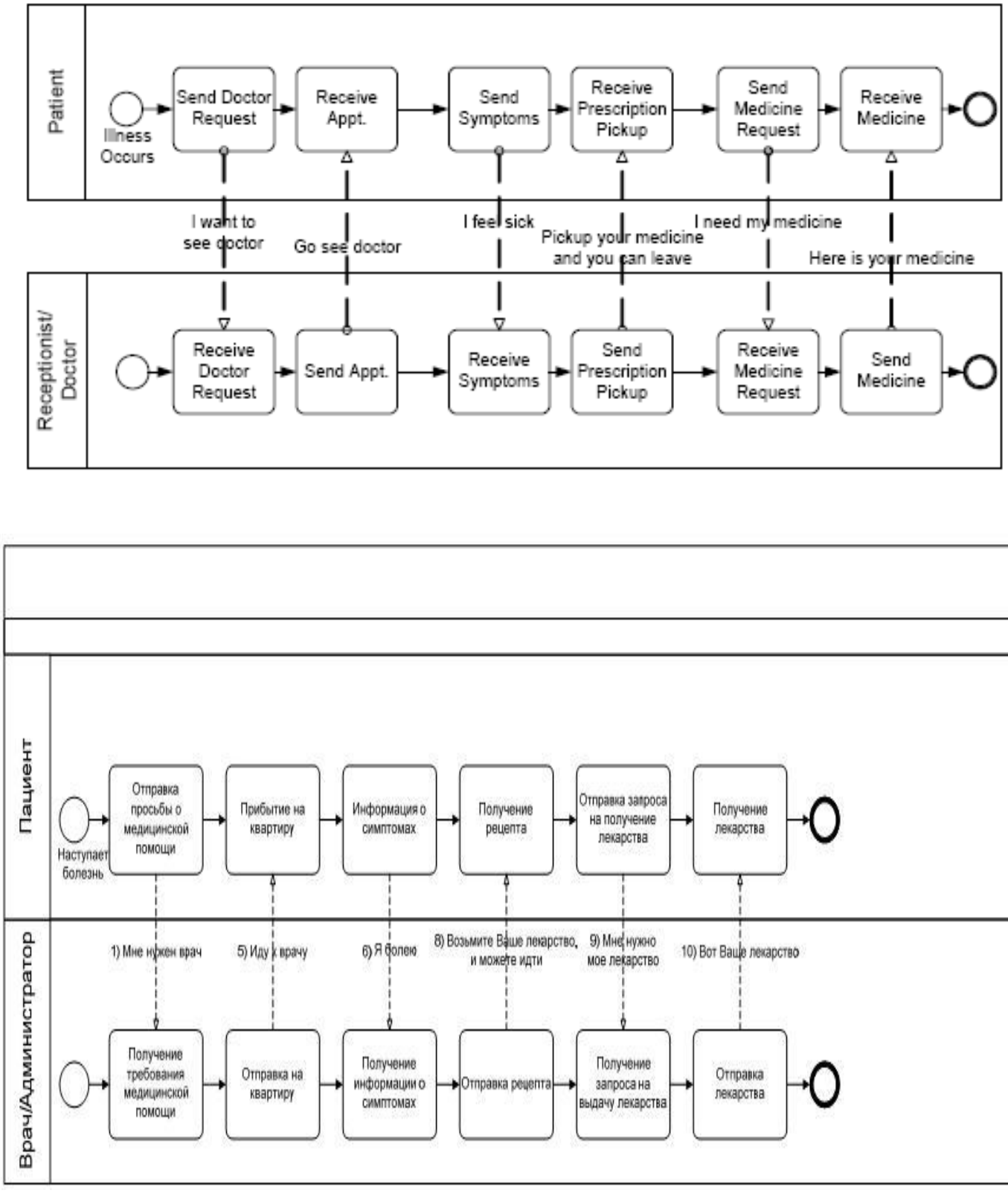


Рис. 9.3 Пример совместного процесса

9.2 Основные типы схем

В рамках и вне рамок этих трех подмоделей BPMN можно создать множество типов схем отбражений другие языки. Ниже перечислены **типы бизнес процессов**, которые можно смоделировать при помощи BPMN (процессы, отмеченные звездочкой, могут не отобразиться на язык реализации):

- Действия частного бизнес процесса высокого уровня (не функциональная схема)*;
- Детальный бизнес процесс;
- Исходный или старый бизнес процесс* ;
- Будущий или новый бизнес процесс;
- Детальный частный бизнес процесс, связанный с одним или более объектов (или бизнес процессами типа «Черного Ящика»);
- Взаимодействие двух или более детальных частных бизнес процессов;
- Взаимосвязь детального частного бизнес процесса с совместным процессом;
- Два или более абстрактных бизнес процесса* ;
- Взаимосвязь абстрактного бизнес процесса с совместным процессом;
- Совместный бизнес процесс (например, ebXML BPSS or RosettaNet)*;
- Взаимодействие двух или более детальных частных бизнес процессов через их абстрактные процессы;
- Взаимодействие двух или более детальных частных бизнес процессов через совместный процесс;
- Взаимодействие двух или более детальных частных бизнес процессов через их абстрактные процессы и совместный процесс.

BPMN допускает все вышеперечисленные типы бизнес процессов для схем отображений. Однако следует помнить, что в случае сочетания слишком большого количества подмоделей, например, три или более частных процесса с потоком сообщений между каждым из них, схема отбражения может стать трудной для понимания. Таким образом, мы рекомендуем разработчику сосредоточиться на выборе схемы отобюражения, например, частном процессе или совместном процессе.

9.2.1 Хореография бизнес процессов

Автономная хореография (не дорожки и оркестровка) является определение ожидаемого поведения, в основном процедурного характера (контракта), взаимодействия между участниками. В то время как нормальный процесс существует в определениях процесса в дорожках и их хореографиимежду участниками бизнес процесса. *Хореография похожа на частный бизнес-процесс, поскольку она состоит из сети деятельности, событий и шлюзов (рис. 9.4).* Тем не менее, хореография отличается тем, что показывает взаимодействие, которое представляют собой набор (одного или более) обмена сообщениями, которое включает в себя двух или более участников. Кроме того, в отличие от нормального

процесса, нет центрального контроллера, ответственного лица или наблюдателя процесса.

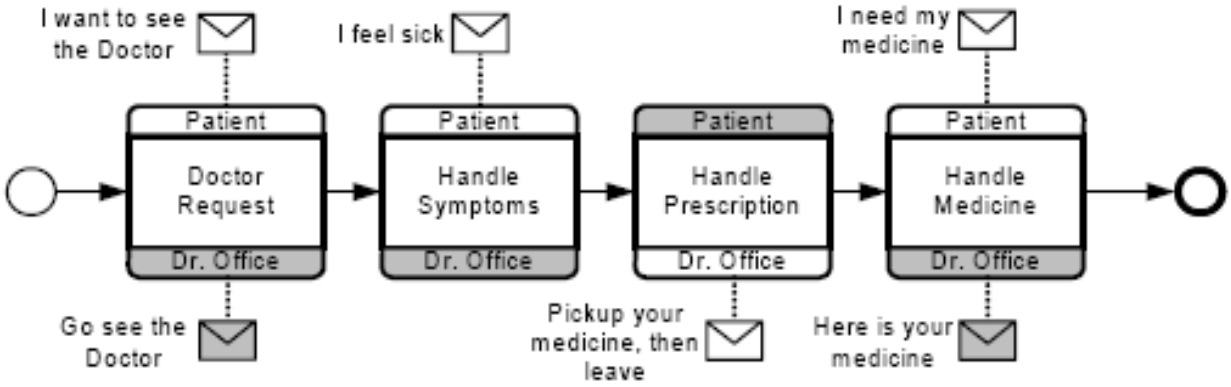


Рис. 9.4 Пример хореографии процессов

9.2.2 Согласование бизнес процесса

Схема согласования бизнес процесса является частным использованием и неформальным описанием совместной схемы (см. рис. 9.5). Тем не менее, дорожки и хореографии в схеме согласования обычно не содержатся. Процесс, как правило, не помещаются в дорожки. ***Схема согласования является логическим отношением обмена сообщениями, логической связи***. На практике, часто относится к интересу бизнес-объекта (там) например, "Заказ", "Отгрузка и доставка", или "Счет". Обмен сообщениями связывает друг с другом различные удаленные бизнес-сценарии. Например, ***в логистике,*** *акции пополнение счета* следует включать в себя следующие ***сценарии*** типа: ***создание заказов, назначение перевозчиков для доставки комбинированных грузов на продажу, взаимодействие с таможенной / карантинной службой; процессинг платежей и расследование проишествий.*** Таким образом, схема согласования, как показано на рисунке 9.5, показывает, согласование (как шестиугольники) между участниками. Это обеспечивает "птичий глаз" точки зрения различных согласование, которые относятся к данному домену.

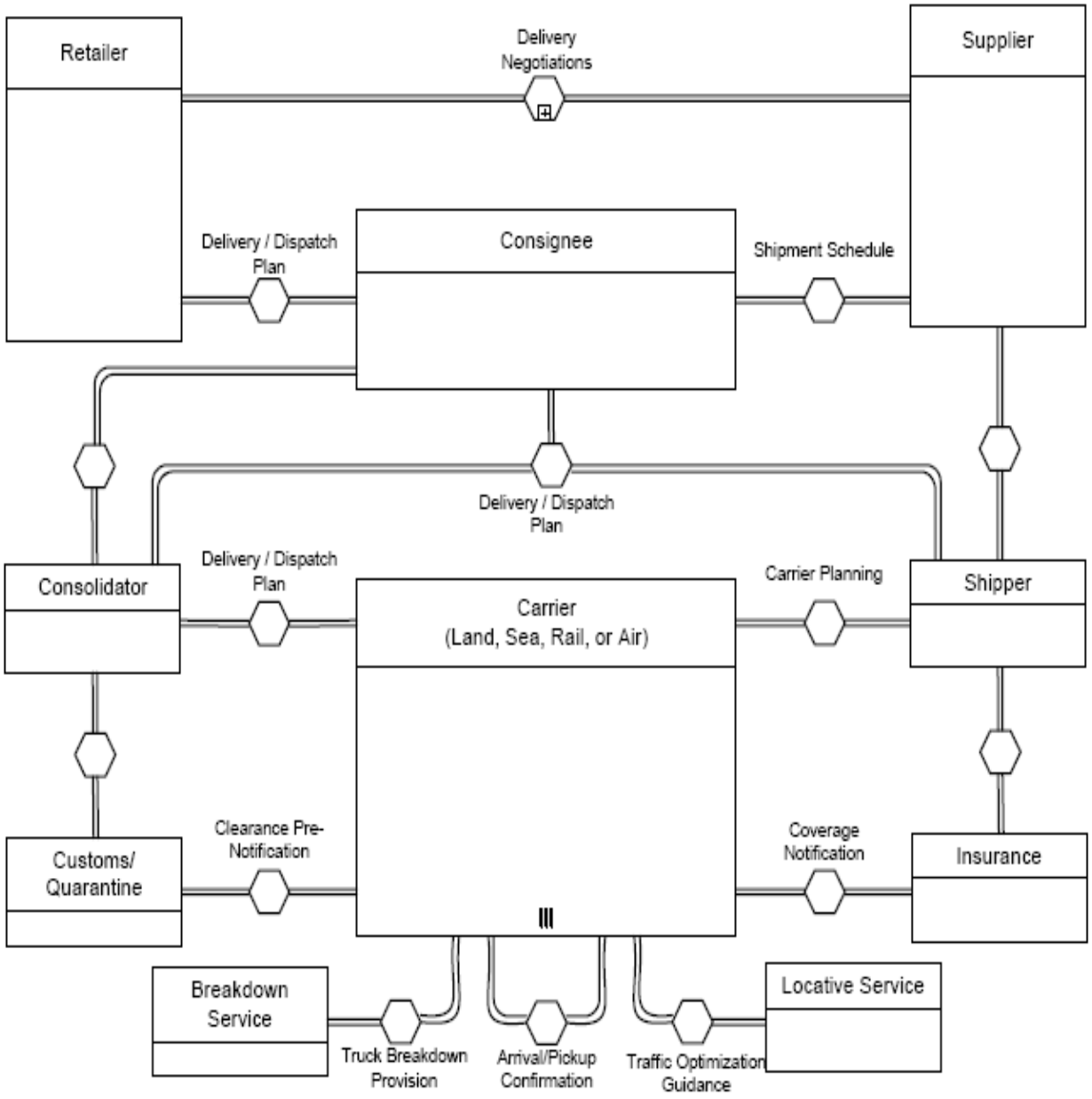


Рис. 9.5 Пример схема согласования бизнес процессов

9.2.3 Точка зрения, бизнес взгляд

Поскольку диаграмма BPMN могут изобразить ***процессы различных участников***, каждый участник может просматривать диаграмму по-разному. Это означает, что ***участники имеют различные точки зрения относительно того, как процессы будут применяться к ним***. Некоторые деятельности будут

внутренние относительно участника (то есть выполняются или под контролем участников) для других участников деятельность будет внешним по отношению к Участнику. Каждый участник будет иметь различные точки зрения о том, какие процессы внутренние и внешние. Во время выполнения, разница между внутренней и внешней деятельностью является важным моментом в том, как участник анализа диаграммы может быть в состоянии деятельности или проблемы. Тем не менее, **сама схема остается той** же. Рисунок 9.3 отображает бизнес-процессы, что имеет две точки зрения. Одна точка зрения пациента, а другой имеет доктор в офисе. Диаграмма показывает, деятельности обоих участников процесса, но когда процесс на самом деле выполняется, каждый участник будет иметь только контроль над своей собственной деятельности. Хотя схема точка зрения важна для просмотра диаграммы, чтобы понять, как поведение процесса будет относиться к этому участнику.

Принято выделять три вида точек зрения:

- **бизнес уровень** (business layer) – подразумевает точку зрения бизнес-аналитика и общее представление различных бизнес - шагов и поток их оркестровки;
- **функциональный уровень** (functional layer) – точка зрения решений и использование ИТ систем. Проектируется аналитиком, имеющим опыт работы с ИТ – системами. На этом уровне необходимо знать взаимодействие с БД, ERP – системами и др.
- **уровень реализации** (implementation layer) – договор (контракт) по реализации деталей процесса. Проектируется ИТ – инженерами, имеющие знания о приложениях взаимодействующих с бизнес – процессами. Здесь появляются исключения, компенсация, события, таймер. Реализуется как уровень сервиса и процесса реализации и WSDL для других процессов.

9.3 Отображение BPMN на другие языки моделирования

Так как BPMN охватывает широкий спектр применения, то существует возможность отображения **BPMN на нескольких языках спецификаций**:

- **BPEL4WS** – основной язык отображения BPMN, но он охватывает только один выполняемый частный бизнес процесс. Если на схеме отбражения BPMN изображено более одного внутреннего бизнес процесса, то для каждого внутреннего бизнес процесса будет отдельное отображение.
- Абстрактные участки схемы отбражения BPMN переводяться в описания интерфейса Web-сервиса.
- Участки модели глобальной работы BPMN могут быть отображены при помощи моделей совместной работы, например, ebXML BPSS, Rosetta-Net, и спецификации W3C Choreography Working Group Specification.

В данном разделе будет затрагиваться только отображение на язык BPEL4WS.

9.4 Основные элементы схемы бизнес процесса

Одним из факторов развития BPMN является создание простого механизма для разработки моделей бизнес процессов, который способен управлять сложными бизнес процессами. Способ решения проблемы этих двух противоречивых требований состоял в создании графических элементов нотации по конкретным категориям.

При этом совокупность категорий нотации получается небольшая, таким образом, читатель схемы отображения BPMN может легко узнать основные типы элементов и понять схему. В рамках основных категорий элементов могут быть добавлены дополнительные изменения и информация для обеспечения соответствия требованиям сложности без значительных изменений основных ощущений и впечатлений от схемы отображения. **Есть пять основных категорий элементами**, которых являются:

1. **Поток объектов** (Flow Objects)

Flow Objects основные графические элементы для определения бизнес – процесса:

1.1 Events (событие) 1.2, Activities (деятельность), 1.3. Gateways (узел, шлюз).

2. **Данные** (Data),

Data (данные) представлены четырьмя элементами:

2.1. Data Objects (объекты данных) 2.2. Data Inputs (входные данные)
2.3. Data Outputs (выходные данные) 2.4. Data Stores (хранилища)

3. **Соединители объектов** (Connecting Objects)

Четыре типа соединений объектов. Соединяют один к одному объекты или информацию:

3.1. Sequence Flows (последовательный поток), 3.2. Message Flows (поток сообщений), 3.3. Associations (ассоциация), 3.4. Data Associations (ассоциация данных).

4. **Области, дорожки** (Swimlanes),

Есть два способа группировки первичных данных моделирования:

4.1. Pools (пулы), 4.2. Lanes (дорожки).

5. **Артефакты** (Artifacts).

Артефакты используются для указания дополнительной информации о процессе. Артефакты включают:

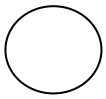
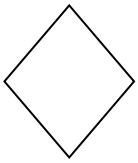
5.1 Group (группа),
5.2 Text Annotation (текст - аннотация).

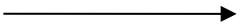
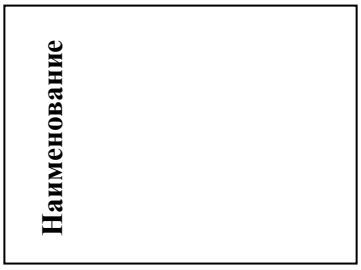
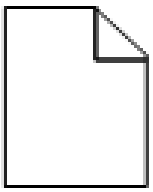
Артефакты используются для введения дополнительной информации по процессу. Разработчик или инструмент моделирования могут добавить столько

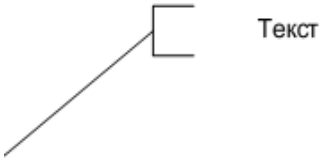
артефактов, сколько требуется. BPMN может предпринять дополнительную попытку привести большую совокупность артефактов к единому стандарту для общего использования, или для вертикальных рынков.

Ниже в таблице 9.1 приведены основные элементы моделирования BMPL.

Таблица. 9.1 Основные элементы моделирования

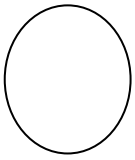
Элемент	Описание	Нотация
Event (событие)	Событие – это нечто, что «происходит» в ходе бизнес процесса. События влияют на ход бизнес процесса и обычно имеют причину (триггер) или воздействие (результат). События – это круг с открытым центром для обеспечения возможности внутренним маркерам различать разные триггеры или результаты. Существует три типа событий, классифицированных по времени воздействия на ход процесса: Начало, Промежуточные события и Конец.	
Activity (деятельность)	Деятельность – термин, характерный для обозначения работы, Деятельность может быть элементарной и неэлементарной (составной). Типы деятельности, являющиеся частью модели процесса: Процесс, Подпроцесс и Задача. Задачи и подпроцессы – закругленные прямоугольники. Процессы – либо безграничны, либо содержатся в пределах области.	
Gateway (узел, шлюз).	Шлюзы используется для контроля расхождения и схождения последовательного потока. Таким образом, оно будет обозначать ветвление, раздвоение, слияние и соединение маршрутов. Внутренние маркеры будут указывать на тип контроля развития процесса.	

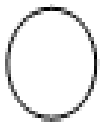
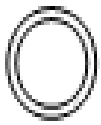
Sequence Flows (последовательный поток)	Последовательный поток показывает порядок, согласно которому будут выполняться действия процесса.	
Message Flows (поток сообщений)	Поток сообщений символизирует поток сообщений между двумя участниками, готовыми к их отправке и получению. В BPMN две отдельных области на диаграмме будут символизировать двух участников (например, бизнес объекты или бизнес роли).	
Association (ассоциация)	Ассоциация используется для связывания информации и артефактов с объектами. Текстовые и графические объекты, не относящиеся к схеме, могут быть связаны с объектами схемы.	
Pool (пул)	Область представляет собой участника в процессе. Она также играет роль «дорожки» и графического контейнера для разделения совокупности действий из других областей, обычно в контексте ситуаций «бизнес для бизнеса».	
Lanes(дорожка)	Дорожка – это подраздел, в пределах области которого протяженность равна длине области, как по вертикали, так и по горизонтали. Дорожки организовывают и классифицируют действия.	
Message (сообщение)	Сообщение используется для передачи контента изображения между двумя участниками бизнес – процесса.	
Data Objects (объекты данных)	Объекты данных рассматриваются как артефакты, так как они не влияют непосредственно на последовательный поток или поток сообщений процесса, но они	

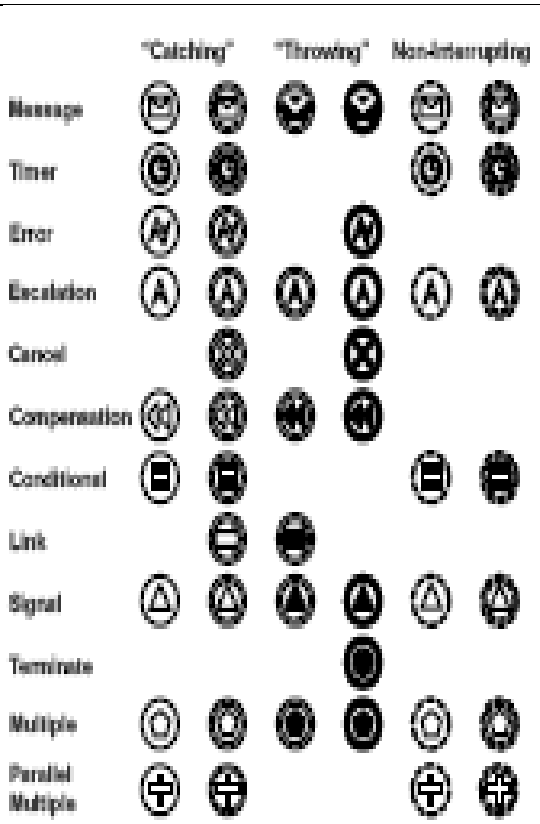
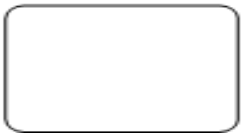
	обеспечивают ввод информации о том, какие действия требуют выполнения и/или что они производят.	
Group (группа) (прямоугольник вокруг группы объектов в целях документирования)	Группировка действий, не оказывающих влияния на последовательный поток. Группировка может использоваться в целях документирования или анализа. Группы могут также использоваться для распознавания действий операции, распределенной по ширине области.	
Text annotation (текстовая аннотация) (связана с ассоциацией)	Текстовая аннотация – способ предоставления дополнительной информации для изучающего схему BPMN.	

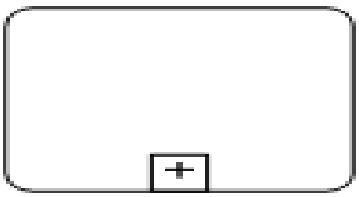
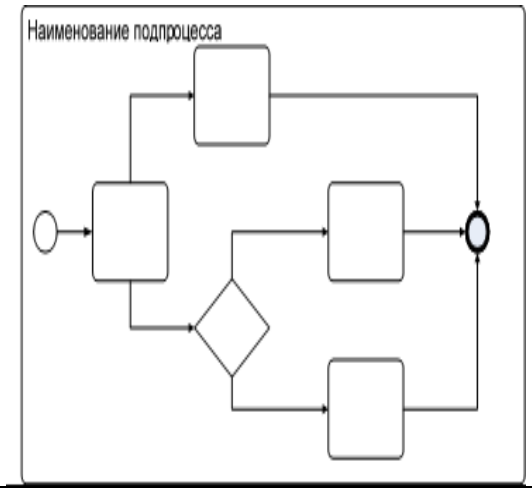
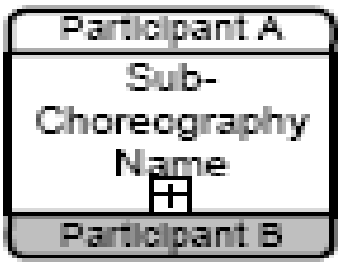
Ниже в таблице 9.2 приведен полный список элементов схемы моделирования бизнес процессов, которые могут быть описаны посредством нотации моделирования бизнес процессов (BMPL).

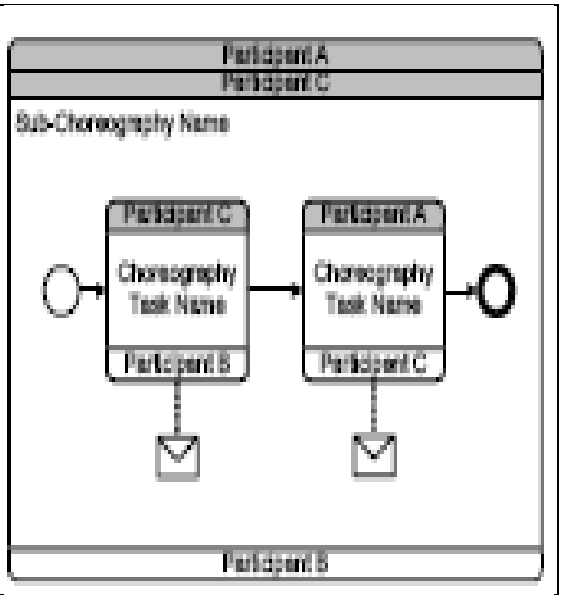
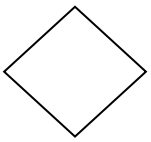
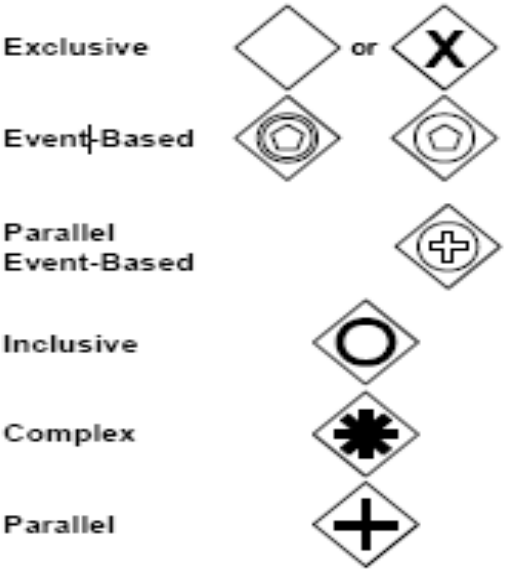
Таблица 9.2 Совокупность всех элементов моделирования

Элемент	Описание	Нотация
Событие (Event)	Событие – это нечто, что «происходит» в ходе бизнес процесса. События влияют на течение бизнес процесса и обычно имеют причину (триггер) или воздействие (результат). События – это круги с открытым центром для обеспечения возможности внутренним маркерам различать разные триггеры или результаты. Существует три типа событий, классифицированные по времени воздействия на ход процесса: Начало,	 Наименование источника

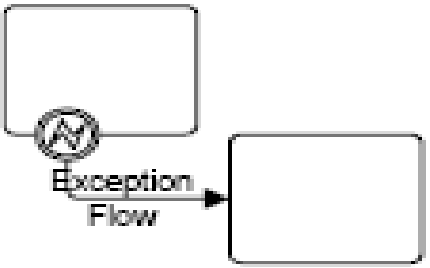
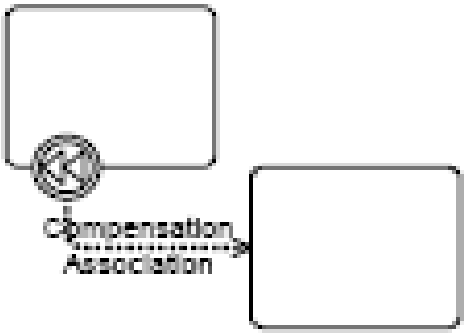
	Промежуточные события и Конец.	
Виды (категория) потока событий (Flow Dimension) (например, Старт, Промежуточные события, Конец) Старт	Событие Старт указывает место, где начинается данный процесс. Событие Старт может быть: Пустое событие начала, Сообщение, Таймер, Правило, Связь, Сложные элементы	 Старт
Промежуточные события	Промежуточные события находятся между событиями Старт и Конец. Они влияют на ход процесса, но не являются непосредственно началом или концом процесса. Промежуточные события могут быть: Пустое событие промежуточное, Сообщение, Таймер, Ошибка, Отмена, Компенсация, Правило, Связь, Сложные элементы.	 Промежуточные события
Конец	Событие Конец указывает, где заканчивается процесс. Событие Конец может быть: Пустое событие конец, Сообщение, Ошибка, Отмена, Компенсация, Связь, Завершение, Сложные элементы.	 Конец

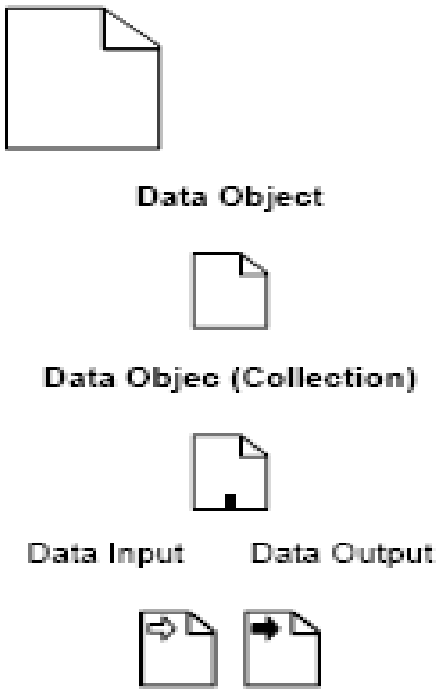
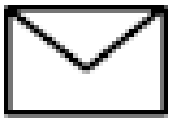
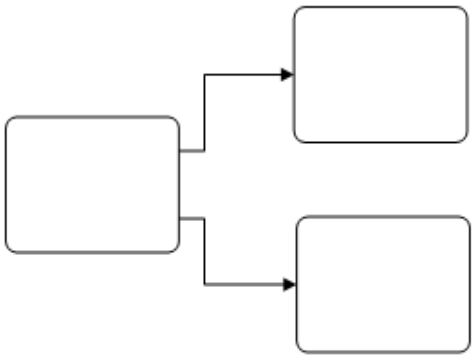
Типы триггеров событий (Сообщение, Таймер, Ошибка, Отмена, Компенсация, Правило, Связь, Сложные, Завершение. И др.)	Старт и большинство промежуточных процессов обладают «триггерами», определяющими причину события. Есть много способов инициировать процесс. Событие Конец может определять «результат», следствие завершения последовательного потока. BPMN. 	
Задача (элементарное действие)	Задача – элементарное действие в пределах процесса. Задача используется, когда работа в процессе не может быть разбита на составляющие	
Деятельность	Деятельность может быть элементарной и неэлементарной (составной). Типы деятельности, являющиеся частью модели процесса: Процесс, Подпроцесс и Задача. Задачи и подпроцессы – закругленные прямоугольники. Процессы – либо безграничны, либо содержатся в пределах области.	

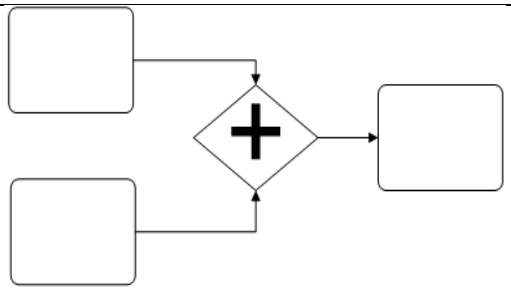
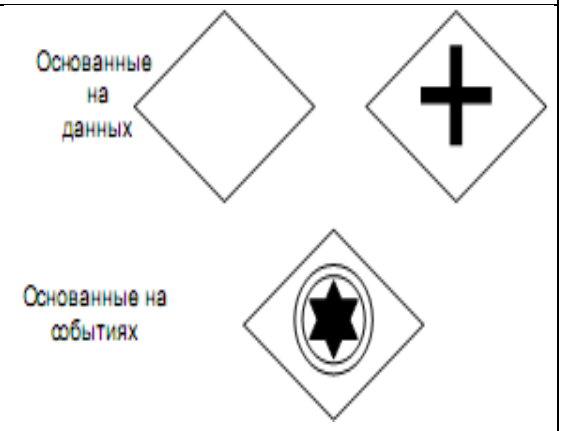
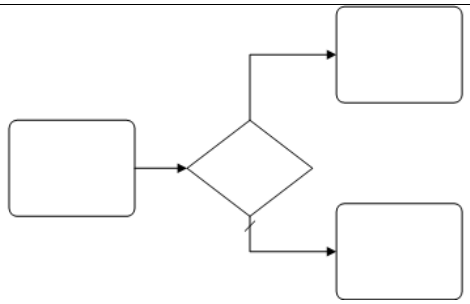
Процесс/подпроцесс	Подпроцесс – это сложное действие в пределах процесса. Он может быть разбит на составляющие при помощи совокупности под-действий.	Смотрите два следующих рисунка
Сжатый подпроцесс	Элементы подпроцесса не видны на схеме. Знак «плюс» в середине нижней части блока означает, что действие – подпроцесс, и его элементы находятся уровнем ниже.	
Расширенный подпроцесс	Границы подпроцесса раздвигаются и элементы (процесса) видны в пределах его границ. Следует отметить, что последовательный поток не может пересекать границы подпроцесса.	
Сжатый подпроцесс хореографии Collapsed Sub-Choreography	Знаком «плюс» в нижнем центре Целевой Название группы формы указывает, что Активность является подпроцессов и имеет более низкий уровень детализации.	
Расширенный подпроцесс хореографии Expanded Sub-Choreography	Границы подпроцесса хореографии расширение и деталей (хореография) видны в пределах его границ. Обратите внимание, что последовательность потоков не может пересечь	

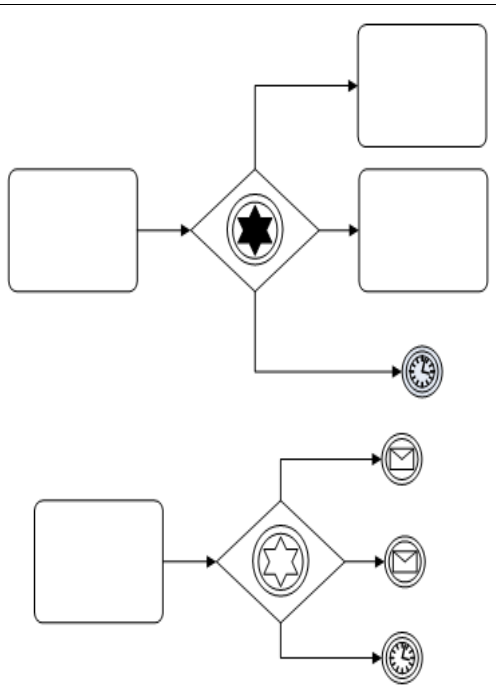
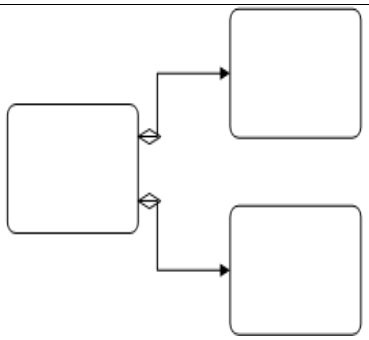
	границы подпроцессов хореографии	
Шлюз, узел	Объединение используется для контроля расхождения и схождения множественных последовательных потоков. Таким образом, данный элемент будет определять разветвление, раздвоение, слияние и соединение маршрутов.	
Типы контроля узлов	Значки внутри ромба обозначают тип контроля потока. Типы контроля включают в себя: •Исключающее ИЛИ – исключающее решение и слияние. Может быть как основанным на данных, так и основанным на событиях Основанный на данных может изображаться с маркером X или без него • ИЛИ – включающее решение и слияние • Сложный - сложные условия и ситуации (например, 3 из 5 потоков) • И – раздвоение и соединение Каждый тип контроля оказывает влияние	

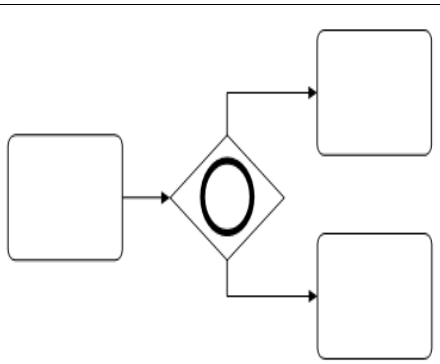
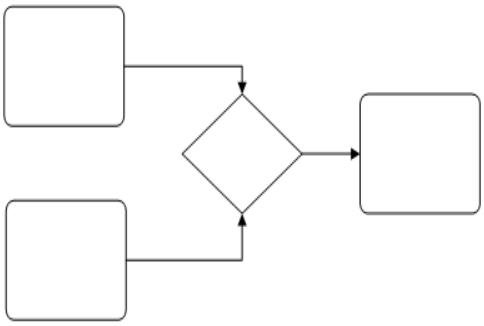
	как на входящие, так и на исходящие потоки	
Последовательный поток	Последовательный поток указывает порядок, в котором будут выполняться действия процесса.	Смотрите семь следующих рисунков
Стандартный поток	Стандартный последовательный поток – поток, берущий начало от события Старт и идущий по действиям через альтернативные и параллельные маршруты до своего завершения в событии Конец.	
Неконтролируемый поток	Неконтролируемый поток - поток, не подверженный влиянию каких-либо условий или не проходящий через объединение. Простейший пример – единичный последовательный поток, соединяющий два действия. То же можно отнести и к множественным последовательным потокам, сходящимся в действии или расходящимся от него. Для каждого неконтролируемого потока будет появляться маркер от объекта-источника до целевого объекта.	
Условный поток	Последовательный поток может иметь условные выражения, которые измеряются по времени выполнения, с целью определить – будет ли использоваться поток. Если условный поток исходит от действия, то в начале линии последовательного потока будет ромбик (рисунок справа). Если условный	

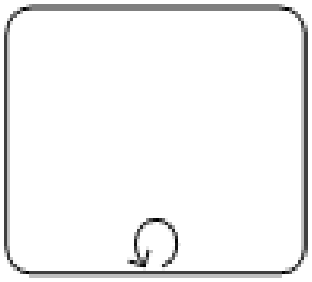
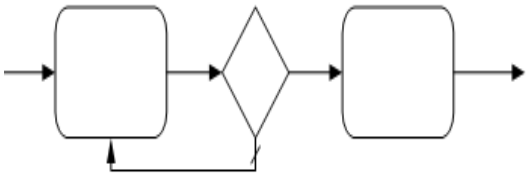
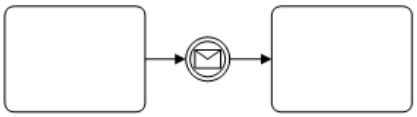
	поток выходит из соединения, то на линии не будет ромбика (рисунок в предыдущей строке).	
Условный поток по умолчанию.	Как для исключающих, так и для включающих ветвлений характерен один тип потока – условный поток по умолчанию. Данный поток будет использоваться в случае, если все другие условные потоки неверны при выполнении. В начале линии у таких последовательных потоков будет диагональная черта (рисунок справа).	
Исключающий поток	Исключающий поток находится вне стандартного потока процесса и основывается на промежуточном событии, имеющем место при выполнении процесса.	
Поток сообщений	Поток сообщений символизирует поток сообщений между двумя участниками, готовыми к их отправке и получению. В BPMN две отдельных области на диаграмме будут символизировать двух участников (например, бизнес объекты или бизнес роли).	
Компенсирующая ассоциация	Компенсирующая ассоциация находится вне стандартного потока процесса и основывается на событии (промежуточное событие Отмена), которое инициируется через сбой в групповой операции или событии Компенсация. Цель	

	ассоциации - действие Компенсация.	
Объекты данных	Объекты данных рассматриваются как артефакты, так как они не влияют непосредственно на последовательный поток или поток сообщений процесса, но они обеспечивают информацию о том, какие действия требуют выполнения и/или что они производят.	
Сообщение	Сообщение используется для описания содержимого связи между двумя участников (как определяется PartnerRole бизнеса или бизнес-PartnerEntity).	
Раздвоение (И-разбиение)	BPMN использует термин «раздвоение» относительно деления маршрута на два или более параллельных маршрута (также известных как И-разбиение). Это участок процесса, где действия могут выполняться одновременно, а не последовательно. Существует два варианта: Может использоваться множественный исходящий последовательный поток (рисунок справа вверху). Это неконтролируемый поток –	

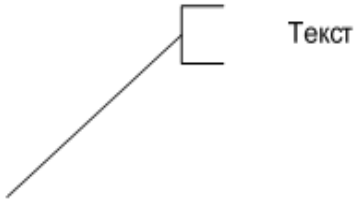
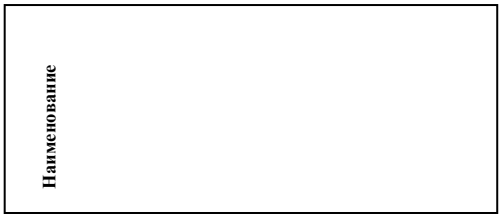
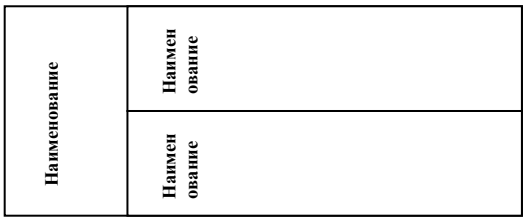
	предпочтительный метод для большинства ситуаций. Может использоваться параллельное (И) объединение (рисунок справа внизу). Этот способ используется редко, обычно в сочетании с другими объединениями.	
Соединение (И Соединение)	BPMN использует термин «соединение» относительно соединения двух или более параллельных маршрутов в один маршрут (так же известно как И-Соединение или синхронизация). Параллельное (И) объединение показывает соединение множественных потоков.	
Ветвление, точка ветвления (ИЛИ-Разбиение)	Ветвления – это объединения в рамках бизнес процесса, в которых контрольный поток может пойти по одному или более альтернативному маршруту.	Смотрите следующие пять рисунков
Исключающие объединения	Исключающее объединение (XOR) ограничивает поток, таким образом, что во время выполнения можно выбрать только одну группу альтернативных маршрутов. Существует два типа исключающих объединений: основанные на данных и основанные на событиях.	
Объединения, основанные на данных	Данное ветвление представляет собой точку ветвления, где альтернативные маршруты основаны на условных выражениях, содержащихся в исходящем последовательном потоке.	

	Может быть выбран только один из альтернативных маршрутов.	
Объединения, основанные на событиях	Данное ветвление представляет собой точку ветвления, где альтернативные маршруты основываются на событии, имеющим место в данной точке процесса. Данное событие - обычно это получение сообщения - определяет маршрут. Могут использоваться другие типы событий, например, таймер. Можно выбрать только один из альтернативных маршрутов. Существует два способа получения сообщений: Могут использоваться Задачи с типом «Получение» (рисунок справа вверху) Могут использоваться промежуточные события типа «Сообщение» (рисунок справа внизу).	
Включающие объединения	Данное ветвление представляет собой точку ветвления, где альтернативные маршруты основываются на условных выражениях, содержащихся в пределах исходящего последовательного потока. В некотором смысле, это группировка связанных независимых двойных (Да/Нет) ветвлений. Так как каждый маршрут независим, могут быть задействованы все комбинации маршрутов, от нуля до бесконечности. Однако ветвление должно	

	быть построено так, чтобы был задействован хотя бы один маршрут. С целью проверки наличия хотя бы одного маршрута, может использоваться условие по умолчанию. Существует два варианта данного типа ветвления: Первый тип использует совокупность условных последовательных потоков, отмеченных ромбиками (рисунок справа вверху). Второй тип использует объединение OR (ИЛИ), обычно в сочетании с другими объединениями (рисунок справа внизу).	
Слияние (ИЛИ-соединение)	BPMN использует термин «слияние» в отношении исключающего сочетания двух или более маршрутов в один (так же известного как OR (ИЛИ)-соединение). Объединение Слияние (XOR) показывает слияние множественных потоков. Если все входящие потоки альтернативные, то в Объединении нет необходимости. Таким образом, неконтролируемый поток обеспечивает аналогичное развитие.	
Цикличность	BPMN поддерживает два типа цикличности в рамках процесса.	Смотрите следующие два рисунка

Цикличность действия	Атрибуты задач и подпроцессов определяют, носят ли они повторяющийся характер или выполнены один раз. Существует два типа циклов: стандартный и многовариантный. В середине нижней части действия появится небольшой значок цикличности.	
Цикличность последовательного потока	Циклы могут создаваться путем соединения последовательного потока с каким-либо «противоположным» объектом. Объект считается «противоположным», если у данного объекта есть исходящий последовательный поток, ведущий к ряду других последовательных потоков, последний их которых - входящий последовательный поток по отношению к исходному объекту.	
Перерыв в процессе (нечто, не попадающее под контроль процесса, приводит к временному прекращению процесса)	Перерыв в процессе – участок процесса, где произойдет предполагаемая задержка. Промежуточное событие показывает фактическое развитие процесса (рисунок справа вверху). Кроме того, артефакт перерыва в процессе, исходя из проектирования разработчиком или инструментом моделирования, может быть связан с событием для указания места задержки в пределах потока.	

Вложенный/Встроенный подпроцесс (совпадающий блок)	Вложенный (или встроенный) подпроцесс – это действие, обладающее тем же набором данных, что и его родительский процесс. Ему противопоставляется процесс независимый, повторно используемый, идущий от родительского процесса. Данные должны быть переданы основному, а не вложенному подпроцессу.	Для вложенных подпроцессов нет специального обозначения.
Группа (блок вокруг группы объектов в целях документирования)	Группировка действий, не оказывающая влияние на последовательный поток. Группировка может использоваться для документирования или анализа. Группы могут так же использоваться для обозначения действий распределенных групповых операций, расположенных по ширине областей.	
Соединитель страниц	Данный объект обычно используется при распечатывании и указывает место, где заканчивается последовательный поток, а затем вновь берет начало на следующей странице. В качестве соединителя страниц может использоваться промежуточное событие Связь.	
Ассоциация	Ассоциация связывает информацию с объектами схемы. Объекты схемы могут быть связаны с текстом и графическими объектами не относящимися к данной схеме.	

Текстовая аннотация (объединенная с ассоциацией)	Текстовая аннотация – это возможность для разработчика привести дополнительную информацию для читателя схемы BPMN.	
Пул	Пул (Область) олицетворяет участника процесса. Он так же может играть роль «дорожки» и графического контейнера для разделения совокупности действий из других областей, обычно в контексте ситуаций «бизнес для бизнеса».	
Дорожки	Дорожка – это подраздел в пределах области, его протяженность равна длине области, как по вертикали, так и по горизонтали. Дорожки организовывают и классифицируют действия.	

9.5 Использование текста, цвета, размера и линий на схеме

Текстовые аннотации объектов могут отражать дополнительную информацию о процессе или атрибутах объектов в рамках процесса.

У объектов и потоков могут быть признаки (например, название и/или другие атрибуты), находящиеся внутри формы, а также над, либо под формой, в любом направлении и месте, в зависимости от пожеланий разработчика или продавца инструмента моделирования.

Заливка, предназначенная для графических элементов, может быть белой или прозрачной.

Могут использоваться другие цвета заливки в соответствии с целями разработчика или продавца инструмента моделирования (например, в целях подчеркивания значимости атрибута объекта).

Объекты схемы и маркеры могут быть любого размера, в соответствии с целями разработчика или инструмента моделирования.

Линии, используемые для рисования графических объектов, могут быть черными.

Могут использоваться другие цвета линии в соответствии с целями разработчика или инструмента (например, в целях подчеркивания значимости атрибута объекта).

Могут использоваться другие стили линии в соответствии с целями разработчика или инструмента (например, в целях подчеркивания значимости атрибута объекта) при условии, что стиль линии не должен вступать в конфликт ни с одним данным стилем линии, определенным BPMN. Таким образом, не должен изменяться стили линий последовательного потока, потока сообщений и ассоциаций.

9.6 Правила соединения объектов

Входящий последовательный поток может соединяться с любым участком на объекте (левая и правая стороны, верхняя и нижняя части), то же относится и к исходящему последовательному потоку. Поток сообщений также обладает этим свойством. BPMN допускает наличие нескольких вариантов, но разработчикам рекомендуется использовать лучшие методы связывания объектов, с целью обеспечить легкое и быстрое понимание схемы. Это еще более важно, когда схема содержит последовательный поток и поток сообщений. В таких ситуациях лучше всего выбрать направление последовательного потока, не важно, вправо, влево, вверх или вниз, а затем направить поток сообщений под углом 90 градусов к последовательному потоку. В результате получится значительно более понятная схема.

9.7 Примеры

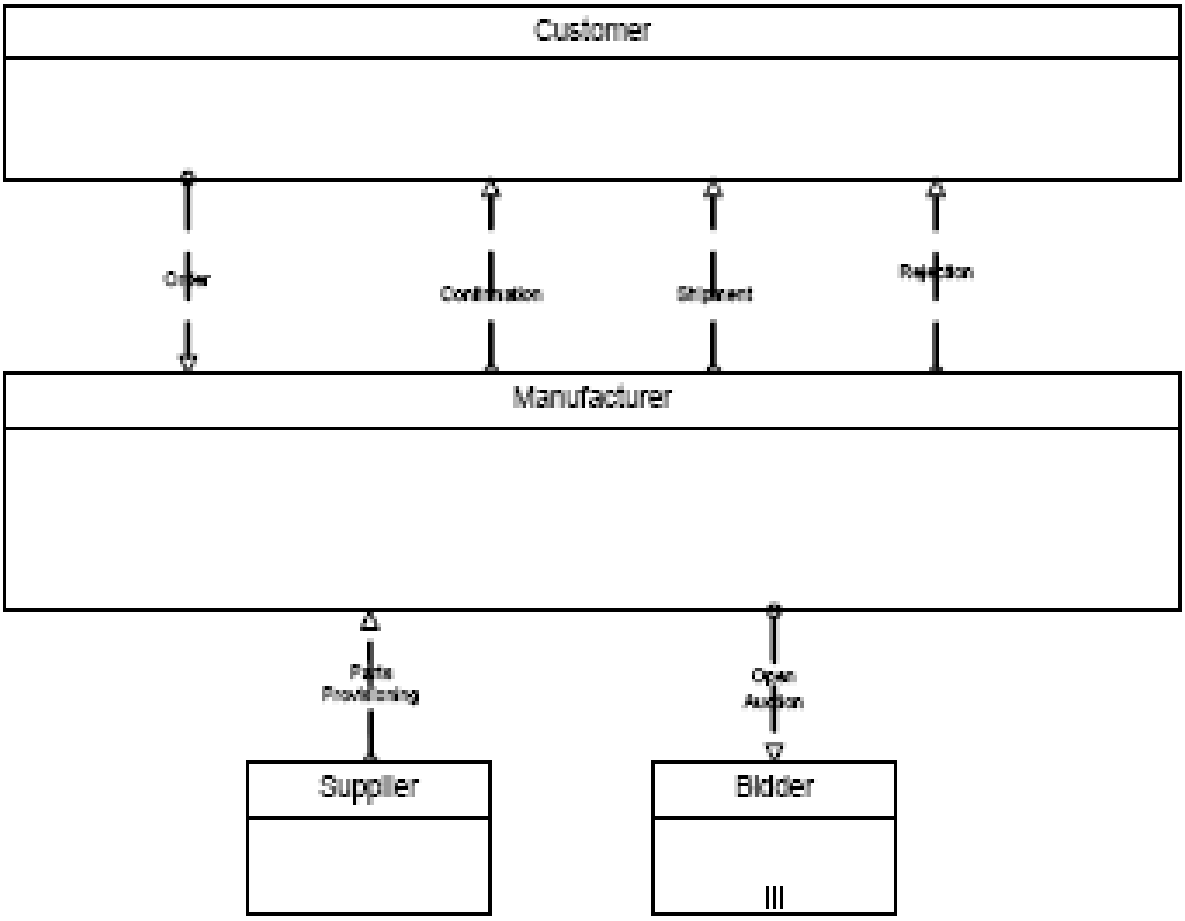


Рис. 9.6 Пример совместной диаграммы с пулами

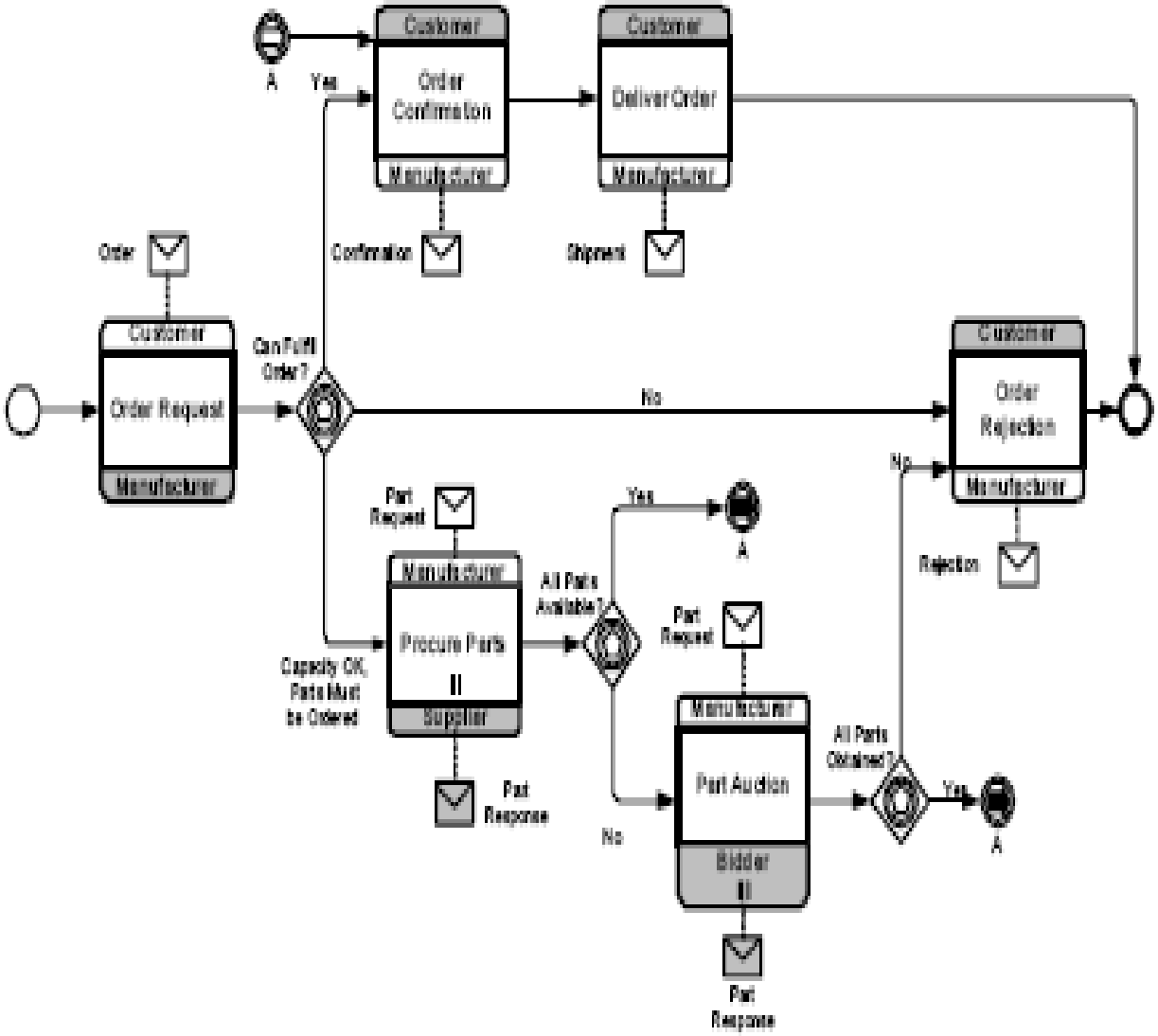


Рис. 9.7 Пример диаграммы хореографии

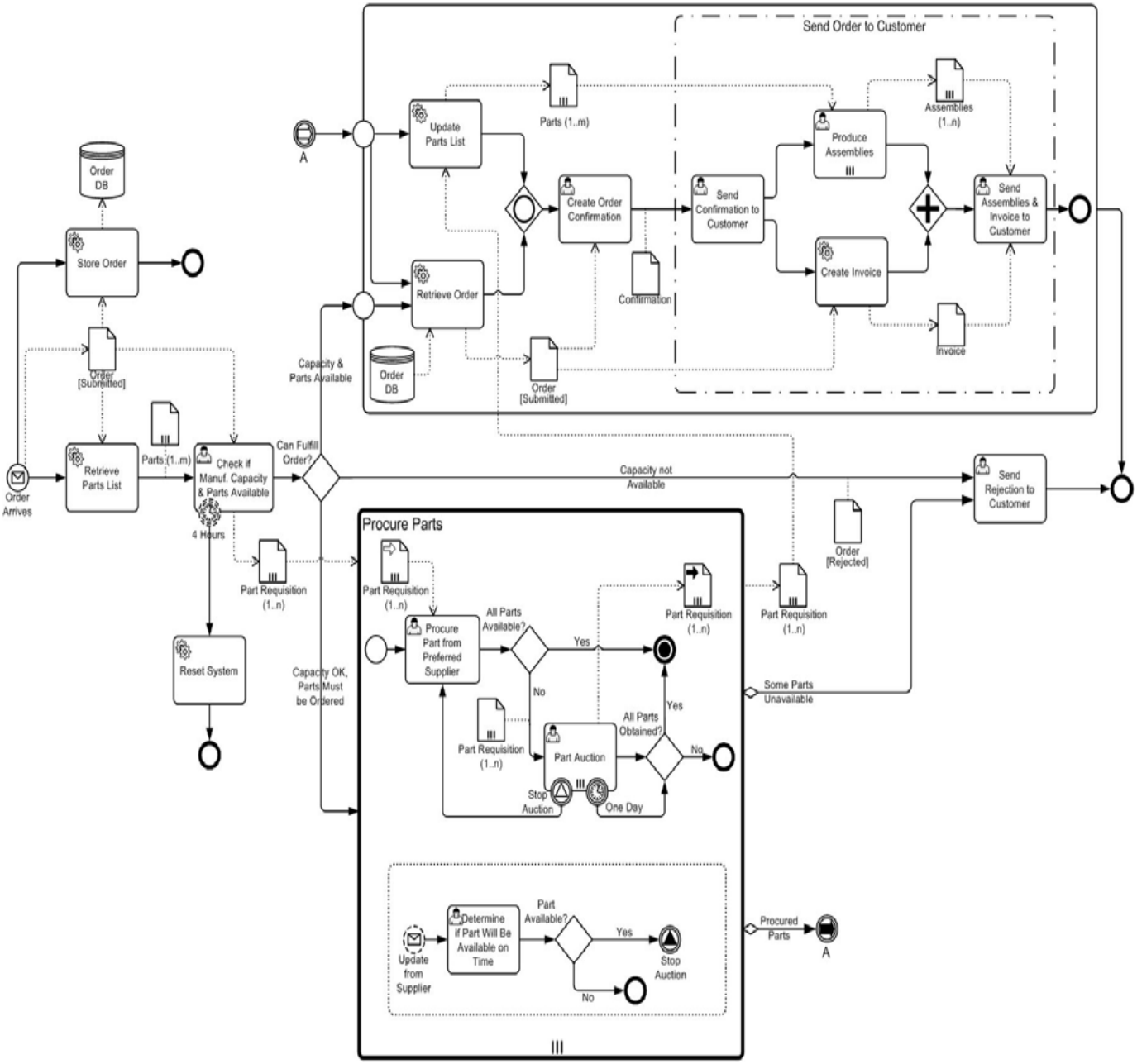


Рис. 9.8 Пример диаграммы процесса (оркестровка)

Модель бизнес-процесса компонента «Формирование месячного портфеля заявок» приведена на рисунках 9.3 – 9.11

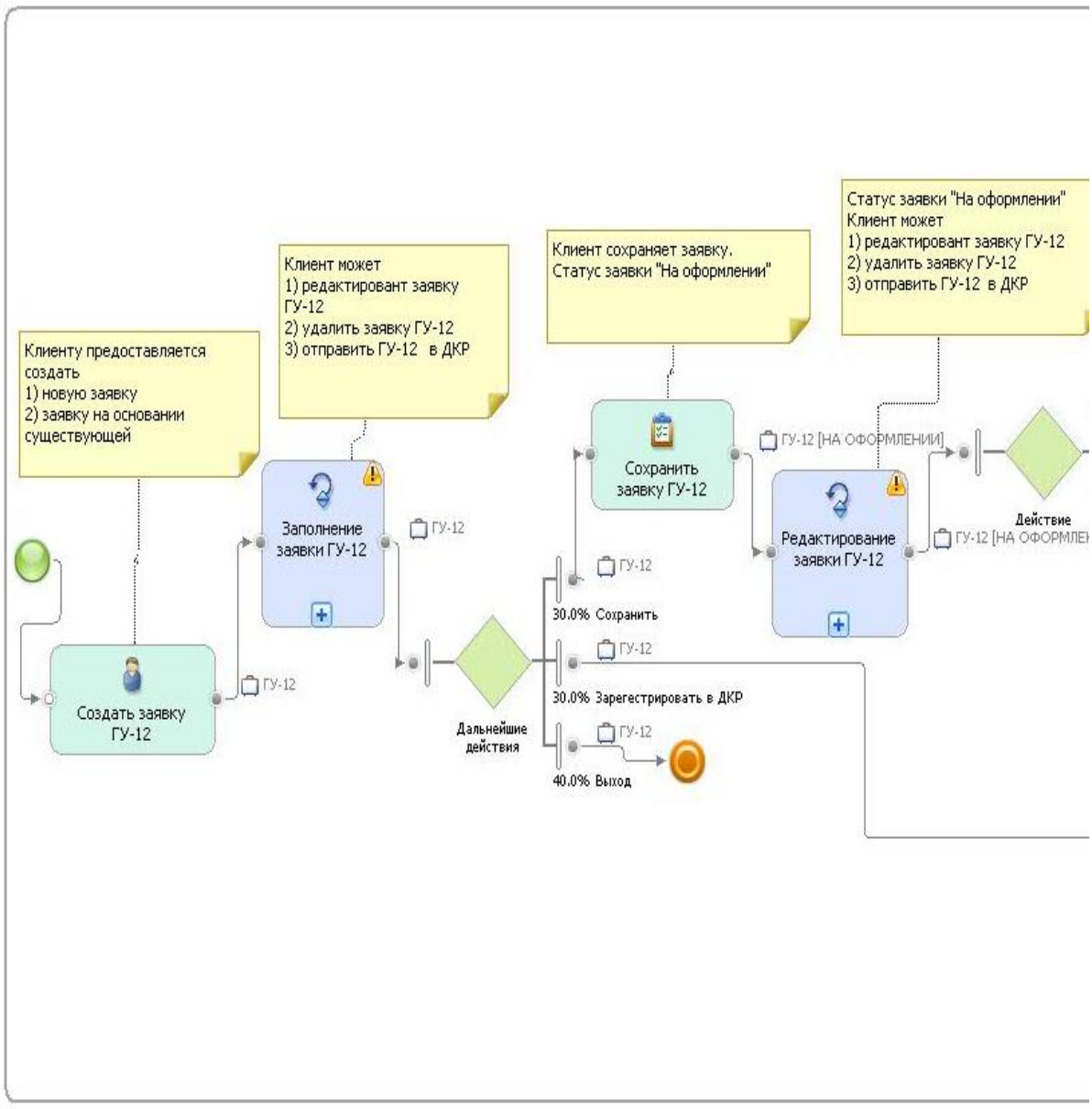
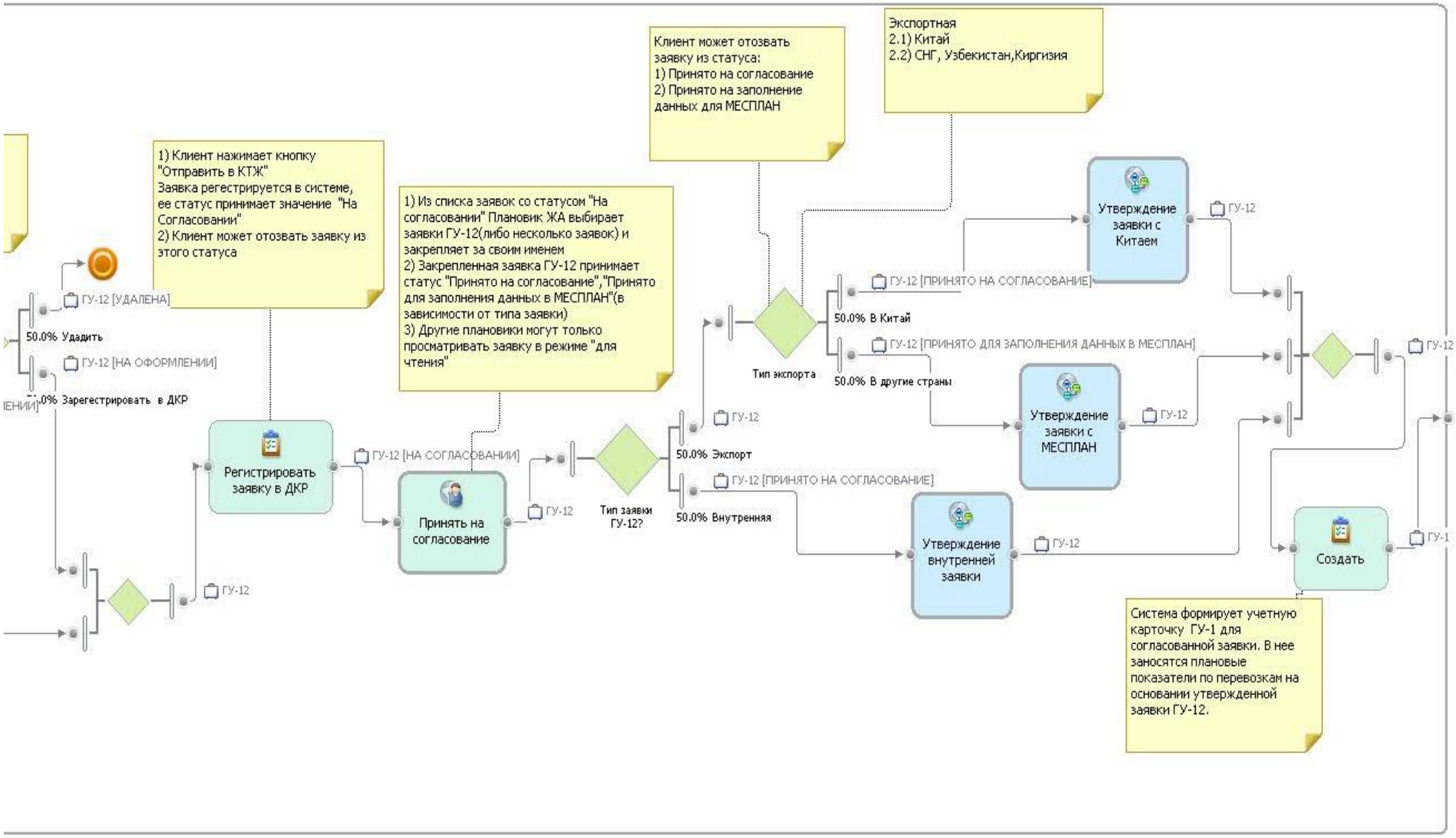


Рисунок 9.3 – Формирование месячного портфеля заявок общий план


1) Клиент нажимает кнопку "Отправить в КТЖ"
Заявка регестрируется в системе, ее статус принимает значение "На Согласовании"
2) Клиент может отозвать заявку из этого статуса
1) Из списка заявок со статусом "На согласовании" Плановик ЖА выбирает заявки ГУ-12(либо несколько заявок) и закрепляет за своим именем
2) Закрепленная заявка ГУ-12 принимает статус "Принято на согласование","Принято для заполнения данных в МЕСПЛАН"(в зависимости от типа заявки)
3) Другие плановики могут только просматривать заявку в режиме "для чтения"
Клиент может отозвать заявку из статуса:
1) Принято на согласование
2) Принято на заполнение данных для МЕСПЛАН
Экспортная
2.1) Китай
2.2) СНГ, Узбекистан,Киргизия
ГУ-12 [УДАЛЕНА]
50.0% Удадить
ГУ-12 [НА ОФОРМЛЕНИИ]
50.0% Зарегестрировать в ДКР
Регистрировать заявку в ДКР
ГУ-12 [НА СОГЛАСОВАНИИ]
Принять на согласование
ГУ-12
Тип заявки ГУ-12?
50.0% Экспорт
50.0% Внутренняя
Тип экспорта
ГУ-12 [ПРИНЯТО НА СОГЛАСОВАНИЕ]
50.0% В Китай
ГУ-12 [ПРИНЯТО ДЛЯ ЗАПОЛНЕНИЯ ДАННЫХ В МЕСПЛАН]
50.0% В другие страны
Утверждение заявки с Китаем
Утверждение заявки с МЕСПЛАН
Утверждение внутренней заявки
Создать
Система формирует учетную карточку ГУ-1 для согласованной заявки. В нее заносятся плановые показатели по перевозкам на основании утвержденной заявки ГУ-12.

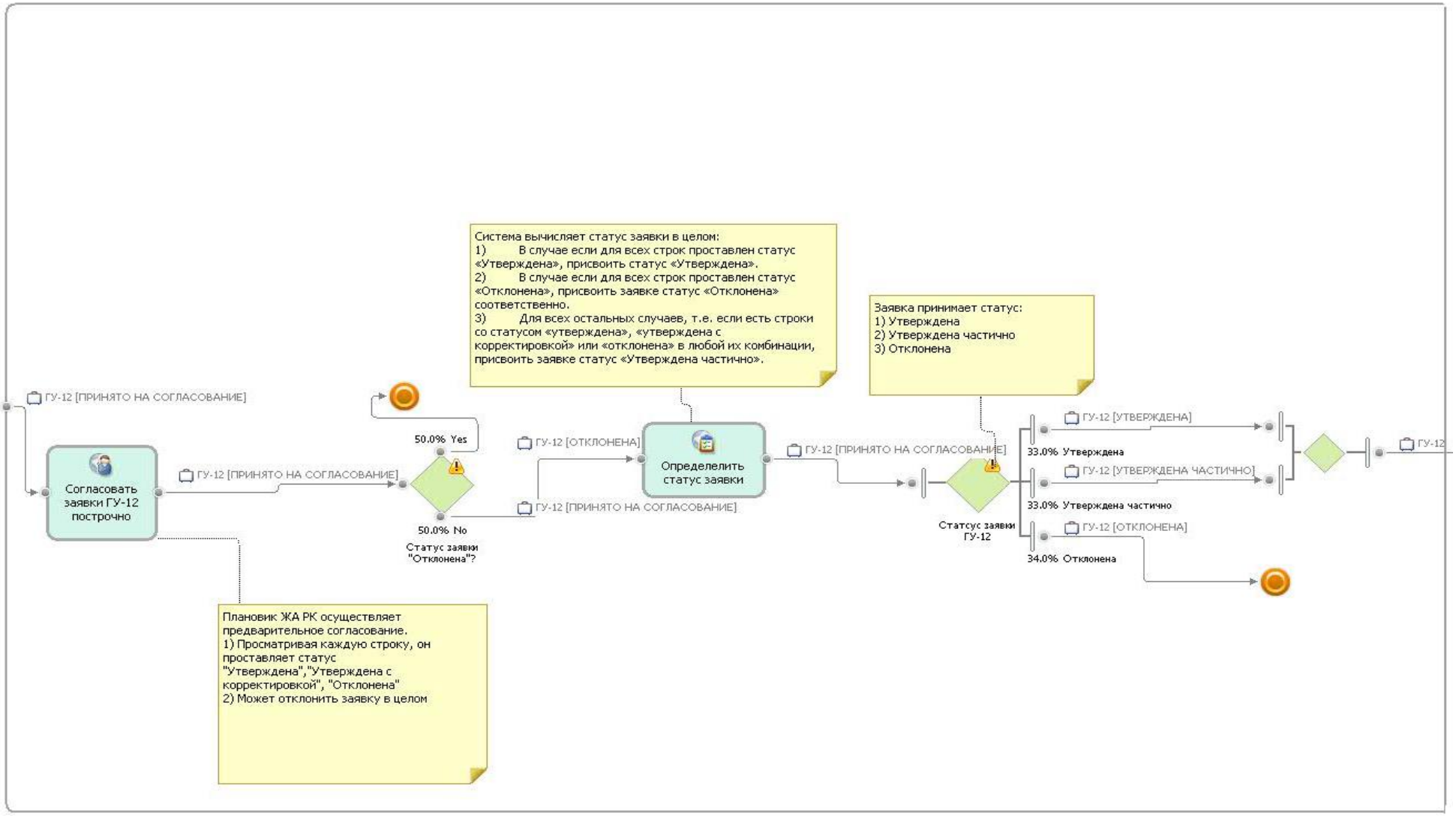

ГУ-12 [ПРИНЯТО НА СОГЛАСОВАНИЕ]
Согласовать заявки ГУ-12 построчно
ГУ-12 [ПРИНЯТО НА СОГЛАСОВАНИЕ]
50.0% Yes
50.0% No
Статус заявки "Отклонена"?
ГУ-12 [ОТКЛОНЕНА]
ГУ-12 [ПРИНЯТО НА СОГЛАСОВАНИЕ]
Определелить статус заявки
Система вычисляет статус заявки в целом:
1) В случае если для всех строк проставлен статус «Утверждена», присвоить статус «Утверждена».
2) В случае если для всех строк проставлен статус «Отклонена», присвоить заявке статус «Отклонена» соответственно.
3) Для всех остальных случаев, т.е. если есть строки со статусом «утверждена», «утверждена с корректировкой» или «отклонена» в любой их комбинации, присвоить заявке статус «Утверждена частично».
Заявка принимает статус:
1) Утверждена
2) Утверждена частично
3) Отклонена
ГУ-12 [ПРИНЯТО НА СОГЛАСОВАНИЕ]
Статусус заявки ГУ-12
ГУ-12 [УТВЕРЖДЕНА]
33.0% Утверждена
ГУ-12 [УТВЕРЖДЕНА ЧАСТИЧНО]
33.0% Утверждена частично
ГУ-12 [ОТКЛОНЕНА]
34.0% Отклонена
ГУ-12
Плановик ЖА РК осуществляет предварительное согласование.
1) Просматривая каждую строку, он проставляет статус "Утверждена","Утверждена с корректировкой", "Отклонена"
2) Может отклонить заявку в целом

Рисунок 9.5.Согласование внутренних перевозок

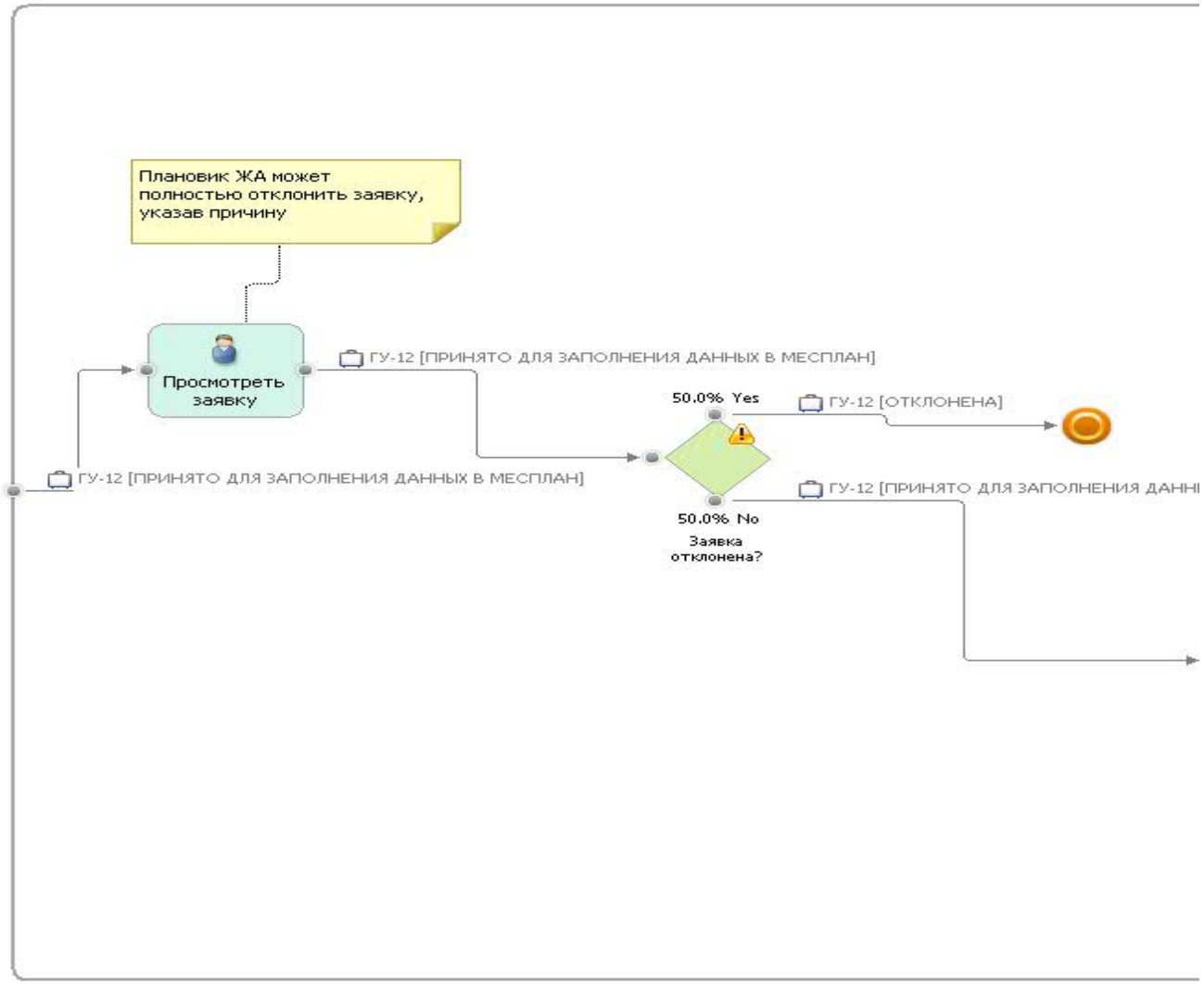


Рисунок 9.6 – Согласование экспорных перевозок СНГ

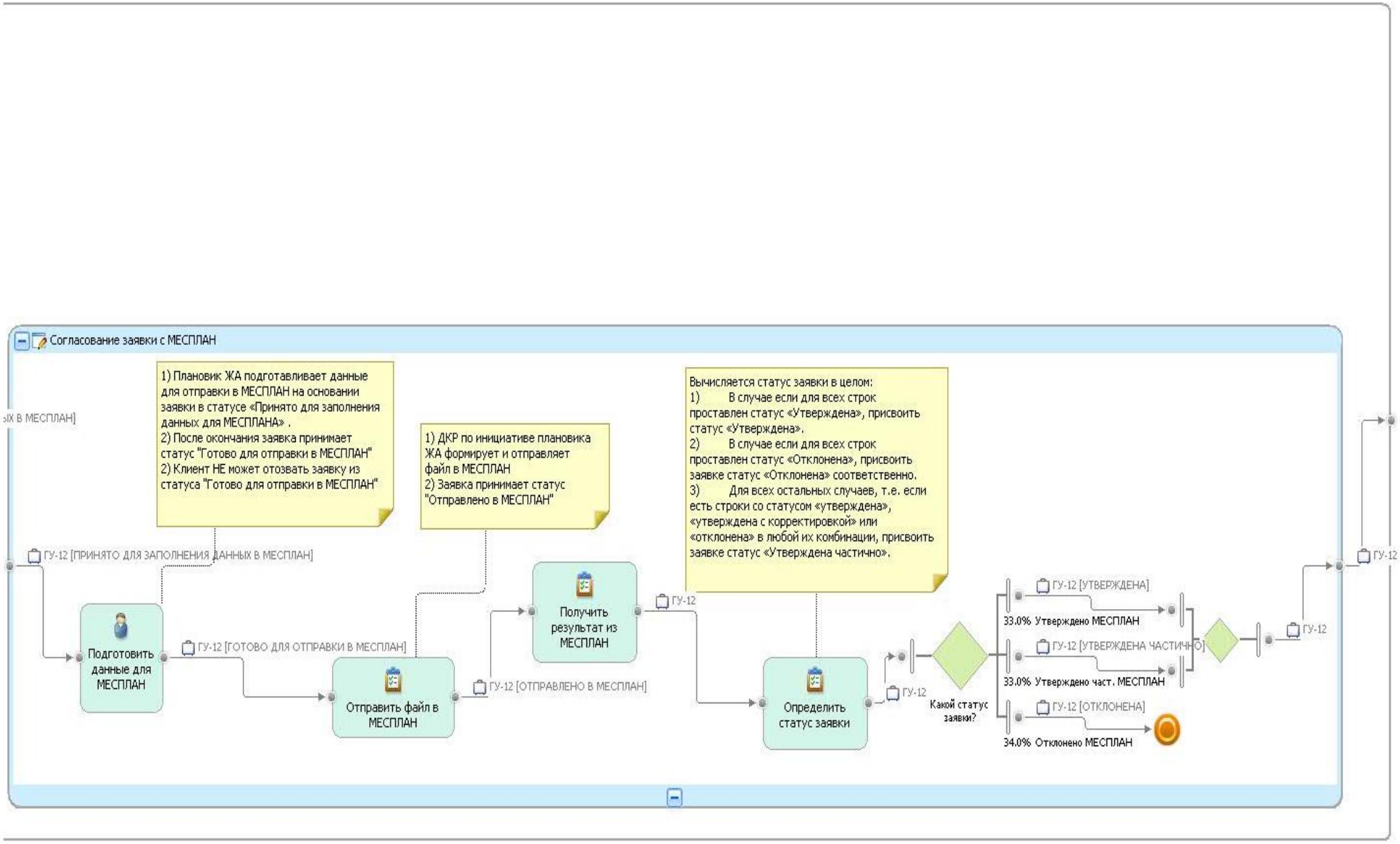
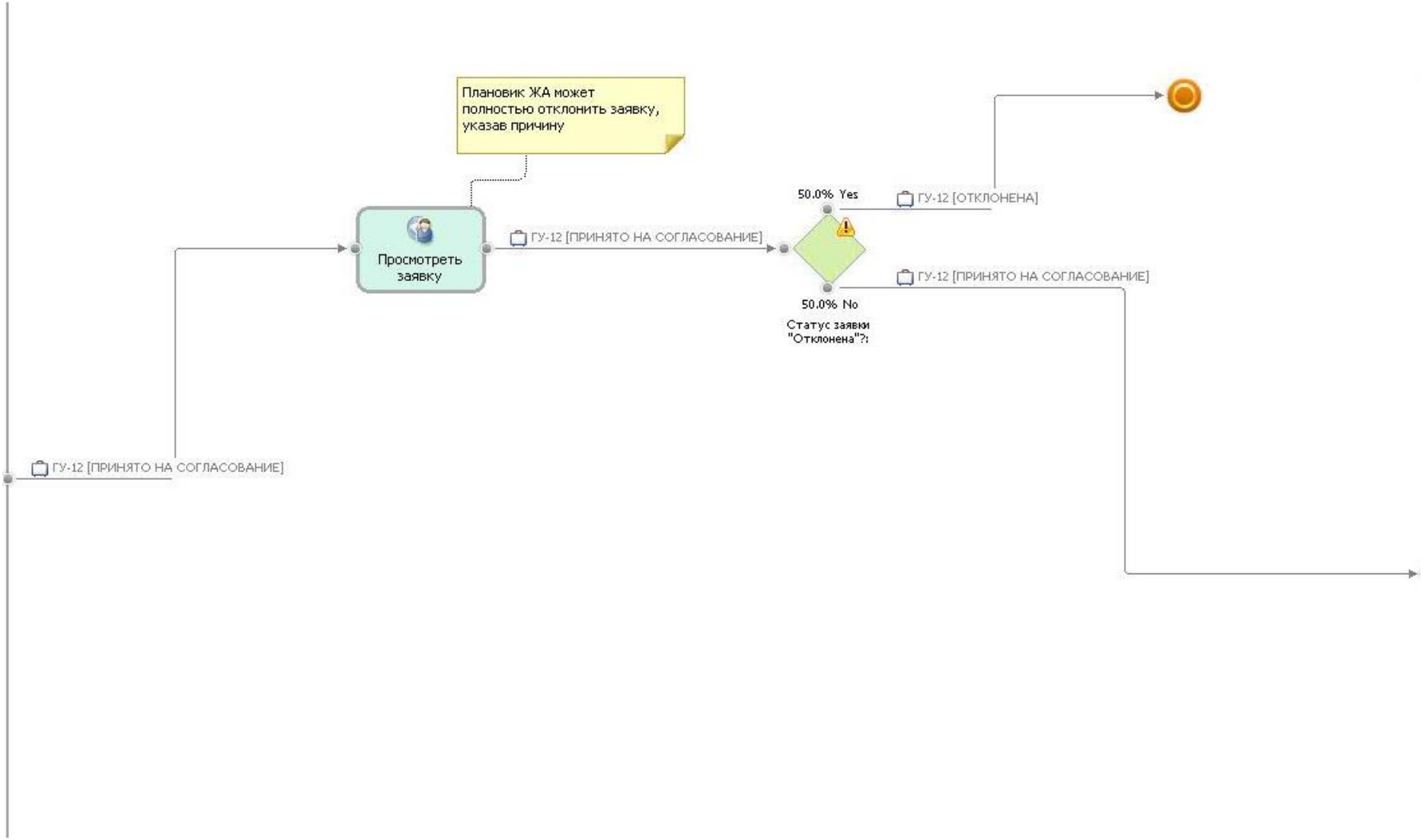
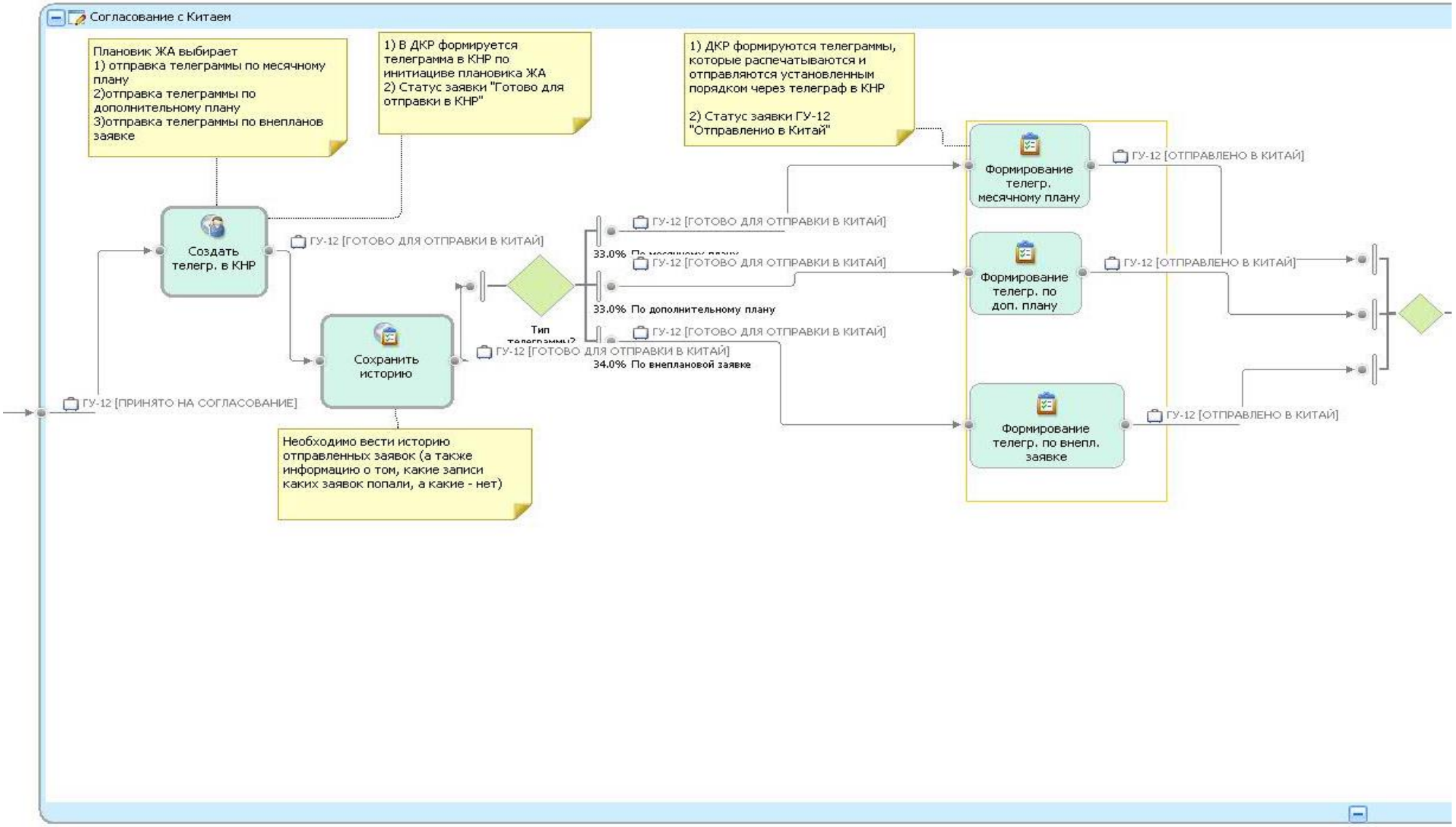


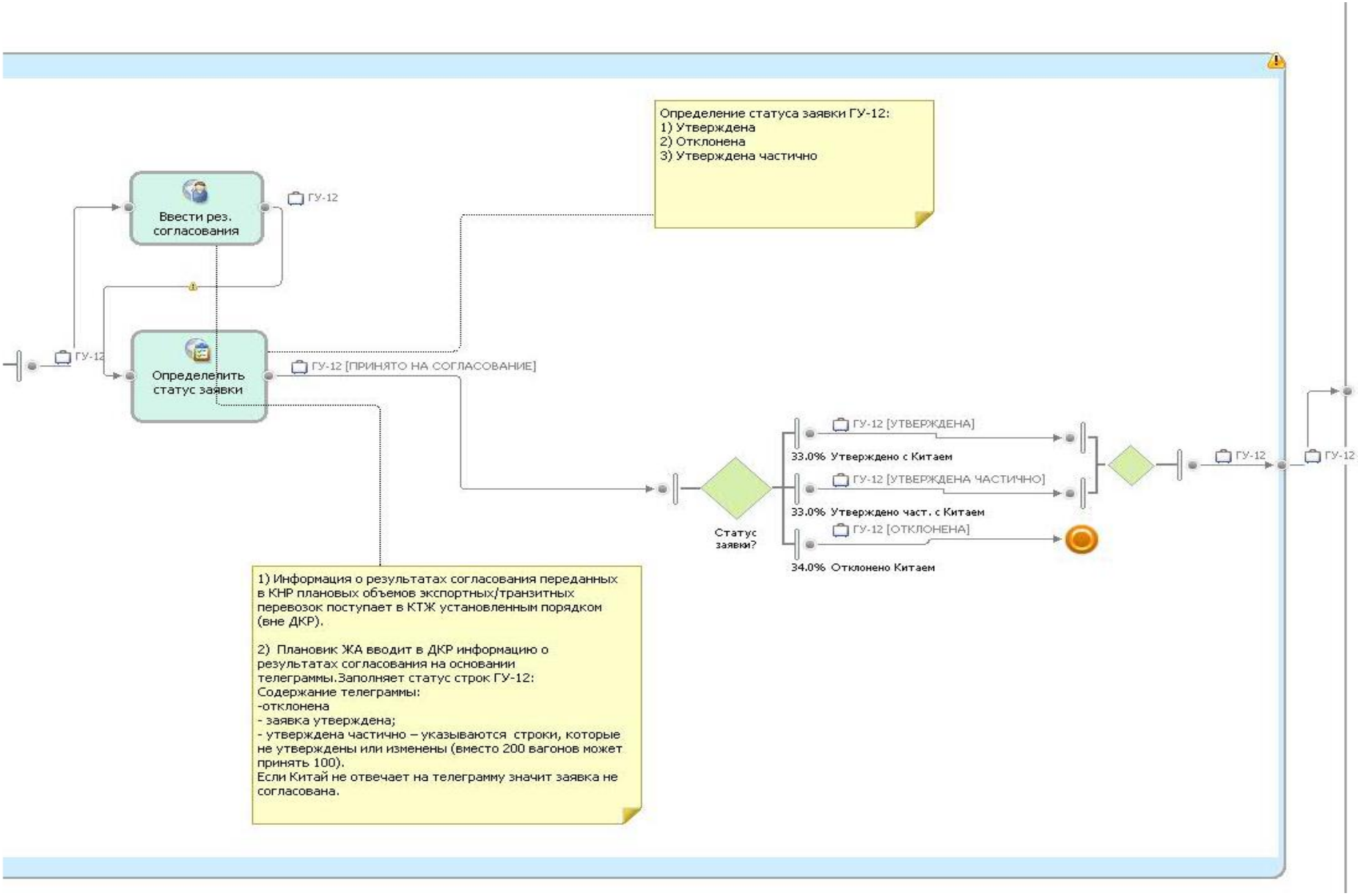
Рисунок 9.7 – Согласование экспорных перевозок СНГ.(Продолжение)



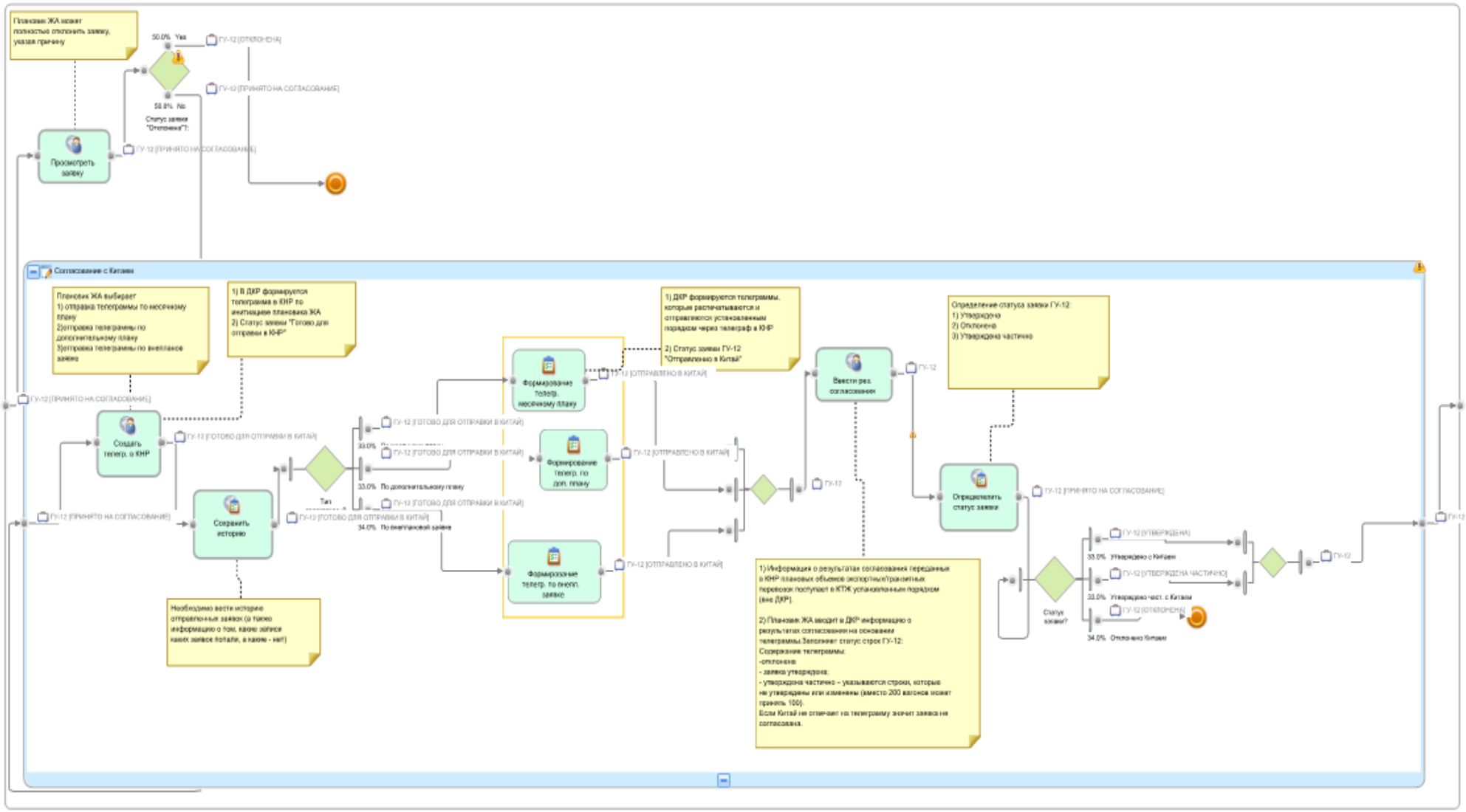
9.8 – Согласование экспорных заявок Китай



9.9– Согласование экспорных заявок, Китай.(Продолжение)



9.10 – Согласование экспорных заявок, Китай. (Продолжение)



9.11 – Согласование экспорных заявок, Китай.(Продолжение)

10. От моделирования на BPMN к моделированию на BPEL и генерации кода

Язык BPMN (Business Process Modeling Notation (BPMN)) был разработан для того, чтобы дать возможность бизнес-пользователям разрабатывать легко понимаемые графические представления бизнес-процессов. BPMN также поддерживает свойства графических объектов, что делает возможным генерацию выполнимого BPEL (Business Process Execution Language) [1]. Таким образом, BPMN создает стандартизованный мост между дизайном бизнес-процессов и их исполнением. Данный материал представляет простой, но наглядный пример того, как BPMN диаграмма может быть использована для создания BPEL процесса.

Когда рисуется BPMN диаграмма для BPEL, необходимо решить, какой будет базовая структура BPEL документа. То есть, будет ли BPEL элемент базироваться на структуре графа (элемент потока) или на блочной структуре (элемент последовательности)? Этот выбор будет влиять на то, сколько стрелок (Sequence Flow) будет изображено для связей элементов. В блочной структуре связи элементов используются только в специфических секциях процесса, где встречаются параллельные действия. В графической структуре наибольшее количество стрелок будет нарисовано для связи элементов, поскольку весь процесс BPEL находится внутри элементов потока. Так как, BPMN 1.0 спецификация [2] берет за основу подход рисования BPMN элементов для BPEL, используя блочную структуру, здесь будет использована графическая структура.

10.1 Моделирование заказа путешествия

Рассмотрим моделирование на примере версии процесса заказа путешествия. Этот пример иллюстрирует несколько ситуаций, которые встречаются внутри BPMN, и как они рисуются для BPEL, такие как параллельные потоки и петли. На Рисунке 10.1 показано оригинальное представление BPEL процесса [3].

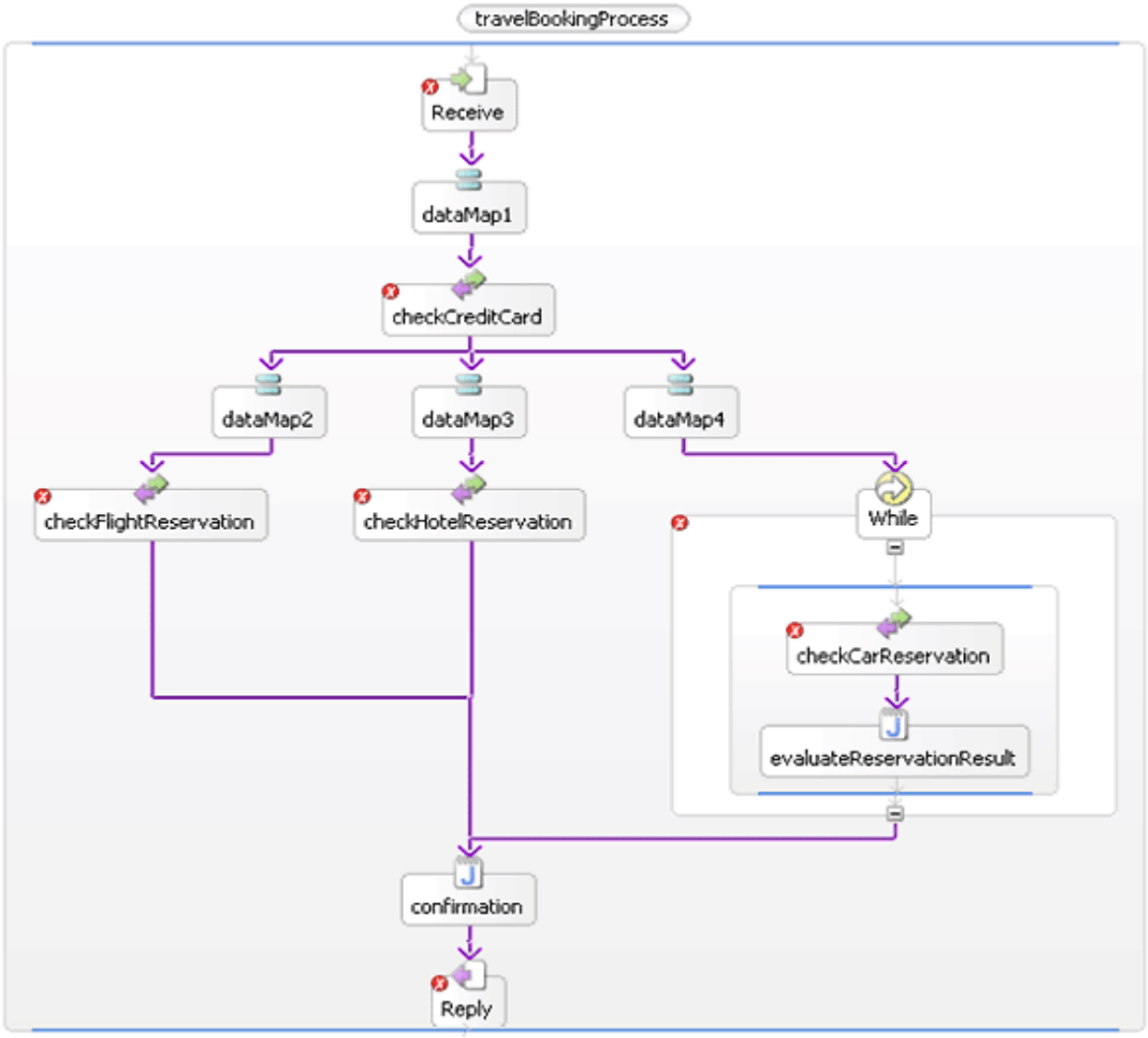


Рисунок 10.1 Модель заказа путешествия, выполнен в WebSphere Studio

Рисунок 2 10.2 показывает, как тот же самый процесс может быть смоделирован с использованием BPMN. Можно отметить, что рисунок 2 показывает модель процесса, выполненную в горизонтальном направлении, слева направо, в то время как оригинальная диагамма на Рисунке 1 выполнена вертикально, сверху вниз. Инструментарий, в котором создается строго BPEL процесс, склонен располагать процесс вертикально. Это не универсально, поскольку бизнес-аналитики предпочитают располагать диаграммы горизонтально, а IT-специалисты — вертикально. BPMN допускает любое расположение диаграмм, однако большинство BPMN-диаграмм горизонтально ориентированы.

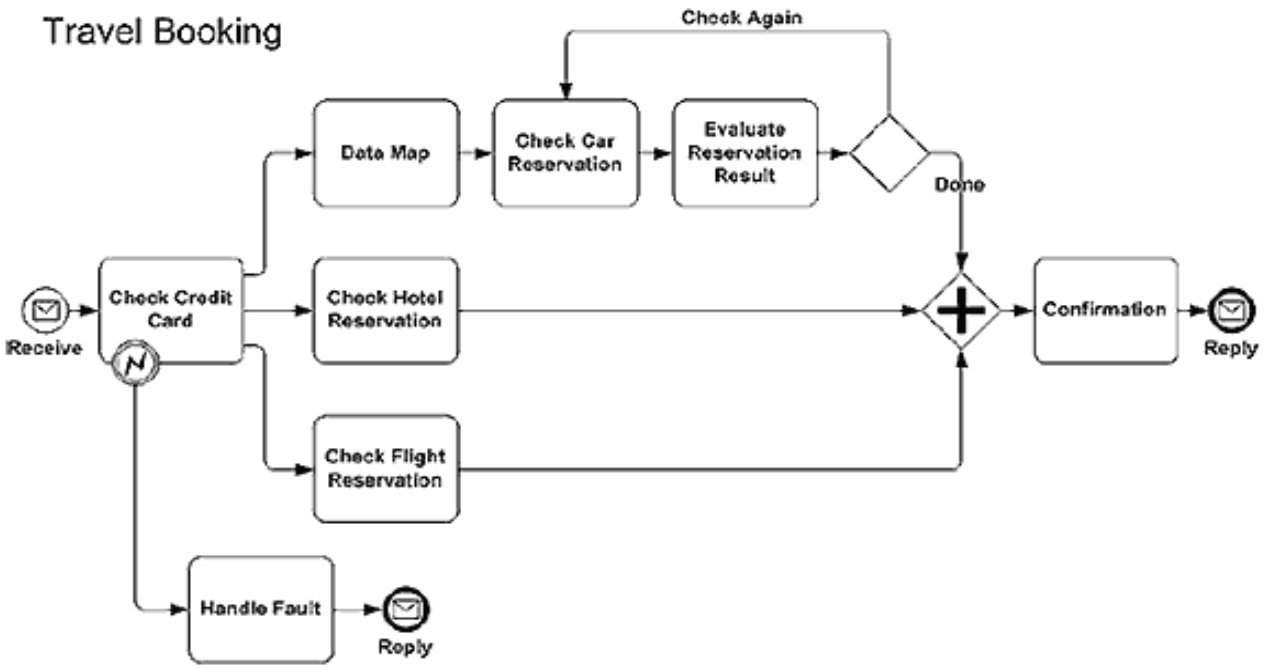


Рисунок 2 10.2. Модель заказа путешествия в BPMN

Процесс начинается с выписки квитанции на заказ путешествия. После выбора кредитной карты выполняется резервирование полета, гостиницы и машины. Для резервирования машины может понадобиться более чем одна попытка. После того, как резервирование выполнено, посылается ответ.

10.2 Определение BPEL информации

BPMN-диаграмма, такая, как на рисунке 2 10.2, может быть использована в разных методологиях и для разных целей, от высокоуровневого описания моделей до детального моделирования, предназначенного для исполнения процессов. Когда одной из целей является определение исполнения процесса и создание BPEL файла для этой цели, тогда модель процесса создается с использованием специально предназначенного для этого инструментария. Диаграмма сама по себе не показывает всей информации, необходимой для создания BPEL файла, т.к. тогда она будет слишком беспорядочной, что сделает ее нечитаемой. BPMN диаграмма предназначена для воспроизведения базовой структуры и потока действий и данных внутри бизнес-процесса. Таким образом, необходим инструмент моделирования, способный «добыть» остальную информацию, необходимую для создания исполняемого BPEL файла. Инструмент моделирования нуждается в определении некоторых базовых типов информации о самом процессе, чтобы заполнить в атрибуты BPEL процесса элементы BPEL документа.

На рисунке 10.3 показано, как основная информация о BPMN диаграмме и о процессе заказа путешествия внутри этой диаграммы будет нанесена диаграмму BPEL, чтобы настроить предварительную информацию для BPEL.

BPMN Object/Attribute	BPEL Element/Attribute
Business Process Diagram	See next row... mapped to attributes of a process element
ExpressionLanguage = "Java"	expressionLanguage="Java"
Business Process	The process element
Name = "Travel Booking Process"	name="travelBookingProcess"
ProcessType = "Private"	abstractProcess="no" or not included
SurpressJoinFailure = "Yes"	suppressJoinFailure="yes"

Рисунок 10.3 Определение базовых атрибутов бизнес-процесса

Инструмент, который создает BPEL файл, нуждается в опредлении параметров, таких как targetNamespace, определении места нахождения поддерживающего WSDL файла, и других параметров, включая переменные окружения, чтобы BPEL мог работать должным образом. Эта информация базируется на конфигурации переменных окружения для инструмента моделирования.

Элементы partnerLink (см. рис. 10.4) определяются до определения процесса (process), а информация об этих элементах будет находиться в Properties of the Tasks BPMN процесса. Tasks of the Process имеют тип Service, и выполняется как WEB-сервис. Определяются в Participant of the Web Service. Participants и их свойства наносятся на карту partnerLink элементов.

На рисунке 10.4 приведено два примера того, как Properties, определенные для выполнения WEB-сервисов для Task соответствуют атрибутам для элементов partnerLink, определенным в заголовке BPEL документа.

BPMN Object/Attribute	BPEL Element/Attribute
Implementation = "Web service"	Invoke, but some properties, below, will map to a partnerLink
Participant = "ProcessStarter"	name="ProcessStarter" partnerLinkType="ProcessStarterPLT"
BusinessRole = "TravelProcessRole"	myRole="TravelProcessRole"
Participant = "HotelReservationService"	name="HotelReservationService" partnerLinkType="HotelReservationServicePLT"
BusinessRole = "HotelReservationRole"	myRole="HotelReservationRole"

Рисунок 10.4 Отображение атрибутов в Web Service Properties для partnerLink

BPEL код для описанного partnerLink приведен в на рис.10.5

```
<partnerLinks>
   <partnerLink myRole="travelProcessRole" name="ProcessStarter" partnerLink-
Type="wsdl5:travelProcess"/>
   <partnerLink name="HotelReservationService" partnerLinkType="wsdl5:HotelReservationPartnerPLT"
                partnerRole="HotelReservationRole"/>
         <!-- Another 3 partnerLinks are defined -->
</partnerLinks>
```

Рисунок 10.5 Установка partnerLink элементов для процесса

Элемент variable также определяется до определения процесса (process). Свойства связываются с процессом внутри BPMN диаграммы. Инструмент моделирования позволяет проектировщику определить эти свойства. Свойства типа structure используются для того, чтобы сгруппировать набор Properties в пакет, который отображается в атрибутах BPEL элемента message. Элемент message на самом деле определяется WSDL документом, который поддерживает BPEL документ. Элемент variable определяется в BPEL документе и ссылается на элемент message.

На рисунке 10.6 демонстрирует два варианта того, как Properties, определенные для Process отображаются для элементов variable и message, которые определяются в заголовке BPEL документа и во вспомогательных WSDL документах.

BPMN Object/Attribute	BPEL Element/Attribute
Property	BPEL variable and WSDL message
Name = "input"	For BPEL variable: name="input" messageType="input" For WSDL message: name="input"
Type = "structure"	The sub-Properties of the structure will map to the WSDL message elements
Property	For WSDL message, in the part element:
Name = "airline"	name="airline "
Type = "string"	type="xsd:string"
Property	For WSDL message, in the part element:
Name = "arrival"	name="arrival"
Type = "string"	type="xsd:string"
Eleven more sub-Properties are included	Eleven more part elements are included
Ten more structure Properties are included	Ten more BPEL variable elements and WSDL message elements will be included

Рисунок 10.6 Отображение атрибутов для элементов Mapping Process Properties для BPEL variable и message

BPEL код для определения variable приведен на рисунке 10.7.

```
<variables>
    <variable messageType="wsdl0:input" name="input"/>
    <variable messageType="wsdl4:doCreditCardCheckingRequest" name="checkCreditCardRequest"/>
    <variable messageType="wsdl4:doCreditCardCheckingResponse" name="checkCreditCardResponse"/>
    <variable messageType="wsdl4:Exception" name="creditCardFault"/>
    <variable messageType="wsdl1:doCarReservationRequest" name="carReservationRequest"/>
        <!-- Another 6  variables are defined -->
</variables>
```

Рисунок 10.7 Установка variable Elements для process

WSDL код для определения message показан на рисунке 10.8.

```
<message name="input">
    <part name="airline" type="xsd:string"/>
    <part name="arrival" type="xsd:string"/>
    <part name="departure" type="xsd:string"/>
        <!-- Another 10 parts are defined -->
</message>
<message name="doFlightReservationRequest">
    <part name="airline" type="xsd:string"/>
    <part name="arrival" type="xsd:string"/>
    <part name="departure" type="xsd:string"/>
        <!-- Another four parts are defined -->
</message>
<!-- Another nine messages are defined -->
```

Рисунок 10.8 Установка элементов message для WSDL документа

Все стрелки (Sequence Flow) на рисунке 10.2, кроме четырех, наносятся на схему BPEL элементов link. Кроме этого процессу (process) понадобятся три элемента link, которых нет на рисунке 10.2. Это исключение будет объяснено ниже. Когда элемент flow для process определен, определение link предшествует определению шагов процесса. BPEL код для определения link показан на рисунке 10.9. Инструментом моделирования для каждого link автоматически генерится name.

```
<flow name="Flow" wpc:id="1"/>
        <links>
                <link name="link1"/>
                <link name="link2"/>
                <link name="link3"/>
                <link name="link4"/>
                <link name="link5"/>
                <link name="link6"/>
                <link name="link7"/>
                <link name="link8"/>
                <link name="link9"/>
                <link name="link10"/>
                <link name="link11"/>
                <link name="link12"/>
        </links>
        <!-- The Main Process will be place here -->
</flow>
```

Рисунок 10.9 Установка элементов link для flow

10.3 Старт процеса

Процесс стартует с получения сообщения о необходимости заказать путешествие, которое инициируется Message Start Event (Рисунок 10.10). После получения требования проверяется информация о кредитной карте. В случае ошибочного номера карты с Check Credit Card действие передается шагу (действию) Handle Fault (рисунок 10.2). Элементы (карта) для этого типа ошибок будет рассмотрена ниже в разделе «Управление ошибками».

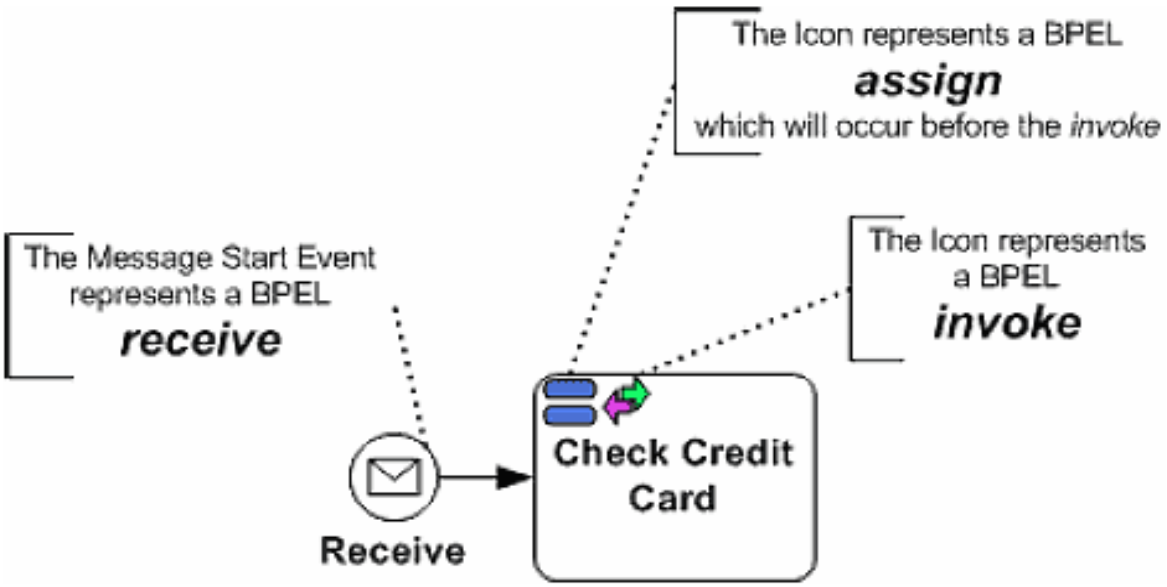


Рисунок 10.10 Начало процесса заказа путешествия

Receive Message Start Event — это механизм, который инициирует процесс при получении сообщения о заявке. Это наносится на карту для BPEL элемента receive.

На рисунке 10.11 показываются свойства Receive Message Start Event, и как эти свойства наносятся на карту для атрибутов элемента receive.

BPMN Object/Attribute	BPEL Element/Attribute
Start Event (EventType: Message)	receive
Name = "Receive"	name="Receive"
Instantiate = "True"	createInstance="yes"
Message = "input"	variable="input"
Implementation = "Web service"	See next three rows...
Participant = "ProcessStarter"	partnerLink="ProcessStarter"
Interface = "travelPort"	portType="wsdl0:travelPort"
Operation = "book"	operation="book"

Рисунок 10.11 Отображение атрибутов для Message Start Event

На рисунке 10.12 показывается результирующий BPEL код, который генерится для Receive Message Start Event.

```
<receive createInstance="yes" operation="book" name="Receive" wpc:displayName="Receive"
        portType="wsdl0:travelPort" variable="input" wpc:id="2">
    <source linkName="link1" />
</receive>
```

Рисунок 10.12 BPEL код для "Receive" start Event

Заметим, что на рисунке 10.10 на задании Check Credit Card в верхнем углу есть маленькие иконки. Они показывают, как нужно печатать задание. Подобные изображения будут и на последующих рисунках. Эти иконки не являются частью стандарта BPMN, но входят в расширенный BPMN. Ожидается, что инструменты для моделирования будут использовать подобные иконки как дополнительный инструментарий. Месторасположение иконок определяется проектировщиком либо инструментарием. Здесь они используются для того, чтобы показать, как диаграмма будет наноситься на карту BPEL.

Задание Check Credit Card следует за Start Event по стрелке (Sequence Flow). Стрелка показывает зависимую связь между элементами, нанесенными на карту от старта до задания. Зависимость примет форму BPEL элемента link ("link1" на рисунке 10.9 Примере 7). Элемент link связывается с шагом с включением элемента source, добавленного к элементу receive (на рисунке 10.12 Пример 9) и элемента target, добавленного к первому элементу, нанесенному на карту от Check Credit Card (шаг assign на рисунке 10.14 Примере 11 ниже).

Задание «Check Credit Card» наносится на карту BPEL элементу invoke. Но это задание, как видно на рисунке 10.9, имеет две иконки в верхнем углу. Первая иконка (две горизонтальных голубых полосы) показывает, что для основного задания необходимо нанести на карту данные. Некоторые данные, полученные при вводе процесса (Message start Event) наносятся на карту к структуре данных сообщения для Check Credit Card сервиса. Как видно на рисунке 10.1, что оригинальный процесс содержит раздельные шаги каждый раз, когда необходимо заносить данные на карту. Однако, многие методологии бизнес-процессов не рассматривают такое занесение данных как раздельные задания, что гарантировало бы им собственную форму и место на диаграмме. Такое занесение данных обычно включается как отдельная пре- или пост-шаговая функция для бизнес-задания, несмотря на то, что она может быть использована в некоторых случаях как автономное задание, как мы позже увидим в процессе. Как и для группы, такое занесение данных может быть определено, как свойство задания и занесено на карту BPEL элемента assign. Индивидуальные свойства занесения на карту комбинируются, чтобы стать набором элементов copy внутри элементов assign.

Вторая иконка (розовые и зеленые стрелочки) показывает, что основные функции задания будут сервисами, осуществленными через WEB-сервисы, которые наносятся на карту к BPEL элементу invoke.

На рисунке 10.13 Пример 10 показывает свойства задания «Check Credit Card», и как эти свойства наносятся на карту атрибутам assign и invoke.

BPMN Object/Attribute	BPEL Element/Attribute
Task (TaskType: Service)	invoke
Name = "Check Credit Card"	name="checkCreditCardRequest"
InMessage	inputVariable="checkCreditCardRequest"
OutMessage	outputVariable="checkCreditCardResponse"
Implementation = Web service	See next three rows...
Participant	partnerLink="creditCardCheckingService"
Interface	portType="wsdl4:creditCardCheckingServiceImpl"
Operation	operation="doCreditCardChecking"
Assignment	assign. The name attribute is automatically generated by the tool creating the BPEL document.
From = input.cardNumber	within a copy element, paired with the next row from part="cardNumber" variable="input"
To = checkCreditCardRequest. cardNumber	within a copy element, paired with the previous row to part="cardNumber" variable="checkCreditCardRequest"
AssignTime = Start	This means that the assign element will precede the invoke
From = input.cardType	within a copy element, paired the next row: from part="cardType" variable="input"
To = checkCreditCardRequest. cardType	within a copy element, paired the previous row to part="cardType" variable="checkCreditCardRequest"
AssignTime = Start	This means that the assign element will precede the invoke. All Assignments with an AssignTime of "Start" will be combined for one assign element.

Рисунок 10.13. Нанесение на карту задания «Check Credit Card»

На рисунке 10.14 Примере 11 показывается результирующий BPEL код, который генерируется для задания «Check Credit Card».

```
<assign name="DataMap1" wpc:displayName="DataMap1" wpc:id="20">
        <target linkName="link1"/>
        <source linkName="link2"/>
        <copy>
                <from part="cardNumber" variable="input"/>
                <to part="cardNumber" variable="checkCreditCardRequest"/>
        </copy>
        <copy>
                <from part="cardType" variable="input"/>
                <to part="cardType" variable="checkCreditCardRequest"/>
        </copy>
</assign>
<invoke inputVariable="checkCreditCardRequest" name="checkCreditCard" opera-
tion="doCreditCardChecking"
                outputVariable="checkCreditCardResponse" partnerLink="CreditCardCheckingService"
                portType="wsdl4:CreditCardCheckingServiceImpl" wpc:displayName="Check Credit Card"
wpc:id="5">
        <target linkName="link2"/>
        <source linkName="link3"/>
        <source linkName="link6"/>
        <source linkName="link9"/>
</invoke>
```

Рисунок 10.14. BPEL код для задания «Check Credit Card»

Существуют логические связи, в которых элемент assign должен предшествовать элементу invoke, как определено свойствами задания. Такие связи будут

результироваться в BPEL элементе link ("link2"). В этом случае нет соответствующих стрелок на BPMN диаграмме, как это было для link ("link1").

10.4 Создание параллельного потока

После задания Check Credit Card возможны три основных действия — резервирование машины, гостиницы и самолета (см. рисунок 10.15). Эти действия не зависят друг от друга, т.к. они могут выполняться одновременно. Резервирование машины наиболее сложное из них, поэтому будет рассмотрено в следующем разделе. В этом разделе затронуто только нанесение на карту данных, которые предшествуют заказу автомобиля.

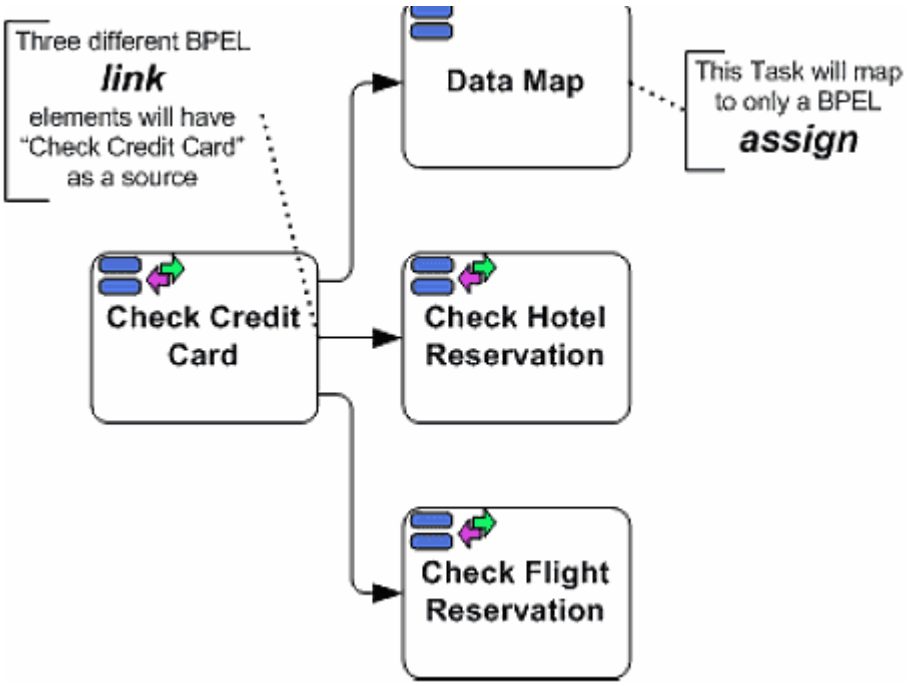


Рисунок 10.15 Распараллеливание действий внутри процесса

Распараллеливание обозначается тремя стрелками, выходящими из Check Credit Card. Три действия, на которые указывают стрелки, могут выполняться одновременно. Каждая из трез стрелок превращается в три BPEL элемента link ("link3", "link6" и "link9"). Эти три элемента link включаются в элемент source в Check Credit Card invoke элемента (см. рисунок 10.14, Пример 11). Они соответствуют target элементу в каждом из трех assign элементов, определенных ниже (см. Рисунки 10.16, 10.17, 10.19 Пример 12, Пример 13, Пример 15).

Начнем снизу рисунка 10.15. Нанесение на карту для Check Flight Reservation и его свойства такие же, как для Check Credit Card (рис. 10.13 Пример 10). В этом случае также результат заносится на карту элемента assign, который предшествует invoke элементу. Элемент link ("link4"), для которого нет связующей стрелки, должен быть добавлен для того, чтобы создать логическую зависимость между assign и invoke.

На рис. 10.16 примере 12 показан результирующий BPEL код, который генерируется для Check Flight Reservation.

```
<assign name="DataMap2" wpc:displayName="DataMap2" wpc:id="21">
        <target linkName="link3"/>
        <source linkName="link4"/>
        <copy>
                <from part="airline" variable="input"/>
                <to part="airline" variable="flightReservationRequest"/>
        </copy>
        <!-- Six additional copy elements are not shown -->
</assign>
<invoke inputVariable="flightReservationRequest" name="checkFlightReservation"
                operation="doFlightReservation" outputVariable="flightReservationResponse"
                partnerLink="FlightReservationService" portType="wsdl3:FlightReservationServiceImpl"

                wpc:displayName="Check Flight Reservation" wpc:id="10">
        <target linkName="link4"/>
        <source linkName="link5"/>
</invoke>
```

Рисунок 10.16. Пример 12. BPEL код для задания "Check Flight Reservation"

Задание Check Hotel Reservation очень похоже на Check Flight Reservation. В этом случае элемент assign также предшествует invoke элементу. И снова, элемент link ("link7") создается для того, чтобы создать зависимость между assign и invoke.

На рисунке 10.17 Пример 13 приводится результирующий BPEL код, который генерируется для Check Hotel Reservation.

```
<assign name="DataMap3" wpc:displayName="DataMap3" wpc:id="22">
        <target linkName="link6"/>
        <source linkName="link7"/>
        <copy>
                <from part="hotelCompany" variable="input"/>
                <to part="name" variable="hotelReservationRequest"/>
        </copy>
        <!-- Six additional copy elements are not shown -->
</assign>
<invoke inputVariable="hotelReservationRequest" name="checkHotelReservation"
                operation="doHotelReservation" outputVariable="hotelReservationResponse"
                partnerLink="HotelReservationService" portType="wsdl2:HotelReservationServiceImpl"
                        wpc:displayName="Check Hotel Reservation" wpc:id="9">
        <target linkName="link7"/>
        <source linkName="link8"/>
</invoke>
```

Рисунок 10.17. Пример 13. BPEL код для задания "Check Hotel Reservation"

Задание в верху рисунка 10.15. готовит данные для Check Car Reservation (см. рисунок 10.20). Другие задания на рисунке 10.15 имеют скрытую картографию данных, на что указывает иконка в верхнем углу фигуры. Это невозможно для задания резервирования автомобиля (Check Car Reservation), т.к. задание для проверки резервирования находится внутри петли. Нанесение данных на карту нужно один раз, в то время как проверка резервирования может производиться несколько раз. Поэтому нанесение данных на карту выделено в отдельное задание, которое кроме этого ничего больше не делает.

На рисунке 10.18 в примере 14 показаны свойства Data Map задания, и как эти свойства наносятся на карту атрибутов элемента assign.

BPMN Object/Attribute	BPEL Element/Attribute
Task (TaskType: None)	None, but assignment properties will create a mapping to an assign. Otherwise a BPEL empty element would have been created.
Name = "Data Map"	None.
Assignment	assign. The name attribute is automatically generated by the tool creating the BPEL document.
From = input.carCompany	within a copy element, paired the next row from part="carCompany " variable="input"
To = carReservationRequest. company	within a copy element, paired the previous row to part="company" variable="carReservationRequest"
AssignTime = Start	This doesn't have any direct effect since the Task is of type "None." All Assignments with an AssignTime of "Start" will be combined for one assign element.
There are four other From/To Assignments that are not shown	These will map to additional from and to elements within a copy.

Рисунок 10.18. Пример 14. Нанесение на карту "Data Map" задания

На рисунке 10.19 пример 15 приводится результирующий BPEL код, который генерируется для Data Map задания.

```
<assign name="DataMap4" wpc:displayName="Data Map" wpc:id="23">
        <target linkName="link9"/>
        <source linkName="link10"/>
        <copy>
                <from part="carCompany" variable="input"/>
                <to part="company" variable="carReservationRequest"/>
        </copy>
        <!-- Four additional copy elements are not shown -->
</assign>
```

Рисунок 10.19. Пример 15. BPEL код для Data Map задания

10.5 Нанесение на карту петли

Петля встречается в той части процесса, где проверяется заказ машины и оценивается результат действия (см. рисунок 10.20).

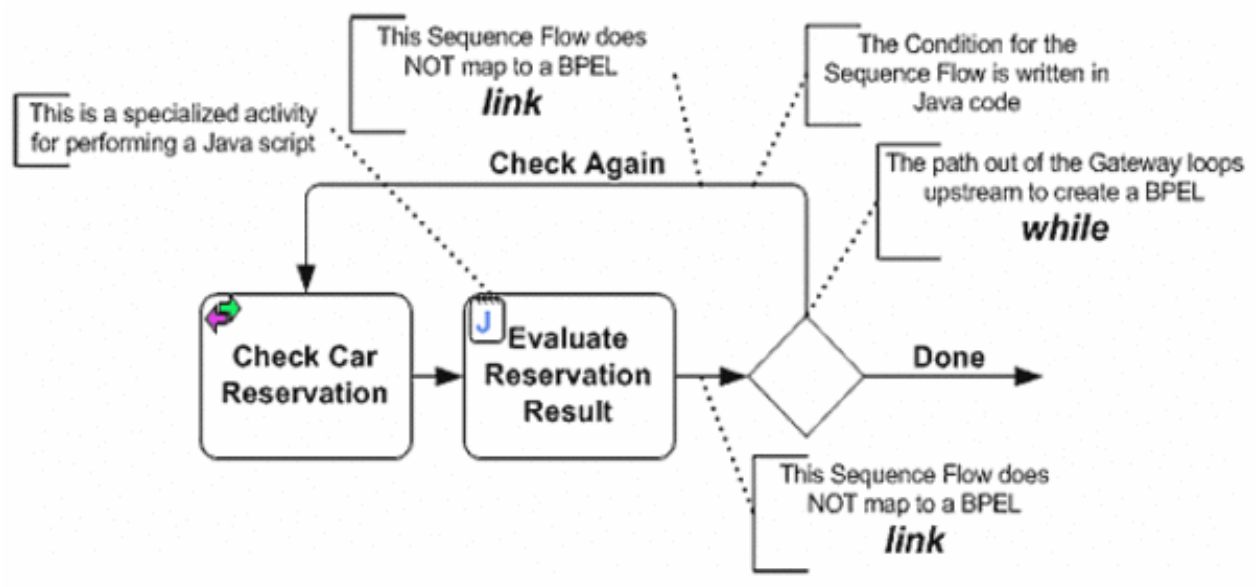


Рисунок 10.20. Петля внутри процесса

Два задания могут выполняться множество раз, пока не будет выполнено условие проверки. Петля состоит из решения Gateway, которое разделяет поток в зависимости от достигнутого результата. Стрелка Check Again выходит из Gateway и связывается с ранеестоящим объектом, образуя петлю.

Для случаев, когда Gateway не образует петлю — в нашем примере такая ситуация не рассматривается — нанесение на карту BPEL будет зависить от того, на какой структуре он базируется — на графической (flow) или блочной (sequence).

Для блочной структуры Gateway показыватся как switch. Каждая исходящая стрелочка наносится на карту case элемента внутри switch, и Expression для стрелочки будет наноситься на карту в condition для case.

Для нанесения на карту графической структуры, каждая выходящая из Gateway стрелка наносится на карту отдельным link элементом (входящие в Gateway стрелочки не наносятся на карту BPEL элементов). Condition для каждой стрелочки будут transitionCondition элемента source на шаге, который предшествует Gateway. В нашем случае это invoke от задания Evaluate Reservation Result.

Однако, т.к. петля создана стрелкой из Gateway, и из-за ациклической природы flow, элементы link не могут быть использованы в target элементе, который находится на верхнем шаге flow, это означает, что элемент while создан, чтобы управлять петлей. Содержимое while задает граничные условия для Gateway. Как видно на рисунке 10.20, задания Check Car Reservation и Evaluate Reservation Result находятся внутри петли, и наносятся на карту в содержимое BPEL while. Согласно решению для всего процесса, содержимое while будет нанесено на карту для графически структуированных элементов. Это означает, что главный элемент while будет flow, и нанесение на карту BPMN заданий будет соответствовать этому flow.

BPMN Check Again стрелка, которая связывает задание Check Car Reservation имеет разветвляющиеся условия (Condition). Эти Condition обычно заносятся на карту как атрибут condition для while. В нашем случае, однако, условия написаны на языке программирования Java, и расширение формы элемента condition используется для поддержки Java кода.

На рисунке 10.21 пример 16 показывает результирующий BPEL код, который генерируется для петли из Gateway.

```
<while condition="DefinedByJavaCode" name="While" wpc:id="11">
        <wpc:condition>
                <wpc:javaCode><![CDATA[
          boolean condition = false;
          try {
                if (getCarReservationResponse().getBooleanPart("result")) {
                      condition = false;
                } else {
                      condition = true;
                }
          } catch (Exception e) {
                e.printStackTrace();
          }
          return condition;
                                                                    ]]>
        </wpc:javaCode>
    </wpc:condition>
    <target linkName="link10"/>
    <source linkName="link11"/>
    <flow wpc:id="16">
        <links>
            <link name="link13"/>
        </links>
          <!-- Two invoke elements are place here, and shown below -->
    </flow>
</while>
```

Рисунок 10.21. Пример 16. BPEL код для петли из Gateway

Задание Check Car Reservation и его свойства заносятся на карту invoke элемента. На рисунке 10.22 пример 17 показывает результирующий BPEL код для задания Check Car Reservation.

```
<invoke inputVariable="carReservationRequest" name="checkCarReservation" operation="doCarReservation"
                outputVariable="carReservationResponse" partnerLink="CarReservationService"
                portType="wsdl1:CarReservationServiceImpl" wpc:displayName="Check Car Reservation"
wpc:id="13">
        <source linkName="link13"/>
</invoke>
```

Рисунок 10.22. Пример 17. BPEL код задания Check Car Reservation

Задание Evaluate Reservation Result отлично от предыдущих заданий процесса. Для BPMN это задание типа Script. Это означает, что когда процесс доходит до этого задания, сервис не будет вызываться, но движок, который исполняет процесс будет выполнять a-скрипт, который будет определен для этого задания. В нашем случае это скрипт, написанный на языке Java. Скрипт будет проверять результат трех резервирований (гостиница, самолет, машина), и определять, выполнены ли все три удачно или поездка не может быть заказана как планировалось.

Чтобы выполнить скрипт, BPMN действие invoke должно поддерживать Java код, который будет исполнять движок процесса. Расширение будет дополнением элемента script, который содержит элемент javaCode, который, в свою очередь, содержит код скрипта.

На рисунке 10.23 пример 18 показывает свойства задания Evaluate Reservation Result, и как эти свойства наносятся на карту атрибутов элемента invoke.

BPMN Object/Attribute	BPEL Element/Attribute
Task (TaskType: Script)	Invoke Since the TaskType is "Script" these attributes are automatically set: partnerLink="null" portType="wpc:null" operation="null" inputVariable is not used outputVariable is not used
Name = "Evaluate Reservation Result"	Name="evaluateReservationRequest"
Script = [Java Script]	The invoke element is extended to add the wpc:script element, which contains a wpc:javaCode element. The Java code is included within the wpc:javaCode element.

Рисунок 10.23. Пример 18. Нанесение на карту задания "Evaluate Reservation Result"

На рисунке 10.24 Пример 19 показывает результирующий BPEL код для задания Evaluate Reservation Result.

```
<invoke name="evaluateReservationRequest" operation="null" partnerLink="null" portType="wpc:null"
                wpc:displayName="Evaluate Reservation Request" wpc:id="14">
        <wpc:script>
                <wpc:javaCode><![CDATA[
                        <!-- Java Code Inserted here -->
                                                                ]]>
                </wpc:javaCode>
        </wpc:script>
        <target linkName="link13"/>
</invoke>
```

Рисунок 10.24. Пример 19. BPEL код для задания "Evaluate Reservation Result"

На рисунке 10.20 есть стрелка между Check Car Reservation и Evaluate Reservation Result. Эта стрелка нанесена на карту элемента link ("link13"), который будет называться в source элементе "carReservationRequest" — invoke, и target элементом в "evaluateReservationRequest" invoke. Стрелка между заданием Evaluate Reservation Result и Gateway обозначает конец петли, и, тем самым, конец while. Таким образом, элемент link не требуется.

10.6 Синхронизация параллельных потоков

Процесс имеет три параллельных пути, следующих за Check Credit Card. Эти три пути сходятся и синхронизируются перед заданием Confirmation, что представлено Parallel Gateway (см. рисунок 10.25). Это означает, что все три пути должны быть завершены в этой точке, прежде чем процесс будет продолжен.

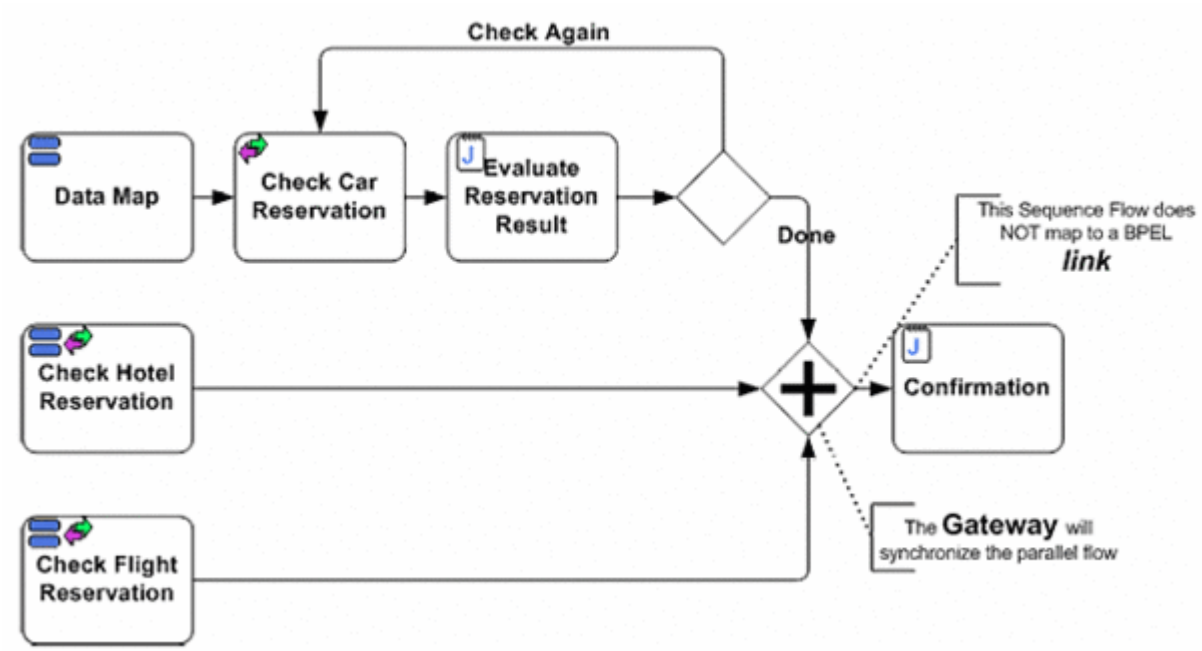


Рисунок 10.25. Синхронизация потока внутри процесса

Задания Confirmation выполняет скрипт, написанный на Java. Поэтому оно наносится на BPEL карту так же, как Evaluate Reservation Result и его свойства (Пример 18).

На рисунке 10.26 пример 20 показан BPEL код для Confirmation.

```
<invoke name="Confirmation" operation="null" partnerLink="null" portType="wpc:null"
        wpc:displayName="Confirmation" wpc:id="12">
    <wpc:script>
        <wpc:javaCode><![CDATA[
            <!-- Java Code Inserted here -->
                                                    ]]>
    </wpc:javaCode>
    </wpc:script>
    <target linkName="link9"/>
    <target linkName="link10"/>
    <target linkName="link11"/>
    <source linkName="link12"/>
</invoke>
```

Рисунок 10.26. Пример 20. BPEL код для Confirmation

Внутри BPEL кода для process синхронизация путей будет происходить в «Confirmation» invoke. Это не отдельный элемент синхронизации, такой как Parallel Gateway в BPMN. BPEL использует элемент link внутри flow, чтобы создать зависимости, включая синхронизацию, между действиями. Из-за того, что «Confirmation» invoke имеет три target элемента (для "link9","link10","link11" — см. рисунок 10.26 Пример 20), этот invoke должен ожидать, пока получит сигнал от всех трех link элементов, прежде чем сможет быть выполнен. Source элементы для этого элемента будут внутри «flightReservationRequest» invoke, «hotelReservationRequest» invoke и while действия соответственно.

Недостаток joinCondition для «Confirmation» invoke в том, что должен быть хотя бы один положительный сигнал, но фактически должны быть все три сигнала, положительные или отрицательные, означающие, что все три действия должны быть завершены, тем самым синхронизируя поток.

Заметим, что стрелка Done из Gateway, которая нанесена на карту "link12" элемента link, имеет условие, которое используется для разветвления от Gateway. Таким образом, source элемент с именем "link12" может иметь определение transitionCondition. В действительности в этом нет необходимости, т.к. link "link12" не будет инициироваться до тех пор, пока while, его source не завершится. Это означает, что для каждого transitionCondition для этого link всегда будет истина, когда он запускается. Если есть другая исходящая стрелка из Gateway, тогда Condition для стрелки влияли бы на прохождение процесса.

10.7 Конец потока

После задания Confirmation процесс заканчивается отсылкой ответа инициатору процесса (см. рисунок 10.27). Сообщение отсылается в Message End Event.

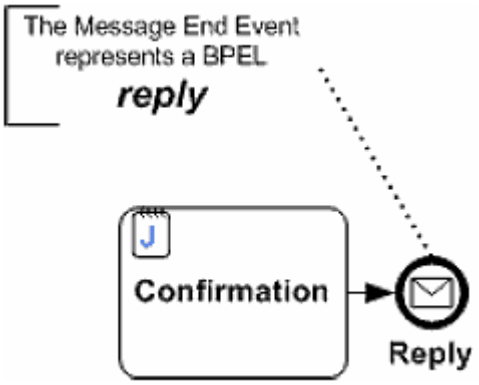


Рисунок 10.27. Завершение процесса

Message End Event наносится на карту Reply элемента. К тому же, стрелка от Confirmation до Reply End Event наносится на карту link элементу "link12" с присвоением имен source и target элементам link соответствующих BPEL действий.

На рисунке 10.28 пример 21 показывает свойства Reply Message End Event, и как эти свойства наносятся на карту атрибутам элемента reply.

BPMN Object/Attribute	BPEL Element/Attribute
End Event (EventType: Message)	reply
Name = "Reply"	name="Reply"
Message = "output"	variable="output"
Implementation = "Web service"	See next three rows…
Participant = "ProcessStarter"	partnerLink="ProcessStarter"
Interface = "travelPort"	portType="wsdl0:travelPort"
Operation = "book"	operation="book"

Рисунок 10.28. Пример 21. Нанесение на карту "Reply" End Event

На рисунке 10.29 пример 22 показывается результирующий BPEL код, который генерируется для Reply Message End Event.

```
<reply name="Reply" operation="book" wpc:displayName="Reply" partnerLink="ProcessStarter"
        portType="wsdl0:travelPort" variable="output" wpc:id="3">
    <target linkName="Link12"/>
</reply>
```

Рисунок 10.29. Пример 22. BPEL код для "Reply" Message End Event

10.8 Обработка ошибок

Как видно на рисунке 10.30, задание Check Credit Card имеет дополнительное событие ошибки, прикрепленное сбоку. Это событие реагирует на специфические ошибки, прерывания задания, и направляет поток по соответствующей стрелке. Ошибка инициируется, если введен неправильный номер кредитной карты.

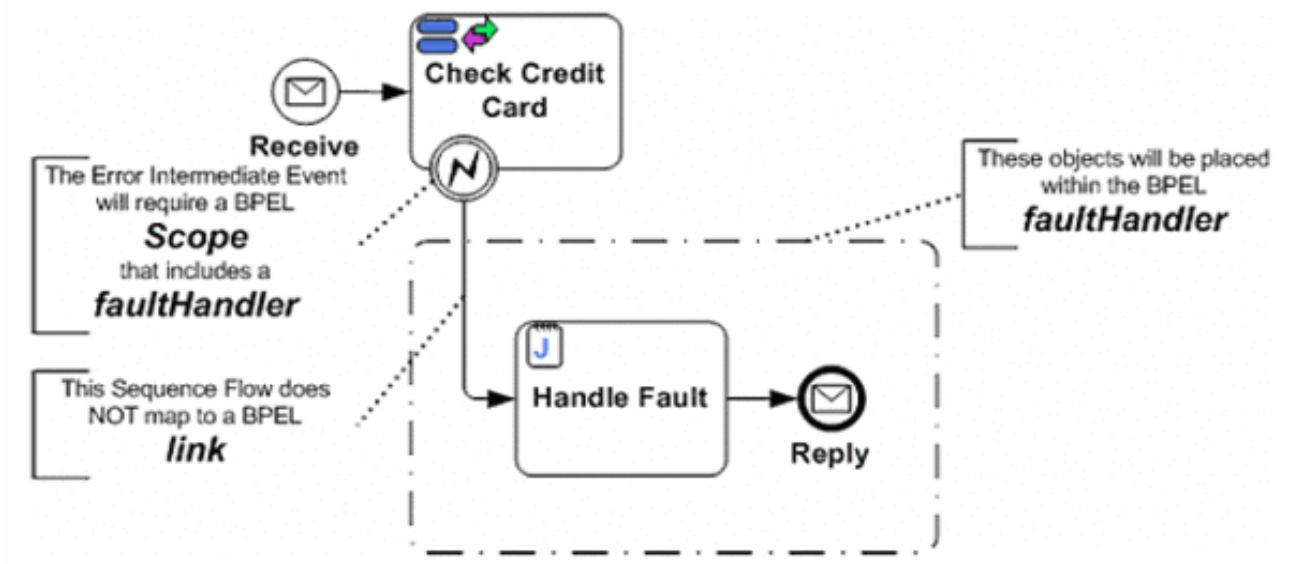


Рисунок 10.30. Обработка ошибки в процессе

При возникновении ошибки задание Handle Fault создаст сообщение об ошибке, которое будет отослано инициатору процесса. Reply Message End Event отсылает это сообщение. Это означает, что процесс будет завершен и все остальные действия не понадобятся. Таким образом, дополнительный Event ведет к End Event, прерывая процесс. BPEL механизм для прерванного процесса — это элемент faultHandlers внутри scope. Этот scope является оболочкой всего содержимого процесса, что означает, что основной flow процесса (см. пример 10.7.) фактически содержится внутри scope-along с faultHandlers. Flow запускается внутри scope, если только он не прерывается faultHandlers.

Содержимое faultHandlers запускается в том случае, если он будет активирован. Это означает, что задание Handle Fault и Reply Message End Event на карте будут находится внутри faultHandlers. Занание Handle Fault — это скрипт, поэтому в BPEL он выглядит так же, как Evaluate Reservation Result (см. рисунок 10.23 пример 18). Карта Reply End Event выглядит также, как и карта Reply End Event для основного процесса (см. рисунок 10.29 пример 22).

Согласно тому, как основной процесс и петля описываются в BPEL, секция обработки ошибок будет размещена внутри flow (внутри faultHandlers). Стрелка от Error Intermediate Event к заданию Handle Fault не будет отображаться link элементом, т.к. задание Handle Fault при нанесении на карту является первым действием внутри flow faultHandlers. А стрелка от Handle Fault к Reply End Event

отображается link ("link14") с source и target наименованиями link-ов в соответствующих BPEL действиях.

На рисуноке 10.31 пример 23 показывается соответствующий BPEL код, который создается для Process Fault Handling, включая код для Handle Fault и Reply End Event.

```
<scope wpc:id="15">
   <faultHandlers>
        <catch faultName="wsdl4:Exception" faultVariable="creditCardFault">
                <flow wpc:id="17">
                        <links>
                                <link name="Link14"/>
                        </links>
                        <invoke name="HandleFault" operation="null" wpc:displayName="handleFault"
                                                partnerLink="null" portType="wpc:null" wpc:id="18">
                                <wpc:script>
                                        <wpc:javaCode><![CDATA[
                                                <!-- Java Code Inserted here -->
                                                                ]]>
                                                </wpc:javaCode>
                                </wpc:script>
                                <source linkName="Link14"/>
                        </invoke>
                        <reply name="Reply" operation="book" wpc:displayName="Reply" partner
                           Link="ProcessStarter" portType="wsdl0:travelPort" variable="output" wpc:id="19">
                              <target linkName="Link14"/>
                        </reply>
                </flow>
        </catch>
   </faultHandlers>
   <flow wpc:id="15">
        <!-- The main process as shown above is placed here -->
   </flow>
</scope>
```

Рисунок 10.31. Пример 23. Нанесение на карту Fault Handling для Process

10.9 Заключение

В данном разделе рассмотрен пример того, как BPMN диаграмма бизнес-процесса может быть использована для изображения выполняемого процесса. Для того, чтобы создать исполняемый процесс, объекты диаграммы и их свойства рассекаются и затем наносятся на карту в соответствующие BPEL элементы. Несмотря на то, что весь этот материал не имел целью охватить все аспекты нанесения на карту BPMN диаграмм для BPEL, он дает пошаговую иллюстрацию для одного конкретного примера — заказа путешествия. Таким образом, пример показывает, как BPMN диаграмма может иметь двойное назначение, обеспечивая

бизнес-представление процесса и позволяя создать исполняемый процессный код для BPEL

11. Язык WS BPEL

11.1 Введение

Прежде, чем приводить описание языка нужно понять, что такое BPEL, важно отметить, что **BPEL полностью основан на Web-сервисах**. Как было сказано в предыдущих главах, Web-сервисы обеспечивают открытый и гибкий способ общения по сети, этот способ основан на XML. Подробно о Web-сервисах см. глава 8.

Основной задачей BPEL (Business Process Execution Language) является предложение открытого стандарта для гибкой связи нескольких модулей, систем, компонет в распределенной и неоднородной среде в новые компоненты для реализации различных сложных бизнес процессов. Взаимодействие BPEL на основе Web-сервисов дает возможность объединять системы, находящиеся в любой точке мира.

В данном учебном пособии будет использоваться стандарт версии WS BPEL 1.1 и стандарт версии WS BPEL 2.0 (см. ниже).

11.2 История BPEL

Язык исполнение бизнес-процессов для Web-сервисов (BPEL4WS) впервые был задуман в июле 2002 года с выпуском BPEL4WS 1.0, совместными усилиями IBM, Microsoft, и BEA. В этом документе предложена оркестровка на основе Web-службам IBM, языка потока (WSFL) и спецификации Microsoft XLANG.

В результате поддержки SAP и Siebel Systems и другими участниками, версия 1.1 спецификации BPEL4WS была выпущена менее чем через год, в мае 2003 года. Эта версия получила больше внимания и поддержки поставщиков, что привело к ряду имеющихся в продаже BPEL4WS-совместимых двигателей оркестровки. Незадолго до этой версии спецификациия BPEL4WS была представлена техническому комитету OASIS, что позволяет использовать ее в качестве официального, открытого стандарта. В апреле 2007 года, WS-BPEL 2.0 была утверждена в качестве стандарта OASIS. Более 37 организаций сотрудничают в разработке WS-BPEL, включая такие как, Adobe Systems, BEA Systems, Booz Allen Hamilton, EDS, HP, Hitachi, IBM, Ионы, Microsoft, NEC, Nortel, Oracle, Red Hat, Rogue Wave , SAP, Sun Microsystems, TIBCO, WebMethods, и другие члены OASIS. В 2008 года OASIS выпустили BPEL4People, который определяет WS-BPEL

расширение по взаимодействию человека («human tasks») в рамках WS-BPEL процесса.

Самым большым конкурент BPEL являся BPML и BPEL4WS. Стандарт BPML предлагает все, что предлагает BPEL и даже больше. Просто по истории получилось, что релиз BPML 1.0 состоялся месяцем позже релиза версии BPEL 1.0, также стандарт BPEL был поддержан большими корпорациями, такими как Microsoft, IBM и SAP. Ранее был разработан Business Process Execution Language for Web Services (BPEL4WS), технический комитет OASIS по разработке языка BPEL анононсировал новую версию WS BPEL 2.0, в которой были обеденены свойства WS BPEL и BPEL4WS.

11.3 Пример практического применения BPEL

Для того чтобы продемонстрировать использование BPEL, приведем короткий пример.

Компания по производству напитков хочет автоматизировать установку и выполнить модернизацию программного обеспечения, основанного на решениях, принятых в Системе планирования ресурсов предприятия (некоторой ERP - системы). Назовем вымышленную компанию S_Water

Компания S_Water использует Систему планирования ресурсов (Electronic Resource Planning). Данная система отслеживает поставку новых машин по производству напитков, также все изменения происходящие с оборудованием: поломка, выход из эксплуатации. В настоящее время, когда новая машина произведена, служащий должен установить программное обеспечение на машине. Важно то, что программное обеспечение различно для различных типов машин. Например, машина, которая будет использоваться в Америке, должна будет использовать доллары, в то время как машина для Франции должна принимать евро. Предположим, что программное обеспечение и оборудование установлено правильно и работает в соответствии с поставленной целью.

Теперь S_Water собирается модернизировать линию изготовления, путем установки связи между машинами. Когда машина изготовлена, она должна связаться по локальной сети по запросу на получение надлежащего программного обеспечения, основанного на всех переменных системы распределения ресурсов. Для установки программного обеспечения компания S_Water купила сервер, который может выполнять поставленную задачу.

Теперь проблема возникает в том, чтобы автоматизировать установку программного обеспечения, система ERP должна связываться с обеспечивающим сервером. Чтобы соединить эти две стороны, мы поместим оркестровый сервер BPEL между ними. Теперь есть одно важное требование: любому ERP и обеспечивающему серверу надо поддерживать Web-сервисы. Система ERP следует изменить так, чтобы, когда устройство установлено и готово к монтажу программного обеспечения, тогда должно произойти обращение к серверу BPEL.

В этом обращении будет содержаться информация о машине, которая готова к установке обеспечения, также ограничивающие настройки ERP. Существует вероятность, что в сообщении будет содержаться информация и другово рода, но в примере откажемся от иных настроек

Все запросы, которые отравляются от ERP на обеспечивающий сервер, должны быть сначало направлены на оркестровый сервер BPEL. Даже если текущий запрос не вызвал никаких действий на обеспечающем сервере, он сохраняется на сервере, чтобы в будущем не надо было запрашивать разрешение у ERP.

Обеспечивающий сервер должен выпополнять все поступающие запросы, которые описаны в виде Web-сервисов, таким образом, оркестровый сервер BPEL может вызывать любое действие на обеспечивающем сервере. На рисунке 11.1 описывается взаимодействие элементов системы.

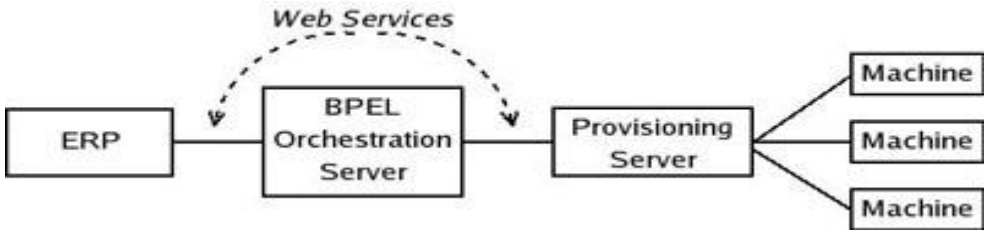


Рисунок 11.1

Возникает вопрос «Зачем использовать оркестровый сервер BPEL, если можно работать напрямую, соединяя систему ERP непосредственно с обеспечивающим сервером?» Ответ очень прост даже при незначительных изменениях обеспечивающего сервера, он запрашивает новые параметры настройки, что вынуждает систему ERP также приспосабливаться к новым изменениям. Это вероятно означало бы, что система ERP должна быть повторно скомпилирована и установлена, что являетмя неприемлимой ситуацией.

Ниже на рис. 11.2. показан очень простой пример добавления нового устройства. После того, как ERP вызывает оркестровый сервер BPEL, процесс устанавливает, какой язык использует ERP. Допустим язык английский, тогда программное обеспечение устанавливается на новом устройстве путем посылки запроса на обеспечивающий сервер. Если язык французский, то устанавливатся другое программное обеспечение. Когда установка завершилась, результат проделанной работы (статус) возвращается на ERP.

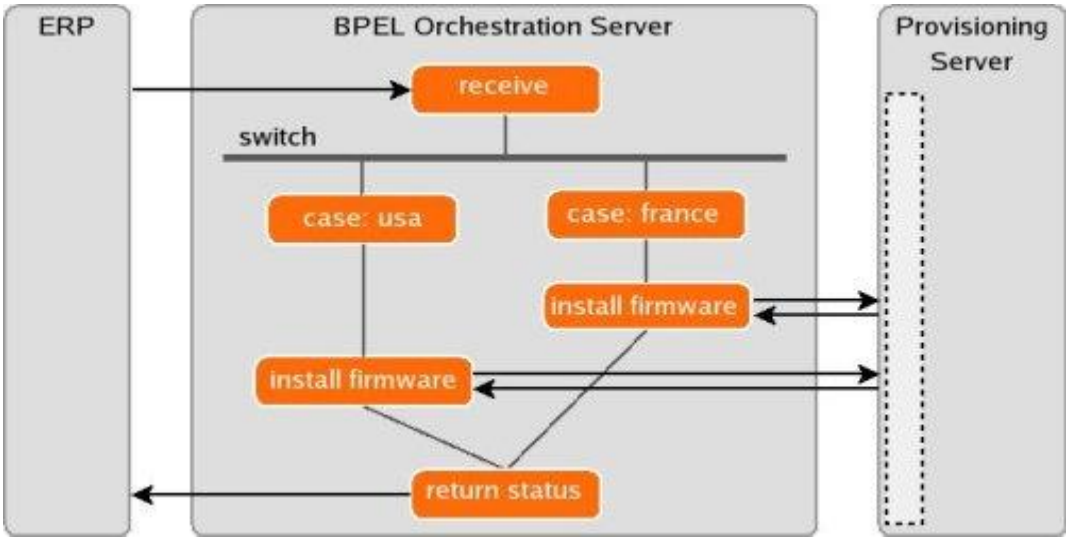


Рисунок 11.2

Конечно же, в примере показана только суть, весь процесс весьма упрощен, но зато можно подчеркнуть, что BPEL процесс работает в зависимости от поступающих переменных. В реальном мире процесс установки нового оборудования выглядел бы иначе, запрос содержал бы множество переменных поступающих на оркестровый сервер описывающих не только информацию о языке использования, но и информацию такого рода как цена продукта, наименование товара, также тара выпуска напитков: бутылки или банки.

Другая важная особенность BPEL процесса заключается в том, что он может проверять корректность установки оборудования. Проверка осуществляется путем отправки запроса на обеспечивающий сервер (отправка запроса осуществляется каждые 30 минут), в который ведет проверять статус установленного оборудования. Если установка оборудования завершилась успешно, то результат «Завершено успешно» отправляется на ERP. Если инсталляция не завершилась в течение 12 часов, то на ERP отправляется сигнал о том, что механик должен разобраться с поломкой.

Конечно же, использования BPEL сервера не ограничивается только проверкой правильности инсталляции оборудования. Можно привести другой пример. Допустим, цены на напиток изменились, тогда механику нодо обойти все автоматы и поменять там цены, но ведь такой подход очень долгий и весьма дорогостоящий. Рассмотрим, как упрощается процесс при использовании BPEL сервера. Все устройства, которые включены в единую сеть, могут быть очень легко менят стоимость на продукцию. ERP отсылает новый прайс на сервер BPEL, он в свою очередь отправляет запрос на обеспечивающий сервер. Обеспечивающий сервер сразу же соединяется со всеми устройствами и отправляет им новый прайс.

Как видно из примера, с помощью данного подхода можно решать различные задачи. Так же подчеркнем, что в BPEL процесс имеет еще некоторые полезные свойства: отслеживание ошибок и откатка процесса. Например, на устройство

устанавливается программное обеспечение и в конце установки имела место ошибка, откатка позволяет вернуть устройство в исходное состояние.

11.4 Нотация BPEL

Соглашение об обозначениях

Данная спецификация BPEL использует неформальный сисинтаксис для описания гамматики языка в XML подобном стиле. Основные моменты:

- синтаксис выглядит как XML, но переменные содержат типы данных, а не значение переменных;
- < -- description -- > место для заполнения другими элементами (например как ## в XSD);
- элементы и атрибуты, разделенные «|» или группы «(» «)» соединенные «and» означают синтаксическую альтернативу;
- примеры, которые начинаются с « <?xml » содержат достаточно информации, чтобы соответствовать данной спецификации, другие примеры теребуют некоторого дополнения, чтобы подходить под спецификацию;
- символы добавления, присоединения элементов, атрибутов и следования: "?" (0 или 1), "*" (0 или больше), "+" (1 или более). Символы "[" и "]" используются для обозначения, содержащие элементы должны рассматриваться как группа относительно "?" "*", или "+";
- элементы и атрибуты, разделенные "|" и сгруппированы по "(" и ")", предназначены для синтаксической альтернативы;
- XML-префиксы пространств имен (см. ниже) используется для указания области действия определенных элементов;
- Имя определенное пользователем указывается как anyElementQName.

Синтаксис спецификации будет выделен следующим образом:

```
<variables>
    <variable name="BPELVariableName"
              messageType="QName"?
              type="QName"?
              element="QName"?>+
        from-spec?
    </variable>
</variables>
```

Спецификация, приведенная ниже, использует ряд префиксов пространства имен, связанных с ними URI, которые перечислены ниже. Заметим, что выбор любого префикса пространства имен является произвольным, ненормативным и семантически не значительным, см. пример ниже.

xsi - "http://www.w3.org/2001/XMLSchema-instance"
xsd - "http://www.w3.org/2001/XMLSchema"
wsdl - "http://schemas.xmlsoap.org/wsdl/"
vprop - "http://docs.oasis-open.org/wsbpel/2.0/varprop"
sref - "http://docs.oasis-open.org/wsbpel/2.0/serviceref"
plnk – "http://docs.oasis-open.org/wsbpel/2.0/plnktype"
bpel – "http://docs.oasis-open.org/wsbpel/2.0/process/executable"
abstract – "http://docs.oasis-open.org/wsbpel/2.0/process/abstract"

11.5 Взаимоотношения с WSDL

WS BPEL зависит от спецификаций, которые основаны на XML: WSDL 1.1, XML Schema 1.0, XPath 1.0 and WS-Addressing.

Нужно учитывать что, WSDL имеет самое большое влияние на язык WS BPEL. Модель процесса WS BPEL лежит на самом высоком уровне сервисной модели, описанной в WSDL. Основой процессной модели WS BPEL является нотация взаимоотношений между сервисами, описанная в WSDL, таким образом, процессы и партнеры смоделированы как сервисы в WSDL. Бизнес процесс определяет, как должны быть скоординированы взаимоотношения между объектами процесса и его партнерами. Таким образом, WS BPEL процесс обеспечивает и/или использует один или несколько WSDL сервисов, также обеспечивает описание поведения взаимоотношения между объектами процесса, партнерами и ресурсами через интерфейсы Web-сервисов. Таким образом, WS BPEL определяет протокол обмена сообщениями в бизнес процессе.

Описание процессов WS BPEL также следует из модели WSDL, разделяется абстрактная информация и данные, доступные для прочтения и использования. WS BPEL процесс описывает всех партнеров и взаимоотношения партнеров с помощью абстрактных интерфейсов WSDL.

Однако абстрактная часть WSDL не определяет ограничения, наложенные на шаблоны коммуникации, поддержанные конкретными связываниями. Поэтому процесс WS BPEL может определить поведение относительно сервиса партнера, который не поддержан всеми возможными связываниями, и может случиться, что некоторые связывания недопустимы для определения процесса WS BPEL.

Процесс WS BPEL - является определением многократного использования, которое может быть использовано по-разному и в различных сценариях, однако поддерживает однородное поведение на уровне приложения.

11.6 Определение бизнес процесса

Перед тем как приступить к доскональному разбору синтаксиса BPEL, рассмотрим простой пример закупки товара. Цель данного примера определить наиболее простые обороты языка, начать возводить фундамент WSDL.

Одна операция процесса очень проста, она представлена на рисунке 11.3 Разрывная линия определяет последовательность. Свободно сгруппированные последовательности определяют параллельные последовательности. Сплошные линии определяют связь между параллельными действиями. Обратим внимание, что описанный пример не является законченной графической нотацией WS BPEL, этот пример приведен для простого понимания.

При получении заказа от заказчика, процесс генерирует три параллельные задачи: подсчет конечной стоимости продукции, выбор перевозчика, а также составление плана перевозки заказа, согласование его с перевозчиком и складом. Обратим внимание, что все задачи являются не только параллельными, они еще и связаны между собой текущими данными: стоимость услуг перевозки должны быть включены в конечную стоимость заказа, также трудоемкой задачей является составление общего расписания. В тот момент, когда все проблемы решаются и выясяется конечная стоимость заказа, выписывается накладная, которая отправляется заказчику.

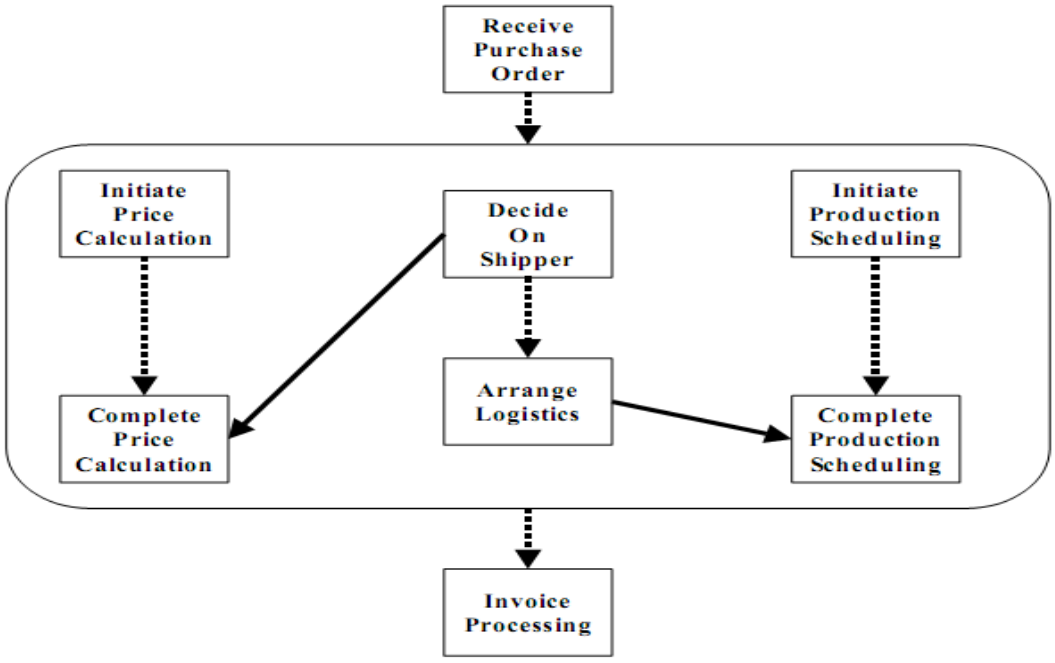


Рисунок 11.3

Опишем документ WSDL, а именно <portType>(напомним <portType> - это элемент, который описывает операции, которые будут поддерживаться Web-сервисом). Операцию заказа товара заказчиком назовем purchaseOrderPT. Обратим внимание, что текущий документ не содержит каких-либо связывающих элементов или сервисных элементов. То есть описываемый WS BPEL процесс является в данном случае абстрактным, содержащим лишь описание операций, но без выполнения. Здесь используется такой метод, для того чтобы данный бизнес процесс можно было бы использовать вторично.

Типы <partner link>, описанные в самом низу документа, показывают взаимоотношение между сервисом «заказ товара» и другими сервисами. Типы <partner link> описываеют зависимость между сервисами. Каждый < partner link> может описывать до двух ролей, также к каждой роли список <portType> (возможный операций) для успешного выполнения связи. В текущем примере имеется два типа <partner link> «purchasingLT» и «schedulingLT». «purchasingLT» определяет взаимосвязь между запросом заказчика и продукцией, назначенной на реализацию. «schedulingLT» определяет взаимосвязь между сервисами «заказ товара» и «составление расписания». «invoicingLT» и «shippingLT» определяют две роли, так как используют сервисы «заказ товара» и «составление расписания», см. листинг на рисунке 11.4.

```
<wsdl:definitions
   targetNamespace="http://manufacturing.org/wsdl/purchase"
   xmlns:sns="http://manufacturing.org/xsd/purchase"
   xmlns:pos="http://manufacturing.org/wsdl/purchase"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:types>
      <xsd:schema>
         <xsd:import namespace="http://manufacturing.org/xsd/purchase"
         schemaLocation="http://manufacturing.org/xsd/purchase.xsd" />
      </xsd:schema>
   </wsdl:types>

   <wsdl:message name="POMessage">
      <wsdl:part name="customerInfo" type="sns:customerInfoType" />
      <wsdl:part name="purchaseOrder" type="sns:purchaseOrderType" />
   </wsdl:message>
   <wsdl:message name="InvMessage">
      <wsdl:part name="IVC" type="sns:InvoiceType" />
   </wsdl:message>
   <wsdl:message name="orderFaultType">
      <wsdl:part name="problemInfo" element=”sns:OrderFault " />
   </wsdl:message>
    <wsdl:message name="shippingRequestMessage">
      <wsdl:part name="customerInfo" element="sns:customerInfo" />
   </wsdl:message>
   <wsdl:message name="shippingInfoMessage">
      <wsdl:part name="shippingInfo" element="sns:shippingInfo" />
   </wsdl:message>
   <wsdl:message name="scheduleMessage">
      <wsdl:part name="schedule" element="sns:scheduleInfo" />
```

```
    </wsdl:message>

<!-- portTypes supported by the purchase order process -->
    <wsdl:portType name="purchaseOrderPT">
      <wsdl:operation name="sendPurchaseOrder">
        <wsdl:input message="pos:POMessage" />
        <wsdl:output message="pos:InvMessage" />
        <wsdl:fault name="cannotCompleteOrder"
        message="pos:orderFaultType" />
      </wsdl:operation>
    </wsdl:portType>

    <wsdl:portType name="invoiceCallbackPT">
      <wsdl:operation name="sendInvoice">
        <wsdl:input message="pos:InvMessage" />
      </wsdl:operation>
    </wsdl:portType>

    <wsdl:portType name="shippingCallbackPT">
      <wsdl:operation name="sendSchedule">
        <wsdl:input message="pos:scheduleMessage" />
      </wsdl:operation>
    </wsdl:portType>

<!-- portType supported by the invoice services -->
<wsdl:portType name="computePricePT">
<wsdl:operation name="initiatePriceCalculation">
<wsdl:input message="pos:POMessage" />
</wsdl:operation>
<wsdl:operation name="sendShippingPrice">
<wsdl:input message="pos:shippingInfoMessage" />
</wsdl:operation>
</wsdl:portType>

<!-- portType supported by the shipping service -->
<wsdl:portType name="shippingPT">
<wsdl:operation name="requestShipping">
<wsdl:input message="pos:shippingRequestMessage" />
<wsdl:output message="pos:shippingInfoMessage" />
<wsdl:fault name="cannotCompleteOrder"
message="pos:orderFaultType" />
</wsdl:operation>
</wsdl:portType>

<!-- portType supported by the production scheduling process -->
```

```
    <wsdl:portType name="schedulingPT">
      <wsdl:operation name="requestProductionScheduling">
        <wsdl:input message="pos:POMessage" />
      </wsdl:operation>
      <wsdl:operation name="sendShippingSchedule">
        <wsdl:input message="pos:scheduleMessage" />
      </wsdl:operation>
    </wsdl:portType>

    <plnk:partnerLinkType name="purchasingLT">
      <plnk:role name="purchaseService"
        portType="pos:purchaseOrderPT" />
    </plnk:partnerLinkType>

    <plnk:partnerLinkType name="invoicingLT">
      <plnk:role name="invoiceService"
        portType="pos:computePricePT" />
      <plnk:role name="invoiceRequester"
        portType="pos:invoiceCallbackPT" />
    </plnk:partnerLinkType>

    <plnk:partnerLinkType name="shippingLT">
      <plnk:role name="shippingService"
        portType="pos:shippingPT" />
      <plnk:role name="shippingRequester"
        portType="pos:shippingCallbackPT" />
    </plnk:partnerLinkType>

    <plnk:partnerLinkType name="schedulingLT">
      <plnk:role name="schedulingService"
        portType="pos:schedulingPT" />
    </plnk:partnerLinkType>

</wsdl:definitions>
```

Рисунок 11.4 Листинг описание документа WSDL

Опишем четыре самые важные секции описания процесса:

- **<Variables>** эта секция предназначена для описания переменных, используемых процессом. Типы переменных определяются в терминах типов сообщений WSDL, типах схемы XML.

- **<PartnerLinks>** - этот секция определяет различных участников, которые взаимодействуют с бизнес-процессом в ходе обработки заказа. Четыре <PartnerLinks>, которые приведены в указанном листинге, соответствуют отправителю заказа (клиент), поставщику товара, который указывает цену продукции (invoicingProvider), пограузчику (shippingProvider) и планировщику расписания (schedulingProvider). Все <PartnerLinks> характеризуются типом и ролью. Эта информация указывает функциональность, которая должна быть выполнена совместно с бизнес-процессом и сервисами.

- **<FaultHandlers>** - этот раздел содержит обработчики ошибок. Обработчик ошибок определяет действия, которые следуют в ответ на возникающие ошибки. Все ошибки в WS BPEL внутренние, или следующие из вызова сервиса, определяются составным именем. В частности каждая ошибка WSDL идентифицирована в WS BPEL составным именем, которое состоит из названия пространства имен документа WSDL, в котором определен соответвующий portType и ошибка, также пространство имен ошибки. Важно отметить, что WSDL не требует уникального наименования ошибки в пределах пространства имен, где имела место некорректная операция. Несмотря на такие серьезные ограничение WSDL, WS BPEL обеспечивает однородную модель наименования ошибок.

- В остальной части определения процесса содержится описание нормального поведения при обработки запроса покупки товара см. листинг на рисунке 11.5.

```
<process name="purchaseOrderProcess"
   targetNamespace="http://example.com/ws-bp/purchase"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:lns="http://manufacturing.org/wsdl/purchase">

   <documentation xml:lang="EN">
      A simple example of a WS-BPEL process for handling a purchase
      order.
   </documentation>

   <partnerLinks>
      <partnerLink name="purchasing"
         partnerLinkType="lns:purchasingLT" myRole="purchaseService" />
      <partnerLink name="invoicing" partnerLinkType="lns:invoicingLT"
         myRole="invoiceRequester" partnerRole="invoiceService" />
      <partnerLink name="shipping" partnerLinkType="lns:shippingLT"
         myRole="shippingRequester" partnerRole="shippingService" />
      <partnerLink name="scheduling"
         partnerLinkType="lns:schedulingLT"
         partnerRole="schedulingService" />
   </partnerLinks>

   <variables>
      <variable name="PO" messageType="lns:POMessage" />
      <variable name="Invoice" messageType="lns:InvMessage" />
      <variable name="shippingRequest"
         messageType="lns:shippingRequestMessage" />
      <variable name="shippingInfo"
         messageType="lns:shippingInfoMessage" />
      <variable name="shippingSchedule"
         messageType="lns:scheduleMessage" />
   </variables>

   <faultHandlers>
      <catch faultName="lns:cannotCompleteOrder"
            faultVariable="POFault"
            faultMessageType="lns:orderFaultType">
         <reply partnerLink="purchasing"
            portType="lns:purchaseOrderPT"
            operation="sendPurchaseOrder" variable="POFault"
            faultName="cannotCompleteOrder" />
      </catch>
   </faultHandlers>

   <sequence>
```

```
<receive partnerLink="purchasing" portType="lns:purchaseOrderPT"
   operation="sendPurchaseOrder" variable="PO"
   createInstance="yes">
   <documentation>Receive Purchase Order</documentation>
</receive>

<flow>
   <documentation>
      A parallel flow to handle shipping, invoicing and
      scheduling
</documentation>

<links>
   <link name="ship-to-invoice" />
   <link name="ship-to-scheduling" />
</links>

<sequence>
   <assign>
      <copy>
         <from>$PO.customerInfo</from>
         <to>$shippingRequest.customerInfo</to>
      </copy>
   </assign>

   <invoke partnerLink="shipping" portType="lns:shippingPT"
      operation="requestShipping"
      inputVariable="shippingRequest"
      outputVariable="shippingInfo">
      <documentation>Decide On Shipper</documentation>
      <sources>
         <source linkName="ship-to-invoice" />
      </sources>
   </invoke>

   <receive partnerLink="shipping"
      portType="lns:shippingCallbackPT"
      operation="sendSchedule" variable="shippingSchedule">
      <documentation>Arrange Logistics</documentation>
      <sources>
         <source linkName="ship-to-scheduling" />
      </sources>
   </receive>
   </sequence>
```

```
<sequence>
   <invoke partnerLink="invoicing"
      portType="lns:computePricePT"
      operation="initiatePriceCalculation"
      inputVariable="PO">
      <documentation>
         Initial Price Calculation
      </documentation>
   </invoke>

   <invoke partnerLink="invoicing"
      portType="lns:computePricePT"
      operation="sendShippingPrice"
      inputVariable="shippingInfo">
      <documentation>
         Complete Price Calculation
      </documentation>
      <targets>
         <target linkName="ship-to-invoice" />
      </targets>
   </invoke>

   <receive partnerLink="invoicing"
      portType="lns:invoiceCallbackPT"
      operation="sendInvoice" variable="Invoice" />
</sequence>

<sequence>
   <invoke partnerLink="scheduling"
      portType="lns:schedulingPT"
      operation="requestProductionScheduling"
      inputVariable="PO">
   <documentation>
      Initiate Production Scheduling
   </documentation>
   </invoke>

   <invoke partnerLink="scheduling"
      portType="lns:schedulingPT"
      operation="sendShippingSchedule"
      inputVariable="shippingSchedule">
      <documentation>
         Complete Production Scheduling
      </documentation>
      <targets>
```

```
            <target linkName="ship-to-scheduling" />
          </targets>
        </invoke>
      </sequence>

      </flow>

      <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder" variable="Invoice">
        <documentation>Invoice Processing</documentation>
      </reply>
    </sequence>
</process>
```

Рисунок 11.5

Структура главного раздела определяется внешним элементом <sequence>, в котором содержатся три действия, выполняющихся параллельно. Запрос клиента поступает в секцию <receive>, далее запрос обрабатывается в секции <flow>, в этой секции возможно параллельное выполнение, конечное сообщение ответа отсылается назад к клиенту (секция <reply>). Обратим внимание, что элементы <receive> и <reply> согласованы в соответсвии с элементами <input> и <output> сообщения "sendPurchaseOrder" операции, вызванной клиентом. Действия, которые выполняются между этими элементами, предпринимаются в ответ на запрос клиенат, ответ высылается обратно в секции (<reply>).

Обработка запроса, происходящая в секции < flow >, состоит из трех блоков <sequence >, работающих одновременно. Для осуществления зависимости между тремя параллельными действиями, используется связь посредством ссылок. Обратим внимание, что ссылки определены в секции < flow > и используются для синхронизиции исходной и целевой деятельности. Отметим, что каждая деятельность объявляет себя в качестве источника или цели, осуществляется такая возможность с помощью ссылки на элементы <source> и <target>. При отсутствии ссылок действия, вложенные непосредственно в поток, выполняются одновременно. Рассмотрим наш пример, в нем имеют место две ссылки, которые осуществляют зависимость между действиями, расположенными внутри каждой последовательности. Например, в то время как вычисления цены началось непосредственно после получения заказа, добавление цены к счету может быть осуществлено только после того, как грузоотправитель получил информацию о заказе. Описанная зависимость может быть выполнена посредством использования ссылки (названный "ship-to-invoice"), которая соединяет первый запрос к грузоотправительному провайдеру ("requestShipping") с отсылкой информации к ценообразовотельному сервису ("sendShippingPrice"). Аналогично, информацию по планирования отгрузки товара можно послать в центр планирования производства

только после того, как была получена информация от центра грузоотправителя, эту функциональность выполняет ссылка ("ship-to-scheduling").

Обратите внимание на переменные, которые объявлены как глобальные, через них передается информация между различными действиями неявным способом. В этом примере зависимости управления, представленные ссылками, связаны с соответствующими зависимостями по данным. Такую связь можно наблюдать в одном случае на готовность работать грузоотправиля, в другом на доступность расписания отпрапки продукции. Информацию передают от деятельности, которая генерирует ее к деятельности, которая использует ее посредством двух переменных, определенные как голобальные ("shippingInfo" и "shippingSchedule").

Обратите внимание, что определенные операции могут возвратить ошибки, как это определено в их WSDL описаниях. Для простоты в данном примере предполагается, что две операции возвращают одинаковую ошибку ("cannotCompleteOrder"). Когда происходит ошибка, нормальная обработка процесса останавливается, и управление передается к соответствующему обработчику ошибки, напомним, что обработчик ошибок расположен в разделе <faultHandlers>. В этом примере обработчик использует элемент <reply>, для того, чтобы возвратить ошибку клиенту (обратите внимание на атрибут "faultName" и элемент < reply >).

Наконец, важно отметить, как осуществляеться передача информации между переменными. В примепе рассмотрено простое присваивание, которое передает часть сообщения от переменной источника в переменную назначения, также возможны более сложные формы присваивания.

11.7 Структура бизнес процесса

Здесь в краткой форме приведен сиснтаксис языка WS BPEL.

На рисунке 11.6, приведен листинг с базовой структурой языка

```
<process name="NCName" targetNamespace="anyURI"
   queryLanguage="anyURI"?
   expressionLanguage="anyURI"?
   suppressJoinFailure="yes|no"?
   exitOnStandardFault="yes|no"?
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

   <extensions>?
      <extension namespace="anyURI" mustUnderstand="yes|no" />+
   </extensions>

   <import namespace="anyURI"?
```

```
   location="anyURI"?
   importType="anyURI" />*

<partnerLinks>?
   <!-- Note: At least one role must be specified. -->
   <partnerLink name="NCName"
      partnerLinkType="QName"
      myRole="NCName"?
      partnerRole="NCName"?
      initializePartnerRole="yes|no"?>+
   </partnerLink>
</partnerLinks>

<messageExchanges>?
   <messageExchange name="NCName" />+
</messageExchanges>

<variables>?
   <variable name="BPELVariableName"
      messageType="QName"?
      type="QName"?
      element="QName"?>+
      from-spec?
   </variable>
</variables>

<correlationSets>?
   <correlationSet name="NCName" properties="QName-list" />+
</correlationSets>

<faultHandlers>?
   <!-- Note: There must be at least one faultHandler -->
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      ( faultMessageType="QName" | faultElement="QName" )? >*
      activity
   </catch>
   <catchAll>?
      activity
   </catchAll>
</faultHandlers>

<eventHandlers>?
   <!-- Note: There must be at least one onEvent or onAlarm. -->
   <onEvent partnerLink="NCName"
```

```
        portType="QName"?
        operation="NCName"
        ( messageType="QName" | element="QName" )?
        variable="BPELVariableName"?
        messageExchange="NCName"?>*

        <correlations>?
          <correlation set="NCName" initiate="yes|join|no"? />+
        </correlations>

        <fromParts>?
          <fromPart part="NCName" toVariable="BPELVariableName" />+
        </fromParts>

        <scope ...>...</scope>

      </onEvent>

      <onAlarm>*
        <!-- Note: There must be at least one expression. -->
        (
        <for expressionLanguage="anyURI"?>duration-expr</for>
        |
        <until expressionLanguage="anyURI"?>deadline-expr</until>
        )?
        <repeatEvery expressionLanguage="anyURI"?>
          duration-expr
        </repeatEvery>?
        <scope ...>...</scope>
      </onAlarm>
    </eventHandlers>
    activity
</process>
```

Рисунок 11.6 Базовая структура языка WS BPEL

Опишем атрибуты, расположенные в начале листинга:

queryLanguage. Этот атрибут определяет язык запросов XML, который используется в выборе узлов в присваивании, определении свойств. и в других сферах. Значение по умолчанию для данного атрибута: "urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0", которое представлено применением [XPath 1.0] в WS-BPEL 2.0. Заметим, что ранее XPath 1.0,

располагался по URI спецификации XPath1.0: http://www.w3.org/TR/1999/REC-xpath-19991116;

expressionLanguage. Этот атрибут определяет язык исполнения, который используется в процессе. Значение по умолчанию для данного атрибута: "urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0", которое представлено применением [XPath 1.0] в WS-BPEL 2.0. Ранее XPath 1.0, располагался по URI спецификации XPath1.0: http://www.w3.org/TR/1999/REC-xpath-19991116;

suppressJoinFailure. Этот атрибут определяет, будет ли ошибка joinFailure подавляться во всех деятельностях процесса. Эффект, принятый на уровне процесса, может быть перекрыт другим значением на уровне деятельности. Значение по умолчания для данного атрибута «no»;

enableInstanceCompensatio. Этот атрибут определяет, сможет ли экземпляр процесса в целом быть скомпенсирован средствами определенной платформы. Значение по умолчания для данного атрибута «no»;

abstractProcess Этот атрибут определяет, будет ли процесс исполняемым или абстрактным. Значение по умолчания для данного атрибута «no».

11.8 Основные конструкции деятельности

Основными конструктивными элементами языка являются:

<receive>, <reply>, <invoke>, <assign>, <throw>, <exit>, <wait>, <empty>, <sequence>, <if>, <while>, <repeatUntil>, <forEach>, <pick>, <flow>, <scope>, <compensate>, <compensateScope> , <rethrow>, <validate>, <extensionActivity>.

Конструкция **<receive>** позволяет бизнес процессу перейти в режим блокирующего ожидания, до тех пор, пока всё сообщение не будет получено. Необязательный атрибут messageExchange используется для связывания <reply> деятельности с <receive> деятельности.

```
<receive partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  variable="BPELVariableName"?
  createInstance="yes|no"?
  messageExchange="NCName"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"? />+
  </correlations>
```

```
  <fromParts>?
    <fromPart part="NCName" toVariable="BPELVariableName" />+
  </fromParts>
</receive>
```

Конструкция **<reply>** позволяет бизнес процессу отправлять сообщение в ответ на полученное сообщение, которое было получено входящее сообщение по передаче сообщений, то есть <receive>, <onMessage> или <onEvent>.

Атрибут <reply> формирует запрос-ответ на операцию WSDL PortType процесса. PortType атрибута <reply> деятельность не является обязательным. Если PortType атрибут присутствует, то значение PortType атрибута должено соответствовать сочетанию указанному в PartnerLink. Необязательный атрибут messageExchange используется для связывания <reply> деятельности с дечтельностью по передаче сообщений.

```
<reply partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  variable="BPELVariableName"?
  faultName="QName"?
  messageExchange="NCName"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"? />+
  </correlations>
  <toParts>?
    <toPart part="NCName" fromVariable="BPELVariableName" />+
  </toParts>
</reply>
```

Конструкция **< invoke >** позволяет бизнес процессу позволяет вызывать одностороннюю операцию или операцию запроса ответа на portType, который предлагается партнером.

```
<invoke partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  inputVariable="BPELVariableName"?
  outputVariable="BPELVariableName"?
  standard-attributes>
  standard-elements
```

```
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"?
      pattern="request|response|request-response"? />+
  </correlations>
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    faultMessageType="QName"?
    faultElement="QName"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
  <toParts>?
    <toPart part="NCName" fromVariable="BPELVariableName" />+
  </toParts>
  <fromParts>?
    <fromPart part="NCName" toVariable="BPELVariableName" />+
  </fromParts>
</invoke>
```

Конструкция **< assign >** предназначена для изменения значения переменных. Эта конструкция может использоваться с любым количеством элементарных назначений.

```
<assign validate="yes|no"? standard-attributes>
  standard-elements
  (
  <copy keepSrcElementName="yes|no"? ignoreMissingFromData="yes|no"?>
    from-spec
    to-spec
  </copy>
  |
  <extensionAssignOperation>
    assign-element-of-other-namespace
  </extensionAssignOperation>
  )+
</assign>
```

Деятельность **<validate>** используется для проверки значений переменных на связанные с ними XML и WSDL определения данных. Конструкция имеет атрибут переменные, который указывает на проверяемые переменные.

```
<validate variables="BPELVariableNames" standard-attributes>
  standard-elements
</validate>
```

Конструкция **<throw>** генерирует исключение в бизнес процессе.

```
<throw faultName="QName"
  faultVariable="BPELVariableName"?
  standard-attributes>
  standard-elements
</throw>
```

Конструкция **<wait>** позволяет ожидать в течение данного периода времени, либо до тех пор пока не наступит определенное время.

```
<wait standard-attributes>
  standard-elements
  (
  <for expressionLanguage="anyURI"?>duration-expr</for>
  |
  <until expressionLanguage="anyURI"?>deadline-expr</until>
  )
</wait>
```

Конструкция **<empty>** позволяет вставить инструкцию "no-op"(нет операции) в бизнес процесс. Такая возможность полезна в синхронизации конкурирующих деятельностей в объекте.

```
<empty standard-attributes>
  standard-elements
</empty>
```

Конструкция **<sequence>** позволяет определить коллекциию деятельностей, которые будут выполнятся последовательно в лексическом порядке.

```
<sequence standard-attributes>
  standard-elements
  activity+
```

```
</sequence>
```

Кострукция **<if>** используется для выбора одной из активностей для выполнения.

```
<if standard-attributes>
  standard-elements
  <condition expressionLanguage="anyURI"?>bool-expr</condition>
  activity
  <elseif>*
    <condition expressionLanguage="anyURI"?>bool-expr</condition>
    activity
  </elseif>
  <else>?
    activity
  </else>
</if>
```

Конструкция **<while>** позволяет посторяться деятельности до тех пор, пока не будет достигнуто условие остановки.

```
<while standard-attributes>
  standard-elements
  <condition expressionLanguage="anyURI"?>bool-expr</condition>
  activity
</while>
```

Деятельность **<repeatUntil>** используется, чтобы определить, что дочерняя деятельность должна быть повторена, пока указанное <условие> не становится истенным. <Условие> провереяетя после того, как дочерняя деятельность заканчивается.

```
<repeatUntil standard-attributes>
  standard-elements
  activity
  <condition expressionLanguage="anyURI"?>bool-expr</condition>
</repeatUntil>
```

Конструкция **<forEach>** запускает итерацию его дочерние деятельности N+1, где N раз равна <finalCounterValue> минус <startCounterValue>. Если

parallel="yes", то процессы выполняются параллельно <forEach> , где N +1 экземпляров деятельности <scope> должны происходить параллельно. В сущности, неявный поток создается динамически для N+1 копии <forEach> в области деятельности <scope>, как дочерние. <completionCondition> могут быть использованы в <forEach> чтобы <forEach> деятельности завершить без выполнения или завершения всех ветвей.

```
<forEach counterName="BPELVariableName" parallel="yes|no"
  standard-attributes>
  standard-elements
  <startCounterValue expressionLanguage="anyURI"?>
    unsigned-integer-expression
  </startCounterValue>
  <finalCounterValue expressionLanguage="anyURI"?>
    unsigned-integer-expression
  </finalCounterValue>
  <completionCondition>?
    <branches expressionLanguage="anyURI"?
      successfulBranchesOnly="yes|no"?>?
      unsigned-integer-expression
    </branches>
  </completionCondition>
  <scope ...>...</scope>
</forEach>
```

Конструкция **<pick>** позволяет блокировать процесс до получения соответствующего сообщения или до истечения времени ожидания. Когда срабатывает один из триггеров, выполняется связанная с ним дочерняя деятельность и деятельность <pick> завершаеся.

```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partnerLink="NCName"
    portType="QName"?
    operation="NCName"
    variable="BPELVariableName"?
    messageExchange="NCName"?>+
    <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
    </correlations>
    <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
    </fromParts>
```

```
    activity
  </onMessage>
  <onAlarm>*
    (
    <for expressionLanguage="anyURI"?>duration-expr</for>
    |
    <until expressionLanguage="anyURI"?>deadline-expr</until>
    )
    activity
  </onAlarm>
</pick>
```

Конструкция **<flow>** позволяет специфицировать одно или несколько деятельностей для их параллельного выполнения. Ссылки могут использоваться в пределах параллельных деятельностей, для того, чтобы определить произвольное управление.

```
<flow standard-attributes>
  standard-elements
  <links>?
    <link name="NCName" />+
  </links>
  activity+
</flow>
```

Конструкция **<scope>** позволяет определить вложенные деятельности вместе со связанными с ними преременными, обработчиками ошибок и обработчиками корректировки.

```
<scope isolated="yes|no"? exitOnStandardFault="yes|no"?
  standard-attributes>
  standard-elements
  <partnerLinks>?
    ... see above under <process> for syntax ...
  </partnerLinks>
  <messageExchanges>?
    ... see above under <process> for syntax ...
  </messageExchanges>
  <variables>?
    ... see above under <process> for syntax ...
  </variables>
  <correlationSets>?
```

```
    ... see above under <process> for syntax ...
  </correlationSets>
  <faultHandlers>?
    ... see above under <process> for syntax ...
  </faultHandlers>
  <compensationHandler>?
    ...
  </compensationHandler>
  <terminationHandler>?
    ...
  </terminationHandler>
  <eventHandlers>?
    ... see above under <process> for syntax ...
  </eventHandlers>
  activity
</scope>
```

Конструкция деятельности **<compensateScope>** используется для запуска компенсации на указанную внутреннию область деятельности, которыя уже успешно завершена. Эта деятельность должна быть использована только в пределах обработчика ошибок, или другого обработчика компенсации, или завершеннго обработчика.

```
<compensateScope target="NCName" standard-attributes>
  standard-elements
</compensateScope>
```

Конструкция **<compensate>** используется для вызова корректировки внутри области действия, в которой к этому времени действия выполнились успешно. Эта конструкция может быть вызвана только из соответствующего обработчика ошибок или другого обработчика компенсации.

```
<compensate standard-attributes>
  standard-elements
</compensate>
```

Конструкция деятельности **<EXIT>** используется для немедленного прекращения работы экземпляра бизнес-процесса, который содержится в <EXIT>.

```
<exit standard-attributes>
  standard-elements
</exit>
```

Конструкция деятельности **<rethrow>** используются для, того чтобы повторно прекратить обработку ошибки, которая была первоначально поймана немедленно объемлещим обработчиком ошибок. Конструкция <rethrow> должна быть использована только в обработчике ошибок (т.е. <catch> и <catchAll> элементов). Это синтаксическое ограничение обязательно.

```
<rethrow standard-attributes>
  standard-elements
</rethrow>
```

Элемент **<extensionActivity>** используется для расширения WS-BPEL, вводя новый тип деятельности. Элементы <extensionActivity> должены содержать простые элементы, которые должны быть стандартными атрбутами и стандартными элементами языка WS-BPEL.

```
<extensionActivity>
  <anyElementQName standard-attributes>
    standard-elements
  </anyElementQName>
</extensionActivity>
```

Стандартные атрибуты (см. выше "***standard-attributes***") это:

name="NCName"? suppressJoinFailure="yes|no"?

где значения по умолчанию являются:
- name: нет значения по умолчанию (то есть, по умолчанию не имеет названия)
- suppressJoinFailure: если этот атрибут не указан для деятельности, он наследует свое значение из ближайших объемлющих деятельностей или процессов, если нет объемлющих деятельностей, указывается этот атрибут.

Стандартные элементы (см. выше "***standard- elements*** ") это:

```
<targets>?
  <joinCondition expressionLanguage="anyURI"?>?
    bool-expr
  </joinCondition>
  <target linkName="NCName" />+
</targets>
<sources>?
  <source linkName="NCName">+
```

```
    <transitionCondition expressionLanguage="anyURI"?>?
      bool-expr
    </transitionCondition>
  </source>
</sources>
```

Обратите внимание, что стандартные атрибуты «standard-attributes», для приведенных выше конструкций следующие:

Для данных атрибутов значениями по умолчанию будут следующие:

- **name**. Нет имени по умолчанию (атрибут без имени)
- **joinCondition**. По умолчанию значение логического ИЛИ(OR).
- **suppressJoinFailure**. По умолчанию (No)

11.5 Расширяемость языка WS BPEL

Язык WS BPEL содержит достаточное количество конструкций, которые могут описать абстрактные и выполняемые бизнес процессы. Однако, иногда все таки приходиться расширить возможности WS BPEL с помощью добавления конструкций из пространства имен XML.

WS BPEL поддерживает расширяемость путем использования атрибутов и элементов из пространства имен XML.

Дополнительно, WS-BPEL имеет две явные конструкции расширения: <extensionAssignOperation> и <extensionActivity>.

Расширения языка не должны противоречить семантике любого элемента или атрибута, который определяется в WS-BPEL спецификации.

Расширения допускаются в конструкциях WS-BPEL, используемых в определениях WSDL, типа <partnerLinkType>, <роли>, <vprop:property> и <vprop:propertyAlias>. Тот же самый образец синтаксиса и семантические правила для расширений конструкций WS-BPEL применены к этим расширениям. Для определений WSDL допускается, транзитивные ссылаеться на процесс WS-BPEL, директивы декларации расширения этого процесса WS-BPEL применены ко всем расширениям, используемым в конструкциях WS-BPEL на этих определениях WSDL.

Дополнительная конструкция <documentation> применима к любой WS-BPEL расширяемой конструкции. Как правило, содержимое <documentation> являются аннотации для человека. Пример для тех типов контента: текст, HTML и XHTML. Инструментов реализации конкретной информации (например, графические детали макета) должны быть добавлены с помощью элементов и атрибутов из других пространств имен, используя общие WS-BPEL механизмы расширения.

11.6 Взаимодействие бизнес парнеров

Одним из самых важных, если не самым важным требованием к WS BPEL является взаимодействие бизнес процессов между предприятиями, в ходе которого процессы одной организации взаимодействует с процессами другой организации через интерфейсы Web-сервисов. Для выполнения такого функционала необходимо иметь возможность моделирования специальных зависимостей с процессом-партнером WSDL описывает функциональность предоставляемых сервисов, как на абстрактном, так и на реальном уровне. Отношения бизнес процесса к партнеру чаще всего выражаются в виде односторонних запросов либо двухсторонних зависимостей. Другими словами партнер выступает в роли потребителя и поставщика сервиса относительно другого бизнес процесса. Взаимоотношение между процессами осуществляется посредством асинхронных сообщений, а не с помощью вызова удаленных процедур. Partner links созданы для оформления отношений между процессами, путем определения сообщений и типов портов, используемых в обоих направлениях. Обратим внимание, что реально существующий партнерский сервис может быть вызван непосредственно внутри процесса. WS BPEL использует понятие ссылки на оконечные точки для динамического трансферта между процессами партнерами.

Типы Partner links

Partner links характеризуют диалог между двумя сервисами, эта функциональность выполняется путем определения ролей для каждого из участников диалога, также типов портов для получения сообщений в ходе взаимодействия.

Пример определения синтаксиса <partnerLinkType>:

```
<plnk:partnerLinkType name="BuyerSellerLink">
  <plnk:role name="Buyer" portType="buy:BuyerPortType" />
  <plnk:role name="Seller" portType="sell:SellerPortType" />
</plnk:partnerLinkType>
```

Отметим, что одна роль определяет один порт WSDL.

Обычно тип порта каждой роли берет начало из разных пространств имен. Однако случается, что тип порта ролей определен из одного пространства имен, это значит, что тип partner link использует отношения обратного вызова «callback» между сервисами.

Тип partner link может быть определен независимо от какого-либо документа WSDL, альтернативно тип partner link может быть определен в пределах документа WSDL, определяющего тип порта из которого уже определены типы ролей. Такой механизм позволяет использовать WSDL по несколько раз и, что еще более важно: позволяет импортировать различные типы портов (portTypes). В случае использования partnerLinkType в качестве связующего звена различных сервисов,

а именно их portTypes, объявление partnerLinkType может происходить в отдельном документе WSDL (с собственным targetNamespace).

Синтаксис определения <partnerlinkType>:

```
<wsdl:definitions name="NCName" targetNamespace="anyURI" ...>
  ...
  <plnk:partnerLinkType name="NCName">
    <plnk:role name="NCName" portType="QName" />
    <plnk:role name="NCName" portType="QName" />?
  </plnk:partnerLinkType>
  ...
</wsdl:definitions>
```

Ссылки Partner Links

Сервисы, с которыми взаимодействует бизнес-процесс, в WS BPEL выглядят как партнерские ссылки (Partner Links). Каждый <partnerLink> характеризован <partnerLinkType>. Больше чем один <partnerLink> может быть иметь один и тот же <partnerLinkType>. Например, процесс закупки товара может привлекать более одного продавца, но тем ни менее использует один partnerLinkType .

```
<partnerLinks>
  <partnerLink name="NCName"
    partnerLinkType="QName"
    myRole="NCName"?
    partnerRole="NCName"?
    initializePartnerRole="yes|no"? />+
</partnerLinks>
```

Каждая партнерская ссылка поименована, и это имя используется во всех сервисах, которые используют эту партнерскую ссылку.

Роль бизнес процесса определяется посредством атрибута myRole, роль партнерской ссылки определяется атрибутом partnerRole. В некоторых случаях, когда partnerLinkType есть только одна роль, один из этих атрибутов опускается.

Заметим, что партнерская ссылка (partnerLink) определяет статическую форму отношений, которые WS BPEL будет использовать в своем поведении. До того, как операция на партнерском сервисе будет вызвана с помощью партнерской ссылки, все коммуникационные данные и связи должны быть уже доступны. Основная информация о партнерском сервисе может уточняться в процессе разработки бизнес процесса, такое уточнение выходит за рамки WS BPEL. Тем не менее, в WS BPEL существует метод вызова интересующего сервиса динамически, такая возможность осуществляется посредством вызова оконечных точек. Так как партнеры, скорее всего, участвуют в каких-то зависимостях, оконечные точки

сервисов должны быть описаны в специальной информации объекта. WS BPEL позволяет оконечным точкам неявным образом присутствовать в партнерских ссылках, такой подход позволяет осуществлять динамический вызов (вызов может осуществляться несколько раз).

Бизнес партнеры

В то время как взаимоотношение между процессами устанавливается посредством партнерских ссылок, отношение между бизнес партнерами требует другой связи. WS BPEL предоставляет возможность установки связи между бизнес партнерами, рассмотрим элемент partner (партнер). Элемент partner описывается как подмножество партнерских ссылок, ниже приведен пример:

```
<partner name="SellerShipper"
        xmlns="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
       <partnerLink name="Seller"/>
       <partnerLink name="Shipper"/>
</partner>
```

Элемент partner является опциональным и не требует вхождения всех партнерских ссылок, описанных в процессе.

Определения партнера являются дополнительными и не должны касаться всех ссылок партнера, определенных в процессе. В примере выше описывается партнер, который обязан выполнять сервисы, связанные с ролями продавца и грузоотправителя. Элемент партнер (partners) НЕ ДОЛЖЕН ПЕРЕКРЫВАТЬСЯ, из этого следует, что партнерская ссылка не должна появляться в более чем одном описании партнера.

```
<partners>
       <partner name="ncname">+
         <partnerLink name="ncname"/>+
       </partner>
</partners>
```

Ссылки на оконечные точки (Endpoint References)

WSDL делает большое и значительное различие между понятиями тип порта (portTypes) и порт (ports). Тип порта (PortTypes) определяет абстрактную функциональность посредством использования абстрактных сообщений. Порт осуществляет реальный доступ к информации, описание оконечных точек, а также предоставляет информацию о ключах общего доступа к зашифрованным данным. В то время как пользователь сервиса статически зависит от абстрактного интерфейса, описанного типом порта, некоторая информация, расположенная в порте, может быть прочитана и использована динамически.

Основной задачей ссылок на оконечные точки является предоставление механизма динамической связи определенных данных порта в сервисе. Ссылки на оконечные точки делают возможным динамический выбор провайдера для определенного сервиса, также вызов принадлежащую сервису операцию. WS BPEL предоставляет механизм коррелирования сообщений для зависимых объектов сервиса, по этой причине ссылки на оконечные точки несут в себе информацию о порте (эта информация является достаточной).

Каждая роль партнера в партнерской ссылке в объекте процесса WS BPEL описывается уникальной ссылкой на оконечную точку, которая назначается в ходе развертывания процесса или динамически с помощью деятельности в пределах процесса.

Ссылки конечной точки, связанные с partnerRole и myRole <partnerLink> появляются как контейнера ссылок сервисов (<sref:service-ref>). Этот контейнер используется как конверт, чтобы обернуть фактическую ценность ссылки конечной точки. Здесь шаблон подобен языку выражения, например:

```
<sref:service-ref reference-scheme="http://example.org">
  <foo:barEPR xmlns:foo="http://example.org">...</foo:barEPR>
</sref:service-ref>
```

Здесь <sref:service-ref> имеет необязательный атрибут ссылка-схема для обозначения URI схемы ссылка интерпретации конечной точки службы, которая является дочерним элементом <sref:service-ref>.

URI ссылка-схемы и пространства имен URI дочерняя - элемент <sref:service-ref> не обязательно будет такой же. Дополнительную атрибут ссылку на схему следует использовать, когда ребенок элемент <sref:service-ref> ссылается неоднозначно. Этот необязательный атрибут предоставляет дополнительную информацию для устранения неоднозначности использования контента. Например, если WSDL <wsdl:service> используется в качестве ссылки на конечную точку, могут произойти различные трактовки WSDL<wsdl:service>.

Когда WS-BPEL реализация не в состоянии интерпретировать сочетание ссылку на схему атрибутов и содержимое элемента или просто контент элемента то только стандартные "unsupportedReference" должны быть отброшены.

Элемент <sref:service-ref> не всегда видимый в WS-BPEL определении процесса. Например, он не видимый в назначении из ссылки конечной точки myRole partnerLink-A к тому из partnerRole partnerLink-B. Напротив, он видимый в присвоении MessageType или базированной переменной элемента через выражение или как литерал <sref:service-ref>.

Приложение. Описание XML

XML (англ. *eXtensible Markup Language* — расширяемый язык разметки; произносится [икс-эм-э́ль]) — рекомендованный Консорциумом Всемирной паутины язык разметки, фактически представляющий собой свод общих синтаксических правил. XML — текстовый формат, предназначенный для хранения структурированных данных (взамен существующих файлов баз данных), для обмена информацией между программами, а также для создания на его основе более специализированных языков разметки (например, XHTML). XML является упрощённым подмножеством языка SGML.

Годом рождения XML можно считать 1996 год, в конце которого появился черновой вариант спецификации языка, или 1998 год, когда эта спецификация была утверждена.

Правильно построенные и действительные документы XML

Стандартом определены два уровня правильности документа XML:

- *Правильно построенный* (англ. *well-formed*). Правильно построенный документ соответствует всем общим правилам синтаксиса XML, применимым к любому XML-документу. И если, например, начальный тег не имеет соответствующего ему конечного тега, то это неправильно построенный документ XML. Документ, который неправильно построен, не может считаться документом XML; XML-процессор (парсер) не должен обрабатывать его обычным образом и обязан классифицировать ситуацию как фатальная ошибка.

- *Действительный* (англ. *valid*). Действительный документ дополнительно соответствует некоторым семантическим правилам. Это более строгая дополнительная проверка корректности документа на соответствие заранее определённым, но уже внешним правилам, в целях минимизации количества ошибок, например, структуры и состава данного, конкретного документа или семейства документов. Эти правила могут быть разработаны как самим пользователем, так и сторонними разработчиками, например, разработчиками словарей или стандартов обмена данными. Обычно такие правила хранятся в специальных файлах — схемах, где самым подробным образом описана структура документа, все допустимые названия элементов, атрибутов и многое другое. И если документ, например, содержит не определённое заранее в схемах название элемента, то XML-документ считается *недействительным*; проверяющий XML-процессор (валидатор) при проверке на соответствие правилам и схемам обязан (по выбору пользователя) сообщить об ошибке.

Данные два понятия не имеют достаточно устоявшегося стандартизированного перевода на русский язык, особенно понятие *valid*, которое можно также перевести, как *имеющий силу*, *правомерный*, *надёжный*, *годный*, или даже *проверенный на соответствие правилам, стандартам, законам*. Некоторые программисты применяют в обиходе устоявшуюся кальку «*Валидный*».

Синтаксис XML

В этом разделе рассматривается лишь *правильное построение* документов XML, то есть их синтаксис.

XML — это описанная в текстовом формате иерархическая структура, предназначенная для хранения любых данных. Визуально структура может быть представлена как дерево элементов. Элементы XML описываются тегами.

Рассмотрим пример простого кулинарного рецепта, размеченного с помощью XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<recipe name="хлеб" preptime="5" cooktime="180">
  <title>Простой хлеб</title>
  <ingredient amount="3" unit="стакан">Мука</ingredient>
  <ingredient amount="0.25" unit="грамм">Дрожжи</ingredient>
  <ingredient amount="1.5" unit="стакан">Тёплая вода</ingredient>
  <ingredient amount="1" unit="чайная ложка">Соль</ingredient>
  <instructions>
   <step>Смешать все ингредиенты и тщательно замесить.</step>
   <step>Закрыть тканью и оставить на один час в тёплом помещении.</step>
   <!-- <step>Почитать вчерашнюю газету.</step> - это сомнительный шаг... -->
   <step>Замесить ещё раз, положить на противень и поставить в духовку.
   </step>
  </instructions>
</recipe>
```

Объявление XML

Первая строка XML-документа называется *объявление XML* (англ. *XML declaration*) — это строка, указывающая версию XML. В версии 1.0 *объявление XML* может быть опущено, в версии 1.1 оно обязательно. Также здесь может быть указана кодировка символов и наличие внешних зависимостей.

```
<?xml version="1.0" encoding="UTF-8"?>
```

Спецификация требует, чтобы процессоры XML обязательно поддерживали Юникод-кодировки UTF-8 и UTF-16 (UTF-32 не обязателен). Признаются допустимыми, поддерживаются и широко используются (но не обязательны) другие кодировки, основанные на стандарте ISO/IEC 8859, также допустимы

другие кодировки, например, русские Windows-1251, KOI-8. Часто в тегах принципиально не используют не-латинские буквы, в этом случае UTF-8 является очень удобной кодировкой — объём, как правило, меньше, чем при UTF-16; декодирование может быть выполнено как для всего документа, так и для конкретных атрибутов и текстов; весь документ не содержит запрещённых символов при попытке разбора с неправильной кодировкой.

Корневой элемент

Важнейшее обязательное синтаксическое требование заключается в том, что документ имеет только один *корневой элемент* (англ. *root element*) (так же иногда называемый *элемент документа* (англ. *document element*)). Это означает, что текст или другие данные всего документа должны быть расположены между единственным начальным корневым тегом и соответствующим ему конечным тегом.

Следующий простейший пример — правильно построенный документ XML:

<book>Это книга: "Книжечка"</book>

Следующий пример не является корректным XML-документом, потому что имеет два *корневых элемента*:

<!-- ВНИМАНИЕ! Некорректный XML! -->
<thing>Сущность №1</thing>
<thing>Сущность №2</thing>

Комментарий

В любом месте дерева может быть размещен элемент-*комментарий*. XML-комментарии размещаются внутри специального тега, начинающегося с символов <!-- и заканчивающегося символами -->. Два знака дефис (--) внутри комментария присутствовать не могут.

<!-- Это комментарий. -->

Теги внутри комментария обрабатываться не должны.

Теги

Остальная часть этого XML-документа состоит из вложенных *элементов*, некоторые из которых имеют *атрибуты* и *содержимое*. *Элемент* обычно состоит из открывающего и закрывающего тегов, обрамляющих текст и другие элементы. *Открывающий тег* состоит из *имени элемента* в угловых скобках, например, <step>, а *закрывающий тег* состоит из того же имени в угловых скобках, но перед именем ещё добавляется косая черта, например, </step>. Имена элементов, как и

имена атрибутов, не могут содержать *пробелы*, но могут быть на любом языке, поддерживаемом кодировкой XML-документа. Имя может начинаться с буквы, подчёркивания, двоеточия. Остальными символами имени могут быть те же символы, а также цифры, дефис, точка.

Содержимым элемента (англ. *content*) называется всё, что расположено между открывающим и закрывающим тегами, включая текст и другие (вложенные) элементы. Ниже приведён пример XML-элемента, который содержит открывающий тег, закрывающий тег и содержимое элемента:

```
<step> Замесить ещё раз, положить на противень и поставить в духовку.
</step>
```

Кроме содержания у элемента могут быть *атрибуты* — пары имя-значение, добавляемые в открывающий тег после названия элемента. Значения атрибутов всегда заключаются в кавычки (одинарные или двойные), одно и то же имя атрибута не может встречаться дважды в одном элементе. Не рекомендуется использовать разные типы кавычек для значений атрибутов одного тега.

```
<ingredient amount="3" unit="стакан">Мука</ingredient>
```

В приведённом примере у элемента «ingredient» есть два атрибута: «amount», имеющий значение «3», и «unit», имеющий значение «стакан». С точки зрения XML-разметки, приведённые атрибуты не несут никакого смысла, а являются просто набором символов.

Кроме текста, элемент может содержать другие элементы:

```
<instructions>
 <step>Смешать все ингредиенты и тщательно замесить.</step>
 <step>Закрыть тканью и оставить на один час в тёплом помещении.</step>
 <step>Замесить ещё раз, положить на противень и поставить в духовку.
</step>
</instructions>
```

В данном случае элемент «instructions» содержит три элемента «step».

XML не допускает перекрывающихся элементов. Например, приведённый ниже фрагмент некорректен, так как элементы «em» и «strong» перекрываются.

```
<!-- ВНИМАНИЕ! Некорректный XML! -->
<p>Обычный <em>акцентированный <strong>выделенный и акцентированный</em> выделенный</strong></p>
```

Для обозначения элемента без содержания, называемого *пустым элементом*, необходимо применять особую форму записи, состоящую из одного тега, в котором

после имени элемента ставится косая черта. Если в DTD элемент не объявлен пустым, но в документе он не имеет содержания, для него допускается применять следующие (три) формы записи. Например:

```
<foo></foo>
<foo />
<foo/>
```

Спецсимволы

В XML определены два метода записи специальных символов: ссылка на сущность и ссылка по номеру символа.

Сущностью (англ. *entity*) в XML называются именованные данные, обычно текстовые, в частности, спецсимволы. *Ссылка на сущность* (англ. *entity references*) указывается в том месте, где должна быть сущность и состоит из амперсанда (&), имени сущности и точки с запятой (;).

В XML есть несколько предопределённых сущностей, таких как lt (ссылаться на неё можно написав <) для левой угловой скобки и amp (ссылка — &) для амперсанда, возможно также определять собственные сущности. Помимо записи с помощью сущностей отдельных символов, их можно использовать для записи часто встречающихся текстовых блоков.

Ниже приведён пример использования предопределённой сущности для избежания использования знака амперсанда в названии:

```
<company-name>AT&amp;T</company-name>
```

Полный список предопределённых сущностей состоит из & (&), < (<), > (>), ' (') и " (") — последние две полезны для записи разделителей внутри значений атрибутов. Определить свои сущности можно в DTD-документе.

Иногда бывает необходимо определить неразрывный пробел, который очень часто используется в HTML и обозначается как . В XML такой предопределённой сущности нет, его записывают , а использование   вызывает ошибку. Отсутствие этой весьма распространённой сущности у множества программистов зачастую вызывает удивление и это создаёт некоторые трудности при миграции своих HTML-разработок в XML.

Ссылка по номеру символа (англ. *numeric character reference*) выглядит как ссылка на сущность, но вместо имени сущности указывается символ # и число (в десятичной или шестнадцатеричной записи), являющееся номером символа в кодовой таблице Юникод. Это обычно символы, которые невозможно закодировать напрямую, например, буква арабского алфавита в ASCII-

кодированном документе. Амперсанд может быть представлен следующим образом:

<company-name>AT&T</company-name>

Существуют и другие правила, касающиеся составления корректного XML-документа.

Достоинства

- XML — язык разметки, позволяющий стандартизировать вид файлов-данных, используемых компьютерными программами, в виде текста, понятного человеку;
- XML поддерживает Юникод;
- в формате XML могут быть описаны такие структуры данных как записи, списки и деревья;
- XML — это самодокументируемый формат, который описывает структуру и имена полей так же как и значения полей;
- XML имеет строго определённый синтаксис и требования к анализу, что позволяет ему оставаться простым, эффективным и непротиворечивым. Одновременно с этим, разные разработчики не ограничены в выборе экспрессивных методов (например, можно моделировать данные, помещая значения в параметры тегов или в тело тегов, можно использовать различные языки и нотации для именования тегов и т. д.);
- XML — формат, основанный на международных стандартах;
- Иерархическая структура XML подходит для описания практически любых типов документов, кроме аудио и видео мультимедийных потоков, растровых изображений, сетевых структур данных и двоичных данных;
- XML представляет собой простой текст, свободный от лицензирования и каких-либо ограничений;
- XML не зависит от платформы;
- XML является подмножеством SGML (который используется с 1986 года). Уже накоплен большой опыт работы с языком и созданы специализированные приложения;
- XML не накладывает требований на порядок расположения атрибутов в элементе и вложенных элементов разных типов, что существенно облегчает выполнение требований обратной совместимости;
- В отличие от бинарных форматов, XML содержит метаданные об именах, типах и классах описываемых объектов, по которым приложение может обработать документ неизвестной структуры (например, для динамического построения интерфейсов);
- XML имеет реализации парсеров для всех современных языков программирования;

- Существует стандартный механизм преобразования XSLT, реализации которого встроены в браузеры, операционные системы, веб-серверы.
- XML поддерживается на низком аппаратном, микропрограммном и программном уровнях в современных аппаратных решениях.
-

Недостатки

- Синтаксис XML избыточен.

- Размер XML-документа существенно больше бинарного представления тех же данных. В грубых оценках величину этого фактора принимают за 1 порядок (в 10 раз).
- Размер XML-документа существенно больше, чем документа в альтернативных текстовых форматах передачи данных (например JSON[2], YAML, Protocol Buffers) и особенно в форматах данных, оптимизированных для конкретного случая использования.
- Избыточность XML может повлиять на эффективность приложения. Возрастает стоимость хранения, обработки и передачи данных.
- XML содержит метаданные (об именах полей, классов, вложенности структур), и одновременно XML позиционируется как язык взаимодействия открытых систем. При передаче между системами большого количества объектов одного типа (одной структуры), передавать метаданные повторно нет смысла, хотя они содержатся в каждом экземпляре XML описания.
- Для большого количества задач не нужна вся мощь синтаксиса XML и можно использовать значительно более простые и производительные решения.

- Неоднозначность моделирования.

- Нет общепринятой методологии для моделирования данных в XML, в то время как для реляционной модели и объектно-ориентированной такие средства разработаны и базируются на реляционной алгебре, системном подходе и системном анализе.
- В природе есть множество объектов и явлений, для описания которых разные структуры данных (сетевая, реляционная, иерархическая) являются естественными, и отображение объекта в неестественную для него модель является болезненным для его сути. В случае с реляционной и иерархической моделями определены процедуры декомпозиции, обеспечивающие относительную однозначность, чего нельзя сказать о сетевой модели.
- В результате большой гибкости языка и отсутствия строгих ограничений, одна и та же структура может быть представлена множеством способов (различными разработчиками), например, значение может быть записано как атрибут тега или как тело тега и т. д. Например: или или <a>1<c>1</c> или <a><c value="1"/> или <a><fields b="1" c="1"/> и т. д.

- Поддержка многих языков в именовании тегов дает возможность назвать, например вес русским словом, в таком случае компьютер никак не сможет установить соответствия этого поля с полем weight в англоязычной версии программы и с полями в версиях модели объекта на множестве других языков.
- XML не содержит встроенной в язык поддержки типов данных. В нём нет строгой типизации, то есть понятий «целых чисел», «строк», «дат», «булевых значений» и т. д.
- Иерархическая модель данных, предлагаемая XML, ограничена по сравнению с реляционной моделью и объектно-ориентированными графами и сетевой моделью данных.
- Выражение неиерархических данных (например графов) требует дополнительных усилий
- Кристофер Дейт, специалист в области реляционных баз данных, автор классического учебника «An Introduction to Database Systems», отмечал, что «…XML является попыткой заново изобрести иерархические базы данных…»[] (в 1980-е года иерархические базы данных были вытеснены реляционными базами данных).
- Пространства имён XML сложно использовать и их сложно реализовывать в XML-парсерах.
- Существуют другие, обладающие сходными с XML возможностями, текстовые форматы данных, которые обладают более высоким удобством чтения человеком (YAML, JSON, SweetXML, XF).

Отображение XML во Всемирной паутине

Наиболее распространены три способа преобразования XML-документа в отображаемый пользователю вид:

1. Применение стилей CSS;
2. Применение XSL;
3. Написание на каком-либо языке программирования обработчика XML-документа.

Без использования CSS или XSL XML-документ отображается как простой текст в большинстве веб-браузеров. Некоторые браузеры, такие как Internet Explorer и Mozilla Firefox отображают структуру документа в виде дерева, позволяя сворачивать и разворачивать узлы с помощью нажатий клавиши мыши.

Применение стилей CSS

Процесс аналогичен применению CSS к HTML-документу для отображения.

Для применения CSS при отображении в браузере, XML-документ должен содержать специальную ссылку на таблицу стилей. Например:

```
<?xml-stylesheet type="text/css" href="myStyleSheet.css"?>
```

Это отличается от подхода HTML, где используется элемент <link>.

Применение XSL

XSL является семейством рекомендаций, описывающих языки преобразования и визуализации XML-документов. Документ трансформируется в формат, подходящий для отображения в браузере. Браузер — это наиболее частое использование XSL, но не стоит забывать, что с помощью XSL можно трансформировать XML в любой формат, например VRML, PDF, текст.

Для задания XSL трансформации (XSLT) на стороне клиента требуется наличие в XML инструкции следующего вида:

<?xml-stylesheet type="text/xsl" href="transform.xsl"?>

Словари XML

Так как XML является достаточно абстрактным языком, были разработаны словари XML.

Словарь позволяет разработчикам договориться о некотором конечном наборе имен тегов и атрибутов этих тегов. Одним из первых словарей является XHTML, который понимают большинство браузеров. XHTML часто используют для хранения и редактирования контента в CMS.

Были созданы более специализированные словари, например протокол передачи данных SOAP, который не является человеко-ориентированным и достаточно трудно читаем. Есть коммерческие словари, такие как CommerceML, xCBL и cXML которые используются для передачи данных, ориентированных на торговую деятельность, эти словари включают в себя описание системы заказов, поставщиков, продуктов и прочее.

Обычно, описывая какой-либо документ, человек для себя придумывает некоторый словарь, который потом описывается посредством DTD или просто объясняется «на пальцах» заинтересованным лицам.

Одним из словарей, получивших широкое распространение, является FB2 — словарь, описывающий формат книги, со всевозможными сносками, цитатами, даже картинками.

Версии XML

- XML 1.0
- XML 1.1

Аннотированные схемы SQLXML

Схемы XSD часто используются для описания содержимого и структуры XML-данных, следовательно, можно создать XSD-схему, которая описывает структуру RSS-каналов. В нашем приложении мы не будем создавать XSD-схему для общих RSS-каналов, а сосредоточим внимание на подмножестве структуры XML, используемой большинством каналов. Канал, использовавшийся при написании этой статьи, основан на Yahoo! News RSS Feed (EN). Нам необходимо аннотировать XSD-схему с помощью реляционных метаданных, которые применяются в SQLXML для преобразования XML-кода в формат реляционной БД. Ознакомьтесь с аннотированной схемой, используемой для нашего приложения.

Листинг 1. Аннотированная схема, используемая для нашего приложения

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
 <xsd:element name="rss" sql:is-constant="1">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="channel" sql:is-constant="1">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="title" type="xsd:string" sql:mapped="false" />
       <xsd:element name="link" type="xsd:string" sql:mapped="false"/>
       <xsd:element name="description" type="xsd:string" sql:mapped="false"/>
       <xsd:element name="language" type="xsd:string" sql:mapped="false"/>
       <xsd:element name="lastBuildDate" type="xsd:string"
sql:mapped="false"/>
       <xsd:element name="ttl" type="xsd:int" sql:mapped="false"/>
       <xsd:element name="image" sql:mapped="false">
        <xsd:complexType>
         <xsd:sequence>
          <xsd:element name="title" type="xsd:string" />
          <xsd:element name="width" type="xsd:int" />
          <xsd:element name="height" type="xsd:int" />
          <xsd:element name="link" type="xsd:string" />
          <xsd:element name="url" type="xsd:string" />
         </xsd:sequence>
        </xsd:complexType>
       </xsd:element>
       <xsd:element name="item" sql:relation="NewsFeed">
        <xsd:complexType>
         <xsd:sequence>
          <xsd:element name="title" type="xsd:string"/>
          <xsd:element name="link" type="xsd:string"/>
```

```
                    <xsd:element name="guid" sql:field="guid">
                      <xsd:complexType>
                        <xsd:simpleContent>
                          <xsd:extension base="xsd:string">
                            <xsd:attribute name="isPermaLink" type="xsd:string"
sql:field="IsPermanent"/>
                          </xsd:extension>
                        </xsd:simpleContent>
                      </xsd:complexType>
                    </xsd:element>
                    <xsd:element name="pubDate" type="xsd:string"
sql:field="PublicationDate" />
                    <xsd:element name="description" type="xsd:string"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="version" type="xsd:string" sql:mapped="false"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Приведенная выше XSD-схема содержит аннотации реляционных данных, необходимые для преобразования. Все аннотации преобразования, относящиеся к SQLXML, принадлежат пространству имен xmlns:sql="urn:schemas-microsoft-com:mapping-schema". Наиболее важными являются аннотация sql:relation элемента item и различные аннотации sql:field, относящиеся к его дочерним элементам. Аннотация sql:relation обеспечивает преобразование элемента item в таблицу NewsFeed, а аннотация sql:field — преобразование дочерних элементов в столбцы этой таблицы. Атрибуты элемента также можно преобразовывать в столбцы таблицы. На самом деле аннотация sql:field является необязательной, поскольку отображение «таблица-элемент» охватывает все дочерние элементы и атрибуты данного элемента. В приведенной выше XSD-схеме только элементы isPermaLink и pubDate снабжены аннотацией sql:field, поскольку имена соответствующих столбцов отличаются. К именам остальных элементов применяются стандартные правила преобразования: они автоматически преобразуются в одноименные столбцы.

В то же время RSS-канал включает множество других элементов, таких как channel, ttl и image, которые в нашем случае не требуется преобразовывать в формат

реляционной базы данных. Для решения этой проблемы в SQLXML можно использовать аннотации sql:is-constant или sql:mapped.

Аннотация sql:is-constant в основном используется для создания элементов-обёрток, которые не отображаются в базу данных. В нашем случае элементы rss и channel являются элементами-обёртками в RSS-канале, поэтому для них используется аннотация sql:is-constant. При использовании аннотации sql:is-constant необходимо принимать во внимание следующее. Во-первых, она может применяться только к элементам сложного типа. Во-вторых, если постоянный (constant) элемент обладает атрибутами, эти атрибуты не могут быть преобразованы в столбцы базы данных.

Аннотация sql:mapped позволяет явным образом указать, что элемент или атрибут не нужно преобразовывать в какие-либо объекты реляционной базы данных. Эта аннотация несколько раз использовалась в схеме. Я пометил атрибут version аннотацией sql:mapped="false" явным образом, поскольку он является частью постоянного элемента. Я также аннотировал с помощью sql:mapped="false" все дочерние элементы channel, имеющие простой тип. Эту аннотацию я также использую для элемента image. Необходимо отметить, что image — это элемент сложного типа, поэтому, аннотировав его, я предотвратил преобразование самого этого элемента, а также всех его дочерних элементов и атрибутов. Аннотацию sql:mapped можно использовать как для элементов простого, так и для элементов сложного типа. В последнем случае все дочерние элементы и атрибуты, входящие в состав сложного типа, также получают значение аннотации sql:mapped.

Преимущество использования схемы преобразования для отображения входных XML-данных в базу данных заключается в том, что одно и то же отображение можно использовать как для запросов к базе данных, так и для генерирования различных XML-представлений данных. Для этих целей в SQLXML предусмотрена функцию XPath.

Использование Bulkload из управляемого кода

Функция SQLXML Bulkload позволяет загружать входные данные XML в реляционную базу данных. Эта функция использует процесс bcp сервера SQL Server и является оптимальным механизмом для загрузки больших объемов входных XML-данных на сервер. Функция реализована как COM-объект и использует поставщиков SQLOLEDB. Благодаря этому ее удобно использовать при программировании через ActiveX Data Objects (ADO), а также при использовании языков сценариев, например VBScript. В нашем случае функция Bulkload используется в приложении .NET. Для реализации этой функции можно воспользоваться одним из указанных ниже способов:

использовать программу tlbimp.exe (EN), чтобы импортировать определения типов COM в runtime-сборку, а затем добавить сгенерированную Interop-сборку в проект;

добавить в проект Visual Studio.NET ссылку непосредственно на DDL-библиотеку Bulkload (xblkld3.dll). При этом будет создана Interop-сборка, которую можно использовать в проекте.

После выполнения любой из приведенных процедур использование Bulkload не составляет особого труда:

Листинг 2. Использование Bulkload

```
...
SQLXMLBULKLOADLib.SQLXMLBulkLoad3Class objBL = new SQLXMLBULKLOADLib.SQLXMLBulkLoad3Class();
objBL.ConnectionString = "Provider=sqloledb;server=server;database=databaseName;integrated security=SSPI";
objBL.ErrorLogFile = "error.xml";
objBL.Execute ("schema.xml","data.xml");
...
```

Загрузка базы данных с помощью SQLXML Bulkload

Итак, выше продемонстрирован процесс преобразования и описание основных принципов работы функции Bulkload. Теперь попытаемся применить полученные знания на практике. В приведенном выше фрагменте кода входные XML-данные считываются из файла на локальном диске. В нашем приложении XML-данные для RSS-канала доступны через URL-адрес. В большинстве случаев входные данные следует обрабатывать в виде потока, поскольку не всегда возможно сохранить их в виде файла для последующей загрузки. Метод Execute в Bulkload существует в нескольких перегруженных вариантах; в одном случае в качестве источника данных используется поток, а в другом — файл данных. Однако все не так просто, как может показаться на первый взгляд. В нашем случае мы не можем просто воспользоваться одним из управляемых классов Stream из пространства имен System.IO.Stream. Поскольку функция Bulkload — это COM-объект, поток для нее должен быть естественным типом IStream, однако ни один из классов потоков, являющихся производными класса System.IO.Stream в структуре .NET, не реализует IStream. Это несколько усложняет наше приложение, поскольку теперь нам необходимо преобразовывать входные XML-данные в поток, распознаваемый Bulkload.

Ниже приведен пример реализации данного интерфейса:

Листинг 3. Объект-оболочка потока, реализующий интерфейс UCOMIStream

```
internal class UCOMStreamWrapper : UCOMIStream
{
```

```
Stream innerStream;
  String _file;
  public UCOMStreamWrapper(Stream inner, string file)
  {
    innerStream = inner;
    _file       = file;
  }
  public UCOMStreamWrapper(Stream inner)
  {
    innerStream = inner;
    _file = null;
  }
  public void Read(byte[] pv, int cb, IntPtr pcbRead)
  {
    Marshal.WriteInt32(pcbRead, innerStream.Read(pv, 0, cb));
  }
  public void Stat(out STATSTG pstatstg, int grfStatFlag)
  {
    DateTime curTime;
    long     curFileTime;
    pstatstg = new STATSTG();
    if (_file == null)
    {
      curTime = DateTime.Now;
    }
    else
    {
      if (grfStatFlag == 0)   // default
      {
        pstatstg.pwcsName=_file;
      }
      else                    // noname
      {
        pstatstg.pwcsName=null;
      }
      curTime = File.GetLastWriteTime(_file);
    }
    curFileTime = curTime.ToFileTime();
    pstatstg.cbSize = innerStream.Length;
    pstatstg.type = 2;                // STGM_READ
    pstatstg.grfMode = 0;             // STGM_READ
    pstatstg.grfLocksSupported = 2;     // LOC_EXCLUSIVE
    pstatstg.mtime.dwHighDateTime = (int) (curFileTime >> 32);
    pstatstg.mtime.dwLowDateTime  = (int) (curFileTime & 0xffffffff);
    pstatstg.atime = pstatstg.mtime;
```

```
    }
    public void Clone(out UCOMIStream ppstm)
    {
      throw new NotImplementedException("UCOMIStream: Clone");
    }
    public void Commit(int grfCommitFlags)
    {
      throw new NotImplementedException("UCOMIStream: Commit");
    }
    public void CopyTo(UCOMIStream pstm, long cb, IntPtr pcbRead, IntPtr
pcbWritten)
    {
      throw new NotImplementedException("UCOMIStream: CopyTo");
    }
    public void LockRegion(long libOffset, long cb, int dwLockType)
    {
      throw new NotImplementedException("UCOMIStream: LockRegion");
    }
    public void Revert()
    {
      throw new NotImplementedException("UCOMIStream: Revert");
    }

    public void Seek(long dlibMove, int dwOrigin, IntPtr plibNewPosition)
    {
      long newPosition = 0;
      if (innerStream.CanSeek == true)
      {
        SeekOrigin origin = SeekOrigin.Begin;
        switch (dwOrigin)
        {
          case 1:
            origin = SeekOrigin.Current;
            break;
          case 2:
            origin = SeekOrigin.End;
            break;
        }
        newPosition = innerStream.Seek(dlibMove, origin);
      }
      if (plibNewPosition != (IntPtr)0)
      {
        Marshal.WriteInt64(plibNewPosition, newPosition);
      }
    }
```

```
  public void SetSize(long libNewSize)
  {
    throw new NotImplementedException("UCOMIStream: SetSize");
  }
  public void UnlockRegion(long libOffset, long cb, int dwLockType)
  {
    throw new NotImplementedException("UCOMIStream: UnlockRegion");
  }
  private void Write(byte[] buf, int offset, int len)
  {
    innerStream.Write(buf, offset, len);
  }
  public void Write(byte[] pv, int cb, IntPtr pcbWritten)
  {
    Write(pv, 0, cb);
    if (pcbWritten != (IntPtr)0)
    {
      Marshal.WriteInt32(pcbWritten, cb);
    }
  }
}
```

Как видно из приведенного фрагмента кода, это не полная реализация интерфейса UCOMIStream. Для функции Bulkload требуется реализовать только следующие методы: Read, Write, Stat и Seek. Можно передать класс UCOMStreamWrapper непосредственно методу Execute в Bulkload. Приведенное выше решение является эффективным способом передачи потоковых данных функции Bulkload в приложении .NET.

Литература

1. Integration Definition for Functional Modeling (IDEF0), NIST USA, 80p.
2. Integration Definition for Information Modeling (IDEF1x), NIST USA, 148p.
3. Information Integration for Concurrent Engineering (IICE), IDEF3 Process Description Capture Method, Report (IDEF3), Knowledge Based System, Inc USA, 224p.
4. Information Integration for Concurrent Engineering (IICE), IDEF4 Object – Oriented Design Method, Report, Knowledge Based System, Inc USA, 152p.
5. Официальный сайт Knowledge Based Systems, Inc. (KBSI), ссылки
   http://www.kbsi.com/solutions-and-services/idef-methods-and-standards
   http://www.idef.com/idef0.htm
   http://www.idef.com/IDEF3.htm
   http://www.idef.com/IDEF1x.htm
6. Официальный сайт компании CA в России
   http://erwin.com/worldwide/russian-russia
7. МЕТОДОЛОГИЯ ФУНКЦИОНАЛЬНОГО МОДЕЛИРОВАНИЯ IDEF0**.** © ИПК Издательство стандартов, 2000. – 75с.
8. ИСО/МЭК 12207 – 95 «Информационная технология. Процессы жизненного цикла программных средств» или ISO/IEC 12207 (ISO – International Organization of Standardization)
9. ГОСТ 34.601–90 Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы, стадии создания.
10. Маклаков С.В. Создание информационных систем с AllFusion Modeling Suite. – М.:ДИАЛОГ–МИФИ, 2003. – 432с.
11. Пилецкий И.И. Пособие по курсу «МЕТОДЫ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ» для студ. спец. 1-31 03 04 «Информатика» Проектирование, разработка и сопровождение баз данных с использованием CASE средств. Минск: БГУИР, 2009. –116 с.
12. Трофимов С.А. Case–технологии: Практическая работа в Rational Rose. Изд. 2–е – М.:Бином –Пресс, 2002 г. – 228с.
13. Кратчен Ф. Введение в Rational Unified Process. 2–е изд., Пер. с англ. – М.: Издательский дом «Вильямс», 2002. –240с.
14. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. М.: ДМК, 2000 г. – 420с.
15. Кролл П., Крачтен Ф. Rational Unified Process – это легко. Руководство по RUP. Пер. с англ. – М.: КУДИЦ-ОБРАЗ, 2004. – 432 с.
16. Кватрани Т. Rational Rose 2000 и UML. Визуальное моделирование: Пер. с англ. – М.: ДМК Пресс, 2001. – 176 с.
17. Фаулер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования. Пер. с англ. – М.: Мир, 1999. – 191 с.
18. Смит К., Уильямс Л. Эффективные решения: практическое руководство по созданию гибкого и масштабируемого программного обеспечения. Пер. с англ. – М.: Издательский дом «Вильямс», 2003. –448с.

19. Бобровский С. Технологии Пентагона на службе российских программистов. Программная инженерия. – СПб.: Питер, 2003. – 222с.

20. Майерс Г. Искусство тестирования программ. М., "Финансы и статистика", 1982. – 174 с.

21. Myers G., Glenford J. The art of software testing/ G.J. Myers; — 2-nd ed. Published by John Wiley & Sons, Inc., Hoboken, New Jersey, 2004. – 234 p.

22. Официальный сайт компании IBM
http://www.ibm.com/developerworks/ru/webservices/newto

http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

23. Сайт OMG http://www.omg.org/spec/BPMN/2.0

24. Официальный сайт компании Mercury Interactive http://www.mercury.com

25. Официальный сайт компании Segue Software http://www.segue.com